29.9.2020 | Desarrollo web (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/)

Refactorización: cómo mejorar el código fuente

Durante el proceso de desarrollo de una aplicación, en el código fuente (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/codigo-fuente-definicion-con-ejemplos/) se van acumulando **elementos con una estructura deficiente** que ponen en riesgo la aplicabilidad y la compatibilidad del programa. Para solucionar este problema, hay dos opciones: o bien volver a escribir el código desde cero o realizar una **reestructuración en pequeños pasos**. Cada vez más programadores y empresas optan por la segunda opción, es decir, la refactorización del código, para optimizar un *software* activo a largo plazo y hacerlo más legible y claro para otros programadores.

La refactorización (o *refactoring*, en inglés) plantea una pregunta: ¿qué problema se quiere solucionar usando qué método? Hoy en día, la refactorización ya forma parte de las lecciones básicas del aprendizaje de la programación (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/aprender-a-programar-introduccion-y-conceptos-basicos/) y es cada vez más relevante, pero ¿qué métodos utiliza y **qué ventajas e inconvenientes trae consigo**?

Índice

1 ¿Qué es la refactorización?
2 Cuando el código fuente decae: el código espagueti
3 ¿Cuál es el objetivo de la refactorización?
4 ¿Qué fuentes de error corrige la refactorización?
5 ¿Cómo se aplica la refactorización?
6 ¿Qué técnicas existen?
7 Ejemplo de refactorización: cambiar el nombre a un método
8 Refactorización: ¿qué ventajas y desventajas presenta?

¿Qué es la refactorización?

Programar *software* conlleva un proceso largo que suele involucrar, al menos parcialmente, a varios desarrolladores, por lo que durante el desarrollo el código fuente a menudo se amplía y modifica. Si además de estos cambios, tenemos en cuenta que se trabaja a contrarreloj y utilizan ciertas prácticas anticuadas, observamos como resultado una acumulación de elementos defectuosos en el código fuente, los **llamados *code smells***. Estos puntos débiles, que van aumentando a medida que avanza el proceso, ponen en peligro la funcionalidad y la compatibilidad del *software* en cuestión. Para evitar esta erosión continua del programa, se utiliza la refactorización o *refactoring*.

La refactorización podría compararse con la **corrección** de un libro: el producto final de la corrección no es un nuevo libro, sino el mismo texto, pero más comprensible. Así, al igual que en la corrección de un libro se usan procedimientos como la reformulación y la reestructuración o eliminación de frases, en la refactorización de código se aplican **métodos como la encapsulación, el reformateo o la extracción** para optimizar el código sin cambiar su contenido.

Este proceso resulta mucho más económico que reescribir la estructura del código desde cero. La refactorización desempeña un papel especialmente importante en el **desarrollo iterativo e incremental**, así como en el desarrollo ágil de software (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/agile-development/), ya que este modelo cíclico lleva a los programadores a modificar el *software* una y otra vez. En este proceso, la refactorización es un paso clave.

Cuando el código fuente decae: el código espagueti

Antes que nada, es importante entender que un código puede degenerar a lo largo del tiempo y convertirse en un infame *código espagueti*. Ya sea por falta de tiempo, por falta de experiencia o por directrices poco claras, las órdenes innecesariamente complejas en la programación del código acaban obstaculizando su funcionalidad. Cuanto más rápido y complejo sea el ámbito de aplicación de un código, más se erosionará.

El término *código espagueti* hace referencia a **códigos fuente confusos y de difícil lectura**, cuya estructura es difícil de comprender para los programadores. Algunos ejemplos típicos de elementos que complican el código son las órdenes de salto (*GOTO*) redundantes, que indican al programa que vaya saltando de un sitio a otro en el código; los bucles *for/while* y los comandos *if*.

