

DISEÑO DE SOFTWARE

TALLER:

Refactoring 1

EQUIPO T3:

Bryan Paul Álava Calderón

Tommy David Beltrán Borbor

Rafael Alfonso Montalvo Velásquez

Tyrone Eduardo Rodríguez Motato

PARALELO: 1

PROFESOR: Dr. Carlos Mera

FECHA DE PRESENTACIÓN:

13 de agosto del 2020

GUAYAQUIL - ECUADOR



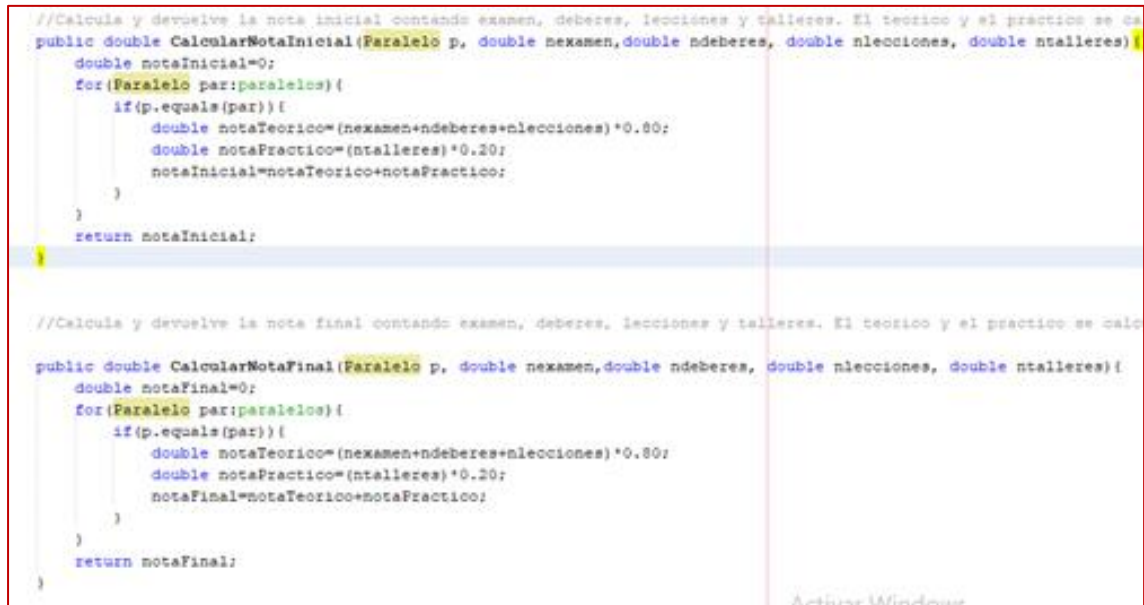
TABLA DE CONTENIDO

SECCIÓN A	3
DUPLICATE CODE.....	3
DATA CLASS	5
TEMPORARY FIELD	8
MESSAGE CHAINS.....	9
LONG PARAMETER LIST	11
FEATURE ENVY.....	14
PRIMITIVE OBSESSION	18
SECCIÓN B.....	20
ENLACE DEL REPOSITORIO.....	20

