Ejercicio 2 Refactoring

2.1 Empleados

```
public class EmpleadoTemporario {
    public String nombre; // Codigo repetido con resto de empleados
    public String apellido; // Codigo repetido con resto de empleados
    public double sueldoBasico = 0; // Codigo repetido con resto de empleados
    public double horasTrabajadas = 0;
    public int cantidadHijos = 0; // Codigo repetido con EmpleadoPlanta
    //

public double sueldo() {
return this.sueldoBasico
(this.horasTrabajadas * 500)
(this.cantidadHijos * 1000)
(this.sueldoBasico * 0.13);
}
}


public class EmpleadoPlanta {
    public String nombre; // Codigo repetido con resto de empleados
    public String apellido; // Codigo repetido con resto de empleados
    public double sueldoBasico = 0; // Codigo repetido con resto de empleados
    public int cantidadHijos = 0; // Codigo repetido con EmpleadoTemporario
    //

    public double sueldo() {
        return this.sueldoBasico
                + (this.cantidadHijos * 2000)
                - (this.sueldoBasico * 0.13);
    }
}

public class EmpleadoPasante {
    public String nombre; //
```

```
    public String apellido; //
    public double sueldoBasico = 0; //
    // Codigo repetido con los otros dos empleados

    public double sueldo() {
        return this.sueldoBasico - (this.sueldoBasico * 0.13);
    }
}
```

Mal olor: **Código duplicado** entre las distintas variables de los empleados (indicado en el código). Refactor a aplicar: **Extract Superclass** y **Pull Up Field**.

Mal olor: Violate Intimacy, TODAS las variables de instancia son públicas lo que rompe encapslamiento. Refactor: **Encapsulate Field**.

```
public abstract class Empleado {
    private String nombre;
    private String apellido;
    private String sueldoBasico = 0;

}

public class EmpleadoTemporario extends Empleado{
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0; // Codigo repetido con EmpleadoPlanta

    public double sueldo() { // Logica de sueldo basico duplicada con los otros
    return this.sueldoBasico
        + (this.horasTrabajadas * 500)
        + (this.cantidadHijos * 1000)
        - (this.sueldoBasico * 0.13);
    }
}


public class EmpleadoPlanta extends Empleado{
    public int cantidadHijos = 0; // Codigo repetido con EmpleadoTemporario

    public double sueldo() { // Logica de sueldo basico duplicada con los otros
```

```
        return this.sueldoBasico
                + (this.cantidadHijos * 2000)
                - (this.sueldoBasico * 0.13);
    }
}

public class EmpleadoPasante extends Empleado{

    public double sueldo() { // Logica de sueldo basico duplicada con los otros
        return this.sueldoBasico - (this.sueldoBasico * 0.13);
    }
}
```

Mal olor: **Duplicated Code**, se puede observar en la lógica de sueldo básico con descuento y en el campo cantidadHijos de EmpleadoTemporario y EmpleadoPlanta. Refactor a aplicar: **Extract Method** y **Pull Up Method**.

Mal olor: **Long Method**, para sueldo() en EmpleadoTemporario y EmpleadoPlanta, sería mejor tener por separado lo asignado por hijo u horas. Refactor: **Extract Method**.

- También aplico un Rename Method al método de sueldo() al que se realiza el Pull Up Method, ya que el nombre sueldo() no es muy descriptivo.
- En este caso considero que solucionar el Duplicated Code en la variable de instancia de EmpleadoTemporario y Empleado Planta complejizaría el código por una única variable de instancia (ya que habría que hacer otro Extract Superclass que extienda a Empleado). Me parece mejor dejarlo con ese mal olor. También porque la lógica de extra por hijo difiere entre ambas.

```
public abstract class Empleado {
    private String nombre;
    private String apellido;
    private String sueldoBasico = 0;

    public double sueldoConDescuento() {
        return this.sueldoBasico - (this.sueldoBasico * 0.13);
    }

}
```

```
public class EmpleadoTemporario extends Empleado{
   private double horasTrabajadas = 0;
   private int cantidadHijos = 0;

  @Override
   public double sueldo() {
      return this.sueldoConDescuento()
                  + this.extraPorHoras()
                  + this.extraPorHijos();
   }

   private double extraPorHoras() {
      return this.horasTrabajadas * 500;
   }

   private double extraPorHijos() {
      return this.cantidadHijos * 1000;
   }
}


public class EmpleadoPlanta extends Empleado{
   public int cantidadHijos = 0;

   public double sueldo() {
      return this.sueldoConDescuento()
               + this.extraPorHijos();
   }

   private double extraPorHijos() {
      return this.cantidadHijos * 2000;
   }
}

public class EmpleadoPasante extends Empleado{
  // ...
}
```

