ESCUELA SUPERIOR POLITECNICA DEL LITORAL

DISEÑO DE SOFTWARE

TALLER DE CODE SMELLS

INTEGRANTES

ALEX VELEZ

JAIME PIZARRO

VALERIA BARZOLA

CARLOS LOJA

EDDO ALVARADO

TERMINO

2020 – 1S

Contents

Sección A

Code Smell 1: Duplicate Code

```java
//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. El teorico y el practico se calcu
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

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 se mantiene el código duplicado, hacemos el código más largo y difícil de mantener. En caso de que se quiera hacer cambio en la forma de calcular las notas, habrá que hacer los cambios en ambos códigos.

Técnicas de Refactorización

Extract Method:

Extraer el código repetido en ambos métodos y ubicarlo en uno nuevo con un nombre que pueda generalizarse para ambos casos.

Después

Esta parte del código tenía varios code smells que son revisados más abajo, aplicando todas las técnicas de refactorización el código final queda de la siguiente manera:

```java
//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y 
public double calcularNota(){
    double notaTeorico=(getNexamen()+getNdeberes()+getNlecciones())*0.80;
    double notaPractico=(getNtalleres())*0.20;
    return notaTeorico+notaPractico;
}
```

Code Smell 2: Long Parameter List

```
public double CalcularNotaFinal(Paralelo p, double nexamen,double ndeberes, double nlecciones, double ntalleres){
```

Consecuencias

Las consecuencias de mantener el método con todos los parámetros son muy negativas.

En primer lugar, hace el código menos entendible, mientras más parámetros tiene el método, hay mayor probabilidad de que no se entienda el por qué debe recibir esos métodos. En segundo lugar, el costo de mantenimiento del código aumenta. También hace el código menos extensible.

Técnicas de Refactorización

Extract Class

Primero creamos una clase Nota que convierta los parámetros del método y los convierte en atributos de la clase. Esto lo hacemos ya que el estudiante no debe tener la responsabilidad de realizar los cálculos de las notas.

Move Method

Creamos un nuevo método en la clase Nota que se llame calcularNota() que se base en el método calcularNotaFinal() o calcularNotaInicial() pero ya no tenga la necesidad de recibir las notas como parámetros sino simplemente calcule la nota promedio usando los atributos de la clase.

Self Encapsulate Field

Usar los getters para acceder a los atributos de la clase.

Después

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


 //Calcula y devuelve la nota inicial contando examen, deberes, lecciones y 1
 public double calcularNota(){
     double notaTeorico=(getNexamen()+getNdeberes()+getNlecciones())*0.80;
     double notaPractico=(getNtalleres())*0.20;
     return notaTeorico+notaPractico;
 }
```

Code Smell 3: Inappropriate Intimacy

```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();
    }
```

Consecuencias:

Esta clase Ayudante usa los campos y métodos internos de la clase Estudiante contribuyendo así a un acoplamiento excesivo entre clases. Haciendo el código difícil de organizar y mantener.

Técnicas de Refactorización:

Replace Delegation With Inherintance

La técnica de refactorización utilizada para solucionar el code smell es Replace Delegation With Inherintance**,** esto se da porque la clase Ayudante utiliza a los métodos públicos de Estudiante, en este caso al utilizar este método evitamos la creación de métodos delegados del objeto de tipo Estudiante.

Por ello convertimos la clase Estudiante en una superclase de la clase ayudante, para así heredar directamente los métodos y no crear nuevos métodos

Después

```java
public class Ayudante extends Estudiante{

    Ayudante(){

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

Code Smell 4: Data Class

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

}
```

Consecuencias:

Esta clase contiene solo campos que son simplemente contenedores de datos utilizados por otras clases. Estas clases no contienen ninguna funcionalidad adicional y no pueden operar de forma independiente con los datos que poseen.

Dejar esta clase en el proyecto significaría aumentar el acoplamiento entre las clases, de manera innecesaria

Técnicas de Refactorización:

Revisando el código de cliente que usa la clase InforamacionAdicionalProfesor que es calcularSueldoProfesor, puede encontrar una funcionalidad que estaría mejor ubicada en la clase de datos en sí (InforamacionAdicionalProfesor). Dado este caso se utilizaría dos técnicas de refactorización:

Move method: Crear un nuevo método en la clase que más usa el método, luego mover el código del método anterior allí.