SECCIÓN A

Duplicate Code

En la clase Estudiante encontramos este mal olor:



```
//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 cala
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:

- Presencia de código duplicado en los métodos CalcularNotaInicial y CalcularNotaFinal que dificultan enormemente el mantenimiento.
- Partes específicas del código se ven diferentes, pero en realidad realizan el mismo trabajo; ya que pesar de tener métodos con distintos nombres, el algoritmo para calcular la nota inicial y final es el mismo.
- Un montón de código afecta la comprensión del mismo, pues además de tener largas secciones idénticas de código pueden ocultar como difieren unas de las otras, y por lo tanto cuál sería su propósito.
- Por último, podríamos encontrarnos con anomalías de actualización, ya que el código duplicado incurre en redundancias.

Técnicas de Refactorización aplicadas:

- **Extract Method:** Ya que encontramos fragmentos de código repetidos en los métodos CalcularNotaInicial y CalcularNotaFinal, que se pueden agrupar en uno nuevo método llamado únicamente CalcularNota, eliminando así los otros métodos con código repetidos.
- **Rename Method:** Se necesita cambiar el nombre de los métodos (CalcularNotaInicial y CalcularNotaFinal) por uno más centralizado como CalcularNota

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

Beneficios:

- Combinar código duplicado simplificará la estructura del código, haciéndolo más corto y entendible.
- Simplificación + brevedad = código más factible de soportar.

Data Class

Este mal olor se encuentra en las siguientes clases:

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

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

Consecuencias:

- Son clases que únicamente sirven como contenedores de datos, por lo cual desaprovechan completamente el poder que nos provee el lenguaje de programación (Java) para crear objetos con atributos y comportamientos determinados.
- En el caso específico de la clase InformaciónAdicionalProfesor, existe un alto acoplamiento con la clase Profesor, puesto que como su nombre lo indica, solo almacena información adicional de objetos de tipo Profesor, por lo cual no puede operar independientemente.
- Adicionalmente, los atributos de ambas clases son públicos, lo que compromete el principio de Encapsulación.
- De mantenerse el código de la misma forma, es posible que se encuentren otros code smells como el Inappropriate Intimacy, ya que otras clases podrían acceder directamente a los atributos.

Técnicas de Refactorización aplicadas:

- **Encapsulate field:** Para comenzar a eliminar este code smell, se empezará encapsulando la clase, de modo que sus atributos sean privados y cuenten con sus respectivos getters y setters.

Aplicación en la clase InformaciónAdicionalProfesor:

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

```
    public int getAñosdeTrabajo() {  
        return añosdeTrabajo;  
    }  
  
    public void setAñosdeTrabajo(int añosdeTrabajo) {  
        this.añosdeTrabajo = añosdeTrabajo;  
    }  
  
    public String getFacultad() {  
        return facultad;  
    }  
  
    public void setFacultad(String facultad) {  
        this.facultad = facultad;  
    }  
  
    public double getBonoFijo() {  
        return BonoFijo;  
    }  
  
    public void setBonoFijo(double BonoFijo) {  
        this.BonoFijo = BonoFijo;  
    }  
}
```

Aplicación en la clase Materia:

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

```

public String getCodigo() {
    return codigo;
}

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

public String getNombre() {
    return nombre;
}

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

public String getFacultad() {
    return facultad;
}

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

```

```

public double getNotaInicial() {
    return notaInicial;
}

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

public double getNotaFinal() {
    return notaFinal;
}

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

public double getNotaTotal() {
    return notaTotal;
}

public void setNotaTotal(double notaTotal) {
    this.notaTotal = notaTotal;
}

```

Beneficios:

- Mejora la comprensión y organización del código.
- La clase InformaciónAdicionalProfesor dejará de ser una Clase de Datos al brindársele una nueva funcionalidad reflejada en las siguientes técnicas de refactorización utilizadas.

Temporary Field

En la clase calcularSueldoProfesor existe este mal olor:

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

Consecuencias:

- Se tiene un campo temporal que no es necesario en la implementación del método ya que es una expresión muy simple que no necesita ser asignada a una variable.
- Solo se emplearán dichas campos o variables dentro de un determinado algoritmo.

Técnicas de Refactorización aplicadas:

- Inline Temp es la técnica que se va a aplicar para modificar ese smell.

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

Beneficios:

- Haciendo uso de la técnica de refactoring Inline Temp evitamos que este método contenga una variable innecesaria y que sea usada solo para devolver lo que requiere el método.
- De igual forma al borrar una variable innecesaria pensando a futuro para grandes algoritmos donde declaremos más campos temporales llega a convertirse en un ahorro de memoria eliminando o aplicando técnicas de refactorización.
- Incrementamos la legibilidad del código, esto en caso de que a futuro existan más líneas de código y necesitemos devolver diferentes valores o se realicen operaciones simplificadas que no necesiten el uso de la variable antes propuesta.

Message Chains

En la clase calcularSueldoProfesor existe este mal olor:

