TEMA 12.
REFACTORIZAR

ED
CFGS DAW

Francisco Aldarias Raya
paco.aldarias@ceedcv.es
2016/2017
Versión:170505.1259

ÍNDICE DE CONTENIDO

UD012. REFACTORIZAR

1. REFACTORIZAR

1.1 DEFINICIÓN

Programar código no únicamente es buscar funcionalidad, sino también limpieza y elegancia de escritura. Muchas veces se programa comprobando resultados, sin importar si el código está enmarañado. Esto tiene varias problemáticas: es más difícil hacer una actualización o corrección, pasarlo a otra persona implica que ha de perder tiempo intentado leerlo, y al cliente no se debe presentar con este aspecto.

Así, aunque debería irse programando ya con una estructura inicial y buscando la limpieza de código, una vez terminado siempre se pasa a la fase de Refactorización.
En ella le damos “pintura y chapa” a nuestro código. Es decir, cambiamos la estructura interna de nuestro software para hacerlo más fácil de comprender, de modificar, más limpio y sin cambiar su comportamiento observable.

Por tanto, refactorizar es mejorar el código fuente sin cambiar el resultado del programa. Nuestro objetivo es mantener el código sencillo y bien estructurado.

1.2 ¿POR QUÉ Y CUÁNDO REFACTORIZAR?

Conforme se modifica, el software pierde su estructura. Así, refactorizamos por varios motivos:

- Para eliminar código duplicado simplificar su mantenimiento.
- Para hacerlo más fácil de entender (por ejemplo la legibilidad del código facilita su mantenimiento)
- Para encontrar errores (por ejemplo al reorganizar un programa, se pueden apreciar con mayor facilidad las suposiciones que hayamos podido hacer)
- Para programar más rápido: al mejorar el diseño del código, mejorar su legibilidad y reducir los errores que se cometen al programar, se mejora la productividad de los programadores.

Con esto, conseguimos las siguientes ventajas:

- Calidad. Código sencillo y bien estructurado, legible y entendible sin necesidad de haber

estado integrado en el equipo de desarrollo durante varios meses.

- Eficiencia. Mantener un buen diseño y un código estructurado es sin duda la forma más eficiente de desarrollar. El esfuerzo invertido en evitar la duplicación de código y en simplificar el diseño se ve compensado a la hora de las modificaciones
- Evitar la reescritura de código. 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 (ahorro de costos).

Sin embargo, todo cambio en un código que funciona implica un riesgo. ¿Cuándo debemos refactorizar? cuando se identifique código mal estructurado o diseños que supongan un riesgo para la futura evolución del sistema.

Si vemos que el diseño se vuelve complicado y difícil de entender, y que cada cambio empieza a ser muy costoso, lo mejor es parar de desarrollar y analizar si el rumbo tomado es correcto.

Los síntomas que indican que algún código de software tiene problemas se conocen como “Bad Smells” (Código Duplicado, Métodos largos, Clases Largas, Cláusulas Switch, Comentarios, etc.). Se debe aplicar una refactorización que permita corregir ese problema.

Es imprescindible que el proyecto tenga pruebas automáticas, que nos permitan saber en cualquier momento al ejecutarlas, si el desarrollo sigue cumpliendo los requisitos que implementaba. Sin ellas, refactorizar conlleva un alto riesgo.

1.3 DIFICULTADES EN LA REFACTORIZACIÓN

No siempre resulta evidente para todos los desarrolladores la necesidad de una refactorización, por tanto el uso de herramientas (javastyle, jcsc, pdm, findbug) pueden servir de ayuda para identificar bad smells además de unificar criterios entre todo el equipo. Pero siempre debemos hacer caso al sentido común.

Si se adoptan posturas excesivamente exigentes o criterios excesivamente personales respecto a la calidad del código 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.

