GENERALITAT VALENCIANA
Conselleria d'Educació, Investigació, Cultura i Esport

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

I. E. S. MARE NOSTRUM
Beato Fco. Castelló Aleu s/n • ☎ 965936520
03008 ALICANTE
www.iesmarenostrum.com
correo@iesmarenostrum.com

UNIDAD 8

Refactorización

I. E. S. MARE NOSTRUM
Beato Fco. Castelló Aleu s/n • ☎ 965936520
03008 ALICANTE
www.iesmarenostrum.com
correo@iesmarenostrum.com

Contenido

1 INTRODUCCIÓN

La refactorización es uno de los nuevos conceptos que se han introducido en la terminología del mundo del desarrollo apoyado por las metodologías ágiles. Extreme Programming, una de las metodologías ágiles más extendida, la incluye dentro del decálogo de prácticas que propone como fundamentales.

Refactorizar es realizar modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo. De esta definición se obtienen conceptos que estaban equivocados. Refactorizar no es una técnica para encontrar y corregir errores en una aplicación, puesto que su objetivo no es alterar su comportamiento externo.

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 siendo completamente controlado por nosotros sin miedo a equivocarnos. Es el efecto acumulativo de todas estas modificaciones lo que hace de la refactorización una potente técnica. El objetivo final es mantener nuestro código sencillo y bien estructurado. Algunas definiciones podrían ser las siguientes:

REFACTORIZACIÓN

Cambio realizado a la estructura interna del software para hacerlo:
más fácil de comprender
y más fácil de modificar
sin cambiar su comportamiento observable.

REFACTORIZAR

Reestructurar el software aplicando una secuencia de refactorizaciones.

2 POR QUÉ REFACTORIZAR

Dado que la refactorización no supone, tal y como hemos visto en las definiciones, mejora alguna en el comportamiento del software, la primera pregunta que nos puede venir a la cabeza es ¿para qué invertir tiempo del desarrollo en refactorizar? Algunas de las razones que justifican el utilizar esta técnica son:

- **Calidad**. Un código de calidad es un código sencillo y bien estructurado, que cualquiera puede leer y entender sin necesidad de haber estado integrado en el equipo de desarrollo durante varios meses. Se debe terminar el tiempo en que imperaban los programas escritos en una sola línea en la que se hacía de todo, y en el que se valoraba la concisión aún a costa de la legibilidad.
- **Eficiencia**. Mantener un buen diseño y un código estructurado es sin duda la forma más eficiente de desarrollar. El esfuerzo que se invierta en evitar la duplicación de código y en simplificar el diseño se verá compensado cuando se tengan que realizar modificaciones, tanto para corregir errores como para añadir nuevas funcionalidades.
- **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 costos y la existencia de sistemas lo hacen imposible.

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? (si funciona no lo toques) ¿Por qué correr el riesgo de introducir nuevos errores?, ¿Cómo se puede justificar el costo de modificar el código sin desarrollar ninguna nueva funcionalidad?

No siempre es justificable modificar el código. Cada refactorización que se realice debe estar justificada. Sólo debemos refactorizar cuando se identifique código mal estructurado o

diseños que supongan un riesgo para la futura evolución del sistema. Si se detecta que el diseño empieza a ser complicado y difícil de entender, y cada cambio empieza a ser muy costoso, en ese preciso momento, cuando se debe frenar la inercia de seguir desarrollando, ya que de hacerlo, el software se convertirá en algo inmantenible (es imposible o demasiado costoso para realizar un cambio).

Los síntomas que indican que algún código de software tiene problemas se conocen como “Bad Smells”. Hay una lista de ellas 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* (unidad 5), 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. Sin pruebas automáticas nunca estaremos convencidos de no haber introducido nuevos errores en el código, al término de una refactorización, y poco a poco dejaremos de hacerlo por miedo a estropear lo que ya funciona.