```
package modelos;

public class calcularSueldoProfesor {

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

Consecuencias:

- Al tener este code smell presente pues se requiere hacer una “navegación” a través de distintas clases para encontrar aquello que se requiere para el método de calcularSueldo.
- Al realizar una navegación por distintas clases se crea una dependencia de esta por parte de la clase calcularSueldoProfesor y por lo que cualquier modificación en las relaciones que tuvieran las clases que forman parte de esta estructura incurrirá en hacer cambios en el método o la clase.

Técnicas de Refactorización Aplicadas:

- **Move Method:** para mover ese método a la InformacionAdicionalProfesor clase.
- **Inline Class:** para eliminar la clase calcularSueldoProfesor

```
public double getBonoFijo() {
    return BonoFijo;
}

public void setBonoFijo(double BonoFijo) {
    this.BonoFijo = BonoFijo;
}

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

Beneficios:

- Al mover este método la clase InformacionAdicionalProfesor lo que hacemos es quitar por completo la cadena de mensajes y así de esta forma evitar esa dependencia en la estructura de las clases, es decir que evitamos que haya un alto acoplamiento entre estas clases.

- Hacemos el código más legible y fácil de mantener, por lo que si a futuro si se desea actualizar o implementar nuevas funcionalidades dentro del algoritmo se reduce la complejidad y se aumenta su comprensión.

Long Parameter List

En la clase Estudiante existe este mal olor:

Como se puede observar, el método CalcularNotaInicial tiene demasiados parámetros, por lo cual se debería buscar la manera de juntarlos en uno solo para así reducir los parámetros.

```
81 //Calcula y devuelve la nota inicial contando examen, deberes, lecciones y talleres. El teorico y el practico se calculan
82 public double CalcularNotaInicial(Paralelo p, double nexamen, double ndeberes, double nlecciones, double ntalleres){
83     double notaInicial=0;
84     for(Paralelo par:paralelos){
85         if(p.equals(par)){
86             double notaTeorico=(nexamen+ndeberes+nlecciones)*0.80;
87             double notaPractico=(ntalleres)*0.20;
88             notaInicial=notaTeorico+notaPractico;
89         }
90     }
91     return notaInicial;
92 }
```

También está en la clase Profesor.

Como se puede observar el constructor de la clase profesor se tiene demasiados parámetros, por ello se debería buscar la manera de reagrupar dichos parámetros para reducir el método constructor.

```
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<>();
}
```

Consecuencias:

- Hace que el código sea mucho más difícil de leer.
- Puede aumentar la probabilidad de errores difíciles de depurar, especialmente. si la lista contiene varios parámetros del mismo tipo que pueden transponerse inadvertidamente.
- Hace que el programa sea difícil de mantener.

Técnicas de Refactorización:

- **Introduce Parameter Object:** para agrupar los diferentes tipos de notas en una sola clase Notas, de las cuales se obtiene los diferentes tipos de notas con un getter.

```
public double CalcularNota(Paralelo p, Notas notas) {
    double nota=0;
    for(Paralelo par:paralelos){
        if(p.equals(par)){
            double notaTeorico=(notas.getNexamen()+notas.getNdeberes()+notas.getNlecciones())*0.80;
            double notaPractico=(notas.getNtalleres())*0.20;
            nota=notaTeorico+notaPractico;
        }
    }
    return nota;
}
```

```

public class Notas {
    double nexamen;
    double ndeberes;
    double nlecciones;
    double ntalleres;

    public double getNexamen() {
        return nexamen;
    }

    public double getNdeberes() {
        return ndeberes;
    }

    public double getNlecciones() {
        return nlecciones;
    }

    public double getNtalleres() {
        return ntalleres;
    }
}

```

- **Introduce Parameter Object y Extract Superclass:** para agrupar los diferentes los parámetros en la clase Profesor se crea un padre Persona. Con ello, los parámetros que recibe el constructor profesor sería el código y un objeto tipo Persona.

