

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

## 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

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

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 compresión 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 lectocomprendión 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.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

## 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 pueda mantener de mejor 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 strikes<sup>1</sup> (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.

<sup>1</sup> 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

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

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

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

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 pasa 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)*: Ocurre cuando una clase tiene atributos que se usan solo en algunos métodos específicos y no son esenciales para la clase en general. Estos atributos pueden hacer que la clase sea más difícil de entender y mantener, ya que no son verdaderamente parte del estado permanente del objeto.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

- (2) *Message chains (cadena de mensajes)*: Ocurre cuando se tiene múltiples niveles de invocaciones de métodos a través de objetos, generando una cadena larga de llamadas como `objetoA.getObjetoB().getObjetoC().hacerAlgo()`. Este patrón indica un fuerte acoplamiento entre las clases, lo que puede llevar a un código frágil y difícil de mantener.
- (3) *Divergent change (cambio divergente)*: Ocurre cuando una clase se modifica frecuentemente por diferentes razones o para diferentes propósitos, violando el principio de responsabilidad única. Esto significa que la clase tiene demasiadas responsabilidades y, como consecuencia, se ve afectada por múltiples tipos de cambios.
- (4) *Shotgun surgery (cambio en cadena)*: Ocurre cuando hacer un pequeño cambio en una clase provoca la necesidad de hacer cambios en muchas otras clases. Esto generalmente indica que el comportamiento o los datos están dispersos por todo el sistema, creando un fuerte acoplamiento entre las clases y dificultando el mantenimiento.  
Este problema puede hacer que el código sea difícil de modificar y propenso a errores, ya que cada pequeño cambio implica realizar modificaciones en varias clases, violando el **principio de encapsulación** y el **principio de responsabilidad única**.
- (5) *Data class (clase de datos)*: Aparece cuando una clase solo tiene atributos y métodos de acceso (getters y setters) y no tiene ninguna lógica o comportamiento significativo. Estas clases actúan simplemente como contenedores de datos y, a menudo, son un síntoma de falta de diseño, ya que no encapsulan adecuadamente el comportamiento asociado a los datos.  
Este tipo de clase puede generar varios problemas:
- No encapsula bien la lógica, ya que la lógica de negocio puede terminar dispersa en otras clases que utilizan esta clase de datos.
  - Puede haber duplicación de lógica en múltiples lugares.
  - Hace que el sistema sea más difícil de mantener y extender.
- (6) *Dead code (código muerto)*: Aparece cuando variables, atributos, métodos o clases ya no se usan, ya sea porque cambiaron los requerimientos del proyecto o por correcciones posteriores que hicieron que ya no fueran necesarios. Mantener este tipo de código es perjudicial, ya que añade ruido innecesario, complica el mantenimiento y puede confundir a los desarrolladores.
- (7) *Large class (clase grande)*: Aparece cuando una clase tiene demasiados atributos, métodos y líneas de código, lo que la convierte en una clase que intenta manejar demasiadas responsabilidades. Esto viola el **principio de responsabilidad única (SRP)** y conduce a un código difícil de mantener, propenso a errores y posiblemente con código duplicado. Este tipo de clase tiende a acumular lógica no relacionada, lo que puede provocar la aparición de otros bad smells.
- (8) *Lazy class (clase perezosa)*: Una "Lazy Class" es una clase que no aporta valor significativo al programa y está subutilizada. Cada clase en un sistema de software conlleva costos de mantenimiento (documentación, pruebas, y esfuerzo para comprender su propósito).  
Si una clase no tiene suficientes responsabilidades o su funcionalidad es tan pequeña que no justifica su existencia, debe ser eliminada o fusionada con otra clase más apropiada.

 UNIVERSIDAD <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
APUNTE DE REFACTORIZACIÓN		

- (9) *Comments (comentarios)*: Los comentarios son útiles en el código, pero cuando son excesivos o innecesarios, pueden indicar que algo está mal en el diseño. Si un bloque de código necesita comentarios extensos para explicarlo, probablemente el código no es lo suficientemente claro o no está bien estructurado. El código bien diseñado debería ser lo suficientemente autoexplicativo, es decir, que su intención sea clara sin necesidad de añadir comentarios extensivos.
- (10) *Duplicated code (código duplicado)*: El código duplicado es uno de los bad smells más comunes y una de las principales razones para refactorizar. Se refiere a situaciones en las que el mismo bloque de código aparece en más de un lugar dentro de un proyecto, o cuando varias partes del código realizan operaciones similares, pero no idénticas. El código duplicado genera problemas de mantenimiento porque cualquier cambio debe replicarse en todos los lugares donde existe la duplicación. Tipos de duplicación:
  - **Duplicación directa**: El mismo código aparece en varios lugares.
  - **Duplicación sutil**: Fragmentos de código que son diferentes en apariencia, pero en realidad realizan la misma tarea o tienen la misma lógica subyacente.
- (11) *Feature envy (envidio de características)*: Ocurre cuando un método de una clase accede más frecuentemente a los atributos o métodos de otra clase que a los de su propia clase. Este comportamiento indica que el método probablemente esté en la clase incorrecta y debería ser movido a la clase de la que más depende. Este bad smell viola el **principio de cohesión** de las clases y es un síntoma de mal diseño.
- (12) *Switch statements (estructura de agrupación condicional)*: Aparece cuando en un programa se encuentran sentencias switch o múltiples if-else anidados en diferentes partes del código para manejar decisiones basadas en el tipo de un objeto. Este tipo de código es propenso a errores porque, cada vez que se agrega una nueva condición o tipo, es necesario modificar todas las sentencias switch o if-else en todas las partes donde se encuentran, lo que rompe el principio Abierto/Cerrado (abierto para la extensión, cerrado para la modificación).
- (13) *Speculative generality (generalidad especulativa)*: Ocurre cuando se agregan funcionalidades al código "por si acaso" que podrían ser necesaria en el futuro, pero que actualmente no tiene uso. Este tipo de código introduce complejidad innecesaria, lo que lo hace más difícil de mantener, entender y probar. Si este comportamiento no se utiliza o es innecesario, debe eliminarse para mantener el código simple y directo.
- (14) *Data clumps (grupo de datos)*: Ocurre cuando ciertos elementos de datos (por ejemplo, atributos o parámetros) siempre aparecen juntos en varios lugares del código, ya sea como parámetros en múltiples métodos o como atributos en varias clases. Esto indica que esos elementos están relacionados entre sí y deberían agruparse en una clase propia para mejorar la cohesión y facilitar el mantenimiento.
- (15) *Middleman (intermediario)*: Ocurre cuando una clase no hace mucho por sí misma y delega la mayor parte de sus responsabilidades o comportamientos a otra clase. Esto rompe con el principio de encapsulación, ya que la clase intermediaria no tiene un propósito claro y simplemente actúa como un puente innecesario.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
APUNTE DE REFACTORIZACIÓN		

Este bad smell puede aparecer como resultado de haber aplicado demasiada delegación para evitar otros bad smells como "**Message Chains**" (cadenas de mensajes), pero termina generando una clase que añade poca o ninguna funcionalidad.

