ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

TAREA 11 - Smells

AUTORES:

- Alan Ariel Aguilar Morocho
- Victor Roberto Bravo Lopez
- Geovanny Andres Lacouture Velasquez
- Eddy Teodoro Arriaga Arreaga

TUTOR:
Ing. David Alonso Jurado Mosquera

13 DE AGOSTO DEL 2025

Composing Methods.....3
Extract Method.....3
Inline Temp.....4
Simplifying Conditional Expressions.....5
Replace Nested Conditional with Guard Clauses.....5
Replace Conditional with Polymorphism.....6
Dealing with Generalization.....7
Pull Up Field.....7
Push Down Field.....8
Pull Up Constructor Body.....8
Pull Up Method.....8
Moving Features between Objects.....9
Move Method.....9

Composing Methods.

Extract Method.

Código sin refactorización:

```
public double calcularSalario() {
    double salarioTotal = salarioBase;
    if (salarioBase>0) {
        if (horasTrabajadas >= 0) {
            // Horas trabajadas normales = 40;
            if (horasTrabajadas > 40) {
                salarioTotal += (horasTrabajadas - 40) * 50; // Pago de horas extra
            }
        }else {
            throw new IllegalArgumentException(s:"Las horas trabajadas deben ser mayor o igual a 0");
        }
    } else {      alan-am, 32 minutes ago • ADD base project
        throw new IllegalArgumentException(s:"El salario debe ser mayor o igual a 0");
    }
    switch (departamento) {
        case "Sistemas":
            salarioTotal += 20;
            break;
        case "Contabilidad":
            salarioTotal += 10;
            break;
        default:
            break;
    }
    return salarioTotal;
}
```

Consecuencias:
Mantener el código sin aplicar Extract Method provoca que calcularSalario() sea largo y difícil de entender. Además, la validación de datos está incrustada dentro del mismo método, lo que impide su reutilización y fomenta la duplicación de código si se necesita en otros lugares. Por último, el mantenimiento se vuelve más costoso, pues cualquier ajuste requiere intervenir en un método extenso, lo que aumenta el riesgo de romper funcionalidades ya existentes y va en contra del principio de responsabilidad única (SRP).

Código factorizado:

```
public double calcularSalario() {
    validarSalarioYHoras();
    double salarioTotal = salarioBase;
    salarioTotal += calcularHorasExtra();
    salarioTotal += calcularBonoDepartamento();
    return salarioTotal;
}
```

```java
private void validarSalarioYHoras() {
    if (salarioBase <= 0) {
        throw new IllegalArgumentException(s:"El salario debe ser mayor o igual a 0");
    }
    if (horasTrabajadas < 0) {
        throw new IllegalArgumentException(s:"Las horas trabajadas deben ser mayor o igual a 0");
    }
}

private double calcularHorasExtra() {
    if (horasTrabajadas > 40) {
        return (horasTrabajadas - 40) * 50;
    }
    return 0;
}       You, 17 seconds ago • Uncommitted changes

private double calcularBonoDepartamento() {
    switch (departamento) {
        case "Sistemas":
            return 20;
        case "Contabilidad":
            return 10;
        default:
            return 0;
    }
}
```

Beneficios:
El beneficio del código refactorizado es la mejora significativamente la legibilidad, ya que calcularSalario() queda reducido a una secuencia de pasos claros y descriptivos. Además, cumple mejor con el principio de responsabilidad única (SRP), separando la validación, el cálculo de horas extras y el cálculo de bonos por departamento en métodos independientes, lo que facilita el mantenimiento y reduce el riesgo de errores al realizar cambios.

Inline Temp.

Código sin refactorización:

```java
@Override
public double calcularSalario() {
    double salario = super.getHorasTrabajadas() * super.getTarifaHora();
    return salario;
}
```

Consecuencias:

Mantener la variable temporal salario innecesaria hace que el método sea ligeramente más largo y menos directo. Al no aplicar Inline Temp, se pierde la oportunidad de simplificar el código y facilitar su lectura, especialmente en métodos cortos donde la asignación directa en la sentencia return es más clara y expresiva. También se dificulta el mantenimiento, ya que se añade un elemento más a considerar sin necesidad real.