3 DIFICULTADES DE LA REFACTORIZACIÓN

En un proyecto real hay que ser consciente de que no refactorizar a tiempo un diseño degradado puede tener consecuencias muy negativas, pero a la vez debe tener en cuenta que el tiempo dedicado a refactorizar no suele ser considerado como un avance del proyecto, por los usuarios, clientes o gestores del mismo.

3.1. El Factor Humano

Una realidad a veces olvidada es que cualquier desarrollador prefiere hacer código de calidad. También es cierto que esta realidad a veces queda oculta debido a condiciones de presión excesiva o menosprecio del trabajo técnico. En este sentido refactorizar es una forma de mejorar la calidad y por tanto es una forma de hacer que el desarrollador esté más orgulloso de su trabajo. Puede decirse que la refactorización bien realizada, rápida y segura, genera satisfacción. Pero también hay un peligro en este hecho: el exceso de celo para conseguir un resultado de calidad puede llegar a suponer un número excesivo de refactorizaciones dando vueltas sobre diseños similares una y otra vez.

Una posible solución consiste en mantener la refactorización bajo control, definir claramente los objetivos antes de comenzar a refactorizar y estimar su duración. Si se excede el tiempo planificado es necesario un replanteamiento. Además de la evidente pérdida de tiempo, refactorizar demasiado suele llevar a dos efectos negativos. El primero es que la modificación del código incremente la complejidad de nuestro diseño, que es justo el efecto contrario del que intentábamos lograr al refactorizar. Por otro, es habitual fomentar la sensación bien del desarrollador o bien de todo el equipo de que no se está avanzando. Una sensación que conduce a repercusiones anímicas negativas.

3.2. Trabajo en equipo

Un equipo de desarrollo debe ser uno. Todos los miembros del equipo deben conocer la arquitectura en cada momento, el estado actual, los problemas que tenemos y el objetivo que se busca. Una reunión diaria es una práctica que facilita la comunicación entre todos los miembros del equipo, y supone una práctica que proporciona un momento a todos los miembros del grupo para plantear sus ideas, dudas e inquietudes respecto al proyecto.

Cuando un desarrollador refactoriza, afecta al resto del equipo. En este sentido, las refactorizaciones internas a una clase son menos problemáticas, pero con las arquitecturales es necesario tener mucho cuidado. Suponen cambios en un gran número de archivos y por tanto es probable que se afecte a archivos que están siendo modificados por otros desarrolladores. La comunicación y coordinación entre los afectados es fundamental para que esto no se convierta en un problema. Por tanto, es necesario que en la reunión diaria del equipo (u otro mecanismo de comunicación equivalente) se planteen las refactorizaciones de este tipo. De esta forma se sabrá en todo momento quienes son los afectados y cuál es el objetivo de la refactorización. De esta forma se promueve la colaboración de todos en la refactorización arquitectural que se va a llevar a cabo. A veces, el comentar con el resto del equipo una refactorización que parece muy necesaria, puede hacer ver que lo que parecía una muy buena idea, no lo es tanto por otros factores que ven otros compañeros y de los que no se era consciente. Esto evita también refactorizaciones en sentidos contrarios propias de equipos en los que falta comunicación.

Las refactorizaciones en sentidos contrarios suelen estar motivadas por tener cada miembro del equipo una idea diferente del diseño hacia el que debe dirigirse ese código. Este problema de comunicación puede tener repercusiones bastante negativas en el diseño resultante. Una vez que ocurre, debe convertirse a una situación en la que todo el equipo comparta una misma visión de la arquitectura y diseño del sistema. Para ello la mejor manera es detener el desarrollo y reunir a todo equipo para alcanzar un acuerdo común. En estas reuniones es muy útil el uso de pizarras y diagramas UML si son conocidos por el equipo.

Incluso los miembros del equipo que no se habían visto involucrados en la situación aprenderán de la experiencia para evitar que vuelvan a darse en el futuro.