Concretamente, los proyectos en los que trabajan muchos desarrolladores suelen generar un código poco legible. Si un código que ya de por sí presentaba ciertas imperfecciones pasa por muchas manos, es difícil evitar que se encadenen modificaciones a modo de parche y que, finalmente, se requiera **una cara revisión o review del código (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/code-review/)** para corregirlo. En el peor de los casos, el *código espagueti* puede poner en peligro todo el proceso de desarrollo del *software*, llegando a un punto en el que ni siquiera la refactorización puede resolver el problema.

Los llamados ***code smells*** y el ***code rot*** (es decir, los defectos y la erosión del *software*) no tienen por qué ser tan preocupantes. Con el paso del tiempo, si un código contiene muchos elementos innecesarios, puede empezar a *apestar*, por así decirlo. Las secciones poco claras de la estructura empeoran cada vez que un programador nuevo se pone a trabajarlas o amplía el código. Es por tanto necesario realizar un *refactoring* en cuanto aparezcan los primeros *code smells*, ya

que si no el código seguirá erosionándose y perderá su funcionalidad a causa del *code rot* (el proceso de *putrefacción*, traducido literalmente).

¿Cuál es el objetivo de la refactorización?

La refactorización siempre tiene el sencillo y claro propósito de **mejorar el código**. Con un código más efectivo, puede facilitarse la integración de nuevos elementos sin incurrir en errores nuevos. Además, cuanto más fácil les resulte a los programadores leer el código, más rápido se familiarizarán con él y podrán identificar y evitar los bugs de forma más eficiente. Otro objetivo de la refactorización es **mejorar el análisis de errores y la necesidad de mantenimiento** del *software*. Poner a prueba el código ahorra esfuerzo a los programadores.

¿Qué fuentes de error corrige la refactorización?

Los métodos aplicados en la refactorización son tan variados como los errores que tratan de corregir. De manera general, la refactorización del código se guía por sus errores y va mostrando los pasos necesarios para acortar o eliminar procesos de corrección. Algunas de las fuentes de error que pueden corregirse mediante *refactoring* son las siguientes:

- **Estructuras complicadas o demasiado largas:** cadenas y bloques de comandos tan largos que la lógica interna del *software* se vuelve incomprensible para lectores externos.
- **Redundancias en el código**: los códigos poco claros suelen contener repeticiones que han de corregirse una a una durante el mantenimiento, por lo que consumen mucho tiempo y recursos.
- **Listas de parámetros demasiado largas**: los objetos no se asignan directamente a un método, sino que se indican sus atributos en una lista de parámetros.
- **Clases con demasiadas funciones**: clases con demasiadas funciones definidas como método, también llamadas *god objects*, que hacen que adaptar el *software* se vuelva casi imposible.
- **Clases con funciones insuficientes**: clases con tan pocas funciones definidas como método que se vuelven innecesarias.
- **Código demasiado general con casos especiales**: funciones con casos especiales demasiado específicos que apenas se usan y que, por lo tanto, dificultan la incorporación de ampliaciones necesarias.
- ***Middle man***: una clase separada actúa como intermediaria entre los métodos y las distintas clases, en lugar de direccionar las solicitudes de los métodos directamente a una clase.

¿Cómo se aplica la refactorización?

La refactorización debe llevarse a cabo antes de modificar una función del programa. En el mejor de los casos, debe realizarse **en muy pocos pasos** y comprobando cada modificación del código mediante procesos de desarrollo de *software* como el desarrollo guiado por pruebas (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-el-test-driven-development/) (TDD, por sus siglas en inglés) y la integración continua (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/integracion-continua/) (CI). En pocas palabras, el TDD y la CI se encargan de poner a prueba los nuevos segmentos de código creados por los programadores, que luego son integrados y cuya funcionalidad es evaluada mediante procesos de prueba, a menudo automatizados.

Por regla general, un programa ha de ser modificado en pocos pasos y desde dentro, sin que su función externa se vea afectada. Tras cada cambio, debe realizarse un test que esté automatizado en la medida de lo posible.