- (16) *Inappropriate intimacy (intimidad inadecuada)*: Se presenta cuando una clase accede directamente a los atributos o métodos privados o protegidos de otra clase, violando el principio de encapsulación. Esto crea un acoplamiento muy fuerte entre las clases, lo que dificulta el mantenimiento, reduce la modularidad y aumenta la complejidad. Idealmente, las clases deberían interactuar a través de interfaces o métodos públicos claramente definidos, respetando los límites de encapsulación.
- (17) *Parallel inheritance hierarchies (jerarquías paralelas)*: Ocurre cuando dos jerarquías de clases crecen en paralelo. Esto significa que cada vez que se crea una subclase en una jerarquía, también se debe crear una subclase correspondiente en la otra jerarquía. Este tipo de diseño causa problemas de duplicación de código y hace que el sistema sea más difícil de mantener, ya que cualquier cambio o adición en una jerarquía requiere cambios en la otra.
- (18) *Refused bequest (legado rechazado)*: Ocurre cuando una subclase hereda métodos o atributos de una clase base que no necesita o no utiliza. Esto puede suceder porque la subclase no está completamente alineada con el propósito de la clase base, lo que conduce a una relación de herencia inadecuada. En estos casos, la subclase "rechaza" parte del comportamiento heredado, lo que indica que quizás la herencia no es la solución adecuada. Cuando esto sucede, puede ser mejor **refactorizar el código** utilizando **composición en lugar de herencia**, o crear una jerarquía de clases más apropiada.
- (19) *Long parameter list (lista de parámetros larga)*: Ocurre cuando un método o constructor recibe demasiados parámetros. Esto no solo aumenta la complejidad de la comprensión del código, sino que también incrementa el acoplamiento entre clases, ya que los métodos que dependen de muchos parámetros son difíciles de mantener y cambiar. Idealmente, en programación orientada a objetos, los métodos deben recibir solo los parámetros necesarios, y se deben utilizar objetos para encapsular conjuntos de datos relacionados.
- (20) *Long method (método largo)*: Ocurre cuando un método tiene demasiadas líneas de código, lo que lo hace difícil de entender, mantener y reutilizar. Este problema suele ser un legado de la programación estructurada, donde era común tener métodos o funciones extensos que realizaban múltiples tareas. Sin embargo, en la **programación orientada a objetos**, uno de los principios clave es tener **métodos más cortos y específicos**, ya que estos son más fáciles de leer, reutilizar y probar.
- (21) *Primitive obsession (obsesión primitiva)*: Ocurre cuando se usan excesivamente tipos primitivos (enteros, cadenas de texto, booleanos, etc.) en lugar de crear objetos que representen conceptos más complejos. Este bad smell se manifiesta cuando se usan tipos primitivos para representar entidades que deberían tener un comportamiento específico o cuando se agrupan varios tipos primitivos relacionados que podrían encapsularse en objetos. El problema con "**Primitive Obsession**" es que el uso de tipos primitivos tiende a dispersar la lógica y la validación en el código, en lugar de centralizarla dentro de un objeto especializado. Esto dificulta la legibilidad, reutilización y mantenimiento del código.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

(22) *Alternative Classes with Different Interfaces* (*clases alternativas con diferentes interfaces*): Ocurre cuando dos o más clases realizan esencialmente la misma función, pero exponen métodos con diferentes nombres o firmas (parámetros). Esto genera inconsistencia en el código, lo que puede dificultar su uso y mantenimiento, ya que los desarrolladores deben recordar diferentes formas de llamar métodos que hacen lo mismo.

Este bad smell también puede ocurrir cuando diferentes clases están diseñadas para el mismo propósito, pero tienen interfaces incompatibles, lo que complica la integración de estas clases.

(23) *Incomplete Library Class* (*biblioteca de clases incompleta*): Se presenta cuando usamos una biblioteca de clases de terceros o del propio sistema que no ofrece toda la funcionalidad que necesitamos, pero no podemos modificar directamente el código de la biblioteca. Este problema ocurre con frecuencia cuando una clase externa tiene algunas limitaciones o carece de ciertos métodos o funcionalidades que son esenciales para nuestro requerimiento.

Dado que las bibliotecas externas suelen ser **código cerrado** (no se pueden modificar) o **no es recomendable alterarlas** (porque podríamos perder compatibilidad o soporte con futuras versiones), enfrentamos el desafío de **extender** o **adaptar** estas clases sin modificar directamente la biblioteca.

### 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*.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
APUNTE DE REFACTORIZACIÓN		

(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):

<b>Add Parameter</b> : Se aplica cuando un método no tiene suficientes datos para realizar determinadas acciones.	<b>Pull Up Constructor Body</b> : Se aplica cuando las subclases tienen constructores con código que en su mayoría es idéntico.
<b>Change Bidirectional Association to Unidirectional</b> : Se aplica cuando se tiene una asociación bidireccional entre clases, pero una de las clases no usa las características de la otra.	<b>Pull Up Field</b> : Se aplica cuando dos clases tienen los mismos atributos.
<b>Change Reference to Value</b> : Se aplica cuando se tiene un objeto de referencia que es demasiado pequeño y que se cambia con poca frecuencia para justificar la gestión de su ciclo de vida.	<b>Pull Up Method</b> : Se aplica cuando las subclases tienen métodos que realizan un trabajo similar.
<b>Change Unidirectional Association to Bidirectional</b> : Se aplica cuando se tiene dos clases y cada una necesita usar las características de la otra, pero la asociación entre ellas es solo unidireccional.	<b>Push Down Field</b> : Se aplica cuando ciertos atributos de la superclase se usan solo en pocas subclases.
<b>Change Value to Reference</b> : Se aplica cuando se tienen muchas instancias idénticas de una sola clase que necesita reemplazarse con un solo objeto.	<b>Push Down Method</b> : Se aplica cuando la lógica del método es aplicable solo a ciertas subclases.
<b>Collapse Hierarchy</b> : Se aplica cuando se tiene una jerarquía de clases en la que una subclase es prácticamente igual que su superclase.	<b>Remove Assignments to Parameters</b> : Se aplica cuando se asigna algún valor a un parámetro dentro del cuerpo del método.
<b>Consolidate Conditional Expression</b> : Se aplica cuando se tienen múltiples condiciones que conducen al mismo resultado o acción.	<b>Remove Control Flag</b> : Se aplica cuando se tiene una variable booleana que actúa como un indicador de control para múltiples expresiones booleanas, en lugar de utilizar la sentencia break, continue o return.
<b>Consolidate Duplicate Conditional Fragments</b> : Se aplica cuando se encuentra	<b>Remove Middleman</b> : Se aplica cuando una clase tiene demasiados métodos que simplemente delegan a otros objetos.

 UNIVERSIDAD <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
<b>APUNTE DE REFACTORIZACIÓN</b>		

<p>un código idéntico en todas las ramas de bloque de condiciones.</p> <p><b>Decompose Conditional:</b> Se aplica cuando se tiene una condición compleja de if/else o switch.</p> <p><b>Duplicate Observed Data:</b> Se aplica cuando los datos de dominio se almacenan en clases responsables de la IGU.</p> <p><b>Encapsulate Collection:</b> Se aplica cuando una clase contiene un atributo de colección y los métodos get y set permiten acceder a la colección completa, en lugar de contar con los métodos para agregar y eliminar elementos de la misma.</p> <p><b>Encapsulate Downcast:</b> Se aplica cuando un método devuelve un objeto que necesita ser casteado por quienes lo invocan, en lugar de castear dentro de dicho método.</p> <p><b>Encapsulate Field:</b> Se aplica cuando los atributos de una clase son públicos.</p> <p><b>Extract Class:</b> Se aplica cuando una clase hace el trabajo de dos.</p> <p><b>Extract Interface:</b> Se aplica cuando varios clientes utilizan el mismo subconjunto de la interfaz de una clase o dos clases tienen parte de sus interfaces en común.</p> <p><b>Extract Method:</b> Se aplica cuando se tiene un fragmento de código que se puede agrupar.</p> <p><b>Extract Subclass:</b> Se aplica cuando una clase tiene características que se usan solo en ciertos casos.</p> <p><b>Extract Superclass:</b> Se aplica cuando se tienen dos clases con campos y métodos comunes.</p>	<p><b>Remove Parameter:</b> Se aplica cuando un parámetro no se utiliza en el cuerpo del método.</p> <p><b>Remove Setting Method:</b> Se aplica cuando el valor de un atributo se establece solo cuando se crea y no cambia nunca por lo que no tiene sentido mantener el método set.</p> <p><b>Rename Method:</b> Se aplica cuando el nombre del método no explica para qué sirve.</p> <p><b>Rename Field:</b> Se aplica cuando el nombre del atributo no explica para qué propósito tiene.</p> <p><b>Replace Array with Object:</b> Se aplica cuando un vector contiene distintos datos que representan al objeto.</p> <p><b>Replace Conditional with Polymorphism:</b> Se aplica cuando una estructura condicional realiza varias acciones según el tipo de objeto o propiedades.</p> <p><b>Replace Constructor with Factory Method:</b> Se aplica cuando se tiene un constructor complejo que hace algo más que establecer valores de parámetros en atributos del objeto.</p> <p><b>Replace Data Value with Object:</b> Se aplica cuando una clase (o grupo de clases) contiene un atributo que tiene su propio comportamiento y datos asociados.</p> <p><b>Replace Delegation with Inheritance:</b> Se aplica cuando una clase contiene muchos métodos simples, que delegan todo a los métodos de otra clase.</p> <p><b>Replace Error Code with Exception:</b> Se aplica cuando un método devuelve un valor especial que indica un error.</p>
--	--