```

4
5 public class Profesor extends Persona {
6     public String codigo;
7     public InformacionAdicionalProfesor info;
8
9     private Profesor(String codigo, Persona persona) {
10         this.codigo = codigo;
11         this.nombre = persona.getNombre();
12         this.apellido = persona.getApellido();
13         this.edad = persona.getEdad();
14         this.direccion = persona.getDireccion();
15         this.telefono = persona.getTelefono();
16         paralelos = new ArrayList<>();
17     }
18
19     public void anadirParalelos(Paralelo p){
20         paralelos.add(p);
21     }
22 }

```

- **Remove Parameter:** Sin embargo, la clase Persona sigue teniendo los parámetros que tenía profesor. para ello vamos a extraer del constructor los parámetros menos relevantes y dejarlos simplemente como setters.

```

13  * @author Tyrone
14  */
15  public class Persona {
16      public String nombre;
17      public String apellido;
18      public int edad;
19      public String direccion;
20      public String facultad;
21      public String telefono;
22      public ArrayList<Paralelo> paralelos;
23
24      public Persona(String nombre, String apellido, String facultad, ArrayList<Paralelo> paralelos) {
25          this.nombre = nombre;
26          this.apellido = apellido;
27          this.facultad = facultad;
28          this.paralelos = paralelos;
29      }

```

Beneficios:

- Se evita la duplicación de código. Los campos y métodos comunes ahora pertenecen a una sola clase padre, en este caso la clase Notas.
- Haciéndolo de esta manera es posible implementar a futuro un Buldier para facilitar la construcción de las personas.

Feature Envy

Este code smell se puede encontrar en la clase Ayudante:

```
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();
    }
}
```

Y adicionalmente en métodos de la clase Estudiante:

```
public double getNotaTeorico(Notas notas) {
    return (notas.getNexamen()+notas.getNdeberes()+notas.getNlecciones())*0.80;
}

public double getNotaPractico(Notas notas) {
    return (notas.getNtalleres())*0.20;
}

//Calcula y devuelve la nota final contando examen, deberes, lecciones y talleres. El te
public double CalcularNota(Paralelo p, Notas notasP, Notas notasT) {
    double nota=0;
    for(Paralelo par:paralelos) {
        if(p.equals(par)) {
            nota= getNotaTeorico(notasT)+getNotaPractico(notasP);
        }
    }
    return nota;
}

//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().getNotaTotal()+p.getMateria().getNotaFinal())/2;
        }
    }
    return notaTotal;
}
```

Consecuencias:

- Existe duplicación de código, puesto que, los métodos `getMatricula`, `setMatricula`, `getNombre` y `getApellido` son propios del atributo `Estudiante` y como tal, deben pertenecer a esa clase.
- Los métodos de la clase `Estudiante` acceden a métodos y atributos de otras clases a través de los parámetros y no hacen uso de los atributos propios, por lo que esto aumenta el acoplamiento entre estas clases.
- No existe una forma para acceder correctamente a los atributos de la clase `Ayudante`.

Técnicas de Refactorización Aplicadas:

- **Move Method:** Los métodos `getMatricula`, `setMatricula`, `getNombre` y `getApellido` se removerán de la clase `Ayudante` y pertenecerán a la clase `Estudiante`. Los métodos `getNotasTeorico` y `getNotasPractico` pasarán a la clase `Notas` y los métodos `calcularNota` y `calcularNotaTotal` pasarán a la clase `Paralelo`.
- **Encapsulate Field:** Los atributos de la clase serán privados y contarán con sus respectivos getters y setters.
- **Encapsulate Collection:** Puesto que la clase contiene un arreglo de paralelos, es conveniente aplicar esta técnica para que se puedan añadir y eliminar paralelos al objeto `Ayudante` y no al atributo.

