

**ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL**

**Diseño de Software**

***Taller 6 -Refactoring***

**INTEGRANTES:**

Perdomo Ordoñez Paul Isaac

Herrera Nieto Christian Alexander

Muñoz Sanchez Salvador Gabriel

Rosado Alcivar Enrique Gabriel

**Grupo:**

9

**Profesor:**

Jurado Mosquera David Alonso

**Periodo Académico:**

PAO II – 2024

Tabla de contenido

[**Problema Identificado #1** 3](#_Toc184936915)

[**1.** **Repetición de código de atributos en las clases de empleados** 3](#_Toc184936916)

[**2.** **Técnica sugerida** 3](#_Toc184936917)

[**3.** **Beneficio** 3](#_Toc184936918)

[**Problema Identificado #2** 3](#_Toc184936919)

[***1.*** **Uso innecesario de atributo tarifaHora** 3](#_Toc184936920)

[**2.** **Técnica sugerida** 3](#_Toc184936921)

[**3.** **Beneficio** 4](#_Toc184936922)

[**Problema Identificado #3** 4](#_Toc184936923)

[**1.** **Método buscarEmpleadoPorNombre en BuscadorEmpleado** 4](#_Toc184936924)

[**2.** **Técnica sugerida** 4](#_Toc184936925)

[**3.** **Beneficio** 4](#_Toc184936926)

[**Problema Identificado #4** 4](#_Toc184936927)

[**1.** **Uso innecesario de variable temporal en calcularSalario** 4](#_Toc184936928)

[**2.** **Técnica sugerida** 4](#_Toc184936929)

[**3.** **Beneficio** 4](#_Toc184936930)

[**Problema Identificado #5** 5](#_Toc184936931)

[**1.** **Métodos duplicados y mal ubicado** 5](#_Toc184936932)

[**2.** **Técnica sugerida** 5](#_Toc184936933)

[**3.** **Beneficio** 5](#_Toc184936934)

[**Problema Identificado #6** 5](#_Toc184936935)

[**1.** **Uso innecesario de estructura compleja en calcularSalario** 5](#_Toc184936936)

[**2.** **Técnica sugerida** 5](#_Toc184936937)

[**3.** **Beneficio** 6](#_Toc184936938)

# **Problema Identificado #1**

## **Repetición de código de atributos en las clases de empleados**

* Se repiten atributos que ya existen en la clase Empleado y EmpleadoTemporario. Si se continua con esta implantación, será difícil modificar o añadir nuevos atributos ya que se tendrá que siempre hacer en dos lugares a la vez, además se tendrán atributos duplicados lo que causa que se ocupe más memoria y sea ambiguo a cuál atributo se refiere.
* También se repite *género* en todas las clases de empleados

## **Técnica sugerida**

* Pull Up Field: Incorporar campos redundantes a la clase principal. Esto es mover mesesContrato, departamento, horasTrabajadas, salarioBase, genero a la clase Empleado.

## **Beneficio**

* Reduce la redundancia y centraliza la lógica en la clase principal, facilitando el mantenimiento y asegurando consistencia.

public class Empleado {

    private String nombre;

    private double salarioBase;

    private int horasTrabajadas;

    private String departamento;

    private String genero;

}

public class EmpleadoTemporario extends Empleado {

    private int mesesContrato;

}

public class EmpleadoFijo extends Empleado {

    private double bonoAnual;

}

public class EmpleadoPorHoras extends Empleado {

    private double tarifaHora;

}

# **Problema Identificado #2**

## **Uso innecesario de atributo tarifaHora**

* La clase principal Empleado incluye atributos como tarifaHora, que no son aplicables a todos los empleados, como los fijos o temporarios. Esto genera un modelo inconsistente, ya que la tarifa por hora no tiene sentido en un contrato mensual o temporal basado en un plazo definido.
* Mantener atributos irrelevantes en la clase principal aumenta la complejidad del diseño, hace que el modelo sea menos claro y puede llevar a confusiones al implementar o extender funcionalidades.

## **Técnica sugerida**

* **Push Down Field**: Mover atributos específicos como tarifaHora a EmpleadoPorHora

## **Beneficio**

* Facilita el mantenimiento del código, reduce la ambigüedad al evitar atributos innecesarios en clases generales y permite una extensión más sencilla y lógica del sistema, al agregar o modificar funcionalidades sin afectar a otras clases.

public class Empleado {

    private String nombre;

    private double salarioBase;

    private int horasTrabajadas;

    private String departamento;

    private String genero;

}

public class EmpleadoPorHoras extends Empleado {

    private double tarifaHora;

}

# **Problema Identificado #3**

## **Método buscarEmpleadoPorNombre en BuscadorEmpleado**

* El método buscarEmpleadoPorNombre está definido en una clase BuscadorEmpleado, pero podría estar directamente en la clase Empresa, que ya maneja la lista de empleados. Este método pertenece más naturalmente a Empresa, ya que es allí donde reside la lista de empleados.

## **Técnica sugerida**

* Move Method: se moverá el método buscarEmpleadoPorNombre de la clase BuscadorEmpleado a la clase Empresa. Esto simplifica el diseño y centraliza la gestión de los empleados en una sola clase.