¿Qué técnicas existen?

Hay muchísimas técnicas concretas de refactorización. Para conocerlas todas, se puede consultar la exhaustiva obra sobre este tema de Martin Fowler y Kent Beck: ***Refactoring: Improving the Design of Existing Code***. A continuación, presentamos un resumen:

Desarrollo rojo-verde:

El llamado desarrollo *rojo-verde* es un método ágil de desarrollo de *software* basado en test. Suele aplicarse cuando se quiere integrar una nueva función en un código existente. El *rojo* es el símbolo del primer test, realizado antes de la implementación de la nueva función. El *verde*, por su parte, se refiere al segmento de código más sencillo que requiere la función para superar el test. A continuación, se realiza una ampliación **con test constantes** para descartar el código defectuoso y aumentar así la funcionalidad. El desarrollo *rojo-verde* es un **elemento clave para la refactorización continua** en el desarrollo continuo de *software*.

Branching by abstraction

Este método de refactorización aplica cambios graduales a un sistema y va **modificando elementos anticuados del código y sustituyéndolos por segmentos nuevos**. El *branching by abstraction* suele utilizarse cuando se realizan cambios grandes que afectan a la **jerarquía de clases, a las herencias y a la extracción**. Al implementar una abstracción que se mantiene enlazada con una implementación antigua, pueden enlazarse con la abstracción otros métodos y clases. De esta manera, es posible sustituir la funcionalidad del segmento antiguo por la abstracción.

A menudo, este proceso se lleva a cabo mediante **métodos de *pull-up* o *push-down***. La función nueva y mejorada se enlaza con la abstracción y se le transmiten los enlaces. Al hacerlo, o bien se transforma una subclase en una clase superior (*pull-up*) o se divide una clase superior en subclases (*push-down*).

Finalmente, pueden borrarse las funciones antiguas sin poner en peligro la funcionalidad del conjunto. Gracias a estos **cambios a pequeña escala**, el sistema funciona igual que antes mientras se van remplazando uno a uno los segmentos defectuosos del código por otros mejorados.

Combinar métodos

La refactorización de código tiene el objetivo de que los métodos puedan leerse de la manera más fácil posible. En el mejor de los casos, los programadores externos que lean el código deberían poder captar la lógica interna del método. Para combinar métodos de forma eficiente, el *refactoring* cuenta con diversas técnicas. El objetivo de cada cambio es unificar métodos, eliminar redundancias y dividir métodos largos en segmentos separados que puedan ser modificados fácilmente en el futuro.

Algunas de estas técnicas son las siguientes:

- Extraer métodos
- Convertir métodos a *inline*
- Eliminar variables temporales
- Sustituir variables temporales por métodos de solicitud
- Introducir variables descriptivas
- Separar variables temporales
- Eliminar redireccionamientos a variables de parámetro
- Sustituir métodos por un objeto método
- Sustituir el algoritmo

Mover propiedades entre clases

Para mejorar un código, es necesario poder mover atributos o métodos entre clases. Las siguientes técnicas sirven para realizar estos cambios:

- Mover el método
- Mover el atributo
- Extraer la clase
- Convertir una clase a *inline*
- Ocultar el *delegate*
- Eliminar una clase en el centro
- Introducir métodos ajenos
- Introducir una ampliación local

Organización de los datos

El objetivo de este método es clasificar los datos en clases, que deben ser tan pequeñas y fáciles de comprender como sea posible. Deben ser eliminados y divididos en clases lógicas los enlaces innecesarios entre clases que perjudiquen la funcionalidad del *software* ante pequeños cambios.