Mal olor: **Lazy Class**, en EmpleadoPasante ya que toda su lógica se implementó en la superclase. Refactor necesario (es un refactoring?): Se elimina la clase EmpleadoPasante y la superclase Empleado deja de ser abstracta

```
public class Empleado {
    private String nombre;
    private String apellido;
    private String sueldoBasico = 0;

    public double sueldoConDescuento() {
        return this.sueldoBasico - (this.sueldoBasico * 0.13);
    }

}

public class EmpleadoTemporario extends Empleado{
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0;

  @Override
    public double sueldo() {
        return this.sueldoConDescuento()
                    + this.extraPorHoras()
                    + this.extraPorHijos();
    }

    private double extraPorHoras() {
        return this.horasTrabajadas * 500;
    }

    private double extraPorHijos() {
        return this.cantidadHijos * 1000;
    }
}


public class EmpleadoPlanta extends Empleado{
```

```
    public int cantidadHijos = 0;

    public double sueldo() {
        return this.sueldoConDescuento()
                + this.extraPorHijos();
    }

    private double extraPorHijos() {
        return this.cantidadHijos * 2000;
    }
}

public class EmpleadoPasante extends Empleado{
  // ...
}
```

2.2 Juego

```
public class Juego {
    public void incrementar(Jugador j) {
        j.puntuacion = j.puntuacion + 100;
    }
    public void decrementar(Jugador j) {
        j.puntuacion = j.puntuacion - 50;
    }
}

public class Jugador {
    public String nombre;
    public String apellido;
    public int puntuacion = 0;
}
```

Mal olor: Romper encapsulamiento, en las variables de instancia de Jugador, lo que rompe el encapsulamiento. Refactoring: **Encapsulate Field**.

- Primero habría que solucionar la envidia, porque en este paso intermedio el código no funcionaría (el Juego estaba accediendo a las variables de instancia sin getter ni setter).

```
public class Juego {
    public void incrementar(Jugador j) {
        j.puntuacion = j.puntuacion + 100;
    }
    public void decrementar(Jugador j) {
        j.puntuacion = j.puntuacion - 50;
    }
}

public class Jugador {
    private String nombre;
    private String apellido;
    private int puntuacion = 0;
}
```

Mal olor: **Feature Envy**, en los métodos de incrementar(jugador) y decrementar(jugador) de la clase Juego, ya que esa lógica debería ser manejada dentro de Jugador. Refactoring a aplicar: Extract Method? y Move Method.

Mal olor: Nombre de métodos poco explicativos, para el incrementar y decrementar. Refactoring: **Rename Method**.

```
public class Juego {
    public void incrementarPuntuacion(Jugador j) {
        j.incrementarPuntacion();
    }
    public void decrementarPuntacion(Jugador j) {
        j.decrementarPuntuacion();
    }
}
```

```
public class Jugador {
    private String nombre;
    private String apellido;
    private int puntuacion = 0;

    public void incrementarPuntuacion() {
        this.puntuacion += 100;
    }

    public void decrementarPuntuacion() }
         this.puntuacion -= 50;
}
```

2.3 Publicaciones

```
/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPosts(Usuario user, int cantidad) {

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    for (Post post : this.posts) {
        if (!post.getUsuario().equals(user)) {
            postsOtrosUsuarios.add(post);
        }
    }

   // ordena los posts por fecha
   for (int i = 0; i < postsOtrosUsuarios.size(); i++) {
      int masNuevo = i;
      for(int j= i +1; j < postsOtrosUsuarios.size(); j++) {
          if (postsOtrosUsuarios.get(j).getFecha().isAfter(
     postsOtrosUsuarios.get(masNuevo).getFecha())) {
```

```
            masNuevo = j;
          }
        }
      Post unPost = postsOtrosUsuarios.set(i,postsOtrosUsuarios.get(masNuevo
      postsOtrosUsuarios.set(masNuevo, unPost);
    }

    List<Post> ultimosPosts = new ArrayList<Post>();
    int index = 0;
    Iterator<Post> postIterator = postsOtrosUsuarios.iterator();
    while (postIterator.hasNext() &&  index < cantidad) {
        ultimosPosts.add(postIterator.next());
    }
    return ultimosPosts;
}
```

Bad smell: Método largo, comentarios. Refactoring a aplicar: **Extract Method** en 3 que realizan el trabajo y eliminar comentarios.

