# Taller Recfactoring

Integrantes:

- Samira Suárez
- Kelly Vaque
- Eduardo Mora

# Contenido

# Data Clumps

## Consecuencias.

Se tendrían variables repetidas como el nombre y apellidos entre las clases de estudiante y profesor incluido los métodos getters y setters por lo que tendríamos un código extenso.

- **Antes**

```
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;

    //Getter y setter de Matricula

    public String getMatricula() {
        return matricula;
    }

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

    //Getter y setter del Nombre
    public String getNombre() {
        return nombre;
    }

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

    //Getter y setter del Apellido
    public String getApellido() {
        return apellido;
    }

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

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

## Solución.

Para mejorarlo creamos una clase Persona que contiene las variables y métodos que se repetían en ambas clases.

- **Después**

```java
public class Persona {

    protected String nombre;
    protected String apellido;
    protected int edad;
    protected ArrayList<Paralelo> paralelos;
    protected String direccion;
    protected String telefono;

    public Persona(String nombre, String apellido, int edad, ArrayList<Paralelo> paralelos, String direccion, String telefono) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
        this.paralelos = paralelos;
        this.direccion = direccion;
        this.telefono = telefono;
    }

    public String getNombre() {
        return nombre;
    }

    public String getApellido() {
        return apellido;
    }

    public int getEdad() {
        return edad;
    }

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

    public String getDireccion() {
        return direccion;
    }

    public String getTelefono() {
        return telefono;
    }
```

# Tempory Field

## Consecuencia.

Los campos temporales obtienen sus valores (y, por lo tanto, los objetos los necesitan) solo en determinadas circunstancias. Fuera de estas circunstancias, están vacías. Variables que no tienen sentido crearlas.

- **Antes**

```
public class calcularSueldoProfesor {

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

## Solución

Podemos eliminar las variables innecesarias y poner el código en una sola sentencia

- **Después**

```
public class calcularSueldoProfesor {

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

# Comments

## Consecuencia.

El código está lleno de comentarios innecesarios en métodos muy obvios cuando solo se deben usar en métodos complejos.

- **Antes**

```
//Getter y setter de Matricula

public String getMatricula() {
    return matricula;
}

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

//Getter y setter del Nombre
public String getNombre() {
    return nombre;
}

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

//Getter y setter del Apellido
public String getApellido() {
    return apellido;
}

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

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

## Solución.

Se eliminan los comentarios innecesarios de los getters y setters solo se conservan en los algunos métodos medianamente complejos.

- **Después**

```java
public String getNombre() {
    return nombre;
}

public String getApellido() {
    return apellido;
}

public int getEdad() {
    return edad;
}

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

public String getDireccion() {
    return direccion;
}

public String getTelefono() {
    return telefono;
}
```

# Inappropiate Intimacy

En este caso la clase Ayudante necesita acceder a los parámetros de la clase Estudiante para su funcionamiento generando un acoplamiento entre las clases.

## Consecuencia

Esto genera que el desarrollador no pueda entender de una manera correcta ninguna de las dos clases. En caso de que se desee realizar un cambio en la clase Estudiante afectaría directamente a la clase Ayudante.

- **Antes**

```
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 duplic
    public String getNombre() {
        return est.getNombre();
    }

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

    //Los paralelos se aÃ±aden/eliminan directamente del Arraylist de
```

```
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;

    //Getter y setter de Matricula
```

## Solución

En este caso lo más optimo seria extraer la clase y hacer uso de la herencia, se toma en cuenta al ayudante como un caso “especial” de estudiante haciendo que extienda directamente de dicha clase.

- **Después**

```
import java.util.ArrayList;

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

    Ayudante(){
        super();
    }

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

# Long Parameter List

## Consecuencia

Los métodos para calcular notas tienen demasiados parámetros lo que hace inentendible el método.

- **Antes**

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

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;
}
```

## Solución

Lo ideal es crear una clase que se encargue de maneja cada parámetro del método calcular notas, para que sea más sencillo acceder a ellas y evitar la sobrecarga en el método.

- **Después**

**Nota.java**

```
package modelos;
import java.util.ArrayList;

public class Nota {
    private double nexamen;
    private double ndeberes;
    private double nlecciones;
    private double ntalleres;
    private ArrayList<Paralelo> paralelos;

    public double calcularNota(Paralelo p) {
        double notaTeorico=0;
        double notaPractico=0;
        for(Paralelo par:paralelos){
            if(p.equals(par)) {
                notaTeorico=(nexamen+ndeberes+nlecciones)*0.80;
                notaPractico=(ntalleres)*0.20;

            }
        }
        return notaTeorico+notaPractico;

    }
}
```

**Estudiante.java**

```java
//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. El
public double CalcularNotaInicial(Paralelo p,Nota n){
    double notaInicial=n.calcularNota(p);
    return notaInicial;
}

//Calcula y devuelve la nota final contando examen, deberes, lecciones y talleres. El t

public double CalcularNotaFinal(Paralelo p,Nota n){
    double notaFinal=n.calcularNota(p);
    return notaFinal;
}

//Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. Es
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;

}
```

# Lazy Class

## Consecuencias

Debido al pequeño aporte que genera esta clase, nos conviene remover y colocar ese método en otra clase, ya que nos ahorraríamos el mantenimiento que deba tener una clase nueva.

- **Antes**

```java
package modelos;

public class calcularSueldoProfesor {

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

## Solución

Ya que esta clase no presenta muchas funcionalidades, e incluso la clase utiliza los datos del Profesor que recibe, podríamos mover el método a la clase Profesor y remover la clase.

- **Después**

```java
import java.util.ArrayList;

public class Profesor extends Persona{
    protected int añosdeTrabajo;
    protected String facultad;
    protected double bonoFijo;

    public Profesor(String codigo, String nombre, String apellido,int edad) {
        super(codigo,nombre,apellido,edad);
    }

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

    public double calcularSueldo(){
        return añosdeTrabajo*600 + bonoFijo;
    }

    //Métodos Getters and Setters de los campos
}
```

**Sección B:**

https://github.com/KellyVaque/TallerRefactoring