Algunos ejemplos de este tipo de técnicas son los siguientes:

- Encapsular los propios accesos a atributos
- Sustituir un atributo propio por una referencia de objeto
- Sustituir un valor por una referencia
- Sustituir una referencia por un valor
- Agrupar datos observables
- Encapsular atributos
- Sustituir un conjunto de datos por una clase de datos

Simplificación de fórmulas condicionadas

Las fórmulas condicionadas deberían simplificarse tanto como sea posible durante la refactorización del código. Para hacerlo, existen varias técnicas:

- Dividir las condiciones
- Agrupar fórmulas condicionadas
- Agrupar comandos redundantes en fórmulas condicionadas
- Eliminar elementos de control
- Sustituir condiciones activas por guardias
- Sustituir distinciones de caso por polimorfismo
- Introducción de objetos cero

Simplificación de las llamadas a método

Las llamadas o solicitudes a método pueden ejecutarse más fácil y rápidamente con las siguientes técnicas:

- Cambiar el nombre de los métodos
- Añadir parámetros
- Eliminar parámetros
- Sustituir parámetros por métodos explícitos
- Sustituir código defectuoso por excepciones

Ejemplo de refactorización: cambiar el nombre a un método

En el siguiente ejemplo, el nombre original del método no deja clara su función. El método debería revelar el código postal de la dirección de una oficina, pero esta tarea no está indicada directamente en el código. Para formularlo de manera más clara, puede refactorizarse el código cambiándole el nombre al método.

Antes:

```
String getPostalCode() {
    return (theOfficePostalCode+“/“+theOfficeNumber);
}
System.out.print(getPostalCode());
```

Después:

```
String getOfficePostalCode() {
    return (theOfficePostalCode+“/“+theOfficeNumber);
}
System.out.print(getOfficePostalCode());
```


Refactorización: ¿qué ventajas y desventajas presenta?

Ventajas	Desventajas
Una mejor comprensión facilita el mantenimiento y la ampliación del software.	Una refactorización imprecisa podría generar nuevos bugs y errores en el código.
La reestructuración del código fuente puede realizarse sin cambiar la funcionalidad.	No existe una definición clara de qué es un código limpio o bien estructurado.
La mejora en la legibilidad del código facilita que otros programadores lo comprendan.	Los clientes no suelen percatarse de las mejoras en el código, ya que la funcionalidad no varía, de forma que la utilidad de la refactorización no es siempre obvia.
La eliminación de las redundancias aumenta la eficiencia del código.	Cuando la refactorización es realizada por equipos grandes, llegar a acuerdos podría suponer más trabajo del esperado.
Los métodos cerrados en sí mismos impiden que los cambios locales afecten a otras partes del código.	
Un código bien estructurado, con métodos y clases más cortas y cerradas en sí mismos, puede ponerse a prueba más fácilmente.	