3.3. Refactorización a posteriori

El término refactorización a posteriori se refiere a todas aquellos cambios estructurales que se realizan un tiempo después de la implementación de la funcionalidad existente. Existen varios motivos para encontrarse con esta situación. Algunos de ellos son:

- Un equipo comienza a trabajar con código desarrollado por un equipo anterior que o bien no tiene buena calidad o bien no está preparado para que se incorporen nuevas funcionalidades.
- Se ha aplicado refactorización continua pero la calidad del código o el diseño se ha degradado porque no se identificó a tiempo la necesidad de una refactorización o bien se equivocó la refactorización necesaria.

Otra característica de estas refactorizaciones es que afectan no sólo al desarrollador o desarrolladores que van a aplicar la refactorización, sino a todos aquellos que trabajan con la parte del código afectada: por un lado no pueden trabajar con el código o deben hacerlo con mayor precaución durante un tiempo; y por otro cuando vuelvan a trabajar con él se encontrarán con un diseño desconocido.

La mejor estrategia a seguir en estos casos es tratar de dividir la refactorización en el mayor número de pequeños pasos posible. Tras cada paso el código debe seguir funcionando sin fallos. Antes de abordar cada uno de estos pasos se debe comunicar a todas las personas afectadas los cambios que se van a realizar. De esta forma se evita que cuando vuelvan a trabajar con ese código se encuentren con un panorama completamente desconocido. En ocasiones también es recomendable dar explicaciones después de la refactorización para explicar el diseño final. Si este tipo de explicaciones son necesarias con demasiada asiduidad,

la solución no es no realizarlas sino identificar los motivos de tanta refactorización a posteriori. Probablemente sea necesario un mayor grado de refactorización continua.

La refactorización por pasos tiene la ventaja de afectar menos al ritmo normal de desarrollo y permite ser intercalada con la implementación de nuevas funcionalidades. El principal inconveniente es que extiende en el tiempo la existencia de código imperfecto. Por ello siempre que sea posible debe completarse la refactorización en el menor tiempo posible.

Cuando una refactorización es difícil técnicamente y lleva un tiempo considerable, se corre el riesgo de entrar en lo que podríamos denominar la espiral refactorizadora. Esta situación ocurre cuando una refactorización se complica y conduce a nuevas refactorizaciones hasta que llega un momento en el que no es posible o es muy difícil estimar cuanto tiempo queda para terminar de refactorizar. Hay varias medidas que pueden adoptarse para evitar caer en la espiral refactorizadora. La primera de ellas es declarar el objetivo de una refactorización antes de empezar. Si durante la misma aparece nuevo código susceptible de ser refactorizado debe dejarse sin modificar, por el momento sólo se anotará esta necesidad para no olvidar de hacer la refactorización más adelante.

Otra práctica muy útil consiste en introducir el código en el sistema de control de versiones antes de emprender cualquier refactorización que podamos considerar grande. Si cuando se está llevando a cabo una refactorización se dispersan los objetivos iniciales, se llevan muchas horas o incluso días sin tener un código que funciona o resulta imposible estimar cuanto queda para terminar es necesario asumir que la refactorización no va por buen camino. Para volver a la normalidad se debe devolver el código a un estado en el funcionara correctamente. Esto resulta muy fácil si se siguió el consejo anterior de comenzar con código guardado en el sistema de control de versiones. No deben tenerse reparos en tirar las modificaciones empezadas, estaremos cambiando un poco de tiempo de trabajo por recuperar una situación estable que nos permite seguir avanzando sin incertidumbres. Lo

más importante será lo aprendido de la refactorización fallida, que nos será muy valioso para volver a intentarlo.

La mejor forma de enfocar correctamente una refactorización y evitar caer en espirales de refactorizaciones, es conocer muy bien los patrones de diseño. Los patrones de diseño no son sino soluciones que funcionan para problemas comunes que todos encontramos cuando estamos desarrollando. Conocer perfectamente los patrones de diseño permite identificar y ver con claridad el objetivo de la refactorización, mejora la comunicación entre los miembros del grupo al compartir una base de conocimiento muy importante y además hace más eficiente la resolución no sólo de problemas conocidos, sino también de problemas desconocidos al haber adquirido una forma mejor arquitecturada de enfocar los problemas.