Código factorizado:

```java
@Override
public double calcularSalario() {
    return super.getHorasTrabajadas() * super.getTarifaHora();
}
```
alan-am, 22 minutes ago • ADD base project

Beneficios:

El uso de Inline Temp en este código ofrece varios beneficios. En primer lugar, mejora la claridad y concisión, ya que elimina la variable temporal innecesaria y reduce el número de líneas, dejando la lógica en una sola instrucción fácilmente entendible. Además, facilita el mantenimiento, ya que no existe una variable extra que pueda ser mal utilizada o modificada de forma accidental en futuras actualizaciones.

Simplifying Conditional Expressions.

Replace Nested Conditional with Guard Clauses.

Código sin refactorizar:

```java
    public double calcularSalario() {
        double salarioTotal = salarioBase;
        if (salarioBase>0) {
            if (horasTrabajadas >= 0) {
                //Horas trabajadas normales = 40;
                if (horasTrabajadas > 40) {
                    salarioTotal += (horasTrabajadas - 40) * 50; //Pago de horas extra
                }
            }else {
                throw new IllegalArgumentException(s:"Las horas trabajadas deben ser mayor o igual a 0");
            }
        } else {
            throw new IllegalArgumentException(s:"El salario debe ser mayor o igual a 0");
        }

        salarioTotal +=departamento.obtenerSalarioDepartamento();

        return salarioTotal;
    }
```

En la clase Empleado, en el metodo calcularSalario() se puede observar una sección de código en el que se declara un If demasiado confuso.

<u>*Consecuencias:*</u> Se hace complicada la legibilidad del código y entender cual seria cada flujo del if, dificultando la optimización y mantenibilidad.

Código Refactorizado:

```java
public double calcularSalario() {
    if (salarioBase <= 0) {
        throw new IllegalArgumentException("El salario debe ser mayor o igual a 0");
    }

    if (horasTrabajadas < 0) {
        throw new IllegalArgumentException("Las horas trabajadas deben ser mayor o igual a 0");
    }

    double salarioTotal = salarioBase;

    // Horas trabajadas normales = 40
    if (horasTrabajadas > 40) {
        salarioTotal += (horasTrabajadas - 40) * 50; // Pago de horas extra
    }

    salarioTotal +=departamento.obtenerSalarioDepartamento();

    return salarioTotal;
}
```

Se utilizó la técnica de “” para simplificar el if y hacerlo más legible.

Beneficios:
Mayor legibilidad en el codigo y mantenibilidad.

Replace Conditional with Polymorphism.

Código Actual:

```java
public double calcularSalario() {
    double salarioTotal = salarioBase;
    if (salarioBase>0) {
        if (horasTrabajadas >= 0) {
            // Horas trabajadas normales = 40;
            if (horasTrabajadas > 40) {
                salarioTotal += (horasTrabajadas - 40) * 50; // Pago de horas extra
            }
        }else {
            throw new IllegalArgumentException("Las horas trabajadas deben ser mayor o igual a 0");
        }
    } else {
        throw new IllegalArgumentException("El salario debe ser mayor o igual a 0");
    }
    switch (departamento) {
        case "Sistemas":
            salarioTotal += 20;
            break;
        case "Contabilidad":
            salarioTotal += 10;
            break;
        default:
            break;
    }
    return salarioTotal;
}
```

En la clase Empleado, en el metodo calcularSalario() se puede observar una seccion de codigo en el que se declara un Switch extenso

Consecuencias: Mantener este Switch extenso hace que se rompa uno de los principios de SOLID, el OCP, haciendo que cada vez que se agregue un nuevo tipo de departamento, haya que agregar un nuevo ‘case’ al switch.