Encapsulate Field: para ocultarlos del acceso directo y requiera que el acceso se realice solo a través de getters y setters.

Después

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

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

Code Smell 5: Primitive Obsession

```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;
```

```java
public class Estudiante{
    //Informacion del 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;
```

Consecuencias:

La consecuencia de mantener a las variables dirección y teléfono como primitivas, es que no permite describir el comportamiento adecuado de esos atributos, pues el tipo primitivo no le brinda una adecuada descripción a su data.

Técnicas de Refactorización:

Revisando ambos atributos que usa las clases Profesor y Estudiante, y se decidió agregar un comportamiento descriptivo que estaría mejor ubicada en nuevas clases. Dado este caso se utilizaría la técnica de refactorización:

Replace Data Value with Object:

Como ambos atributos no cumplen con un comportamiento adecuado, se crean clases de con un nombre descriptivo y se agregan las funcionalidades descriptivas y adecuadas de cada una de las clases.

Después:

```java
*/
public class Dirección {
    private String callePrincipal;
    private String calleSecundaria;
    private String referencia;
    private String codigoPostal;

    public Dirección(String callePrincipal) {
        this.callePrincipal = callePrincipal;
    }

    public String getCallePrincipal() {
        return callePrincipal;
    }

    public void setCallePrincipal(String callePrincipal) {
        this.callePrincipal = callePrincipal;
    }

    public String getCalleSecundaria() {
        return calleSecundaria;
    }

    public void setCalleSecundaria(String calleSecundaria) {
        this.calleSecundaria = calleSecundaria;
    }

    public String getReferencia() {
        return referencia;
    }
```

```java
public class Telefono {
    private String number;
    private String operadora;
    private Persona propietario;

    public Telefono(String number) {
        this.number = number;
    }


    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }

    public String getOperadora() {
        return operadora;
    }

    public Persona getPropietario() {
        return propietario;
    }

    public void setPropietario(Persona propietario) {
        this.propietario = propietario;
    }
```

Code Smell 6: Duplicate Code

```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;
```

```java
public class Estudiante{
    //Informacion del 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;
```

Consecuencias:

Tener atributos de clases repetidos, causan que las clases tengan muchos atributos y por lo tanto sea una clase larga, sea mucho más difícil de mantener, menos mantenible y tenga un mayor costo de mantenibilidad.

Técnicas de Refactorización:

Extract Superclass.

Como ambas clases tienen atributos similares, se debe crear una nueva clase que contenga a todos esos atributos similares, y hacer que ambas clases hereden de esa nueva clase.

En este caso se creó a la clase Persona que es la encargada de tener estos atributos en común, los ponemos con un modificador de acceso protected, con sus getters y setters y Profesor y Estudiante heredan de la clase Persona, y mantienen los atributos que no heredan de Persona.

Después

Luego de aplicar esta técnica de refactoring el código resultante sería el siguiente:

```java
public class Persona {
    protected String nombre;
    protected String apellido;
    protected int edad;
    protected Dirección direccion;
    protected Telefono telefono;
    protected List<Paralelo> paralelos;


    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getApellido() {
        return apellido;
    }
```

```java
public void setApellido(String apellido) {
    this.apellido = apellido;
}

public int getEdad() {
    return edad;
}


public void setEdad(int edad) {
    this.edad = edad;
}

public String getDireccion() {
    return direccion.getCallePrincipal();
}

public void setDireccion(String direccion) {
    this.direccion = new Dirección(direccion);
}

public String getTelefono() {
    return telefono.getNumber();
}

public void setTelefono(String telefono) {
    this.telefono = new Telefono(telefono);
}

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

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

public class Profesor extends Persona{
    public String codigo;
    public InformacionAdicionalProfesor info;

    public Profesor(String codigo) {
        this.codigo = codigo;
    }

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


}
```

```java
public class Estudiante extends Persona{
    //Informacion del estudiante
    public String matricula;
    public String facultad;

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

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


    //Getter y setter de la Facultad
    public String getFacultad() {
        return facultad;
    }

    public void setFacultad(String facultad) {
        this.facultad = facultad;
    }

}
```