4 IMPLANTACIÓN DE LA REFACTORIZACIÓN

4.1. Pasos para implantar la refactorización en un equipo

Implantar la refactorización debe realizarse de una manera progresiva. La experiencia en la implantación de esta técnica nos dice que es necesario seguir los siguientes pasos:

- Escribir pruebas unitarias y funcionales. Refactorizar sin pruebas unitarias y funcionales resulta demasiado costoso y de mucho riesgo.
- Usar herramientas especializadas. Las refactorizaciones en muchas ocasiones obligan a pequeñas modificaciones muy simples en muchas clases de nuestro código. Con herramientas especializadas estas refactorizaciones se realizan automáticamente y sin riesgo.
- Dar formación sobre patrones de refactorización y de diseño. Una buena base teórica sobre las refactorizaciones más comunes permite al programador detectar "Bad Smells" de los que no es consciente y que harán que su código se degrade progresivamente.
- Refactorizar los principales fallos de diseño. La recomendación es comenzar realizando refactorizaciones concretas que corrijan los principales fallos de diseño, como puede ser código duplicado, clases largas, etc que son refactorizaciones sencillas de realizar y de las que se obtiene un gran beneficio.

- Comenzar a refactorizar el código tras añadir cada nueva funcionalidad en grupo. Una vez corregido los errores del código existente, la mejor manera de aplicar Refactorización suele ser al añadir una nueva funcionalidad, de manera que se desarrolle de la manera más eficiente. Realizar discusiones en grupos sobre la conveniencia de realizar alguna refactorización suele ser muy productivo.

- Implantar refactorización continua al desarrollo completo. Esta es la última fase y es cuando cada desarrollador incorpora la Refactorizacion como una tarea más dentro del su proceso de desarrollo de Software.

4.2. 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. Esta es la mayor de las justificaciones de una refactorización continua, que trata de evitar las refactorizaciones grandes haciendo refactorizaciones pequeñas muy a menudo. En concreto, se refactoriza de forma inmediata a la inclusión de una nueva funcionalidad. Esta práctica tiene dos ventajas principalmente:

- El código afectado es el mismo que el que se modificó para añadir la funcionalidad y por tanto se reduce o evita completamente los inconvenientes para otros desarrolladores.
- El tiempo que se debe dedicar es reducido dado que en ese momento se tiene un conocimiento preciso del código que se verá afectado.

Las claves para poder aplicar refactorización continua son:

- Concienciación 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.

Lo más habitual es que este tipo de refactorizaciones surjan solamente al implementar una funcionalidad y dejar los tests pasando satisfactoriamente. Estos mismos tests nos garantizarán que el código sigue funcionando después de la refactorización.

Sin embargo, no siempre resulta evidente para todos los desarrolladores la necesidad de una refactorización. En estos casos el uso de herramientas como CodeRush Express puede emplearse como ayuda para identificar "bad smells" además de unificar criterios entre todo el equipo. En cualquier caso, es importante que estas herramientas no sustituyan nunca al sentido común y se usen únicamente como ayuda.

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 continua la refactorización debe tender a ocupar una parte pequeña en relación al tiempo dedicado a las nuevas funcionalidades.

5 REFACTORIZACIÓN Y PRUEBAS UNITARIAS

La existencia de pruebas automatizadas facilita enormemente las refactorizaciones. Los principales motivos son:

- El riesgo de la refactorización disminuye, dado que es posible comprobar cuando ésta concluye, que todo sigue funcionando satisfactoriamente.
- El desarrollador que realiza la refactorización puede comprobar que no ha estropeado la funcionalidad implementada por otros. Evita efectos colaterales.
- El tiempo necesario para comprobar que todo sigue funcionando es menor lo que permite avanzar en pasos más pequeños si se desea.
- Refactorizar sin tests es una actividad de alto riesgo y no debe hacerse salvo para las refactorizaciones más sencillas y realizadas con herramientas especializadas. En ocasiones incluso algunas refactorizaciones muy pequeñas pueden provocar la aparición de bugs muy difíciles de identificar a posteriori.