Con todo ello, para aplicar esta etapa a nuestro desarrollo debemos tener en cuenta los siguientes puntos:

- Refactorizar no está considerado como avance del proyecto para los clientes

- Refactorizar es una forma de mejorar la calidad. Bien realizada, rápida y segura, genera satisfacción.
- Es importante 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.
- Refactorizar demasiado implica:
  - Pérdida de tiempo
  - Incremento de complejidad de diseño de tanto refactorizar
  - Sensación de no estar avanzando (sensaciones anímicas negativas)
- Un equipo de desarrollo debe ser uno: Conocer el estado actual, los problemas que tenemos y el objetivo que se busca (una reunión diaria facilita la comunicación)
- Si se detectan problemas de comunicación se recomienda detener el desarrollo y reunir a todo el equipo para intentar alcanzar un objetivo común.

2. TIPOS DE REFACTORIZACIÓN

2.1 REFACTORIZACIÓN A POSTERIORI

A veces se da la situación de que 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. Otra situación puede ser que se haya 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. Esto obliga a que tiempo después de implementar una funcionalidad existente se hagan cambios estructurales a posteriori.

Dichos cambios afectan a todos aquellos que trabajan con la parte del código afectada. Así, hay que intentar minimizar los cambios. Para ello, se recomienda dividir la refactorización en el mayor número de pequeños pasos posible (comunicar a las personas afectadas los cambios, para que no se encuentren con un panorama desconocido)

La refactorización a posteriori afecta menos al ritmo normal de desarrollo y permite incorporar nuevas funcionalidades, con el inconveniente de extenderse en el tiempo código imperfecto.

Si una refactorización se complica y conduce a nuevas refactorizaciones, puede que llegue un momento en que no se sepa estimar cuánto tiempo queda para terminar de refactorizar. Esto se conoce como espiral refactorizadora. Debe evitarse, para ello, hay que:

- Plantear claramente el objetivo refactorización antes de empezar
- Si aparece nuevo código susceptible de refactorizar, solo se anota
- Emplear el control de versiones

3. IMPLEMENTACIÓN DE LA REFACTORIZACIÓN

Las siguientes:

- Escribir pruebas unitarias y funcionales, si no se hace así demasiado costoso y mucho riesgo.
- Usar herramientas especializadas. Con herramientas especializadas estas refactorizaciones se realizan automáticamente y sin riesgo.
- Dar formación sobre patrones de refactorización y de diseño. Permite detectar “Bad Smells” de los que no es consciente y que harán que su código se degrade progresivamente.
- Refactorizar los principales fallos de diseño (por ejemplo, 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. 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 refactorización como una tarea más dentro del su proceso de desarrollo de Software.

3.1 REFACTORIZACIÓN Y PRUEBAS UNITARIAS

La existencia de pruebas automatizadas facilita enormemente las refactorizaciones:

- El riesgo de la refactorización disminuye, dado que es posible comprobar cuando esta 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 refactorizaciones más sencillas y realizadas con herramientas especializadas.

Las refactorizaciones grandes, obligarán a realizar cambios en las pruebas. La mejor solución es prevenir aplicando refactorizaciones continuas y más pequeñas.

Hay que 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.

3.2 PATRONES DE REFACTORIZACIÓN MÁS HABITUALES

Los patrones más habituales son:

1. 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.

2. Long method (método largo). Cuanto más corto es un método más fácil de reutilizarlo es.

3. 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.

4. 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.

5. Divergent change (cambio divergente): una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre sí.

6. Shotgun surgery (Cirugía de escopeta): 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.

7. 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.

8. Switch: Lo tipico de un software Orientado a Objetos es el uso mínimo de los “switch”(o case, o switch escondido). El problema es la duplicación, el mismo “switch” en lugares diferentes