```
public class Ayudante {  
    private Estudiante est;  
    private List<Paralelo> paralelos;  
  
    Ayudante(Estudiante e) {  
        est = e;  
        paralelos = new ArrayList<>();  
    }  
  
    public Estudiante getEst() {  
        return est;  
    }  
  
    public void setEst(Estudiante est) {  
        this.est = est;  
    }  
  
    public List<Paralelo> getParalelos() {  
        return Collections.unmodifiableList(paralelos);  
    }  
  
    public boolean addParalelo(Paralelo paralelo) {  
        return paralelos.add(paralelo);  
    }  
  
    public boolean removeParalelo(Paralelo paralelo) {  
        return paralelos.remove(paralelo);  
    }  
}
```

```

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;
}

```

En la clase Notas:


```

public double getNotaTeorico(){
    return (getNexamen() + getNdeberes() + getNlecciones())*0.80;
}
public double getNotaPractico(){
    return getNtalleres()*0.20;
}

```

En la clase Paralelo:

```

//Se puede usar para calcular la nota de cualquiera de los parciales
public double CalcularNotaParcial(Notas notas){
    return notas.getNotaTeorico()+ notas.getNotaPractico();
}

public double CalcularNotaTotal(){
    return (getMateria().getNotaTotal()+getMateria().getNotaFinal())/2;
}

```

Beneficios:

- Existe una mejor organización de los métodos en las clases.
- Los métodos que usaban de los atributos “enviados” ahora corresponden a la clase pertinente para evitar el code smell.
- Con la nueva ubicación del método, existe una mejor distribución de responsabilidades para cada clase.

Primitive Obsession

En la clase Profesor existe este mal olor:

En este caso el constructor profesor tiene dos variables de tipo String dirección y teléfono, Sin embargo, la variable dirección puede llegar a ser muy compleja, ya que puede tener varias calles. Por otro lado, tenemos la variable teléfono, el cual puede llegar a ser muy simple, sin embargo, si se necesita tener un teléfono asociado a una compañía debería poder relacionarse con dicha clase.

```
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<>();  
}
```

Consecuencias:

- El desarrollador pierde los beneficios que vienen con el diseño orientado a objetos, como escribir datos por nombre de clase o sugerencias de tipo.
- Cuando la lógica del tipo de datos no está separada en una clase dedicada, agregar un nuevo tipo o comportamiento hace que la clase básica crezca y se vuelva difícil de manejar.
- Los tipos de datos primitivos son mucho más difíciles de controlar. Como resultado, podemos obtener variables que no son válidas (admitidas por el tipo) o significativas.

Técnicas de Refactorización:

- **Replace Data Value with Object:** Con el reemplazo de un valor de datos con un objeto, tenemos un campo primitivo (Dirección, Teléfono) que ya no es tan simple debido al crecimiento del programa y ahora tiene datos y comportamientos asociados.

```
12 public class Direccion {  
13     public String callePrincipal;  
14     public String calleSecundaria;  
15     public String manzana;  
16     public String villa;  
17  
18     public Direccion(String callePrincipal, String calleSecundaria, String manzana, String villa) {  
19         this.callePrincipal = callePrincipal;  
20         this.calleSecundaria = calleSecundaria;  
21         this.manzana = manzana;  
22         this.villa = villa;  
23     }  
24 }
```

```

public class Telefono {
    String numero;

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

    public String getNumero() {
        return numero;
    }

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

```

```

public class Persona {
    public String nombre;
    public String apellido;
    public int edad;
    public Direccion direccion;
    public String facultad;
    public Telefono telefono;
    public ArrayList<Paralelo> paralelos;
}

```

Beneficios:

- Se obtiene un código más legible, porque en lugar de un grupo de parámetros, verá un solo objeto con un nombre comprensible en este caso “Persona”.
- Grupos idénticos de parámetros dispersos que crean su propio tipo de duplicación de código ya que, constantemente se encuentran grupos idénticos de parámetros y argumentos.

SECCIÓN B

Enlace del Repositorio

<https://github.com/tomdbelt/Refactoring1-G3>