En este sentido es muy importante resaltar la importancia de que las pruebas tanto funcionales como unitarias comprueben que el código bajo prueba funcione correctamente independientemente de cómo esté implementado. Cuanto mayor sea el acoplamiento de la prueba con la implementación, mayores serán los cambios que habrá que realizar en esta cuando se lleve a cabo.

Pero incluso aunque no exista este acoplamiento, las refactorizaciones grandes, que afectan al diseño del código significativamente, obligarán a realizar cambios en las pruebas. Esto provoca una situación incómoda, las pruebas que deben ser una ayuda, se convierten en este caso en un estorbo a la refactorización. La mejor solución es prevenir aplicando refactorizaciones continuas y más pequeñas.

En este sentido también resulta útil pensar antes de refactorizar en cómo van a afectar a los tests y cómo se pueden usar en nuestro beneficio en vez de dejar que se conviertan en un problema.

5.1. Patrones de refactorización más usuales

- Duplicated code (código duplicado): Es la principal razón para refactorizar. Si se detecta el mismo código en más de un lugar, se debe buscar la forma de extraerlo y unificarlo.
- Long method (método largo). Cuanto mas corto es un método más fácil de reutilizarlo es.
- Large class (clase grande). Si una clase intenta resolver muchos problemas, usualmente suele tener varias variables de instancia… lo que suele conducir a código duplicado.
- Long parameter list (lista de parámetros extensa): en la programación orientada a objetos no se suelen pasar muchos parámetros a los métodos, sino sólo aquellos mínimamente necesarios para que el objeto involucrado consiga lo necesario. Éste tipo de métodos, los que reciben muchos parámetros, suelen variar con frecuencia, se tornan difíciles de comprender e incrementan el acoplamiento.
- Divergent change (cambio divergente): una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre si. Este síntoma es el opuesto del siguiente.
- Shotgun surgery: éste síntoma se presenta cuando luego de un cambio en un determinado lugar, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.
- Feature envy (envidia de funcionalidad): un método que utiliza más cantidad de elementos de otra clase que de la propia. Se suele resolver el problema pasando el método a la clase cuyos componentes son más requeridos para usar.
- Data class (clase de datos): Clases que sólo tienen atributos y métodos de acceso a ellos (“get” y “set”). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.

- Refused bequest (legado rechazado): Subclases que usan sólo pocas características de sus superclases. Si las subclases no necesitan o no requieren todo lo que sus superclases les proveen por herencia, esto suele indicar que como fue pensada la jerarquía de clases no es correcto. La delegación suele ser la solución a éste tipo de inconvenientes.

5.2. Ejemplos de refactorizaciones

- Un ejemplo de una refactorización trivial es cambiar el nombre de una variable para que sea más significativo, como una sola letra 't' a 'tiempo'.
- Una refactorización más compleja es transformar el código dentro de un bloque en una subrutina
- Una refactorización todavía más compleja es remplazar una sentencia condicional if por polimorfismo

6 REFACTORIZACIÓN CON VISUAL STUDIO/INTELLIJ IDEA

6.1. Tabulación

La tabulación no es una técnica de refactorización propiamente dicha porque no se modifica código, pero ayuda a clarificar su estructura.

En **Visual Studio** podemos tabular automáticamente el código usando la entrada de menú *Editar → Avanzadas → Dar formato* al documento. Se puede personalizar el formato del código en *Herramientas → Opciones → Editor de texto*. En **IntelliJ IDEA** disponemos de las opciones (vistas en el tema de documentación) *Code → Reformat Code o Rearrange Code*.

6.2. Renombrar

Se usa para cambiar el nombre de variables, campos, métodos, propiedades, clases, namespaces, etc, cuando tienen un nombre no significativo. La potencia de este método está en que la refactorización se extiende a todo el proyecto. Es decir, el nuevo nombre se cambia en todos los lugares donde se hacía referencia al antiguo.

En **Visual Studio** se realiza seleccionando el elemento a renombrar y eligiendo la opción *Editar → Refactorizar → Cambiar Nombre*, o escogiendo la opción directamente con el botón derecho. En **IntelliJ IDEA** podemos localizar la opción en el menú *Refactor → Change Name*, o pulsando *Mays + F6*.