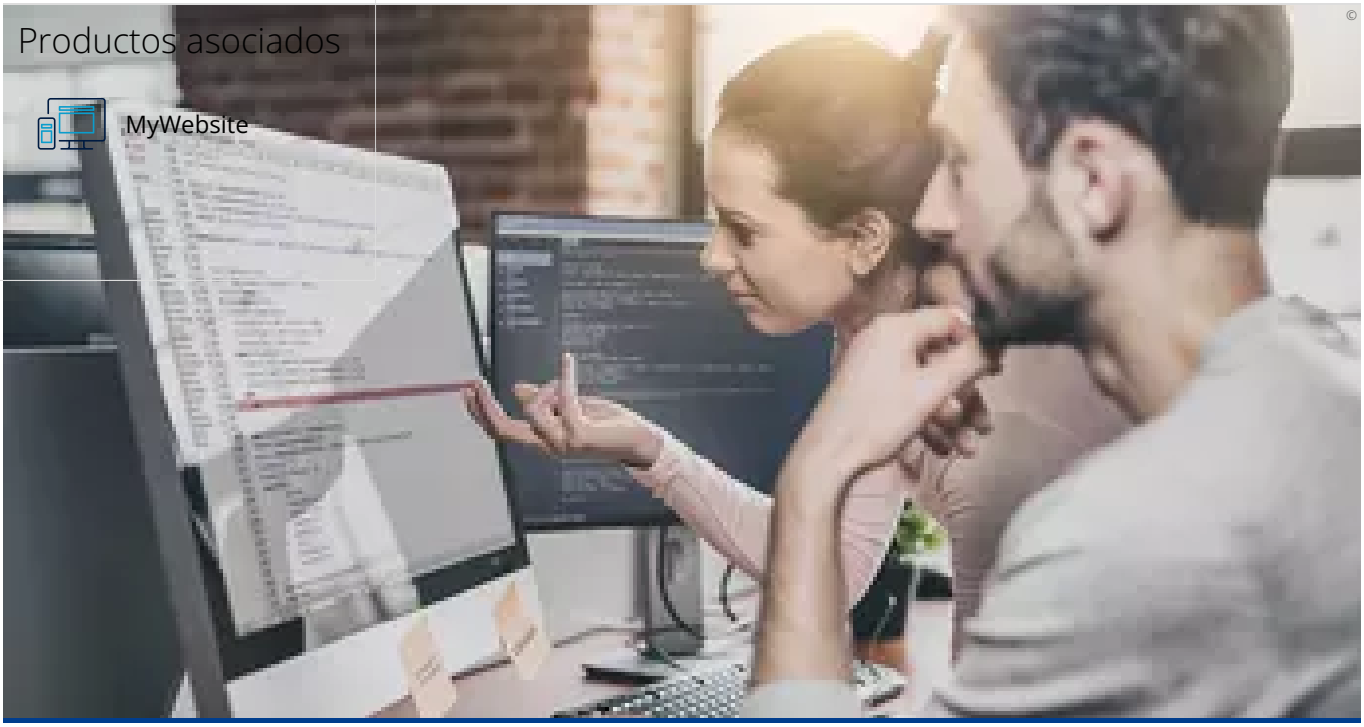
En términos generales, es importante añadir funciones nuevas únicamente si puede mantenerse intacto el código fuente original y, de la misma manera, cambiar el código fuente (es decir, refactorizar) únicamente si no pueden añadirse funciones nuevas.

29.9.2020 | Desarrollo web (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/)

SDK: ¿qué es un software development kit?

25.9.2019 | Desarrollo web

Son muchas las cosas que hay que tener en cuenta en el desarrollo de software: la usabilidad es tan importante como la funcionalidad de la aplicación y, naturalmente, la ejecución no ha de dejar pasar un solo error. Además, el programa tiene que ser compatible con las plataformas y dispositivos para los que esté previsto. ¿Hasta qué punto son útiles los kits de desarrollo de software para cumplir...



