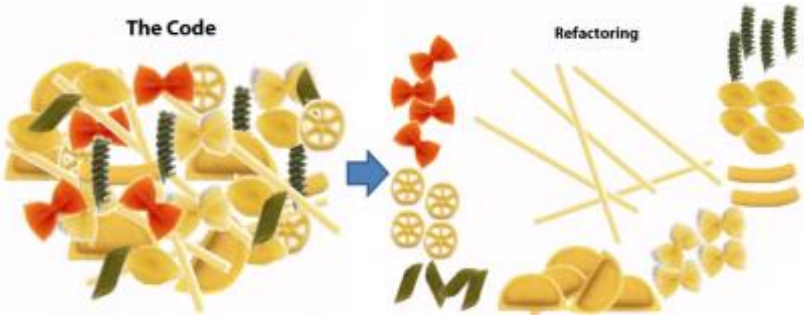
TEMA IV

REFACTORIZACIÓN Y DOCUMENTACIÓN

Refactorización

El término refactorizar dentro del campo de la Ingeniería del Software hace referencia a la modificación del código sin cambiar su funcionamiento. Se emplea para crear un código más claro y sencillo. Se podría entender como el mantenimiento del código, para facilitar su comprensión, pero sin añadir ni eliminar funcionalidades. Refactorizar código consiste en crear un código más limpio.



La refactorización debe ser un paso aislado en el diseño de un programa, para evitar introducir errores de código al reescribir o modificar algunas partes de este. Refactorizamos para:

- Limpiar código, mejorando la consistencia y la claridad.
- Eliminar el código “muerto”, y modularizar.
- Facilitar el futuro mantenimiento y modificación del código.

Cada paso es simple, por ejemplo, mover una propiedad desde una clase a otra, convertir determinado código en un nuevo método, etc. La acumulación de todos estos pequeños cambios puede mejorar el diseño.

Hay que diferenciar la refactorización de la optimización. En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento. Sin embargo, cuando se optimiza, se persigue una mejora del rendimiento, por ejemplo, mejorar la velocidad de ejecución, pero esto puede hacer un código más difícil de entender.

Un área problemática de la refactorización son las bases de datos. Una base de datos presenta muchas dificultades para poder ser modificada, dado la gran cantidad de interdependencias que soporta. Cualquier modificación que se requiera de las bases de datos, incluyendo modificación de esquema y migración de datos, puede ser una tarea muy costosa. Es por ello por lo que la refactorización de una aplicación asociada a una base de datos siempre será limitada, ya que la aplicación dependerá del diseño de la base de datos.

Antes de refactorizar se realizan las pruebas para comprobar el que código "funciona". Después se analizan los cambios a realizar, y se aplican.
Por último, se vuelven a ejecutar las pruebas, de manera que los resultados tienen que ser los mismos que al principio.

Bad Smells

Se conoce como Bad Smell o Code Smell (mal olor) a algunos indicadores del código que posiblemente ocultan un problema más profundo. No son errores de código, bugs, ya que no impiden que el programa funcione correctamente, pero son indicadores de fallos en el diseño del código que dificultan el posterior mantenimiento de este y aumentan el riesgo de errores futuros. Algunos casos:

- Código duplicado (Duplicated code). Si se detectan bloques de código iguales o muy parecidos en distintas partes del programa, se debe extraer creando un método para unificarlo.

- Métodos muy largos (Long Method). Los métodos de muchas líneas dificultan su comprensión. Un método largo probablemente está realizando distintas tareas, que se podrían dividir en otros métodos. Las funciones deben ser los más pequeñas posibles (3 líneas mejor que 15). Cuanto más corto es un método, más fácil es reutilizarlo. Un método debe hacer solo una cosa, hacerla bien, y que sea la única que haga.

- Clases muy grandes (Large class). Problema anterior aplicado a una clase. Una clase debe tener solo una finalidad. Si una clase se usa para distintos problemas tendremos clases con demasiados métodos, atributos e incluso instancias. Las clases deben el menor número de responsabilidades y que esté bien delimitado.

- Atributo público en una clase. Se pone privado y si se tiene que acceder o cambiar el valor del atributo fuera de la clase se codifican métodos get y set para acceder a él fuera de la clase. Sólo deben permanecer públicos los atributos que son constantes numéricas.

- 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.

- Lista de parámetros extensa (Long parameter list). Las funciones deben tener el mínimo número de parámetros posible. Si un método requiere muchos parámetros puede que sea necesario crear una clase con esa cantidad de datos y pasarle un objeto de la clase como parámetro. Del mismo modo ocurre con el valor de retorno, si necesito devolver más de un dato.

- Cambio divergente (Divergent change). Si una clase necesita ser modificada a menudo y por razones muy distintas, puede que la clase esté realizando demasiadas tareas. Podría ser eliminada y/o dividida.

- Shotgun surger: este síntoma se presenta cuando después de un cambio en un determinado lugar, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.

- Envidia de funcionalidad (Feature Envy). Ocurre cuando una clase usa más métodos de otra clase, o un método usa más datos de otra clase, que de la propia.