6.3. Encapsular campo

Se hace cuando tenemos un campo público y lo queremos convertir en propiedad para restringir el acceso. Como norma generar de diseño en POO, no deben existir campos públicos, ya que se pueden manipular directamente y producir datos erróneos.

En **Visual Studio** se hace con la opción *Editar → Refactorizar → Encapsular campo*. En **IntelliJ IDEA**, disponemos de la opción *Refactor → Encapsulate Fields*.

6.4. Extraer método

Extraer método es una operación de refactorización que crea un nuevo método a partir de un fragmento de código existente.

Utilizando *Extraer método* se puede crear un nuevo método extrayendo una selección de código de dentro del bloque de código de un miembro existente. El nuevo método que se ha extraído contiene el código seleccionado, y el código seleccionado del miembro existente se reemplaza por una llamada al nuevo método.

Normalmente se aplica cuando hay código repetido en varios lugares de nuestro programa. Al agrupar ese código en un método, éste puede ser llamado desde donde estaba el código original.

Para aplicarlo en **Visual Studio**, se selecciona el código a extraer y seleccionamos la opción *Editar → Refactorizar → Extraer Método*. En el cuadro de diálogo escribimos el nombre del nuevo método. De manera similar, en **IntelliJ IDEA** hemos de usar la opción *Refactor → Extract… → Method*.

6.5. Quitar parámetros

Quitar parámetros es una operación de refactorización que proporciona una manera sencilla de quitar parámetros de los métodos. Quitar parámetros cambia la declaración; en todas las ubicaciones donde se llama al miembro, el parámetro se quita para reflejar la nueva declaración. *Quitar parámetros* se utiliza cuando ya no necesitamos esos "datos de entrada" en un método y se pueden omitir sin perjudicar al método y de paso la ejecución del programa.

La operación de quitar parámetros en **Visual Studio** se realiza colocando primero el cursor en un método. Cuando el cursor esté en la posición correcta, para invocar la operación Quitar Parámetros, haga clic en el menú *Editar → Refactorizar → Quitar parámetros*. Quitar parámetros permite eliminar un parámetro al que se hace referencia en el cuerpo del miembro, pero no quita las referencias a dicho parámetro en el cuerpo del método. Esto puede producir errores de compilación en el código. Sin embargo, puede usar el cuadro de diálogo *Vista previa de los cambios* para revisar el código antes de ejecutar la operación de refactorización.

En **IntelliJ IDEA**, se puede realizar esta operación (y cualquiera que afecte a la organización de los parámetros, como la del punto 6.6) con la opción *Refactor → Change Signature (CTRL+F6)*.

6.6. Reordenar parámetros

Reordenar Parámetros es una operación de refactorización que cambia el orden de los parámetros de los métodos. Reordenar Parámetros cambia la declaración y, en todas las ubicaciones donde se llama al miembro, los parámetros se reorganizan para reflejar el nuevo orden.

Para realizar la operación *Reordenar Parámetros* en **Visual Studio**, sitúe el cursor sobre o a un lado de un método. Una vez colocado el cursor, invoque la operación *Editar → Refactorizar → Reordenar parámetros*. En el cuadro de diálogo *Reordenar parámetros*, seleccionamos el parámetro que queremos cambiar de orden y, a continuación, lo subimos o bajamos con los botones de flecha. También se puede arrastrar en la lista de parámetros. Si seleccionamos la opción *Vista previa de los cambios de referencia* en el cuadro de diálogo *Reordenar parámetros*, aparecerá un cuadro de diálogo con la vista previa.

6.7. Extraer interfaz

Extraer interfaz sirve crear una nueva interfaz con miembros originados a partir de una clase o interfaz existente. Esto es útil cuando varias clases tienen un subconjunto de métodos en común.