 UNIVERSIDAD <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
APUNTE DE REFACTORIZACIÓN		

<p><b>Form Template Method:</b> Se aplica cuando subclases implementan algoritmos que contienen pasos similares en el mismo orden.</p>	<p><b>Replace Exception with Test:</b> Se aplica cuando se lanza una excepción marcada en una condición que podría haberse verificado previamente por quien invoca.</p>
<p><b>Hide Delegate:</b> Se aplica cuando el cliente obtiene el objeto B de un campo o método del objeto A. Luego, el cliente también llama a un método del objeto B.</p>	<p><b>Replace Inheritance with Delegation:</b> Se aplica cuando se tiene una subclase que usa solo una parte de los métodos de su superclase (o no es posible heredar datos de la superclase).</p>
<p><b>Hide Method:</b> Se aplica cuando un método no es usado por otras clases o solo se usa dentro de su propia jerarquía de clases.</p>	<p><b>Replace Magic Number with Symbolic Constant:</b> Se aplica cuando el código usa un número que tiene cierto significado.</p>
<p><b>Inline Class:</b> Se aplica cuando una clase no hace casi nada y no es responsable de nada, y no se planean responsabilidades adicionales para ella.</p>	<p><b>Replace Method with Method Object:</b> Se aplica cuando se tiene un método largo en el que las variables locales están tan entrelazadas que no puede aplicarse la refactorización "Extract method".</p>
<p><b>Inline Method:</b> Se aplica cuando la lógica de un método es tan obvia que no tiene sentido mantener el método.</p>	<p><b>Replace Nested Conditional with Guard Clauses:</b> Se aplica cuando se tiene bloque de condiciones anidadas y es difícil determinar el flujo normal de ejecución del código.</p>
<p><b>Inline Temp:</b> Se aplica cuando se tiene una variable temporal a la que se le asigna el resultado de una expresión simple y nada más.</p>	<p><b>Replace Parameter with Explicit Methods:</b> Se aplica cuando un método se divide en partes, cada una de las cuales se ejecuta según el valor de un parámetro.</p>
<p><b>Introduce Assertion:</b> Se aplica para que una parte del código funcione correctamente asumiendo que ciertas condiciones o valores deben ser verdaderos.</p>	<p><b>Replace Parameter with Method:</b> Se aplica cuando un objeto invoca un método y luego pasa el resultado como parámetro a otro método cuando el receptor también podría invocar este método.</p>
<p><b>Introduce Explaining Variable:</b> Se aplica cuando se tiene una expresión complicada.</p>	<p><b>Replace Record with Data Class:</b> Se aplica cuando se necesita interactuar con una estructura de registro en un entorno de programación tradicional.</p>
<p><b>Introduce Foreign Method:</b> Se aplica cuando una clase de una biblioteca de clases no contiene el método que se necesita y no se tiene la posibilidad de agregarlo. En este caso, habrá que hacer un método que reciba como parámetro un objeto del tipo correspondiente.</p>	<p><b>Replace Subclass with Fields:</b> Se aplica cuando se tiene subclases que difieren solo en sus métodos (de retorno constante).</p>
<p><b>Introduce Local Extension:</b> Se aplica cuando una clase de una biblioteca de clases</p>	<p><b>Replace Temp with Query:</b> Se aplica cuando se coloca el resultado de una</p>

 UNIVERSIDAD <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
<b>APUNTE DE REFACTORIZACIÓN</b>		

<p>no contiene alguno de los métodos que se necesitan y no puede agregárselos. En este caso, se deberá heredar de la misma para poder definirlos.</p>	<p>expresión en una variable local para su uso posterior en el código.</p>
<p><b>Introduce Null Object:</b> Se aplica cuando algunos métodos retornan null en lugar de objetos reales y se requiere controlar dichos nulos dentro del código.</p>	<p><b>Replace Type Code with Class:</b> Se aplica cuando una clase tiene muchos atributos numéricos para representar distintos tipos y estos no afectan su comportamiento.</p>
<p><b>Introduce Parameter Object:</b> Se aplica cuando los métodos contienen un grupo repetido de parámetros.</p>	<p><b>Replace Type Code with State/Strategy:</b> Se aplica cuando se tiene un atributo codificado que afecta el comportamiento, pero no se puede usar subclases para deshacerse de él.</p>
<p><b>Move Field:</b> Se aplica cuando un atributo se usa más en otra clase que en su propia clase.</p>	<p><b>Replace Type Code with Subclasses:</b> Se aplica cuando se tiene un atributo codificado que afecta directamente el comportamiento del programa (los valores de este atributo activan distintos códigos en los condicionales).</p>
<p><b>Move Method:</b> Se aplica cuando un método se usa más en otra clase que en su propia clase.</p>	<p><b>Self Encapsulate Field:</b> Se aplica cuando se utiliza el acceso directo a los atributos privados dentro de una clase.</p>
<p><b>Parameterize Method:</b> Se aplica cuando múltiples métodos realizan acciones similares que son diferentes solo en sus valores internos, números u operaciones.</p>	<p><b>Split Temporary Variable:</b> Se aplica cuando se tiene una variable local que se usa para almacenar varios valores intermedios dentro de un método.</p>
<p><b>Preserve Whole Object:</b> Se aplica cuando se obtienen varios valores de un objeto y luego se pasan como parámetros a un método, en lugar de enviar el objeto completo.</p>	<p><b>Substitute Algorithm:</b> Se aplica cuando se quiere reemplazar un algoritmo por otro más claro.</p>

### 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.
Cambio en cadena	Move method. Move field. Inline class.
Clase de datos	Move method. Encapsulate field. Encapsulate collection.

 UNIVERSIDAD <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
<b>APUNTE DE REFACTORIZACIÓN</b>		

Código muerto	Inline class. Collapse hierarchy. Remove parameter.
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 middleman. 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.
Clases alternativas con diferentes interfaces	Rename methods. Move method. Add parameter. Parameterize method. Extract superclass.
Biblioteca de clases incompleta	Introduce foreign method. Introduce local extension.

