Taller Refactoring

Paralelo 3

Profesor: David Jurado

GRUPO 6:

Aaron Franco

Fabrizzio Ontaneda

Javier Vega

Lizbeth Vergara

Willy Mateo

Contenido

Inappropiate Intimacy.

Código inicial.

```java
//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y taller
public double CalcularNotaTotal(Paralelo p){
    double notaTotal=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            notaTotal=(p.getMateria().notaInicial+p.getMateria().notaFinal)/2;

        }
    }
    return notaTotal;

}
```

```java
public class calcularSueldoProfesor {

    public double calcularSueldo(Profesor prof){
        double sueldo=0;
        sueldo= prof.info.añosdeTrabajo*600 + prof.info.BonoFijo;
        return sueldo;
    }
}
```

```java
public class Paralelo {
    public int numero;
    public Materia materia;
    public Profesor profesor;
    public ArrayList<Estudiante> estudiantes;
    public Ayudante ayudante;
```

```java
package modelos;

import java.util.ArrayList;

public class Profesor {
    public String codigo;
    public String nombre;
    public String apellido;
    public int edad;
    public String direccion;
    public String telefono;
    public InformacionAdicionalProfesor info;
    public ArrayList<Paralelo> paralelos;
```

```java
public class Materia {
    public String codigo;
    public String nombre;
    public String facultad;
    public double notaInicial;
    public double notaFinal;
    public double notaTotal;
```

Consecuencias.

En el método ***calcularNotaTotal()*** de la clase ***Estudiante*** se accede de forma directa a ***notaInicial*** y ***notaFinal*** de la clase ***Materia*** o en el **método calcularSueldo()** de la clase ***CalcularSueldoProfesor*** se accede de manera directa a el campo ***info*** y esta a su vez a ***añosdeTrabajo*** y ***BonoFijo***. Esto se debe a que, la mayoría de las clases constan con campos con modificadores de acceso público o default, algunas también poseen ***getters*** y ***setters***, sin embargo, esto no impide que se acceda y modifique su contenido de manera directa. Resultará en problemas mayores pues en un futuro se pueden alterar o perder los datos de manera inconsciente y sin ningún tipo de seguridad.

Técnicas de refactorización.

La principal técnica de refactorización utilizada fue ***Extract superclass*** para extraer todos los campos de información adicional a un Profesor para generar una estructura de herencia precisa, seguido de ***Pull up method*** para fusionar los métodos de notas finales e iniciales en la clase ***Notas***.

Código refactorizado.

```java
public class Notas {
    private Paralelo paralelo;
    private Calificacion notaInicial;
    private Calificacion notaFinal;

    public Notas(Paralelo paralelo) {
        this.paralelo = paralelo;
    }

    public Notas(Paralelo paralelo, Calificacion notaInicial, Calificacion notaFinal) {
        this(paralelo);
        this.notaInicial = notaInicial;
        this.notaFinal = notaFinal;
    }

    public double CalcularNotaTotal(){
        return (notaInicial.CalcularNota() + notaFinal.CalcularNota())/2;
    }
```

Message Chains.

Código inicial.

```
public class calcularSueldoProfesor {

    public double calcularSueldo(Profesor prof){
        double sueldo=0;
        sueldo= prof.getInfo().getAñosdeTrabajo()*600 + prof.getInfo().getBonoFijo();
        return sueldo;
    }
}
```

```
//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. Est
public double CalcularNotaTotal(Paralelo p){
    double notaTotal=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            notaTotal=(p.getMateria().getNotaInicial()+p.getMateria().getNotaFinal())/2;

        }
    }
    return notaTotal;
```

Consecuencias.

Una cadena de mensajes ocurre cuando la variable sueldo del método **CalcularSueldo(Profesor prof)** trata de realizar una operación en base otro objeto, ese objeto solicita otro, y así sucesivamente, esto también se puede evidenciar en la variable notaTotal del método **CalcularNotaTotal(Paralelo p).** Estas cadenas significan que el cliente depende de la navegación a lo largo de la estructura de clases. Cualquier cambio en estas relaciones requiere modificar al cliente.

Técnicas de refactorización.

Utilizamos la técnica de ***Extract superclass*** en donde se creó una superclase compartida para ellos y mueva todos los campos y métodos idénticos a ella, se usó ***Extract method*** para ello movimos código a un nuevo método separado y reemplazamos el código anterior con una llamada al método y para finalizar se utilizó **Pull up method**. en donde hicimos que los métodos sean idénticos y luego los movimos a la superclase correspondiente.

Código refactorizado.

```
import modelos.usuarios.Profesor;

public class calcularSueldoProfesor {

    public double calcularSueldo(Profesor prof){
        return prof.getAñosdeTrabajo()*600 + prof.getBonoFijo();
    }
}
```

Lazy Class

Código Inicial

```java
package modelos;

public class InformacionAdicionalProfesor {
    public int añosdeTrabajo;
    public String facultad;
    public double BonoFijo;

}
```

Consecuencias

Como se puede observar en la imagen anterior, la clase **InformacionAdicionalProfesor** no realiza acción alguna, es decir que no es responsable de nada, ni siquiera posee métodos que realicen acción alguna, por lo que no hay responsabilidades extras planeadas para ella, lo único que hace es ocupar espacio de código y podría generar confusiones al momento de querer acceder a dichos atributos presentes en la clase.

