PRINCIPIOS DE DESARROLLO DE SOFTWARE

2019-2020

Universidad Carlos III de Madrid

Ejercicio Guiado - 4

2019/2020

REFACTORING Y DISEÑO SIMPLE

Ejercicio Guiado 4

Universidad Carlos III de Madrid. Escuela Politécnica Superior

Javier García Guzmán

Sección 1

Objetivos

Este ejercicio guiado tiene como objetivos:

1) Poner en práctica la técnica de refactoring.
2) Poner en práctica los principios de diseño simple y patrones de diseño básico.

Caso Práctico

Como punto de partida de este ejercicio guiado se proporcionará el código fuente que implementa satisfactoriamente las funcionalidades presentadas en el ejercicio guiado 3, por tanto, como primer paso, será necesario asegurar que los casos de prueba definidos permiten verificar satisfactoriamente el código fuente. Es muy importante tener en cuenta que el código proporcionado implementa las funcionalidades descritas en el enunciado del ejercicio guiado 3 sin adaptaciones que se hayan podido dar durante las clases de prácticas.

En primer lugar, se deberá reorganizar el código de tal manera que se puedan resolver distintas situaciones predefinidas de refactoring que se han introducido intencionadamente en el código para facilitar el aprendizaje de esta técnica. Los casos propuestos para la aplicación de esta técnica se proporcionan en la sección 2.

Posteriormente, se abordarán diversas mejoras de la arquitectura técnica propuesta para la solución del ejercicio guiado 3 que permiten aplicar los principios fundamentales de diseño simple y algunos patrones de diseño básicos. Las recomendaciones de aplicación de patrones se describen en el capítulo 3 de este enunciado.

Sección 2

Refactoring

Se han identificado las siguientes situaciones susceptibles de reorganización en el código fuente del componente de Token Management de la compañía Transport4Future. Las acciones de reorganización de código se deberán realizar en el orden propuesto a continuación y al finalizar cada etapa se deberá comprobar que las pruebas siguen funcionando correctamente (para ello podría ser necesario actualizar el código de dichas pruebas).

Paso 2.1: Identificación y resolución de las situaciones de refactoring.

Las situaciones de refactoring que están incluidas intencionadamente en el código son las siguientes:

a) Nombres misteriosos. Escribir texto desconcertante es apropiado cuando se está escribiendo o leyendo una novela de detectives, pero no cuando estás leyendo un código. Una de las partes más importantes del código claro son los buenos nombres, por lo que pensamos mucho en las funciones de nomenclatura, módulos, variables, clases, para que comuniquen claramente lo que hacen y cómo usarlos.

   Las refactorizaciones más comunes en este caso son los renombrados: cambiar la declaración de la función (124)[1] (para renombrar una función), Renombrar variable (137) y Renombrar campo (244).

b) Si ve la misma estructura de código en más de un lugar, puede estar seguro de que su programa será mejor si encuentra la manera de unificarlos. La duplicación significa que

[1] Los números hacen referencia al número de página del libro "Refactoring: Improving the existing code" donde se describe la refactorización a la que se hace referencia.

cada vez que lea estas copias, deberá leerlas cuidadosamente para ver si hay alguna diferencia. Si necesita cambiar el código duplicado, debe encontrar y capturar cada duplicación.

El problema de código duplicado más simple es cuando tiene la misma expresión en dos métodos de la misma clase. Entonces todo lo que tiene que hacer es Extraer Función (106) e invocar el código desde ambos lugares. Si tiene un código similar, pero no del todo idéntico, vea si puede usar Slide Statements (223) para organizar el código de modo que los elementos similares estén todos juntos para una extracción fácil. Si los fragmentos duplicados están en subclases de una clase base común, puede usar el Método Pull Up (350) para evitar llamar a uno de otro.

c) Funciones Largas. En nuestra experiencia, los programas que duran más tiempo y tienen menos problemas son aquellos con funciones cortas.

   Desde los primeros días de la programación, las personas se han dado cuenta de que cuanto más larga es una función, más difícil es entenderla.