*Especificación de algunas refactorizaciones*

### **Refactorización que extrae método**

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

<i>Extract Method: Extraer método</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 propósito.	
Mecanismo	(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> <p style="text-align: center;"></p> <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> <p style="text-align: center;"></p> <pre>public void printInvoice(Order order) {</pre>	

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

```

        printInvoiceHeader();

        printCustomerDetails(order.getCustomer());

        //...
    }

    public void printInvoiceHeader() {
        System.out.println ("Factura");
        System.out.println ("=====");
    }

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

```

## Método de refactorización en línea

<i>Inline Method: Método en línea</i>	
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	<pre> public void calculateAreaOfCircle(double radius) {     double area = getValueOfPI() * Math.pow(radius, 2);     System.out.println ("Area: " + area); }  private double getValueOfPI () {     return Math.PI; } </pre>  <pre> public void calculateAreaOfCircle(double radius) {     double area = Math.PI * Math.pow(radius, 2);     System.out.println ("Area: " + area); } </pre>

## Refactorizaciones que eliminan temporales

*Replace Temp with Query: Reemplazar temporales con consultas*

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

<b>Resumen</b>	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.
<b>Motivación</b>	Se está utilizando una variable temporal para manejar el resultado de una expresión.
<b>Mecanismo</b>	(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.
<b>Ejemplo</b>	<pre>public boolean checkWithdrawalStatus(double amoutToWithdraw) {     //Crear temporal     double availableBalance = _balance - _pendingCharges;      //Usa temporal     if(availableBalance &gt; amoutToWithdraw) {         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() &gt; 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*

<b>Resumen</b>	Quitar variable temporal utilizar una única vez.
----------------	--

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

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

### Refactorizaciones que añaden temporales

<i>Split Temporary Variable: División de variables temporales</i>	
Resumen	Hacer una variable temporal por cada asignación.
Motivación	Variable temporal asignada más de una vez.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

Mecanismo	<ul style="list-style-type: none"> <li>(1) Cambiar el nombre de la variable temporal y su primera asignación.</li> <li>(2) Declarar la nueva variable temporal como final.</li> <li>(3) Cambiar todas las referencias de la variable temporal antes de su segunda asignación.</li> <li>(4) Declarar la variable temporal en la segunda asignación.</li> <li>(5) Compilar y probar.</li> <li>(6) Repetir en etapas, en cada etapa renombrar la variable y cambiar las referencias hasta la próxima asignación.</li> </ul>
Ejemplo	<pre>double temp = item.getPrice() * item.getQuantity(); System.out.println("Total por línea: " + temp);  temp = order.getTotal() - order.getDiscount(); System.out.println("Cantidad debida: " + temp);</pre> <p style="text-align: center;"></p> <pre>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);</pre>

<i>Introduce Explaining Variable: Introducir variables explicativas</i>	
Resumen	Hacer el código más entendible.
Motivación	Expresiones complicadas.
Mecanismo	<ul style="list-style-type: none"> <li>(1) Declarar la variable temporal como final, y asignarle el resultado asociado a la parte de la expresión compleja.</li> <li>(2) Reemplazar la parte de la expresión con la variable temporal.</li> <li>(3) Compilar y probar.</li> <li>(4) Repetir pasos 1 al 3 para el resto de la expresión.</li> </ul>
Ejemplo	<pre>if(stock.checkStatus(order.getItem()) &gt; order.getQuantity() &amp;&amp;    order.getTotal() &gt; 99 &amp;&amp;    order.getCustomer().getBillingAddress().getLine1(). equals(order.getShippingAddress().getLine1())) {     System.out.println("Aceptado");     ... } else {     ... }</pre> <p style="text-align: center;"></p> <pre>final boolean stockAvailable = stock.checkStatus(order.getItem()) &gt; order.getQuantity();</pre>

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

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

<i>Remove Assignments to Parameters: Eliminar asignaciones a parámetros</i>	
Resumen	El parámetro es reasignado dentro del método.
Motivación	Evitar la modificación del parámetro recibido.
Mecanismo	<ul style="list-style-type: none"> <li>(1) Crear una variable temporal para el parámetro.</li> <li>(2) Reemplazar todas las referencias hechas al parámetro después de la asignación, a la variable temporal.</li> <li>(3) Cambiar la asignación del parámetro a la asignación de la variable temporal.</li> <li>(4) Compilar y probar.</li> </ul>
Ejemplo	<pre> public void checkStock(OrderItem item, int desired) {     //¿Se puede cumplir con el pedido?     if(item.getStockUnits() &lt; 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");          // ...     }     // ... }  ↓  public void checkStock(OrderItem item, int desired) {     //¿Se puede cumplir con el pedido?     if(item.getStockUnits() &lt; desired) {         //Se usa una temporal en línea en lugar de asignar al parámetro </pre>

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

```

        System.out.println("Solo " + item.getStockUnits() + " unidades disponibles");

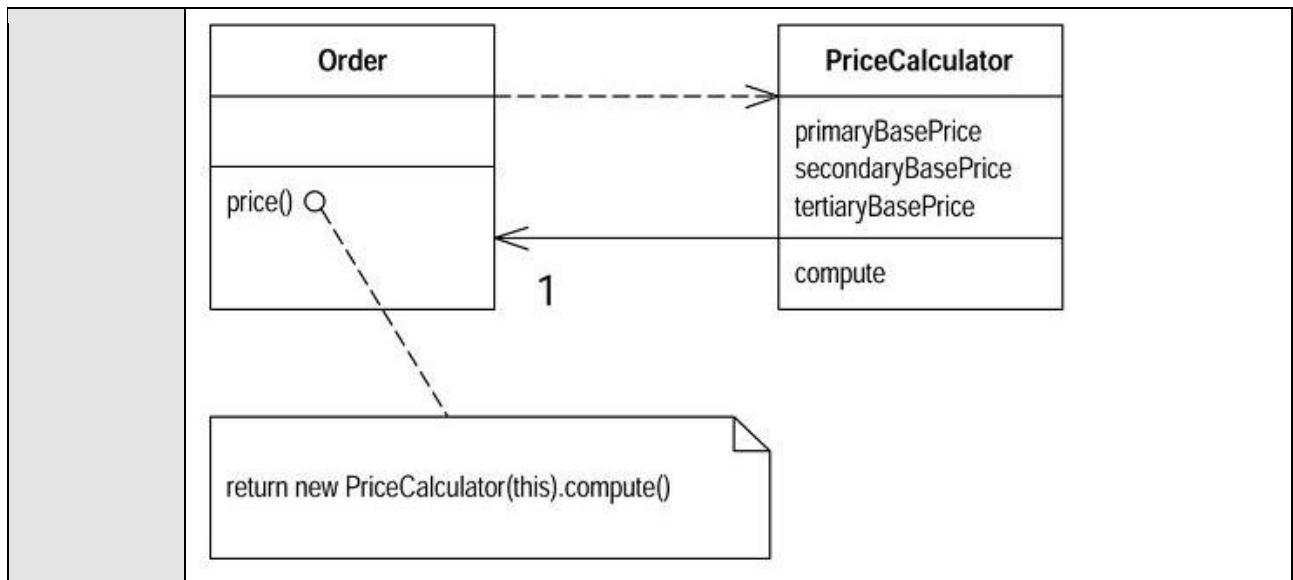
        //Se usa una nueva temporal en lugar de asignar al parámetro
        OrderItem alternate = new OrderItem(item.getAlternateID());
        System.out.println("Sugiero " + alternate.getName() + " en su lugar");

        // ...
    }
    // ...
}

```

### Refactorización que reemplaza método con método de un objeto

<i>Replace Method with Method Object: Reemplazar método con método de un objeto</i>	
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	<ul style="list-style-type: none"> <li>(1) Crear una nueva clase cuyo nombre es el método.</li> <li>(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.</li> <li>(3) Crear un constructor que recibe el objeto fuente y cada parámetro.</li> <li>(4) Crear un nuevo método "compute" en la nueva clase.</li> <li>(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.</li> <li>(6) Compilar.</li> <li>(7) Reemplazar el método viejo por uno que crea el objeto nuevo e invoque el método "compute".</li> </ul>
Ejemplo	<pre> class Order {      public double price() {         double primaryBasePrice;         double secondaryBasePrice;         double tertiaryBasePrice;          // long computation;      } } </pre> 



## Refactorización que sustituye algoritmo

Substitute Algorithm: Sustituir 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 foundPerson(String[] people){     for (int i = 0; i &lt; 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 foundPerson(String[] people){     List candidates = Arrays.asList(new String[] {"Don", "John", "Kent"});     for (int i=0; i&lt;people.length; i++) </pre>

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

	<pre>         if (candidates.contains(people[i]))             return people[i];         return "";     } } </pre>
--	---