Técnicas de Refactorización utilizadas

Se utilizará la técnica de refactorización Inline Class, ya que básicamente lo que haremos será mover todos los atributos de una clase a otra, en este caso a **Profesor**, luego se procederá a eliminar la Lazy Class mencionada.

Código Final

```java
package modelos;

import java.util.ArrayList;

public class Profesor {
    private String codigo;
    private String nombre;
    private String apellido;
    private int edad;
    private String direccion;
    private String telefono;
    private ArrayList<Paralelo> paralelos;

    private int añosdeTrabajo;
    private String facultad;
    private double BonoFijo;

    public Profesor(String codigo, String nombre, String apellido, String facultad, int edad, String direccion, String telefono) {
        this.codigo = codigo;
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
        this.direccion = direccion;
        this.telefono = telefono;
        paralelos= new ArrayList<>();
    }

    public void anadirParalelos(Paralelo p){
        paralelos.add(p);
    }
```

Temporary Field

Código Inicial

```
package modelos;

public class calcularSueldoProfesor {

    public double calcularSueldo(Profesor prof){
        double sueldo=0;
        sueldo= prof.info.añosdeTrabajo*600 + prof.info.BonoFijo;
        return sueldo;
    }
}
```

Consecuencias

Usualmente se crea un capo temporal cuando se desean realizar algoritmos complejos que requieren una gran cantidad de inputs, en ciertos casos es razonable usar estos campos temporales, pero en el caso de este programa, los valores usados para hallar el sueldo se los puede obtener fácilmente con los getters de la clase Profesor, y es que en realidad deberían hacerse uso de los mismos porque si no estarían ocupando espacio en el código innecesario, generando un smell de código muerto en caso de que esta lógica de programación se mantenga en todo el código.

Técnicas de Refactorización utilizadas

La técnica de refactorización ideal a utilizarse es Inline Temp, con ello lo que lograremos es eliminar la variable asignada junto con su línea de código designada, y a su vez lo que se hará es retornar de forma directa el valor calculado de sueldo en una sola línea haciendo uso de los getters que se crearon antes en la clase **Profesor**.

Código Final

```
package modelos;

public class calcularSueldoProfesor {

    public double calcularSueldo(Profesor prof){
        return prof.getAñosdeTrabajo()*600 + prof.getBonoFijo();
    }
}
```

Long Parameter List.

Código inicial.

```java
//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. El teorico y el practico se calcula por parcial.
public double CalcularNotaInicial(Paralelo p, double nexamen,double ndeberes, double nlecciones, double ntalleres){
    double notaInicial=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            double notaTeorico=(nexamen+ndeberes+nlecciones)*0.80;
            double notaPractico=(ntalleres)*0.20;
            notaInicial=notaTeorico+notaPractico;
        }
    }
    return notaInicial;
}

//Calcula y devuelve la nota final contando examen, deberes, lecciones y talleres. El teorico y el practico se calcula por parcial.

public double CalcularNotaFinal(Paralelo p, double nexamen,double ndeberes, double nlecciones, double ntalleres){
    double notaFinal=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            double notaTeorico=(nexamen+ndeberes+nlecciones)*0.80;
            double notaPractico=(ntalleres)*0.20;
            notaFinal=notaTeorico+notaPractico;
        }
    }
    return notaFinal;
}
```

Consecuencias

Como se puede evidenciar, mas de 3 o 4 parámetros se encuentran presentes en ambos métodos en la clase **Estudiante,** haciendo que leer estos métodos resulta confuso para el programador al momento de ingresar los parámetros, sumado a esto se ve una falta de estética y legibilidad en el código.

Técnicas de Refactorización utilizadas

La técnica de refactorización mas adecuada para resolver este problema es reemplazar todos los parámetros por un objeto que incluya todos los parámetros necesarios en dicho método, de esta forma estamos haciendo un código más legible y mas entendible para el programador, ya que el único parámetro que se le debe pasar al método es el objeto **Notas** que se creo como se puede evidenciar en el código final.

Código Final

```java
package modelos;

/**
 *
 * @author test1
 */
public class Notas {
    private Paralelo p;
    private double nexamen;
    private double ndeberes;
    private double nlecciones;
    private double ntalleres;

    public Notas(Paralelo p, double nexamen, double ndeberes, double nlecciones, double ntalleres) {
        this.p = p;
        this.nexamen = nexamen;
        this.ndeberes = ndeberes;
        this.nlecciones = nlecciones;
        this.ntalleres = ntalleres;
    }
```

```java
    //Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. El teorico y el practico se calcula por parcial.
    public double CalcularNotaInicial(Notas notas){
        double notaInicial=0;
        for(Paralelo par:paralelos){
            if(notas.getP().equals(par)){
                notaInicial = notaFinal(notas.getNexamen(),notas.getNdeberes(),notas.getNlecciones(),notas.getNtalleres());
            }
        }
        return notaInicial;
    }

    //Calcula y devuelve la nota final contando examen, deberes, lecciones y talleres. El teorico y el practico se calcula por parcial.

    public double CalcularNotaFinal(Notas notas){
        double notaFinal=0;
        for(Paralelo par:paralelos){
            if(notas.getP().equals(par)){
                notaFinal = notaFinal(notas.getNexamen(),notas.getNdeberes(),notas.getNlecciones(),notas.getNtalleres());
            }
        }
        return notaFinal;
    }
```