Código Refactorizado:

Para la refactorización usaremos la técnica “Replace Conditional with Polymorphism.”

```
public double calcularSalario() {
    double salarioTotal = salarioBase;
    if (salarioBase>0) {
        if (horasTrabajadas >= 0) {
            // Horas trabajadas normales = 40;
            if (horasTrabajadas > 40) {
                salarioTotal += (horasTrabajadas - 40) * 50; // Pago de horas extra
            }
        }else {
            throw new IllegalArgumentException(s:"Las horas trabajadas deben ser mayor o igual a 0");
        }
    } else {
        throw new IllegalArgumentException(s:"El salario debe ser mayor o igual a 0");
    }

    salarioTotal +=departamento.obtenerSalarioDepartamento();

    return salarioTotal;
}
```

Se hizo una sección nueva de clases departamentos, usando polimorfismo entre ellas se creó un método obtenerSalarioDepartamento() comun entre ellos , para ya solo llamar a este método sin tener que usar un switch case.

Beneficios:

Ahora el código cumple el principio OCP, se remueve código duplicado(muchos cases) y promueve una escalabilidad rápida.

Dealing with Generalization.

Pull Up Field.

Antes:

EmpleadoPorHoras.java > EmpleadoPorHoras

```
public class EmpleadoPorHoras extends Empleado {
    private String genero;
```

```
public class EmpleadoTemporario extends Empleado {
    private String nombre;
    private String genero;
    private double salarioBase;
    private int horasTrabajadas;
    private String departamento;
    private int mesesContrato;
```

```
public class Empleado {
    private String nombre;
    private double salarioBase;
    private int horasTrabajadas;
    private String departamento;
    private double tarifaHora;
```

```
public class EmpleadoFijo extends Empleado {
    private double bonoAnual;
    private String genero;
```

Consecuencia: El atributo género estaba duplicado en las subclases de empleado, generando duplicaciones en el código.

Después:

```
public class EmpleadoFijo extends Empleado {
    private double bonoAnual;
```

```
public class EmpleadoPorHoras extends Empleado {
```

```
public class EmpleadoTemporario extends Empleado {
    private String nombre;
    private double salarioBase;
    private int horasTrabajadas;
    private String departamento;
    private int mesesContrato;
```

```
public class Empleado {
    private String nombre;
    private double salarioBase;
    private int horasTrabajadas;
    private String departamento;
    private double tarifaHora;
    protected String genero;
```

Beneficio: Se movió el atributo género a la clase padre y se lo hizo protected para que los hijos que lo hereden puedan acceder a él sin problema y sin tener duplicar el atributo en todas las clases hijos

Push Down Field.

Código sin refactorizar

```
public class Empleado {
    private String nombre;
    private double salarioBase;
    private int horasTrabajadas;
    private double tarifaHora;
    private String departamento;
    protected String genero;
```

```
public class EmpleadoPorHoras extends Empleado {
```

Consecuencia: El atributo tarifaHora solo está siendo usado por EmpleadoPorHoras, incluso ni la misma clase Empleado lo usa, por eso lo mejor es pasarlo directamente a EmpleadoPorHoras para evitar redundancias y código inutil

Código refactorizado:

```
public class Empleado {
    private String nombre;
    private double salarioBase;
    private int horasTrabajadas;
    private double tarifaHora;
    private String departamento;
    protected String genero;
```

Beneficio: Al enviar tarifahra a la clase hija eliminamos redundancia y código inutil

Pull Up Constructor Body.

Antes:

```
public EmpleadoTemporario(String nombre, double salarioBase, int horasTrabajadas, Stri
    this.nombre = nombre;
    this.salarioBase = salarioBase;
    this.horasTrabajadas = horasTrabajadas;
    this.departamento = departamento;
    this.mesesContrato = mesesContrato;
    this.genero = genero;
}
```

Consecuencia: la clase EmpleadoTemporario tiene constructor es idéntico al de la clase padre.