## Refactorización que mueve métodos

<i>Move Method: Mover método</i>	
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	<ul style="list-style-type: none"> <li>(1) Examinar todas las características utilizadas por el método fuente que se definen en la clase origen.</li> <li>(2) Comprobar las subclases y superclases.</li> <li>(3) Declarar el método en la clase destino.</li> <li>(4) Copiar el código del método a la clase destino. Ajustar el método para que funcione en la nueva clase.</li> <li>(5) Compilar la clase destino.</li> <li>(6) Reemplazar las llamadas originales.</li> <li>(7) Compilar y probar.</li> </ul>
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 &lt; items.length; i++) {             System.out.println("* Name: " + items[i].getName());             System.out.println("* ID: " + items[i].getId());         }     } } </pre>

```
        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;

    ...

    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*

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

<b>Resumen</b>	Crear una nueva clase y mover los campos y métodos pertenecientes a la vieja clase en la nueva clase.
<b>Motivación</b>	La clase realiza más cosas que lo que originalmente iba hacer.
<b>Mecanismo</b>	<ul style="list-style-type: none"> <li>(1) Decidir cómo dividir las responsabilidades de la clase.</li> <li>(2) Crear una nueva clase para expresar las responsabilidades a extraer.</li> <li>(3) Hacer un vínculo desde la antigua a la nueva clase.</li> <li>(4) Utilizar “Mover campo” en cada campo que se desea mover.</li> <li>(5) Compilar y probar después de cada movimiento.</li> <li>(6) Utilizar “Mover método” para mover métodos de la clase viaja a la nueva.</li> <li>(7) Compilar y probar después de cada movimiento.</li> <li>(8) Revisar y reducir las interfaces de cada clase.</li> <li>(9) Decidir si se va a exponer la nueva clase.</li> </ul>
<b>Ejemplo</b>	<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() {         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;     } } </pre> 

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

```

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;
    }

    public String getTelephoneNumber(){
        return this.officeTelephone.getTelephoneNumber();
    }

    public TelephoneNumber getOfficeTelephone() {
        return this.officeTelephone;
    }

    public void setOfficeTelephone(TelephoneNumber arg) {
        this.officeTelephone = arg;
    }