- Legado rechazado (Refused bequest). Cuando una subclase extiende (hereda) de otra clase, y utiliza pocas características de la superclase, puede que haya un error en la jerarquía de clases.

Además, hay que refactorizar:

- Código no identado
- Los nombres de las variables y métodos deben proporcionarnos alguna información. Si constan de varias palabras se empieza por minúscula y luego el comienzo de cada nueva palabra con mayúscula. Ej puntoMedio. El nombre de las clases la primera por mayúsculas.

Momentos para refactorizar

Refactorizar no es una actividad que suele planificarse, pero debería hacerse habitualmente, siempre que se necesite. Algunos momentos propicios para refactorizar son:

• Cuando se aplica 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.

• Cuando se resuelve un fallo. El reporte de un fallo del software suele indicar que el código no estaba lo suficientemente claro.

• Cuando se realiza 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 autor del código éste sea claro, pero suele ocurrir que para el resto no lo es.

Momentos para no refactorizar

Cuando se dispone de código que simplemente no funciona. A veces es mejor reescribir el código desde cero.
Cuando se está próximo a una entrega.

En el proceso de refactorización, se siguen una serie de patrones preestablecidos, los más comunes son los siguientes:

•Renombrado(rename): este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.

•Sustituir bloques de código por un método: este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.

•Campos encapsulados: se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.

•Mover la clase: si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización.

•Borrado seguro: se debe comprobar, que cuando un elemento del código ya no es necesario, se han borrado todas las referencias a él que había en cualquier parte del proyecto.

•Cambiar los parámetros del proyecto: nos permite añadir nuevos parámetros a un método y cambiar los modificadores de acceso.

•Extraer la interfaz: crea una nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.

•Mover del interior a otro nivel: consiste en mover una clase interna a un nivel superior en la jerarquía.

Analizadores de código

El análisis estático de código es un proceso que tiene como objetivo, evaluar si nuestro software sigue las normas de estilo predefinidas para el lenguaje de programación usado y las predefinidas por nosotros. Esto facilitará la incorporación de una nueva persona al equipo y el posterior mantenimiento de la aplicación. A este tipo de software se le llama **linter**

Los analizadores de código incluyen analizadores léxicos y sintácticos que procesan el código fuente y de un conjunto de reglas que se deben aplicar sobre determinadas estructuras. Si el analizador considera que nuestro código fuente tiene una estructura mejorable, nos lo indicará y también nos comunicará la mejora a realizar.

Los IDE incorporan esta función, por ejemplo, FindBugs en Netbeans, aunque el análisis se puede hacer manualmente.

Tomando como base el lenguaje de programación Java, tenemos en analizador PDM

Refactorización en los entornos de desarrollo

Los entornos de desarrollo actuales nos proveen de una serie de herramientas que nos facilitan la labor de refactorizar. En puntos anteriores, hemos indicado algunos de los patrones que se utilizan para refactorizar el código. Esta labor se puede realizar de forma manual, pero supone una pérdida de tiempo, y podemos inducir a redundancias o a errores en el código que modificamos.
A continuación, vamos a usar los patrones más comunes de refactorización, usando las herramientas de ayuda del entorno.

Podemos seleccionar diferentes elementos para mostrar su menú de refactorización (una clase, una variable, método, bloque de instrucciones, expresión, etc.). A continuación, se muestras algunos de los métodos más comunes:

1. Rename: Es la opción empleada para cambiar el identificador a cualquier elemento (nombre de variable, clase, método, paquete, directorio, etc). Cuando lo aplicamos, se cambian todas las veces que aparece dicho identificador.

2. Extracting: Es la opción que nos permite coger un fragmento de código y moverlo. Podemos extraer una variable, una constante, una clase, un método …

   - Extract Interface: Este patrón de refactorización nos permite seleccionar los métodos de una clase para crear una interface. Una Interfaz es una plantilla que define los métodos acerca de lo que puede hacer una clase. Define los métodos de una clase (Nombre, parámetros y tipo de retorno) pero no los desarrolla.

   - Extract Superclass: Permite crear una superclase (clase padre) con los métodos y atributos que seleccionemos de una clase concreta. Lo usamos cuando la clase con la que trabajamos podría tener cosas en común con otras clases, las cuales serían también subclases de la superclase creada.

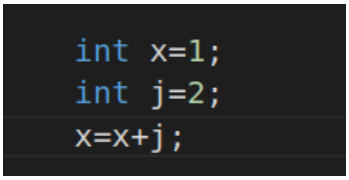
3. Change a method signature: con esta opción puedo cambiar la definición de un método (parámetros de entrada, de salida, nombre)

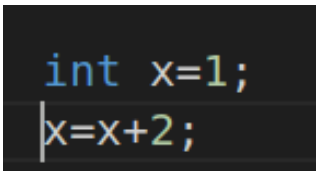
4. Move: Mueve una clase (fichero .java) de un paquete a otro y se cambian todas las referencias. También se realiza la misma operación arrastrando la clase de un paquete a otro en el explorador