   El noventa y nueve por ciento de las veces, todo lo que tiene que hacer para acortar una función es Extraer Función (106). Encuentra partes de la función que parecen ir bien juntas y crea una nueva.

d) Cambios Divergentes. Estructuramos nuestro software para facilitar el cambio; después de todo, el software está destinado a ser adaptable sin problemas. Cuando hacemos un cambio, queremos poder saltar a un punto claro en el sistema y hacer el cambio. Cuando no puedes hacer esto, estás oliendo una de dos picaduras estrechamente relacionadas.

   El cambio divergente ocurre cuando un módulo se cambia a menudo de diferentes maneras por diferentes razones. Si observa un módulo y dice: "Bueno, tendré que cambiar estas tres funciones cada vez que obtenga una nueva base de datos; Tengo que cambiar estas cuatro funciones cada vez que hay un nuevo instrumento financiero ", esto es una indicación de un cambio divergente.

   Si los dos aspectos forman naturalmente una secuencia, por ejemplo, obtiene datos de la base de datos y luego aplica su procesamiento financiero en ella, entonces Split Phase (154) separa los dos con una clara estructura de datos entre ellos. Si hay más de ida y

vuelta en las llamadas, cree los módulos apropiados y use la función de Movement (198) para dividir el procesamiento. Si las funciones combinan los dos tipos de procesamiento dentro de sí mismas, utilice Extraer Función(106) para separarlas antes de moverlas. Si los módulos son clases, entonces Extraer Clase (182) ayuda a formalizar cómo hacer la división.

e) Clases grandes. Cuando una clase está tratando de hacer demasiado, a menudo aparece como demasiados campos y funciones. Cuando una clase tiene demasiados campos, el código duplicado no puede estar muy lejos.

   Puede Extraer la Clase (182) para agrupar un número de las variables. Elija variables para ir juntas en el componente que tenga sentido para cada una.

   Si en la situación la herencia tiene sentido, encontrará Extraer Superclase (375) o Reemplazar código de tipo con subclases (362) (que esencialmente es extraer una subclase) a menudo son más fáciles.

f) Comentarios. La razón por la que mencionamos los comentarios aquí es porque con alta frecuencia ocurre que se encuentra código con comentarios abundantes y te das cuenta de que los comentarios están ahí porque el código es malo. Los comentarios nos llevan a un código erróneo que tiene todos los defectos negativos que deben conducir a refactoring. Cuando terminamos, a menudo encontramos que los comentarios son superfluos.

   Si necesita un comentario para explicar lo que hace un bloque de código, intente Extraer función (106). Si el método ya está extraído pero aún necesita un comentario para explicar lo que hace, use Cambiar declaración de función (124) para cambiar el nombre. Si necesita establecer algunas reglas sobre el estado requerido del sistema, use Presentar aserción (302).

Paso 2.2: Actualización del código según la normativa de código.

Por último, se deberán realizar las actualizaciones correspondientes para que el código fuente incluya convenientemente las reglas y recomendaciones definidas en la normativa de código por el grupo de prácticas.

Paso 2.3: Ejecución de los casos de prueba definidos.

Finalmente se debe verificar que el resultado del refactoring del servidor es satisfactorio y cumple las expectativas marcadas. Con este propósito se deberán volver a ejecutar las pruebas creadas para verificar el correcto funcionamiento del componente TokenManagement.

Es necesario considerar que este paso se debe ejecutar inmediatamente después de la finalización de cada una de las tareas de reorganización de código del servidor establecidas para este ejercicio guiado.

Es posible, que las propias tareas de reorganización de código supongan la actualización o la definición e implementación de nuevos casos prueba. En caso de que esto sea necesario, esta tarea de actualización de casos de prueba se deberá realizar antes de probar la funcionalidad del componente Token Management de la compañía Transport4Future.