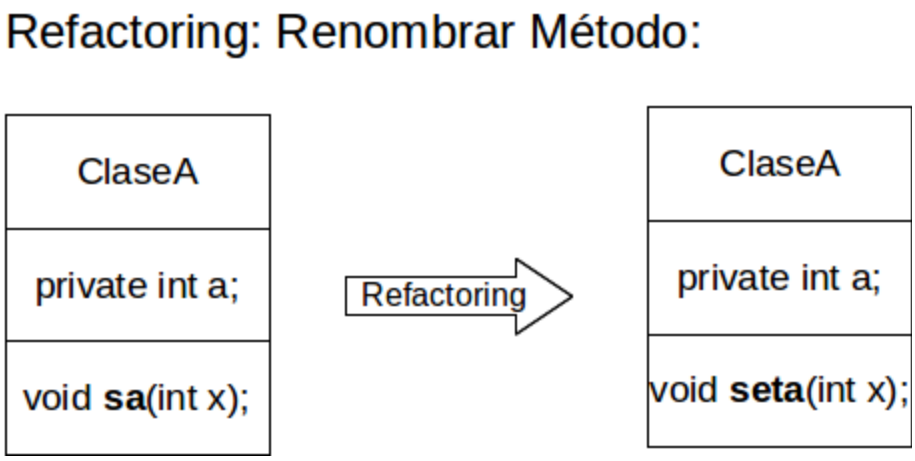
9. Data class (clase de datos): Clases que sólo tienen atributos y métodos de acceso a ellos (“get” y “set”). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.

10. 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.

3.3 MÉTODOS DE REFACTORIZACIÓN

3.3.1 Cambiar de nombre

Se debe refactorizar cuando el nombre no revela que realiza. También llamado rename method. Se puede realizar sobre variables, métodos, comentarios, etc.



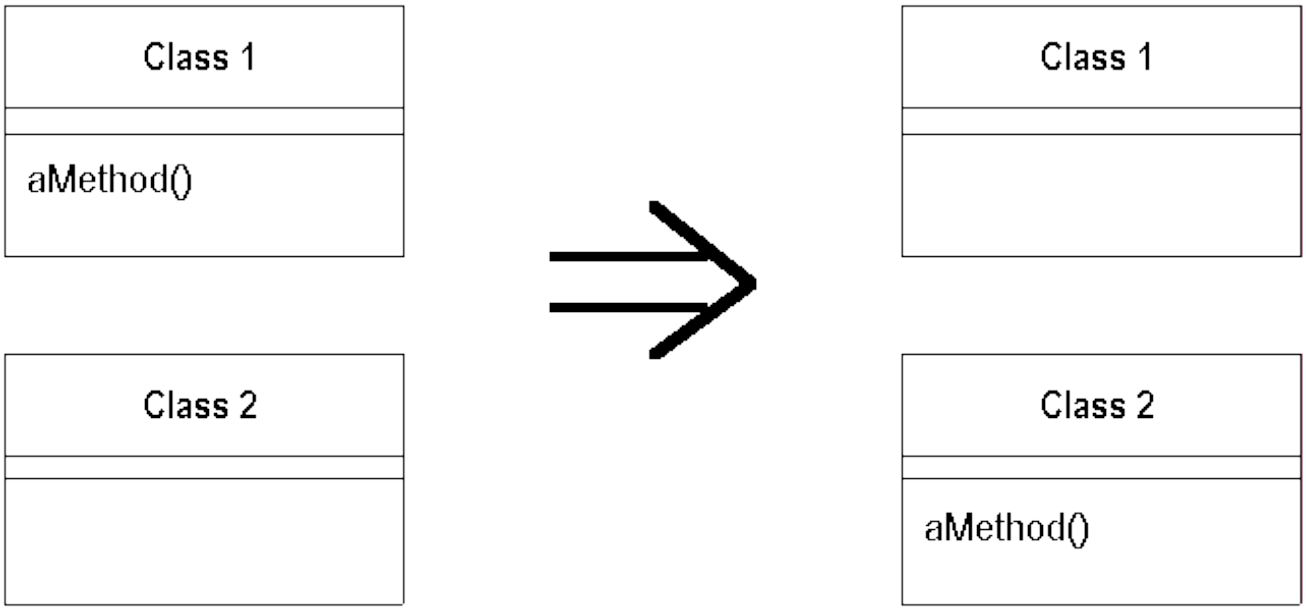
3.3.2 Mover

Un método de una clase será movido a otra otra clase que lo usa más. También llamado Move Method.