}

```

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

<b>Nota</b>	<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

<i>Decompose Conditional: Descomposición condicional</i>	
<b>Resumen</b>	Extraer la expresión condicional y crear un método que luego se utilice en la sentencia condicional.
<b>Motivación</b>	Se tiene una expresión condicional complicada.
<b>Mecanismo</b>	<p>(1) Extraer la condición creando un nuevo método que retorne el resultado de la misma.</p> <p>(2) Modificar la sentencia condicional utilizando el método creado.</p> <p>(3) Compilar y probar.</p>
<b>Ejemplo</b>	<pre>if(order.getCustomer().getStatus().equals("P")         (order.getTotal() &gt; 1000 &amp;&amp; order.getWeight() &lt; 500)) {     shipping = order.getTotal() * 0.05;     if(shipping &gt; 100) shipping = 100; } else {     shipping = order.getTotal() * 0.08;     if(shipping &gt; 200) shipping = 200; }</pre> <div style="text-align: center; margin-top: 10px;">  </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() &gt; 1000 &amp;&amp; order.getWeight() &lt; 500); }  public double discountShippingRate(Order order) {     double result = order.getTotal() * 0.05;     if(result &gt; 100) result = 100;     return result; }  public double normalShippingRate(Order order) {</pre>

 <b>UNIVERSIDAD Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

	<pre>double result = order.getTotal() * 0.08; if(result &gt; 200) result = 200; return result;</pre>
--	--

<i>Consolidate Conditional Expression: Consolidar la expresión condicional</i>	
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	<p>(1) Comprobar que ninguna de las expresiones condiciones tiene un efecto secundario.</p> <p>(2) Armar una única expresión condicional utilizando operadores lógicos.</p> <p>(3) Compilar y probar.</p> <p>(4) Considerar el uso de "Extraer método" para la condición construida.</p>
Ejemplo	<pre>public double calculateShipping() {     if(_isEmployee) return 0;     if(_isPlatinumCustomer) return 0;     if(_isGoldCustomer) return 0;     if(_hasCoupon) return 0;      //Calcular el envío     //... }</pre> <p style="text-align: center; margin-top: 10px;">  </p> <pre>public double calculateShipping() {     if(freeShipping()) return 0;      //Calcular el envío     //... }  public boolean freeShipping() {     return (_isEmployee    _isPlatinumCustomer    _isGoldCustomer        _hasCoupon); }</pre>

<i>Consolidate Duplicate Conditional Fragments: Consolidación de fragmentos de código condicional duplicados</i>	
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.

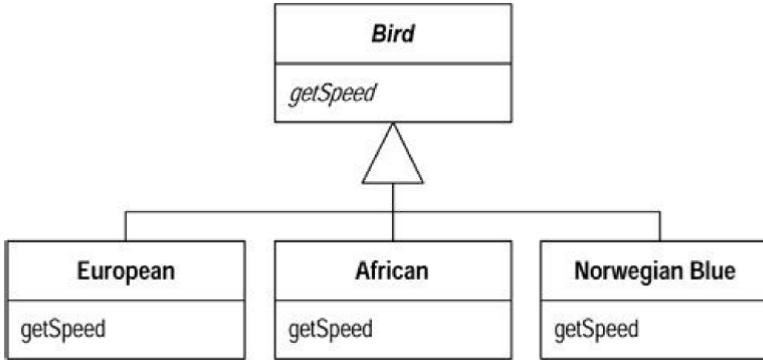
 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

Mecanismo	<ul style="list-style-type: none"> <li>(1) Identificar el código que se ejecuta de la misma manera independientemente de la condición.</li> <li>(2) Si el código común es al principio, moverlo antes del condicional.</li> <li>(3) Si el código común es al final, moverlo después del condicional.</li> <li>(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.</li> <li>(5) Si hay más de una declaración, se debe extraer ese código en un método.</li> </ul>
Ejemplo	<pre>if(largeOrImportant(order)) {     shipping = discountShippingRate(order);     order.addShippingCost(shipping); } else {     shipping = normalShippingRate(order);     order.addShippingCost(shipping); }</pre> <p style="text-align: center;"></p> <pre>if(largeOrImportant(order)) {     shipping = discountShippingRate(order); } else {     shipping = normalShippingRate(order); } order.addShippingCost(shipping);</pre>

### Refactorización que reemplaza condicional con polimorfismo

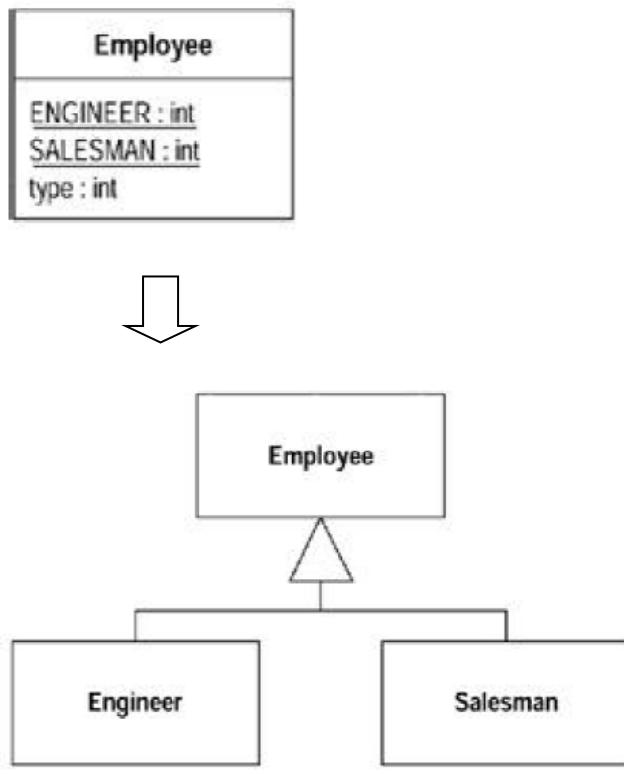
<i>Replace Conditional with Polymorphism: Reemplazar condicional con polimorfismo</i>	
Resumen	El método que contiene la sentencia condicional se sobrescribe en las subclases y se convierte el método de la superclase en abstracto.
Motivación	Se cuenta con una sentencia condicional que elige un comportamiento diferente en función de un tipo de objeto.
Mecanismo	<ul style="list-style-type: none"> <li>(1) Si la sentencia condicional es una parte de un método más grande, desarmar la sentencia condicional y utilizar "Extraer método".</li> <li>(2) Si es necesario, utilizar "Mover método" para colocar el condicional en la superclase.</li> <li>(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.</li> <li>(4) Compilar y probar.</li> <li>(5) Eliminar esa parte de la sentencia condicional.</li> <li>(6) Compilar y probar.</li> </ul>

 <b>UNIVERSIDAD Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

	<p>(7) Repetir el procedimiento con cada parte de la sentencia condicional hasta que todas las condiciones evaluadas se conviertan en métodos de subclases.</p> <p>(8) Hacer que el método se convierta en abstracto en la superclase.</p>
<b>Ejemplo</b>	<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

<i>Replace Type Code with Subclasses: Reemplazar código de tipo con subclases</i>	
<b>Resumen</b>	Se reemplaza la tipificación de código con subclases.
<b>Motivación</b>	Según el tipo del objeto, se determina el comportamiento de la clase.
<b>Mecanismo</b>	<ul style="list-style-type: none"> <li>(1) Encapsular la tipificación del código.</li> <li>(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.</li> <li>(3) Quitar de la superclase el campo que representa cada tipificación. Declarar el método de obtención del objeto como abstracto.</li> <li>(4) Compilar y probar.</li> </ul>

**Ejemplo****Refactorizaciones de encapsulamiento**

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

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

	<pre>}</pre> <p style="text-align: center;"></p> <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 &gt;= this.getLow() &amp;&amp; arg &lt;= this.getHigh();     }      //... }</pre>
--	--

<i>Encapsulate Field:</i> Encapsulamiento de campos	
Resumen	Hacer privado el acceso al campo y programar los métodos accesores.
Motivación	Hay campos con acceso público.
Mecanismo	<ul style="list-style-type: none"> <li>(1) Crear los métodos accesores del campo.</li> <li>(2) Encontrar las llamadas fuera de la clase que hacen referencia al campo. Reemplazar las mismas con el método accesror que corresponda.</li> <li>(3) Compilar y probar después de cada cambio.</li> <li>(4) Una vez que todas las referencias se cambiaron, declarar el campo como privado.</li> <li>(5) Compilar y probar.</li> </ul>
