Escuela Superior Politécnica del Litoral

DISEÑO DE SOFTWARE

TALLER:

Refactoring

Equipo T2

Ricardo Aurelio Rivera Guerra

Michelle Stephania Mujica Meneses

Adriana Lourdes Riofrío Silva

Claudia Sofia Asanza Romero

Xavier Patricio García Baño

PARALELO: 1

PROFESOR: Dr. Carlos Mera

FECHA DE PRESENTACIÓN:

13 de Agosto del 2020

GUAYAQUIL - ECUADOR

Tabla de contenido

Sección A: Code Smells y Técnicas de Refactorización

1. Lazy class

1.1 Consecuencia

Estas clases hacen que el código sea menos entendible y eficiente

- InformacionAdicionalProfesor

  Esta clase tiene algunos atributos que no son requeridos en el sistema, es decir no justifica su existencia

```java
package modelos;

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

}
```

- CalcularSueldo

  Esta clase solo tiene un método que puede estar dentro de Profesor para así disminuir el acoplamiento entre clases

```java
public class calcularSueldoProfesor {

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

        return sueldo;
    }
}
```

1.2 Técnica de Refactorización

Inline Class

Mover las características de la clase InfoAdicional a la clase principal Profesor y a su vez el metodo Calcular

```
public class Profesor {
    public String codigo;
    public String nombre;
    public String apellido;
    public int edad;
    public String direccion;
    public String telefono;
    public ArrayList<Paralelo> paralelos;
    //atributos adicionales
    public int añosdeTrabajo;
    public String facultad;
    public double BonoFijo;

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

    public double calcularSuledoProfesor(){
        return añosdeTrabajo*600 +BonoFijo;

    }
```

2. Speculative Generality

2.1 Consecuencia

Código no utilizado o método no implementado vuelve al código más difícil de mantener como es el método de mostrarListado() que quizá este para futuras implementaciones, pero actualmente en el sistema no es usado.

```
//Imprime el listado de estudiantes registrados
public void mostrarListado(){
    //No es necesario implementar
}
```

2.1 Técnica de Refactorización

Inline Method

Eliminar el método no utilizado

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

    public int getNumero() {
        return numero;
    }

    public void setNumero(int numero) {
        this.numero = numero;
    }

    public Materia getMateria() {
        return materia;
    }

    public void setMateria(Materia materia) {
        this.materia = materia;
    }

    public Profesor getProfesor() {
        return profesor;
    }

    public void setProfesor(Profesor profesor) {
        this.profesor = profesor;
    }


}
```

3. Message Chains

3.1 Consecuencia

ANTES

- Clase CalcularSueldoProfesor

```java
public class calcularSueldoProfesor {

    public double calcularSueldo(Profesor prof){
        double sueldo = 0;
        sueldo = prof.info.aniosdeTrabajo*600 + prof.info.BonoFijo; // code smell -> Message chain
        return sueldo;
    }
}
```

- Aumenta el acoplamiento entre clases, por lo tanto, un cambio en una clase dentro de la cadena puede afectar el funcionamiento del método.

3.2 Técnica de Refactorización

MOVE METHOD

- Clase Profesor

```java
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;

    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);
    }

    public double calcularSueldo(){
        return info.aniosdeTrabajo*600 + info.BonoFijo;
    }
}
```

- Para no llamar a tantas clases es conveniente mover el método completo a la clase Profesor, dado que es esta clase la que utilizará este método siempre. Como consecuencia la clase CalcularSueldoProfesor es eliminada.

4. Data clumps

4.1 Consecuencia

ANTES

- Clase Profesor

```
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;

    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);
    }

    public double calcularSueldo(){
        return info.aniosdeTrabajo*600 + info.BonoFijo;
    }
}
```

- Clase Estudiante

```
public class Estudiante{
    public String matricula;
    public String nombre;
    public String apellido;
    public String facultad;
    public int edad;
    public String direccion;
    public String telefono;
    public ArrayList<Paralelo> paralelos;

    //Getter y setter de Matricula

    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    public double calcularNotaInicial(Paralelo p, Nota nota){
        return calcularPuntaje(p,nota);
    }

    public double calcularNotaFinal(Paralelo p, Nota nota){
        return calcularPuntaje(p,nota);
    }

    public double calcularPuntaje(Paralelo p, Nota nota) {
        double puntaje = 0;
        for(Paralelo par:paralelos){
            if(p.equals(par)){
                puntaje = nota.calcularNota();
            }
        }
```

- Tener código repetido e idéntico en distintas partes del código representa problemas al interpretarlo y aumenta la cantidad de líneas, generando clases excesivamente largas y confusas de entender.

4.2 Técnica de refactorización

EXTRACT CLASS

- Clase Profesor

```
public class Profesor extends Persona {
    public String codigo;
    public int anosTrabajo;
    public double BonoFijo;
    public ArrayList<Paralelo> paralelos;

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

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

    public double calcularSueldoProfesor() {
        return anosTrabajo*600+BonoFijo;
    }
}
```

- Clase Estudiante

```
public class Estudiante extends Persona{
    public String matricula;
    public ArrayList<Paralelo> paralelos;

    //Getter y setter de Matricula
    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    public double calcularNota(Paralelo p, Nota n) {
        double nota = 0;
        for(Paralelo par:paralelos){
            if(p.equals(par)){
                nota = n.calcularNota();
            }
        }
        return nota;
    }

    public double calcularNotaTotal(Paralelo p){
        double notaTotal = 0;
        for(Paralelo par:paralelos){
            if(p.equals(par)){
                notaTotal = p.getMateria().calcularNotaTotal();
            }
        }
        return notaTotal;
    }
}
```

- Clase Persona