Paso 2.4: Registro del código funcional y de pruebas en Git

Como la resolución del ejercicio guiado supondrá diversas sesiones de trabajo, cada sesión debe finalizar con la correcta implementación de un caso de prueba (el que corresponda en cada momento).

Al finalizar la sesión de trabajo se deberá realizar en GIT la ejecución los comandos *Push* y *Commit* para el correcto registro del trabajo realizado, de acuerdo con las instrucciones proporcionadas en el Ejercicio Guiado 2.

Es necesario recordar que también hay que registrar en GIT el registro en XML que evidencia la correcta ejecución de los casos de prueba definidos en el momento de registrar el código en el sistema de control de versiones.

Sección 3

Diseño Simple

En esta sección se abordarán diversas mejoras de la arquitectura técnica propuesta para la solución del ejercicio guiado 3 que permiten aplicar los principios fundamentales de diseño simple y algunos patrones de diseño básicos.

Las mejoras establecidas para la arquitectura del componente diseñado una vez se ha aplicado la técnica de refactoring son las siguientes:

1) Solamente se podrá ejecutar una instancia de las clases TokenManager, TokenRequestsStore y TokensStore. Para ello se abordará la aplicación del patrón Singleton.
2) Debido a que el componente requiere que se lean distintos tipos de ficheros JSON que se deben procesar de manera similar, pero tienen que procesarse de manera ligeramente diferente, se deberá adoptar un patrón Strategy para resolver apropiadamente esta situación. Asimismo, se implementará este patrón para mejorar el diseño del código necesario para generar códigos hash ya sea con MD5 o SHA256.

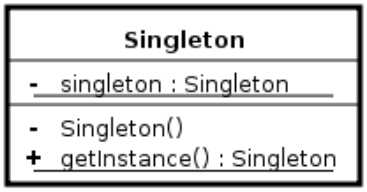
Paso 3.1: Aplicación del patrón Singleton para asegurar que solo se puede generar una instancia de una clase

En ingeniería de software, singleton o instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto solo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada

nuevamente se regula el alcance del constructor (con modificadores de acceso como protegido o privado).



A continuación, se muestra un ejemplo del código en Java que es necesario para implementar el patrón singleton[2]:

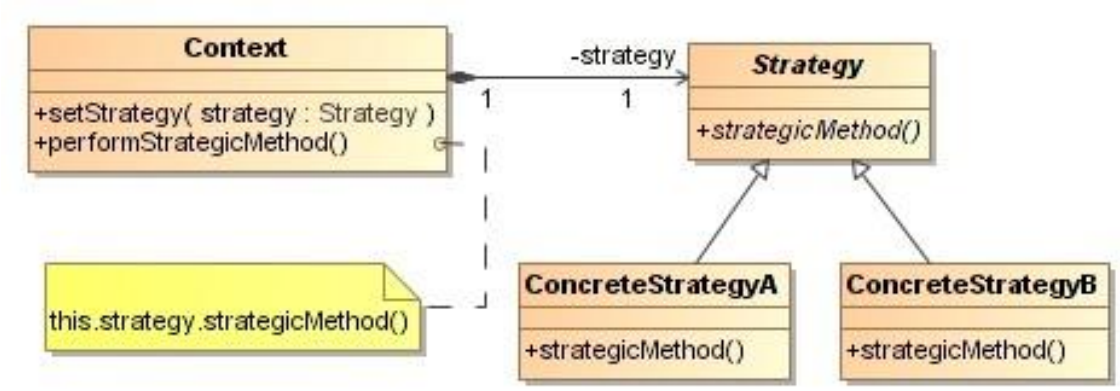
```
public class SoyUnico {

    private String nombre;

    private static SoyUnico soyUnico;

    // El constructor es privado, no permite que se genere un constructor por defecto.

    private SoyUnico(String nombre) {

        this.nombre = nombre;

        System.out.println("Mi nombre es: " + this.nombre);

    }

    public static SoyUnico getSingletonInstance(String nombre) {

        if (soyUnico == null){

            soyUnico = new SoyUnico(nombre);

        }

        else{
```