Legacy code: cómo trabajar con código obsoleto o desconocido

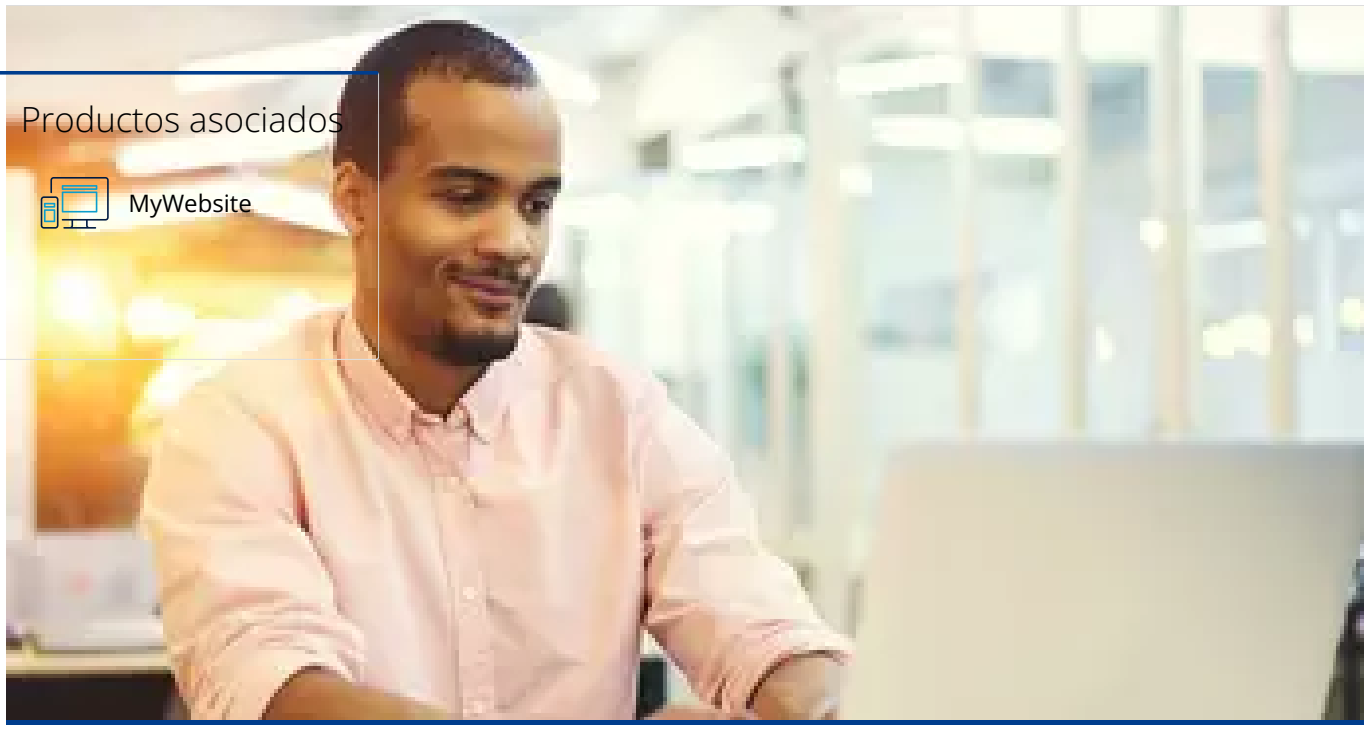
16.9.2020 | Desarrollo web

Legacy code es un término inglés que se traduce como código antiguo o código heredado. Este código suele ser incomprensible para los programadores, además de complicado de mantener y utilizar, ya que se basa en versiones de software anticuadas o escritas por otras empresas. El legacy code no se puede probar mediante pruebas de regresión, si bien existen maneras de trabajar con él.

El behavior-driven development en el desarrollo ágil de software

1.10.2020 | Desarrollo web

El behavior-driven development (BDD) es un componente esencial del desarrollo ágil de software. Esta técnica no se basa en un lenguaje de programación concreto, sino en un formato de texto a partir del cual se pueden realizar pruebas de forma automática. El llamado desarrollo guiado por comportamiento facilita el uso de herramientas complejas sin conocimientos en programación que ponen a prueba la...



Archivo readme: resumen con plantilla

8.10.2020 | Desarrollo web

En general, el archivo readme es el punto de partida para empezar un proyecto, instalar un software o ejecutar una actualización. Es también muy importante para que los desarrolladores puedan integrar repositorios públicos (como, por ejemplo, GitHub) en su propio proyecto. Aquí te contamos en qué consiste un archivo readme y además te ofrecemos una plantilla readme.md.

Cloud native: explicación fácil

31.3.2021 | Desarrollo web

En el ámbito del desarrollo de software, el término de nativo en la nube lleva ya un tiempo en boca de todos. Pero ¿qué significa exactamente cloud native y cómo se implementa con éxito? El objetivo de este modelo ágil es que las aplicaciones puedan incorporarse sin problemas a una infraestructura en la nube a medida que se desarrollan. Descubre las características, ventajas y definición de nativo...

Productos asociados

MyWebsite

Artículos Favoritos