Mecánica:

- Se crea el nuevo método, con el código copiado.
- Se controlar que no haya problemas con el acceso a variables, o si hay que crear un temp.
- Se compila, se ejecuta y testea.

```
public List<Post> ultimosPosts(Usuario user, int cantidad) {
    List<Post> postsOtrosUsuarios = getPostsOtrosUsuarios(user);

    postsOtrosUsuarios = ordenarPorFecha(postsOtrosUsuarios);

    List<Post> ultimosPosts = getUltimosPosts(postsOtrosUsuarios, cantidad);
}

private List<Post> getPostsOtrosUsuarios(Usuario user) {
        List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    for (Post post : this.posts) {
        if (!post.getUsuario().equals(user)) {
            postsOtrosUsuarios.add(post);
```

```
        }
    }
    return postsOtrosUsuarios;
}

private List<Post> ordenarPorFecha(List<Post> postsOtrosUsuarios) {
    for (int i = 0; i < postsOtrosUsuarios.size(); i++) {
       int masNuevo = i;
       for(int j= i +1; j < postsOtrosUsuarios.size(); j++) {
          if (postsOtrosUsuarios.get(j).getFecha().isAfter(
     postsOtrosUsuarios.get(masNuevo).getFecha()) ) {
             masNuevo = j;
          }
       }
      Post unPost = postsOtrosUsuarios.set(i,postsOtrosUsuarios.get(masNuevo
      return postsOtrosUsuarios.set(masNuevo, unPost);
   }
}

private List<Post> getUltimosPosts(List<Post> postsOtrosUsuarios, int cantida
    List<Post> ultimosPosts = new ArrayList<Post>();
    int index = 0;
    Iterator<Post> postIterator = postsOtrosUsuarios.iterator();
    while (postIterator.hasNext() &&  index < cantidad) {
        ultimosPosts.add(postIterator.next());
    }
    return ultimosPosts;
}
```

Bad smell: Reinventando la rueda, Long Class, Cadena de mensajes, Duplicated Code (las 3 iteraciones realizan cosas parecidas). Refactor a aplicar: Replace Loop With Pipeline, y un Extract Method. Se crea un método a partir de los 4. Mecánica:

- Se identifican los bucles de for y while.
- Se reescribe la lógica usando Streams.
- Se agrupa la lógica de los 3 métodos en el mismo streams, concatenando los pipelines.

- Se compila, se ejecuta y se testea. Si todo funciona bien se eliminan los otros métodos.

```
public List<Post> ultimosPosts(Usuario user, int cantidad) {
    return this.posts.stream
        .filter(p → !p.getUsuario().equals(user)) // filtro
        .sorted((p1, p2) → p1.getFecha().compareTo(p2.getFecha())) // ordenamie
        .limit(cantidad) // limitar cantidad
         .collect(Collectors.toList());
}
```

2.4 Carrito de Compras

```
public class Producto {
    private String nombre;
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}

public class ItemCarrito {
    private Producto producto;
    private int cantidad;

    public Producto getProducto() {
        return this.producto;
    }

    public int getCantidad() {
        return this.cantidad;
    }
```

```
}

public class Carrito {
    private List<ItemCarrito> items;

    public double total() {
            return this.items.stream()
                .mapToDouble(item → item.getProducto().getPrecio() * item.getCant
                .sum();
    }
}
```

Mal olor: **Lazy Class** de la clase Producto (también podría ser considerada **Data Class**). Refactor a aplicar: Se mueven los campos de Producto a Item.

- Si se proyecta lógica futura en Producto se podría mantener el mal olor, ya que está justificado. Pero en el código dado Producto no aporta lógica.