[2] Ejemplo obtenido de https://jarroba.com/patron-singleton-en-java-con-ejemplos/

```
            System.out.println("No se puede crear el objeto "+ nombre + " porque ya existe un objet
o de la clase SoyUnico");

        }

        return soyUnico;

    }

}

//Sobreescribimos el método clone, para que no se pueda clonar un objeto de esta clase

@Override

public SoyUnico clone(){

    try {

        throw new CloneNotSupportedException();

    } catch (CloneNotSupportedException ex) {

        System.out.println("No se puede clonar un objeto de la clase SoyUnico");

    }

    return null;

}
```

Paso 3.2: Aplicación del patrón Strategy para la lectura de ficheros JSON

El patrón Estrategia (Strategy) es un patrón de diseño para el desarrollo de software. Se clasifica como patrón de comportamiento porque determina cómo se debe realizar el intercambio de mensajes entre diferentes objetos para resolver una tarea. El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.



A continuación, se muestra un ejemplo del código en Java que es necesario para implementar el patrón strategy[3]:

```
public class Main {
	public static void main(String args[])
	{
		//Usamos la estrategia A
		Strategy estrategia_inicial = new StrategyA();
		Context context = new Context(estrategia_inicial);
		context.some_method();

		//Decidimos usar la estrategia B
		Strategy estrategia2 = new StrategyB();
		context.setStrategy(estrategia2);
		context.some_method();

		//Finalmente,usamos de nuevo la estrategia A
		context.setStrategy(estrategia_inicial);
		context.some_method();
	}
}
public class Context {
	Strategy c;
	public Context( Strategy c )
	{
		this.c = c;
```

[3] Ejemplo obtenido de https://es.wikipedia.org/wiki/Strategy_(patr%C3%B3n_de_dise%C3%B1o)#Java

```
        }

        public void setStrategy(Strategy c) {
                this.c = c;
        }

        //Método de estrategia 'c'
        public void some_method()
        {
                c.behaviour();
        }
}

public Interface Strategy{
       public void behaviour();
}

public class StrategyA implements Strategy{
        @Override
        public void behaviour() {
                System.out.println("Estrategia A");
        }
}

public class StrategyB implements Strategy{
        @Override
        public void behaviour() {
                System.out.println("Estrategia B");
        }
}
```

Paso 3.3: Registro del código funcional y de pruebas en Git

Como la resolución del ejercicio guiado supondrá diversas sesiones de trabajo, cada sesión debe finalizar con la correcta implementación de un caso de prueba (el que corresponda en cada momento).

Al finalizar la sesión de trabajo se deberá realizar en GIT la ejecución los comandos *Push* y *Commit* para el correcto registro del trabajo realizado, de acuerdo con las instrucciones proporcionadas en el Ejercicio Guiado 2.

Es necesario recordar que también hay que registrar en GIT el registro en XML que evidencia la correcta ejecución de los casos de prueba definidos en el momento de registrar el código en el sistema de control de versiones.

Sección 4

Reglas y Procedimientos

Entrega

Este ejercicio se completará en parejas.

1) Se debe registrar en el repositorio de Git que el profesor de prácticas indique para este ejercicio guiado.
   a) El código funcional que los alumnos han generado durante el proceso de desarrollo dirigido por pruebas considerado en el ejercicio guiado.
   b) El código de prueba en JUNIT para la automatización de los casos de prueba considerados.
   c) Un informe generado en XML por JUnit que incluya los resultados de la ejecución de los casos incluidos para cada una de las clases de prueba que contienen los casos considerados para cada una de las funcionalidades y técnicas consideradas.

La fecha límite para la entrega del ejercicio es 15/05/2020 antes de las 23:59.

De acuerdo con las normas de evaluación continua en esta asignatura, si un estudiante no envía la solución del ejercicio antes de la fecha límite, el ejercicio será evaluado con 0 puntos.

Sugerencias

Cada estudiante debe guardar una copia de la solución entregada hasta la publicación de las calificaciones finales de la asignatura.