Ejemplo	<pre>class Person {      public String _name;      //... }</pre> <p style="text-align: center;"></p> <pre>class Person {      private String _name;      public String getName() {         return this._name;     } }</pre>

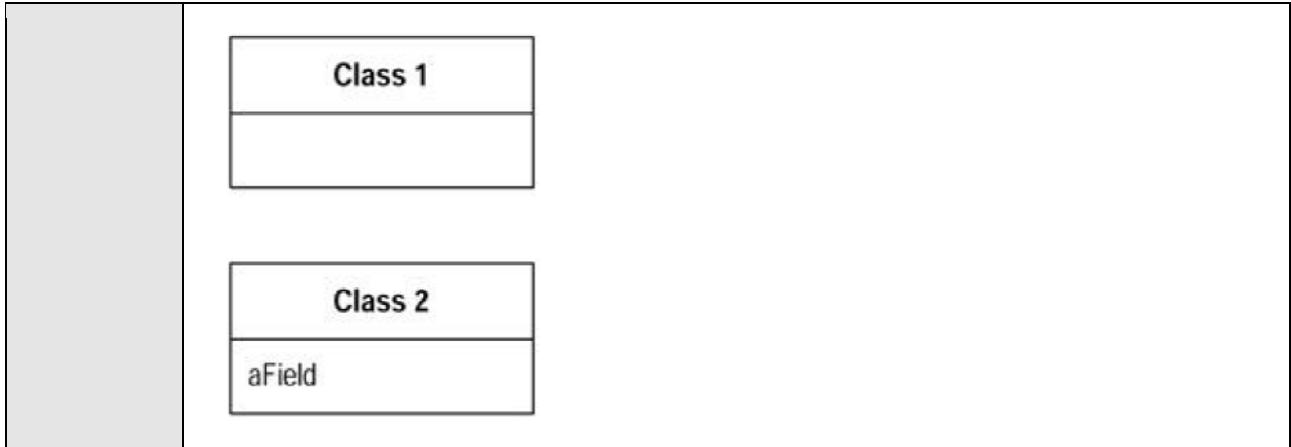
 UNIVERSIDAD <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
<b>APUNTE DE REFACTORIZACIÓN</b>		

	<pre>         }          public void setName(String arg) {             this._name = arg;         }          //...     }   </pre>
--	--

### Refactorización que mueve campos

<i>Move Field: Mover campos</i>	
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	<p>(1) Si el campo es público, utilizar "Encapsulamiento de campo".</p> <p>(2) Compilar y probar.</p> <p>(3) Crear un campo en la clase destino con los métodos accesores correspondientes.</p> <p>(4) Compilar la clase destino.</p> <p>(5) Determinar cómo hacer referencia a la clase destino desde el origen.</p> <p>(6) Remover el campo de la clase origen.</p> <p>(7) Reemplazar la referencia en la clase origen invocando el método accesror correspondiente (Campo autoencapsulado)</p> <p>(8) Compilar y probar.</p>
Ejemplo	<div style="border: 1px solid black; padding: 10px; margin-bottom: 20px;"> <b>Class 1</b> <hr/>         aField       </div> <div style="border: 1px solid black; padding: 10px; margin-bottom: 20px;"> <b>Class 2</b> <hr/> </div> 

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		



## Trabajando con cúmulos de datos

<i>Preserve Whole Object: Preservación del objeto entero</i>	
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	<ul style="list-style-type: none"> <li>(1) Crear un nuevo parámetro que permita enviar el objeto entero.</li> <li>(2) Compilar y probar.</li> <li>(3) Determinar qué parámetros pueden ser reemplazados por el objeto entero.</li> <li>(4) Tomar un parámetro y reemplazar las referencias al mismo dentro del cuerpo del método invocado.</li> <li>(5) Eliminar el parámetro.</li> <li>(6) Compilar y probar.</li> <li>(7) Repetir el procedimiento por cada parámetro que se pueda reemplazar con el objeto entero.</li> <li>(8) Quitar el código en el método de llamada que obtiene los parámetros borrados.</li> <li>(9) Compilar y probar.</li> </ul>
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) {     // ... }  double shippingCost = calculateShipping(order); </pre>

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

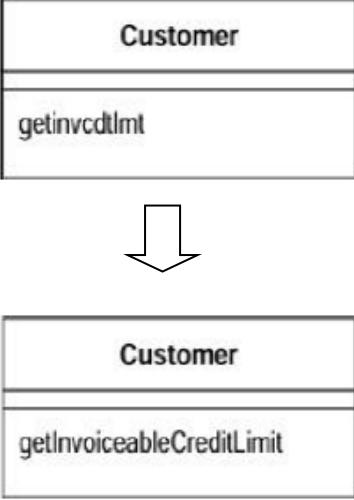
	<pre>public double calculateShipping(Order order) {     // ... }</pre>
--	--

<i>Introduce Parameter Object: Introducir objeto como parámetro</i>	
Resumen	Reemplazar parámetro con un objeto.
Motivación	Se tiene un grupo de parámetros que están fuertemente relacionados.
Mecanismo	<p>(1) Crear una nueva clase que represente el grupo de parámetros que está relacionado.</p> <p>(2) Compilar.</p> <p>(3) Utilizar “Agregar parámetro” para el nuevo grupo de datos. Usar un valor nulo para este parámetro en todas las llamadas al método.</p> <p>(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.</p> <p>(5) Recopilar y probar después de quitar cada parámetro.</p> <p>(6) Una vez que se hayan eliminado los parámetros, buscar qué comportamiento se puede mover dentro del parámetro objeto. Utilizar “Mover método”.</p>
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> <div style="text-align: center; margin-top: 10px;">  </div> <pre>class Frame {      private int x;     private int y;     private int height;     private int width;      // ...  }  public void drawRectangle(Frame frame) {     // ... }</pre>

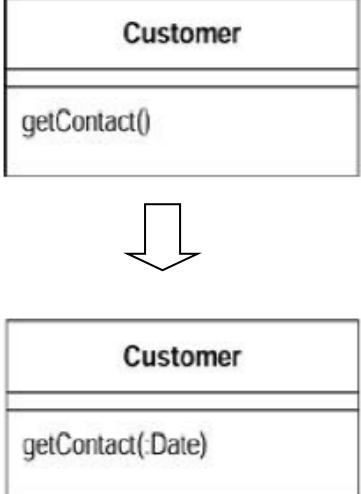
 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

	<pre>public void drawEllipse(Frame frame) {     // ... }  public void drawStar(Frame frame) {     // ... }</pre>
--	--

## Refactorizaciones para simplificar llamado a métodos

<i>Rename Method:</i> 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	<ul style="list-style-type: none"> <li>(1) Revisar la firma del método para determinar si está implementada en una superclase o subclase. Si esto se cumple, seguir los siguientes pasos.</li> <li>(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.</li> <li>(3) Compilar.</li> <li>(4) Cambiar el cuerpo del viejo método de modo que llame al nuevo.</li> <li>(5) Compilar y probar.</li> <li>(6) Encontrar todas las referencias al nombre antiguo del método y cambiarlos por el nuevo. Compilar y probar después de cada cambio.</li> <li>(7) Retirar el antiguo método.</li> <li>(8) Compilar y probar.</li> </ul>
Ejemplo	 <p>The diagram illustrates the refactoring process for the <code>Customer</code> class. It shows two states of the class:</p> <ul style="list-style-type: none"> <li><b>Before Refactoring:</b> The class has a method named <code>getinvcdtlmt</code>.</li> <li><b>After Refactoring:</b> The class has a method named <code>getInvoiceableCreditLimit</code>.</li> </ul> <p>An arrow points from the old method name to the new one, indicating the transformation.</p>

 UNIVERSIDAD <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
<b>APUNTE DE REFACTORIZACIÓN</b>		

<i>Add Parameter: Añadir parámetro</i>		
Resumen	Añadir un parámetro para que el objeto pueda transmitir más información.	
Motivación	El método requiere más información de quién lo invoca.	
Mecanismo	(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	 <pre> classDiagram     class Customer {         getContact()     }     Customer --&gt; Customer     Customer &lt;&lt;getContact(Date)&gt;&gt;   </pre>	

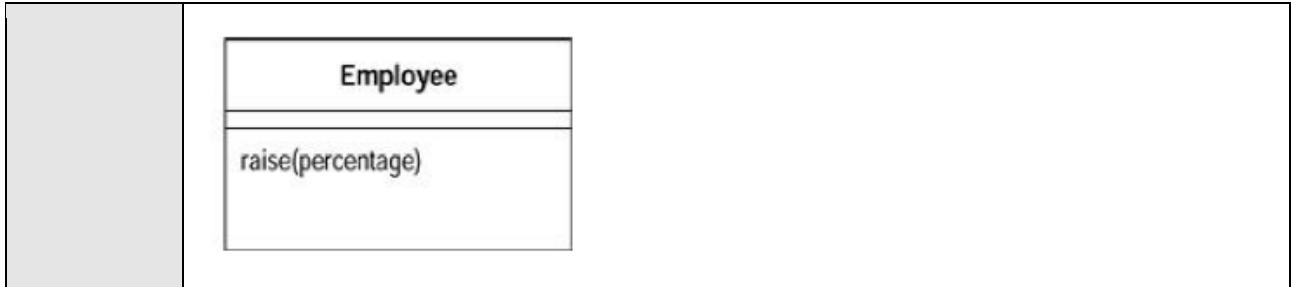
<i>Remove Parameter: Eliminar parámetro</i>		
Resumen	Se elimina parámetro sin uso.	
Motivación	El parámetro no es utilizado en el cuerpo del método.	
Mecanismo	(1) Revisar la firma del método para determinar si está implementada en una superclase o subclase. (2) Revisar si la clase o superclase utiliza 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.	

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

	<ul style="list-style-type: none"> <li>(6) Compilar y probar.</li> <li>(7) Encontrar todas las referencias al método antiguo y cambiarlas para referirse al nuevo. Compilar y probar después de cada cambio.</li> <li>(8) Retirar el antiguo método.</li> <li>(9) Compilar y probar.</li> </ul>