Duplicate Code

Código Inicial

```
//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. El teorico y el practico se calcula por parcial.
public double CalcularNotaInicial(Paralelo p, double nexamen,double ndeberes, double nlecciones, double ntalleres){
    double notaInicial=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            double notaTeorico=(nexamen+ndeberes+nlecciones)*0.80;
            double notaPractico=(ntalleres)*0.20;
            notaInicial=notaTeorico+notaPractico;
        }
    }
    return notaInicial;
}

//Calcula y devuelve la nota final contando examen, deberes, lecciones y talleres. El teorico y el practico se calcula por parcial.

public double CalcularNotaFinal(Paralelo p, double nexamen,double ndeberes, double nlecciones, double ntalleres){
    double notaFinal=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            double notaTeorico=(nexamen+ndeberes+nlecciones)*0.80;
            double notaPractico=(ntalleres)*0.20;
            notaFinal=notaTeorico+notaPractico;
        }
    }
    return notaFinal;
}
```

Consecuencias

Si en algún momento se debe modificar el código duplicado, es muy probable que no se haya cambiado alguna ocurrencia de la duplicación. También el código duplicado lleva a que el código sea más difícil de comprender.

Técnicas de Refactorización utilizadas

Ya que existe un fragmento de código que se puede agrupar, la técnica de refactorización elegida es **Extract method** se crea un método **Nota** y se reemplaza el código duplicado por el método creado, si en algún momento se desea cambiar el “propósito” de la función solo se modifica el método.

Código Final

```
//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. El teorico y el practico se calcula por parcial.
public double CalcularNotaInicial(Notas notas){
    double notaInicial=0;
    for(Paralelo par:paralelos){
        if(notas.getP().equals(par)){
            notaInicial = notaFinal(notas.getNexamen(),notas.getNdeberes(),notas.getNlecciones(),notas.getNtalleres());
        }
    }
    return notaInicial;
}

//Calcula y devuelve la nota final contando examen, deberes, lecciones y talleres. El teorico y el practico se calcula por parcial.

public double CalcularNotaFinal(Notas notas){
    double notaFinal=0;
    for(Paralelo par:paralelos){
        if(notas.getP().equals(par)){
            notaFinal = notaFinal(notas.getNexamen(),notas.getNdeberes(),notas.getNlecciones(),notas.getNtalleres());
        }
    }
    return notaFinal;
}
```

```
public double notaFinal(double nexamen, double ndeberes, double nlecciones, double ntalleres){
    return ((nexamen+ndeberes+nlecciones)*0.80) + ((ntalleres)*0.20);
}
```

Feature Envy

Código Inicial

```java
package modelos;

import java.util.ArrayList;

public class Ayudante {
    protected Estudiante est;
    public ArrayList<Paralelo> paralelos;

    Ayudante(Estudiante e){
        est = e;
    }
    public String getMatricula() {
        return est.getMatricula();
    }

    public void setMatricula(String matricula) {
        est.setMatricula(matricula);
    }

    //Getters y setters se delegan en objeto estudiante para no duplicar código
    public String getNombre() {
        return est.getNombre();
    }

    public String getApellido() {
        return est.getApellido();
    }
```

Consecuencias

Existe código duplicado, debido a que, las clases Ayudante y Estudiante contiene el mismo código. A su vez, se puede notar que existe alto acoplamiento entre ambas clases, es decir, si se llega a eliminar código en la clase Estudiante puede que este le afecte a la operabilidad de la clase Ayudante.

Técnicas de Refactorización utilizadas

La técnica que se aplicará para resolver este code smell será Reemplazar Delegación con Herencia, ya que a través de la Herencia logramos que nuestro código en la clase Estudiante se reúse sin la necesidad de que haya código duplicado, lo cual a su vez hace que la clase Ayudante sea más pequeña y fácil de mantener en un futuro.

Código Final

```java
package modelos;

import java.util.ArrayList;

public class Ayudante extends Estudiante{
    public ArrayList<Paralelo> paralelos;

    public ArrayList<Paralelo> getParalelos() {
        return paralelos;
    }

    public void setParalelos(ArrayList<Paralelo> paralelos) {
        this.paralelos = paralelos;
    }

    //Los paralelos se añaden/eliminan directamente del Arraylist de paralelos


    //Método para imprimir los paralelos que tiene asignados como ayudante
    public void MostrarParalelos(){
        for(Paralelo par:paralelos){
            //Muestra la info general de cada paralelo
        }
    }

    //Getters y Setters de la herencia
        public String getMatricula() {
        return matricula;
    }
```

modelos.Ayudante > setMatricula >