Después:

```
public class EmpleadoTemporario extends Empleado {
    public EmpleadoTemporario(String nombre, double salarioBase, int horasTrabajadas, Stri
        super(nombre, salarioBase, horasTrabajadas, tarifaHora:0, departamento);
        this.mesesContrato = mesesContrato;
    }
```

Beneficios: Reducción de código y más legible.

Pull Up Method.

Antes:

```
public void imprimirDetalles() {
    System.out.println("Nombre: " + nombre);
    System.out.println("Genero: " + super.getNombre());
    System.out.println("Salario: " + salarioBase);
    System.out.println("Horas trabajadas: " + horasTrabajadas);
    System.out.println("Departamento: " + departamento);
    System.out.println("Meses de contrato: " + mesesContrato);
}
```

Consecuencia: Las subclases que heredan de Empleados, todos tienen el método imprimirDetalles() en vez de escribir el método en la superclase.

Después:

```
public class Empleado {

    Tabnine | Edit | Test | Explain | Document
    public void imprimirDetalles() {
        System.out.println("Nombre: " + getNombre());
        System.out.println("Genero: " + getNombre());
        System.out.println("Salario: " + getSalarioBase());
        System.out.println("Horas trabajadas: " + getHorasTrabajadas());
        System.out.println("Departamento: " + getDepartamento());
    }
```

```
public class EmpleadoFijo extends Empleado {

    Tabnine | Edit | Test | Explain | Document
    @Override
    public void imprimirDetalles() {
        super.imprimirDetalles();
        System.out.println("bonoAnual: " + bonoAnual);
    }
```

```
public class EmpleadoTemporario extends Empleado {
    public EmpleadoTemporario(String nombre, double salarioBase, in
    }

    Tabnine | Edit | Test | Explain | Document
    public void imprimirDetalles() {
        super.imprimirDetalles();
        System.out.println("Meses de contrato: " + mesesContrato);
    }
```

Beneficios: Reducción de línea de código y se sabe que sus subclases pueden usar el método imprimirDetalles.

Moving Features between Objects.

Move Method.

Código sin refactorizar:

```
public void imprimirDetalles() {
    System.out.println("Nombre: " + super.getNombre());
    System.out.println("Genero: " + super.getNombre());
    System.out.println("Salario: " + super.getSalarioBase());
    System.out.println("Horas trabajadas: " + super.getHorasTrabajadas());
    System.out.println("Departamento: " + super.getDepartamento());
}
```

Consecuencias:
Mantener este código sin aplicar Move Method puede generar varias consecuencias negativas. En primer lugar, imprimirDetalles() parece estar en una clase que no es la más adecuada para esta funcionalidad, ya que está accediendo directamente a datos del padre (super) que probablemente pertenecen a otra clase, lo que rompe el principio de cohesión y la responsabilidad única (SRP).. Además, si en el futuro la lógica de impresión cambia, habría que modificar esta clase en lugar de hacerlo en la clase que realmente contiene la información, lo que aumenta el riesgo de errores y duplicación de lógica en otras partes del sistema.

Código factorizado:

```
public void imprimirDetallesBase() {
    System.out.println("Nombre: " + getNombre());
    System.out.println("Salario: " + getSalarioBase());
    System.out.println("Horas trabajadas: " + getHorasTrabajadas());
    System.out.println("Departamento: " + getDepartamento());
}
```

```
public void imprimirDetalles() {
    super.imprimirDetallesBase();
    System.out.println("Genero: " + genero);
}
    You, 3 seconds ago • Uncommitted changes
```

Beneficios:

El código refactorizado con Move Method ofrece varios beneficios importantes, mejora la cohesión al ubicar la lógica de impresión básica (imprimirDetallesBase) en la clase que realmente posee los datos, lo que reduce el acoplamiento innecesario entre clases. Esto facilita el mantenimiento, ya que cualquier cambio en los atributos o en la forma de mostrarlos se realiza en un único lugar, evitando duplicaciones y errores. Además, incrementa la reutilización, permitiendo que otras clases que hereden o usen esta funcionalidad puedan invocar el método base sin necesidad de reescribir.