<b>Ejemplo</b>	<pre> graph TD     A[Customer getContact(Date)] --&gt; B[Customer getContact()]   </pre> <p>The diagram illustrates the refactoring of a method. It shows two states of a class named 'Customer'. In the first state, there is a method 'getContact(Date)'. In the second state, this method has been refactored into 'getContact()' (removing the parameter). An arrow points from the original state to the refactored state.</p>

<b>Parameterize Method: Parametrizar método</b>	
<b>Resumen</b>	Se elimina parámetro sin uso.
<b>Motivación</b>	Varios métodos hacen cosas similares pero con valores distintos.
<b>Mecanismo</b>	<ul style="list-style-type: none"> <li>(1) Crear un método con un parámetro que pueda sustituir cada método repetitivo.</li> <li>(2) Compilar.</li> <li>(3) Reemplazar el antiguo método invocando al nuevo.</li> <li>(4) Compilar y probar.</li> <li>(5) Repetir el procedimiento para todos los métodos. Probar después de cada cambio.</li> </ul>
<b>Ejemplo</b>	<pre> graph TD     A[Employee fivePercentRaise() tenPercentRaise()] --&gt; B[Employee calculateRaise()]   </pre> <p>The diagram illustrates the refactoring of multiple methods into one. It shows two states of a class named 'Employee'. In the first state, there are two methods: 'fivePercentRaise()' and 'tenPercentRaise()'. In the second state, these have been refactored into a single method 'calculateRaise()'. An arrow points from the original methods to the refactored method.</p>

 <b>UNIVERSIDAD Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

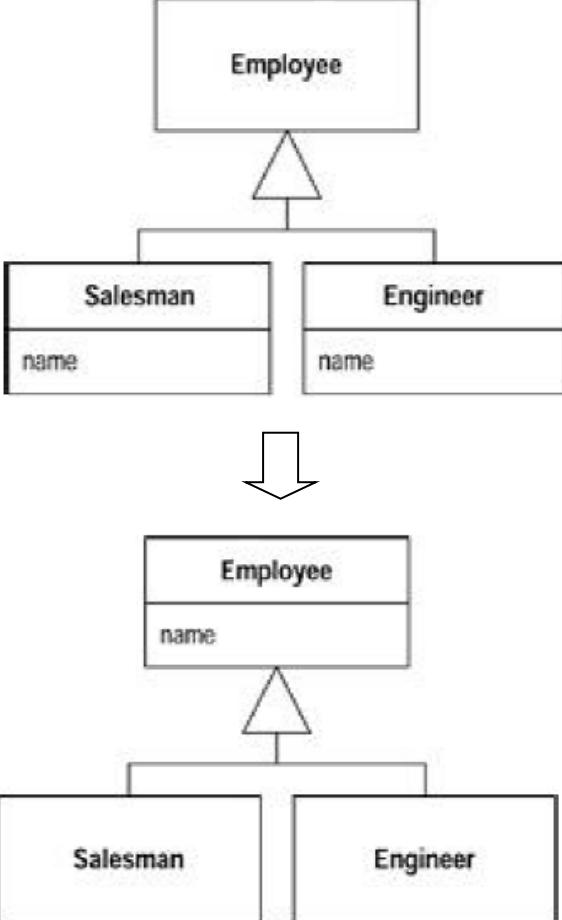


<i>Replace Parameter with Explicit Methods:</i> Reemplazar parámetro con métodos explícitos	
Resumen	Crear un método por cada valor posible del parámetro.
Motivación	Se tiene un método que ejecuta código diferente según el valor de uno de sus parámetros.
Mecanismo	<ul style="list-style-type: none"> 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

## Refactorizaciones que suben y bajan métodos y campos

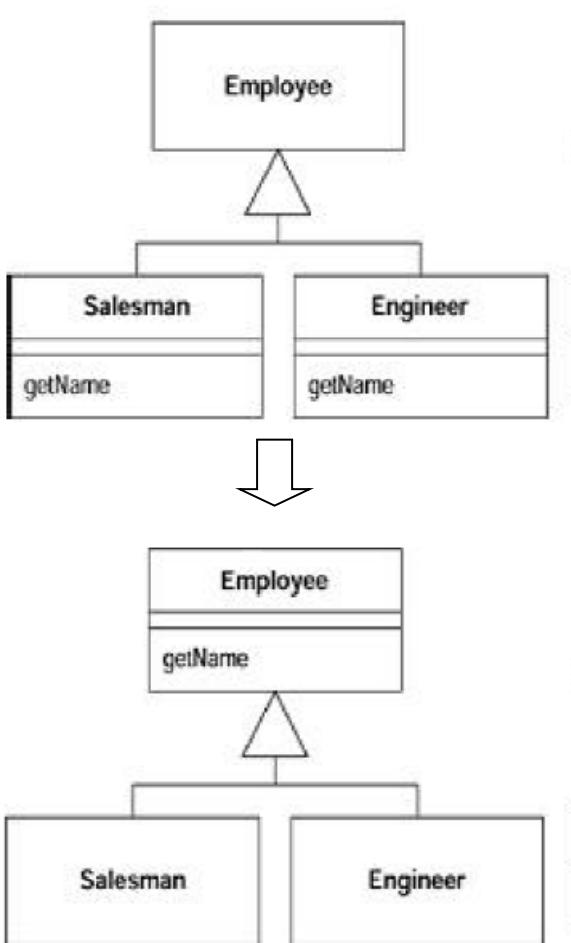
<i>Pull Up Field:</i> Subir campo	
Resumen	Mover el campo a la superclase.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

<b>Motivación</b>	<p>Dos subclases tienen un campo que significa lo mismo.</p>
<b>Mecanismo</b>	<ul style="list-style-type: none"> <li>(1) Revisar todos los usos de los campos candidatos para asegurarse de que se utilizan de la misma manera.</li> <li>(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.</li> <li>(3) Compilar y probar.</li> <li>(4) Crear un nuevo campo en la superclase.</li> <li>(5) Eliminar los campos de las subclases.</li> <li>(6) Compilar y probar.</li> <li>(7) Considerar el uso de la refactorización de "Auto encapsulamiento de campos" para el nuevo campo.</li> </ul>
<b>Ejemplo</b>	 <pre> graph TD     Employee[Employee] --&gt; Salesman[Salesman]     Employee --&gt; Engineer[Engineer]     Salesman --&gt; name1[name]     Engineer --&gt; name2[name]      Employee[Employee] --&gt; Salesman[Salesman]     Employee --&gt; Engineer[Engineer]   </pre>

<i>Pull Up Method: Subir método</i>	
<b>Resumen</b>	Mover los métodos a la superclase.
<b>Motivación</b>	Se tienen métodos con iguales resultados en las subclases.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

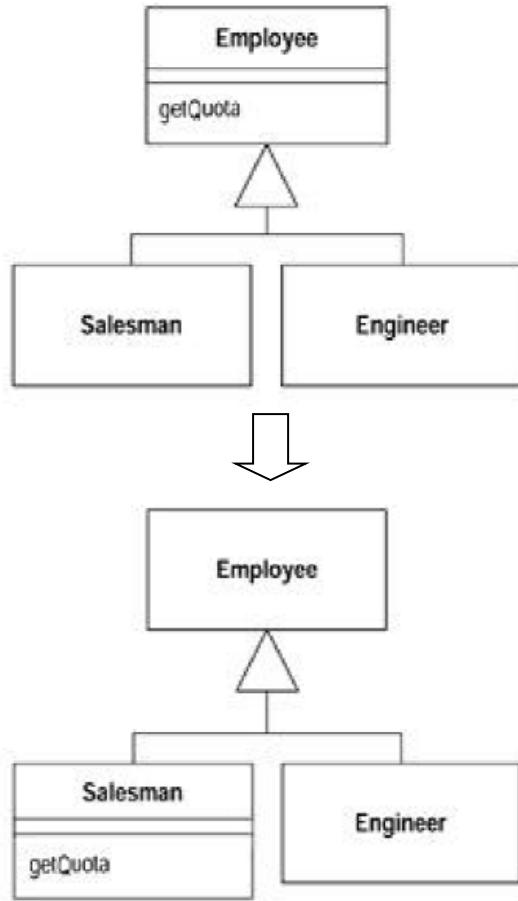
Mecanismo	<ul style="list-style-type: none"> <li>(1) Revisar los métodos para asegurarse de que son idénticos.</li> <li>(2) Si los métodos tienen diferentes firmas, cambiar las firmas a la que se desea utilizar en la superclase.</li> <li>(3) Crear un nuevo método en la superclase, copiar el cuerpo de uno de los métodos, ajustar, y compilar.</li> <li>(4) Encapsular campo y declarar y utilizar un método get abstracto.</li> <li>(5) Eliminar el método de la subclase.</li> <li>(6) Compilar y probar.</li> <li>(7) Borrar métodos de las subclases y probar hasta que sólo quede el método de superclase.</li> <li>(8) Revisar las llamadas al método para determinar si se requiere algún cambio.</li> </ul>
Ejemplo	 <pre> graph TD     Employee[Employee] --&gt; Salesman[Salesman]     Employee --&gt; Engineer[Engineer]     Salesman.getName[getName]     Engineer.getName[getName]      Employee2[Employee] --&gt; Salesman2[Salesman]     Employee2 --&gt; Engineer2[Engineer]     Employee2.getName[getName]   </pre>

<i>Push Down Field:</i> 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.

	<p>(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.</p>
Ejemplo	<pre> graph TD     Employee[Employee] --&gt; Salesman[Salesman]     Employee --&gt; Engineer[Engineer]     Salesman --- quota[quota]          Employee[Employee] --&gt; Salesman[Salesman]     Salesman --- quota[quota]     Salesman --- Engineer[Engineer]   </pre>

*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	<p>(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.</p>

**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.

 <b>UNIVERSIDAD</b> <b>Blas Pascal</b>	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2014</b> <b>DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE</b>	<b>VERSIÓN: 1.5</b> <b>VIGENCIA: 11-10-2011</b>
<b>APUNTE DE REFACTORIZACIÓN</b>		

- *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.

#### 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.
Cambio en cadena	Move method. Move field. Inline class.
Clase de datos	Move method. Encapsulate field. Encapsulate collection.
Código muerto	Inline class. Collapse hierarchy. Remove parameter.
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 middleman. Inline method. Replace delegation with inheritance.
Intimidad inadecuada	Move method. Move field. Change bidirectional association to unidirectional. Replace inheritance with delegation. Hide delegate.

 UNIVERSIDAD <b>Blas Pascal</b>	INGENIERÍA EN INFORMÁTICA – PLAN 2014 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.5 VIGENCIA: 11-10-2011
<b>APUNTE DE REFACTORIZACIÓN</b>		

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.
Clases alternativas con diferentes interfaces	Rename methods. Move method. Add parameter. Parameterize method. Extract superclass.
Biblioteca de clases incompleta	Introduce foreign method. Introduce local extension.

## 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

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