Ejemplo:

Código sin refactorizar	Código refactorizado

class **Project** {
Person [] participants ;
}
 	
class **Person** {
int id;
boolean **participate** (Project p) {
for (int i =0; i<p. participants . length ; i ++) {
if (p. participants [i]. id == id) return (true);
}
return (false);
}
}
... if (x. participate (p)) ...
	class **Project** {
	Person [] participants ;
	boolean **participate** (Person x) {
	for (int i =0; i< participants . length ; i ++) {
	if (participants [i]. id == x.id) return (true);
	}
	return (false);
	}
	}
	class **Person** {
	int id;
	}
	... if (p. participate (x)) ...



3.3.3 Introducir método

Permite introducir un bloque de código en un método. También llamado Extract Method.

Ejemplo:

Código sin refactorizar	Código refactorizado

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

Existen muchos más tipos que se puede ver en la página web Refactorings in Alphabetical Order que pertenece al libro de Martin Fowler: http://www.refactoring.com/catalog/index.html

4. EJEMPLOS DE REFACTORIZACIÓN

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.

5. CASO PRÁCTICO DE REFACTORIZACIÓN

Imaginemos que tenemos una aplicación bancaria que usa extensivamente una clase llamada Transacción, la cual posee una operación cuya firma es:

```
public void asignarCtaCorriente(CuentaCorriente cc);
```

Esta operación es invocada una gran cantidad de veces por distintas operaciones en nuestra aplicación. El programa es que se debe cambiar la firma de la operación para homogenizar el estándar de nomenclatura establecido por la empresa que desarrolló la aplicación.

La nueva firma debería ser:

```
public void asignarCuentaCorriente(CuentaCorriente cc);
```

¿Cuál es la solución?

Debemos iniciar el proceso de refactorización en nuestra aplicación para reflejar este cambio. Modificaremos la firma en la clase Transacción y luego modificaremos en todo el proyecto cada una de las invocaciones. Si la aplicación es realmente grande, implicará un gran esfuerzo llevar a cabo este cambio manualmente.

Probablemente se introduzcan errores.

A medida que se desarrolla una aplicación con NetBeans, de forma transparente para el programador se genera un repositorio de metadata del código que el programador genera. El metadata se podría definir brevemente como “datos que describen otros datos”, por lo cual NetBeans puede contar con esta información para realizar cambios sobre los fuentes automáticamente. Gracias a este repositorio de metadata que NetBeans mantiene es posible automatizar muchas tareas de refactorización, por ejemplo, en el caso que describimos anteriormente, NetBeans se encarga de cambiar la firma de la operación de la clase Cuenta, buscar todas las invocaciones y modificarlas. Adicionalmente NetBeans provee un informe del impacto que tendrá el proceso de refactoring, para permitir al programador tomar decisiones respecto a este proceso. Luego de desplegado dicho informe se puede iniciar el proceso de refactoring de acuerdo a las decisiones tomadas por el programador.

6. REFACTORIZAR CON NETBEANS

En Netbeans al refactorizar le llama reestructurar. Se puede obtener:

- Menú Reestructurar
- Menú Contextual (botón derecho del ratón sobre el texto marcado)

Siempre que sea posible, se han de utilizar las herramientas de refactorización que NetBeans provee, de este modo no sólo será mucho más sencillo este tipo de procedimiento, sino que además ejecutaremos un proceso seguro mediante el cual no introduciremos errores humanos.

Para saber más, véase la práctica 1 de este tema (solucionada).

7. BIBLIOGRAFÍA Y ENLACES

- Critina Alvarez. (2015): “Entornos de desarrollo”, CEEDCV
- Amescua, A.; Cuadrado, JJ; Ernica, E. (2003): Análisis y Diseño Estructurado y
- Orientado a Objetos de Sistemas Informáticos, Mcgraw-hill
- Casado, C. (2012): Entornos de desarrollo, RA-MA, Madrid
- Refactorización con Netbeans. http://cnx.org/content/m17586/latest/
- Refactorización: http://refactoring.com/
- Catálogo de refactorización: http://www.refactoring.com/catalog/index.htm