## **Beneficio**

* Centraliza la lógica de los empleados en una sola clase.
* Facilita el uso del método, ya que los usuarios pueden buscar directamente en la instancia de Empresa.

public class Empresa {

    private List<Empleado> empleados;

    public Empresa() {

        empleados = new ArrayList<>();

    }

*// MOVE METHOD*

    public static Empleado buscarEmpleadoPorNombre(String nombre, List<Empleado> empleados) {

        for (Empleado empleado : empleados) {

            if (empleado.getNombre().equals(nombre)) {

                return empleado;

            }

        }

        return null;

    }

# **Problema Identificado #4**

1. **Uso innecesario de variable temporal en calcularSalario**

* El método calcularSalario utiliza una variable temporal salario que almacena el resultado de una operación simple antes de retornarlo. Este uso es redundante y no aporta claridad al código, ya que la operación es suficientemente sencilla como para ser evaluada directamente en el return.
* El uso de variables temporales innecesarias puede hacer que el código sea más verboso y dificulte su comprensión.

1. **Técnica sugerida**

* **Inline Temp**: Eliminar la variable temporal salario y retornar directamente el resultado de la operación super.getHorasTrabajadas() \* super.getTarifaHora().

1. **Beneficio**

* Simplifica el código, haciéndolo más directo y legible al evitar variables intermedias innecesarias.  
  Reduce el riesgo de errores relacionados con el manejo de variables temporales y hace que el método sea más fácil de mantener.
* *// INLINE TEMP*
* @Override
* public double calcularSalario() {
* return super.getHorasTrabajadas() \* getTarifaHora();
* }

# **Problema Identificado #5**

1. **Métodos duplicados y mal ubicado**

* El método imprimirDetalles se encuentra replicado en varias clases y realiza operaciones que deberían pertenecer a la clase base.

1. **Técnica sugerida**

* **Pull Up Method:** Subir el método imprimirDetalles a la clase base Empleado.
* **Move Method:** Reubicar lógica para respetar el principio de responsabilidad única.

1. **Beneficio**

* Reduce duplicación de lógica y asegura que las responsabilidades estén correctamente distribuidas, siguiendo principios SOLID.

public void imprimirDetalles() {

        System.out.println("Nombre: " + nombre);

        System.out.println("Salario Base: " + salarioBase);

        System.out.println("Horas Trabajadas: " + horasTrabajadas);

        System.out.println("Departamento: " + departamento);

        System.out.println("Género: " + genero);

    }

` En EmpleadoFijo

public void imprimirDetalles() {

        super.imprimirDetalles();

        System.out.println("Bono Anual: " + bonoAnual);

    }

EmpleadoPorHoras:

  public void imprimirDetalles() {

        super.imprimirDetalles();

        System.out.println("Tarifa por Hora: " + tarifaHora);

    }

EmpleadoTemporal:

public void imprimirDetalles() {

        super.imprimirDetalles(); *// Imprime los detalles comunes*

        System.out.println("Meses de Contrato: " + mesesContrato);

    }

# **Problema Identificado #6**

## **Uso innecesario de estructura compleja en calcularSalario**

* El método calcularSalario utiliza múltiples condicionales (if y switch) para calcular aspectos como las horas extra y bonos por departamento, lo que hace que el código sea rígido, difícil de extender y propenso a errores al agregar nuevos departamentos o reglas.
* Esta lógica condicional se puede reemplazar utilizando polimorfismo, delegando el cálculo a subclases específicas que representen diferentes tipos de empleados o departamentos.

1. **Técnica sugerida**

• **Replace Conditional with Polymorphism**: Utilizar polimorfismo para tratar las diferentes maneras en que el salario puede ser calculado según el tipo de empleado.  
• **Replace Conditional with Guard Clauses**: Usar cláusulas de guardia para manejar las validaciones de forma temprana

• **Extract Method**: Extraer las lógicas específicas del cálculo (como la validación de datos, cálculo de horas extra y cálculo de bono por departamento) en métodos independientes, lo que simplifica el código y facilita la reutilización.

1. **Beneficio**

**•** La utilización de polimorfismo simplifica la lógica al delegar el cálculo del salario según el tipo de empleado, lo que elimina la necesidad de condicionales.  
• Las cláusulas de guardia hacen el código más directo y fácil de seguir, mejorando la comprensión del flujo lógico y reduciendo el nivel de anidación.  
• Extraer métodos mejora el modularidad, facilita el mantenimiento y hace que el código sea más legible y testable.

public double calcularSalario() {

        double salarioTotal = salarioBase;

        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");

        }

*// Horas trabajadas normales = 40;*

        if (horasTrabajadas > 40) {

            salarioTotal += (horasTrabajadas - 40) \* 50; *// Pago de horas extra*

        }

*// return calcularBonoPorDepartamento(salarioTotal);*

        return salarioTotal;

    }

No se extrae simplemente el switch case porque no cumple OCP

public class DecoratorContabilidad extends Empleado{

    private Empleado empleado;

    public DecoratorContabilidad(Empleado empleado){

        super(empleado.getNombre(),empleado.getSalarioBase(),empleado.getHorasTrabajadas(),empleado.getDepartamento(),empleado.getGenero());

        this.empleado = empleado;

    }

    @Override

    public double calcularSalario(){

        return super.calcularSalario() + 10;

    }

}

public class DecoratorSistemas extends Empleado {

    private Empleado empleado;

    public DecoratorSistemas(Empleado empleado) {

        super(empleado.getNombre(), empleado.getSalarioBase(), empleado.getHorasTrabajadas(),

                empleado.getDepartamento(), empleado.getGenero());

        this.empleado = empleado;

    }

    @Override

    public double calcularSalario() {

        return super.calcularSalario() + 20;

    }

}