Encapsulate Field

Se cambia el modificador de acceso de los atributos de las clases Profesor y Estudiante de público a privado, y se incluyen los getters y setters para dar acceso a los atributos.

```java
public class Profesor extends Persona{
    private String codigo;
    private InformacionAdicionalProfesor info;

    public Profesor(String codigo) {
        this.codigo = codigo;
    }

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

    public String getCodigo() {
        return codigo;
    }

    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }

    public InformacionAdicionalProfesor getInfo() {
        return info;
    }

    public void setInfo(InformacionAdicionalProfesor info) {
        this.info = info;
    }

}
```

```java
public class Estudiante extends Persona{
    //Informacion del estudiante
    private String matricula;
    private String facultad;

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

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

    //Getter y setter de la Facultad
    public String getFacultad() {
        return facultad;
    }

    public void setFacultad(String facultad) {
        this.facultad = facultad;
    }

}
```

Code Smell 7: Lazy Class

```
public class calcularSueldoProfesor {

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

Consecuencias:

Las consecuencias de tener una Lazy Class es que los costos de mantenimiento pueden elevarse, ya que habrá más código que leer, además que quitará tiempo.

Técnicas de Refactorización:

Inline Class

Para solucionar el code smell se utilizará el método Inline Class lo cual nos permitirá unir una clase con otra, en este caso podremos eliminar la clase CalcularSueldoProfesor.

Move Method

y utilizar otro método de refactoring el cual es move method, el mismo nos ayudara a mover el método que contenía la clase CalcularSueldoProfesor a la clase InformacionAdicionalProfesor**.**

Después

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

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

Code Smell 8: Dead Code

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

Consecuencias:

Provoca que el código sea más complicado de leer, que las clases sean más largas y complejas.

Técnicas de Refactorización:

Como los métodos no son usados en ninguna clase se pueden eliminar sin generar problemas en ninguna otra clase.

Después

La solución de este code smell es eliminar el método, por lo que no existe un bloque de código para mostrar en esta sección.

Code Smell 9: Inappropiate Intimacy

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

}
```

Consecuencias:

Se tienen atributos que no deben pertenecer a Materia, por lo que no cumple el principio de SOLID de Single Responsibility Principle. El tener estos atributos en materia, hace que deban tener instancias de esta clase para poder acceder a estos atributos, lo que produce que exista un acoplamiento probablemente innecesario con esta clase.

Técnicas de Refactorización:

Extract Class:

Aplicamos el método de Extract Class para crear una nueva clase EstudianteNota que sirva como relación entre Estudiante y Paralelo. Esta clase servirá para tener las notas de los estudiantes.

Move Field:

Notamos que es necesario mover los atributos notaInicial, notaFinal y notaTotal de la clase Materia a la nueva clase EstudianteNota.

Move Method:

Movemos el método de calcularNotaTotal() que estaba estudiante a la clase EstudianteNota.

Después

```java
public class EstudianteNota {
    private Nota notaInicial;
    private Nota notaFinal;
    private double notaTotal;
    private Estudiante estudiante;
    private Paralelo paralelo;


    public void calcularNotaTotal(){
        setNotaTotal((getNotaInicial().calcularNota() + getNotaFinal().calcularNota())/2);
    }

    public Nota getNotaInicial() {
        return notaInicial;
    }
```

```java
    public Nota getNotaInicial() {
        return notaInicial;
    }

    public void setNotaInicial(Nota notaInicial) {
        this.notaInicial = notaInicial;
    }

    public Nota getNotaFinal() {
        return notaFinal;
    }

    public void setNotaFinal(Nota notaFinal) {
        this.notaFinal = notaFinal;
    }

public Estudiante getEstudiante() {
    return estudiante;
}

public void setEstudiante(Estudiante estudiante) {
    this.estudiante = estudiante;
}

public Paralelo getParalelo() {
    return paralelo;
}

public void setParalelo(Paralelo paralelo) {
    this.paralelo = paralelo;
}
```

Seccion B

Repositorio de github:

https://github.com/AlexVelezLl/TallerDS_Refactoring