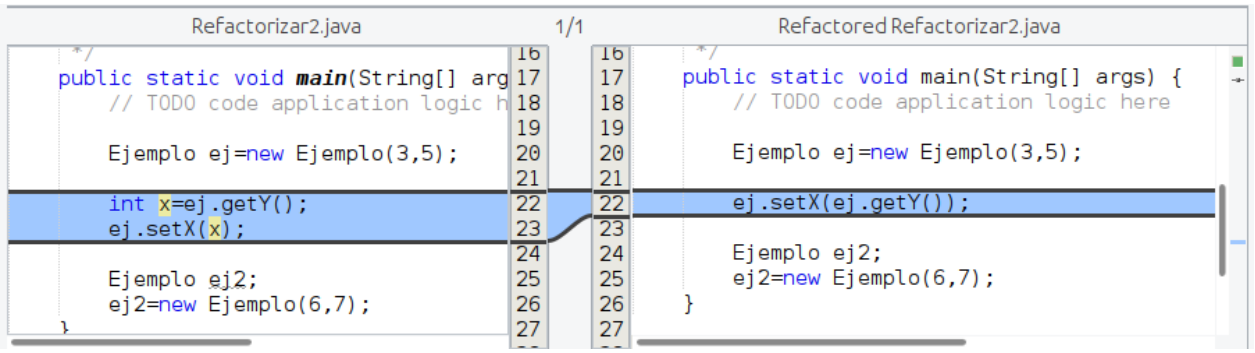
5. Copy: copia una clase (o fichero) de un paquete a otro cambiando todas las referencias.

6. Safely Delete: borra una clase (o fichero) buscando las referencias a la misma en otras clases. Avisa de que hay que eliminarlas, si no lo hacemos y seguimos adelante borrará la clase y tendremos errores sintácticos. Se puede utilizar también para borrar variables

7. Inline**:** Nos permite ajustar una referencia a una variable o método en una sola linea de código.







8. Encapsular campos. Se puede automáticamente generar métodos getter y setter para un campo, y opcionalmente actualizar todas las referencias al código para acceder al campo, usando los métodos getter y setter.

9. Convert a local variable in field.

10. Pull up, down: para mover métodos en una jerarquía de clases.

Refactorización y pruebas

En la actualidad, la refactorización y las pruebas son dos aspectos del desarrollo de aplicaciones que se han convertido en conceptos de gran importancia en la ingeniería del software.

Uno de los enfoques que han surgido, que pretende integrar las pruebas y la refactorización, es el Desarrollo Guiado por Pruebas (TDD, Test Driven Development).

Con el Desarrollo Guiado por Pruebas (TDD), se propone agilizar el ciclo de escritura de código, y realización de pruebas de unidad. Cabe recordar, que el objetivo de las pruebas de unidad es comprobar la calidad de un módulo desarrollado. Existen utilidades que permiten realizar esta labor, pudiendo ser personas distintas a las que los programan, quienes los realicen. Esto provoca cierta competencia entre los programadores de la unidad, y quienes tienen que realizar la prueba. El proceso de prueba supone siempre un gasto de tiempo importante, ya que el programador realiza revisiones y depuraciones de este antes de enviarlo a prueba. Durante el proceso de pruebas hay que diseñar los casos de prueba y comprobar que la unidad realiza correctamente su función. Si se encuentran errores, éstos se documentan, y son enviados al programador para que lo subsane, con lo que debe volver a sumergirse en un código que ya había abandonado.

Con el Desarrollo Guiado por Pruebas, la propuesta que se hace es totalmente diferente. El programador realiza las pruebas de unidad en su propio código, e implementa esas pruebas antes de escribir el código a ser probado.

Cuando un programador recibe el requerimiento para implementar una parte del sistema, empieza por pensar el tipo de pruebas que va a tener que pasar la unidad que debe elaborar, para que sea correcta. Cuando ya tiene claro la prueba que debe de pasar, pasa a programar las

pruebas que debe pasar el código que debe de programar, no la unidad en sí. Cuando se han implementado las pruebas, se comienza a implementar la unidad, con el objeto de poder pasar las pruebas que diseñó previamente.

Cuando el programador empieza a desarrollar el código que se le ha encomendado, va elaborando pequeñas versiones que puedan ser compiladas y pasen por alguna de las pruebas. Cuando se hace un cambio y vuelve a compilar también ejecuta las pruebas de unidad. Y trata de que su programa vaya pasando más y más pruebas hasta que no falle en ninguna, que es cuando lo considera listo para ser integrado con el resto del sistema.
Para realizar la refactorización siguiendo TDD, se refactoriza el código tan pronto como pasa las pruebas para eliminar la redundancia y hacerlo más claro. Existe el riesgo de que se cometan errores durante la tarea de refactorización, que se traduzcan en cambios de funcionalidad y, en definitiva, en que la unidad deje de pasar las pruebas. Tratándose de reescrituras puramente sintácticas, no es necesario correr ese riesgo: las decisiones deben ser tomadas por un humano, pero los detalles pueden quedar a cargo de un programa que los trate automáticamente.