Mal olor: **Feature Envy**(Carrito está calculando el precio del item a partir de su cantidad). Refactor a aplicar: **Extract Method/Move Method**, se crea un método en Item que realice el cálculo. Mecánica:

- Se crea el nuevo método, con el código.
- Se controlar que no haya problemas con el acceso a variables, o si hay que crear un temp.
- Se compila, se ejecuta y testea.

```
public class ItemCarrito {
    private String nombre;
    private double precio;
    private int cantidad;

    public int getCantidad() {
        return this.cantidad;
    }

    public double precioItem() {
        return this.cantidad * this.precio;
    }
```

```
}

public class Carrito {
   private List<ItemCarrito> items;

   public double total() {
         return this.items.stream()
            .mapToDouble(item → item.precioItem())
            .sum();
   }
}
```

2.5 Envío de Pedidos

```
public class Supermercado {
  public void notificarPedido(long nroPedido, Cliente cliente) {
    String notificacion = MessageFormat.format(“Estimado cliente, se le inform

    // lo imprimimos en pantalla, podría ser un mail, SMS, etc..
     System.out.println(notificacion);
    }
}

public class Cliente {
  public String getDireccionFormateada() {
      return
         this.direccion.getLocalidad() + “, ” +
         this.direccion.getCalle() + “, ” +
         this.direccion.getNumero() + “, ” +
         this.direccion.getDepartamento();
}
```

Mal olor: **Feature Envy**, Cliente hace todo el manejo del string pidiéndole todos los campos a dirección, cuando ese formato tendría que hacerlo Dirección. Refactor: Move Method a Dirección.

Mal olor: Variable temporal notificación. Refactor: Replace Temp With Query, para que se pueda reutilizar el string de notificacion en los diferentes formatos.

```
public class Supermercado {
  protected String notificacion(long nroPedido, Cliente cliente) {
   return MessageFormat.format("Estimado cliente, se le informa que hemos
   }

   public void imprimirNotificacion(long nroPedido, Cliente cliente) {
      System.out.println(this.notificacion(nroPedido, cliente));
   }
}

public class Cliente {
  private Direccion direccion;

   public String getDireccionString() {
      return this.direccion.formatearString();
   }
}

public class Direccion {
   private String localidad;
   private String calle;
   private int numero;
   private String departamento;

   public String formatearString() {
      return
         this.localidad() + ", " +
         this.calle() + ", " +
         this.numero() + ", " +
         this.departamento();
```

```
    }
}
```

2.6 Películas

```
public class Usuario {
    String tipoSubscripcion;

    public void setTipoSubscripcion(String unTipo) {
     this.tipoSubscripcion = unTipo;
    }

    public double calcularCostoPelicula(Pelicula pelicula) {
     double costo = 0;
     if (tipoSubscripcion=="Basico") {
         costo = pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
     }
     else if (tipoSubscripcion== "Familia") {
         costo = (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()) *
     }
     else if (tipoSubscripcion=="Plus") {
         costo = pelicula.getCosto();
     }
     else if (tipoSubscripcion=="Premium") {
         costo = pelicula.getCosto() * 0.75;
     }
     return costo;
    }
}

public class Pelicula {
    LocalDate fechaEstreno;
    // ...
```

```
    public double getCosto() {
     return this.costo;
    }

    public double calcularCargoExtraPorEstreno(){
    // Si la Película se estrenó 30 días antes de la fecha actual, retorna un cargo
        return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) )
    }
}
```

Bad smell: Romper encapsulamiento en las v.i de Usuario y de Pelicula.
Refactor: Encapsulate Field para ambas clases.

Bad smell: **Switch Statement**(en este caso If), en calcularCostoPelicula().
Refactoring a aplicar: **Replace Conditional With Polymorphism**. Mecánica:

- Se crea la jerarquía de clases.
- En este caso no es necesario aplicar un Extract Method.
- Por cada subclase se copia el código del condicional en un método que sobreescribe al calcularCostoPelicula(Pelicula).

```
public class Usuario {
    private TipoSuscripcion tipoSuscripcion;
    private String tipoSuscripcionString;

    public void setTipoSuscripcion(String unTipo, TipoSuscripcion tipoSuscripc
     this.tipoSuscripcion = tipoSuscripcion;
     this.tipoSuscripcionString = unTipo;
    }

    public double calcularCostoPelicula(Pelicula pelicula) {
      if (this.tipoSuscripcion != null) {
          return this.tipoSuscripcion.calcularCostoPelicula(pelicula);
      }

     double costo = 0;
     if (tipoSubscripcionString=="Basico") {
         costo = pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
     }
```

```
    else if (tipoSubscripcionString== "Familia") {
        costo = (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()) *
    }
    else if (tipoSubscripcionString=="Plus") {
        costo = pelicula.getCosto();
    }
    else if (tipoSubscripcionString=="Premium") {
        costo = pelicula.getCosto() * 0.75;
    }
    return costo;
   }
}

public class Pelicula {
     private LocalDate fechaEstreno;
   // ...

   public double getCosto() {
    return this.costo;
   }

   public double calcularCargoExtraPorEstreno(){
   // Si la Película se estrenó 30 días antes de la fecha actual, retorna un cargo
      return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) )
   }
}

public class TipoSuscripcion {
   public double calcularCostoPelicula(Pelicula pelicula);
}

public class SuscripcionBasica extends TipoSuscripcion {
   return pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
}

public class SuscripcionFamiliar extends TipoSuscripcion {
   return (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()) * 0.9
}
```

```
public class SuscripcionPlus extends TipoSuscripcion {
   return pelicula.getCosto();
}

public class SuscripcionPremium extends TipoSuscripcion {
   return pelicula.getCosto() * 0.75;
}
```

- Una vez testeado, funcionando correctamente se el borra el condicional de la clase Usuario y el campo agregado para testear.
- Una vez eliminados todos los condicionales se testa todo y se pone el método de la superclase como abstracto.

```
public class Usuario {
   private TipoSuscripcion tipoSuscripcion;

   public void setTipoSuscripcion(TipoSuscripcion tipoSuscripcion) {
    this.tipoSuscripcion = tipoSuscripcion;
   }

   public double calcularCostoPelicula(Pelicula pelicula) {
         return this.tipoSuscripcion.calcularCostoPelicula(pelicula);
      }
}

public class Pelicula {
   private LocalDate fechaEstreno;
   // ...

   public double getCosto() {
    return this.costo;
   }

   public double calcularCargoExtraPorEstreno(){
   // Si la Película se estrenó 30 días antes de la fecha actual, retorna un cargo
      return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) )
   }
```

```
}

public abstract class TipoSuscripcion {
    public abstract double calcularCostoPelicula(Pelicula pelicula);
}

public class SuscripcionBasica extends TipoSuscripcion {
    return pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
}

public class SuscripcionFamiliar extends TipoSuscripcion {
    return (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()) * 0.9
}

public class SuscripcionPlus extends TipoSuscripcion {
    return pelicula.getCosto();
}

public class SuscripcionPremium extends TipoSuscripcion {
    return pelicula.getCosto() * 0.75;
}
```

Bad smell: Feature Envy, al sumarle al costo el extra. Tendría que ser manejado por película. Refactoring: Move Method a Pelicula.

- Otro refactoring que se hizo en el medio es el de eliminar comentarios.

```
public class Usuario {
    private TipoSuscripcion tipoSuscripcion;

    public void setTipoSuscripcion(TipoSuscripcion tipoSuscripcion) {
     this.tipoSuscripcion = tipoSuscripcion;
    }

    public double calcularCostoPelicula(Pelicula pelicula) {
          return this.tipoSuscripcion.calcularCostoPelicula(pelicula);
        }
}
```

```
public class Pelicula {
   private LocalDate fechaEstreno;
   // ...

   public double getCosto() {
    return this.costo;
   }

   private double calcularCargoExtraPorEstreno(){
      return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) )
   }

   public double calcularCostoConExtra() {
      return this.costo + this.calcularCargoExtraPorEstreno();
   }
}

public abstract class TipoSuscripcion {
   public double calcularCostoPelicula(Pelicula pelicula);
}

public class SuscripcionBasica extends TipoSuscripcion {
   return pelicula.calcularCostoConExtra();
}

public class SuscripcionFamiliar extends TipoSuscripcion {
   return pelicula.calcularCostoConExtra(); * 0.90
}

public class SuscripcionPlus extends TipoSuscripcion {
   return pelicula.getCosto();
}

public class SuscripcionPremium extends TipoSuscripcion {
   return pelicula.getCosto() * 0.75;
}
```