En **Visual Studio** se hace con el botón derecho del ratón en la clase de la que queremos extraer los métodos y seleccionando *Editar → Refactorizar → Extraer interfaz*. A continuación, hay que seleccionar los métodos que queremos pasar a la nueva interfaz. También tenemos que escribir el nombre de la nueva interfaz y del archivo que la contiene. Esto genera una nueva interfaz en un archivo nuevo con los métodos abstractos creados a partir de los métodos que hemos seleccionado. De manera similar, en **IntelliJ IDEA** podemos realizar la operación con la opción *Refactor → Extract... → Interface*.

6.8. Reemplazar número mágico por constante simbólica

Un número mágico es un valor codificado que puede cambiar en una etapa posterior, pero que puede ser, por tanto, difícil de actualizar. El término número mágico se refiere a la mala práctica de programación de utilizar números directamente en código fuente sin razón. En la mayoría de los casos esto hace más difícil de leer, comprender y mantener programas. Aunque la mayoría de normas de estilo hacen una excepción para los números cero y uno, es una buena idea definir todos los demás números en código como constantes con nombre.

Visual Studio no proporciona ninguna operación de refactorización para reemplazar número mágico por constante simbólica. En **IntelliJ IDEA**, podemos hacer este reemplazo con la opción *Refactor → Introduce Constant (Ctrl+Alt+C)*.

Ejemplo:

Número mágico

```
double EnergiaPotencial(double masa, double altura) {
        return masa * 9.81 * altura;
    }
```

Constante simbólica

```
static const double IntensidadDeGravedad = 9.81;

double EnergiaPotencial(double masa, double altura) {
          return masa * IntensidadDeGravedad * altura;
     }
```

6.9. Otras herramientas de refactorización

Hemos visto que las herramientas de refactorización de **Visual Studio** son muy básicas y sólo cubren las técnicas más básicas, aunque también son las que se usan con más frecuencia. Si queremos ampliar esta funcionalidad, debemos acudir a extensiones, siendo las más usadas ReSharper y CodeRush (de pago). También podemos encontrar una extensión del equipo de SharpDevelop (Refactoring Essentials for Visual Studio 2017). En **IntelliJ IDEA**, sin embargo, disponemos de un menú dedicado (*Refactor*) con multitud de opciones para explorar.

7 ANEXO: INTERFACES

Una interfaz es una clase que sólo contiene métodos abstractos. En C# también puede tener propiedades, eventos e indizadores. Un método abstracto es un método que sólo tiene declaración, pero no implementación.

Cuando una clase implementa una interfaz, debe implementar (definir el código de) los métodos de esa interfaz. Además, mientras que una clase sólo puede heredar de una sola clase (herencia simple), puede implementar varias interfaces, lo que proporciona un mecanismo parecido a la herencia múltiple.

Las interfaces definen y estandarizan las formas en que pueden interactuar las cosas entre sí, como las personas y los sistemas. Por ejemplo, los controles en una radio sirven como una interfaz entre los usuarios de la radio y sus componentes internos. Los controles

permiten a los usuarios realizar un conjunto limitado de operaciones (por ejemplo, cambiar la estación, ajustar el volumen, seleccionar AM o FM), y distintas radios pueden implementar los controles de distintas formas (por ejemplo, el uso de botones, ruedas, comandos de voz). La interfaz especifica qué operaciones debe permitir la radio que realicen los usuarios, pero no cómo deben hacerse.

Por lo general, una interfaz se utiliza cuando clases dispares (es decir, no relacionadas) necesitan compartir métodos comunes. Esto permite que los objetos de las clases que implementan la misma interfaz puedan responder a las mismas llamadas a métodos.

Por ejemplo, si tenemos la interfaz IPagable:

```
public interface Ipagable
{
     double ObtenerTotalAPagar();
}
```

Puede ser implementada por las clases Factura y Empleado de forma diferente:

```
public class Factura : IPagable
{
     private double cantidad;
     private double precio;

     double ObtenerTotalAPagar()
     {
          return cantidad * precio;
     }
}

public class Empleado : IPagable
{
     private double salario;

     double ObtenerTotalAPagar()
     {
          return salario;
     }
}
```