```java
public class Persona {
    public String nombre;
    public String apellido;
    public int edad;
    public String direccion;
    public String telefono;
    public String facultad;

    public Persona() {
        // Constructor vacio para subclases que no necesiten constructor
    }

    public Persona(String nombre, String apellido, int edad, String direccion, String telefono, String facultad) {
        super();
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
        this.direccion = direccion;
        this.telefono = telefono;
        this.facultad = facultad;
    }
    // Getters y Setters
```

- Como tanto la clase Estudiante como la clase Profesor tienen varios atributos en común, hay que extraer una super clase, la cual sirva para encapsular esta información y evitar que el mismo código exista en ambas clases. Con esta refactorización logramos reducir el tamaño de la clase Estudiante y la clase Profesor.

5. Inappropiate Intimacy

5.1 Consecuencia

Antes

```java
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();
    }

    //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
        }
    }
}
```

- La clase Ayudante esta muy acoplada con la clase Estudiante esto causa que las clases sean difíciles de mantener, comprender, y cambiar.
- La clase Ayudante se vuelve completamente dependiente de Estudiante, donde si se modifica esta clase, la clase Ayudante también debería cambiarse para cumplir con la funcionalidad adecuada.

4.1 Técnica de Refactorización

Replace Delegation with Inheritance

```java
package modelos;

import java.util.ArrayList;

public class Ayudante extends Estudiante {

    public ArrayList Paralelo  paralelos;

    public String getMatricula() {
        return matricula;
    }

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

    public String getNombre() {
        return nombre;
    }

    public String getApellido() {
        return apellido;
    }

    //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
        }
    }
}
```

Haciendo la asuncion de que el Ayudante es un Estudiante que usara sus atributos y métodos en algún momento requerido por lo que se decide usar esta técnica haciendo que extienda de ayudante, por lo que el código se vuelve mas organizado y muy simple de modificar, entender y mantener.

6. Duplicated Code

6.1 Consecuencia

Antes

```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;
    }
```

- La clase estudiante tiene los métodos CalcularNotaInicial(), CalcularNotaFinal(), con código duplicado esto ocasiona que el método y la clase sea más larga de lo que debería ser, por lo que habría una posibilidad de bugs, haciéndolo difícil de mantener.
- Hace que el código sea difícil de cambiar, ya que si se necesita cambiar alguna parte del código repetido este cambio debería hacerse para cada parte en donde se lo implementa.

6.2 Técnica de Refactorización

Extract Method
Despues `public double calcularNotaInicial(Paralelo p, Nota nota){ return calcularPuntaje(p,nota); } public double calcularNotaFinal(Paralelo p, Nota nota){ return calcularPuntaje(p,nota); } public double calcularPuntaje(Paralelo p, Nota nota) { double puntaje 0; for(Paralelo par:paralelos){ if(p.equals(par)){ puntaje = nota.calcularNota(); } } return puntaje; }`
Al momento de extraer un método que se llama en los métodos que tuvieron código duplicado esto hace que el código sea corto, facilitando su lectura, entendimiento, el debbug, y el mantenimiento de este.

7. Long Parameter List

7.1 Consecuencia

Una lista de parámetros mayor a 3 o 4 para un método hace que el código sea menos entendible, lo recomendable es tener un objeto y acceder a sus atributos.

7.2 Técnica de Refactorización

Introduce Parameter Object

Los atributos nexamen, nlecciones y ntalleres está relacionados, así que es mejor agruparlos en un mismo objeto en lugar de enviarlos por separado. Para esto se crea una nueva clase llamada Nota, que además se encarga de hacer el cálculo de la calificación en base a estos parámetros.

Antes

```
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;
}
```

Después

Clase Nota

```
public class Nota {
    private double nexamen;
    private double ndeberes;
    private double nlecciones;
    private double ntalleres;


    public double getNotaExamen() {
        return nexamen;
    }

    public double getNotaDeberes() {
        return ndeberes;
    }

    public double getNotaLecciones() {
        return nlecciones;
    }

    public double getNotaTalleres() {
        return ntalleres;
    }

    public double calcularNota() {
        double notaTeorico = nexamen+ndeberes+nlecciones*0.80;
        double notaPractico = ntalleres*0.20;
        return notaTeorico+notaPractico;
    }
}
```

Clase Estudiante

```
public double calcularNotaInicial(Paralelo p, Nota nota){
    double notaInicial=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            notaInicial = nota.calcularNota();
        }
    }
    return notaInicial;
}

public double calcularNotaFinal(Paralelo p, Nota nota){
    double notaFinal=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            notaFinal = nota.calcularNota();
        }
    }
    return notaFinal;
}
```

8. Temporary field

8.1 Consecuencia

Los campos temporales ocupan espacio en las clases y alarga el código innecesariamente ya que solo se les asigna un valor y son usados bajo ciertas condiciones, el resto del tiempo están vacíos. Además, el código se vuelve más difícil de entender.

8.2 **Técnica de Refactorización**

Inline temp

Para no usar una variable temporal, se pone directamente después de *return* el código con el que se calcula el valor a devolver.

Antes

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

    return sueldo;
}
```

Después

```
public double calcularSueldo(Profesor prof){
    return prof.info.aniosdeTrabajo*600 + prof.info.BonoFijo;
}
```

SECCION B

Repositorio

https://github.com/claudiasofia18/TallerRefactoring.git