Refactoring

1.1 Protocolo del Cliente

Mal olor en los nombres de los métodos. Ya que se requiere de comentarios para entender con claridad el funcionamiento de los métodos.

lmtCrdt(), mtFcE(), mtCbE() son métodos que, si el desarrollador no hubiera puesto los comentarios, no se habria entendido al instante para qué sirve dicho método.

1.2 Participación en proyectos

El cambio que se realizó fué Move Method, se aplicó dicho método de refactoreo por qe la clase persona estaba usando un servicio el cual estaba definido en una clase diferente a la suya (Hay envidia de atributos, se Persona se muestra envidiosa de las capacidades de proyecto), por lo tanto se aplicó Move Method.

1.3 Cálculos

Los code smells que noto son: Métodos largos

En primer lugar,, se deberia utilizar replace temp with query para las variables temporales, ya que hay variables temporales innecesarias.
Despues Extract method ya que dicho método lo que hace es recorrer una lista, sumar los datos, despues crear un string y luego imprimir. MUCHAS COSAS HACE.

Por lo tanto se deberia utilizar Extract Method para cada responsabilidad del imprimirValores() y que ese mismo método solo se dedique a imprimirValores, o los valores que se necesite en el momento invocando OTROS metodos que se encarguen de hacer el laburo correspondiente.

2.1 Empleados

Bad Smells:

Código duplicado: Hay que aplicar Pull Up method y hacer un Template Method

i. Mal olor:
    i. Código duplicado: Las clases EmpleadoTemporario, EmpleadoPlanta y EmpleadoPasante repiten los campos nombre, apellido, sueldoBásico y cantidad de hijos, y la fima del método sueldo().
ii. Refactoring:
    i. Pull Up Method: Extraer los campos similares y la firma de `sueldo()` a una clase abstracta `Empleado`
iii. Código Refactorizado:

```
public abstrct class EmpleadoTemporario {
   public String nombre;
   public String apellido;
   public double sueldoBasico = 0;
   // ......

  public Emplado (String nombre, String apellido, double sueldoBasico) {
     this.nombre = nombre;
     this.apellido = apellido;
     this.sueldoBasio = sueldoBasico;

  }

public abstract double sueldo();


// Cada subclase ya sólo define su lógica particular
public class EmpleadoTemporario extends Empleado {
   private double horasTrabajadas;
   private int    cantidadHijos;

   public EmpleadoTemporario(String nombre, String apellido,
                             double sueldoBasico,
                             double horasTrabajadas,
                             int cantidadHijos) {
      super(nombre, apellido, sueldoBasico);
      this.horasTrabajadas = horasTrabajadas;
      this.cantidadHijos   = cantidadHijos;
   }

   @Override
   public double sueldo() {
      return this.sueldoBasico
         + (this.horasTrabajadas * 500)
```

```
            + (this.cantidadHijos   * 1000)
            + (this.sueldoBasico    * 0.13);
    }
}

public class EmpleadoPlanta extends Empleado {
    private int cantidadHijos;

    public EmpleadoPlanta(String nombre, String apellido,
                          double sueldoBasico, int cantidadHijos) {
        super(nombre, apellido, sueldoBasico);
        this.cantidadHijos = cantidadHijos;
    }

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

public class EmpleadoPasante extends Empleado {
    public EmpleadoPasante(String nombre, String apellido,
                           double sueldoBasico) {
        super(nombre, apellido, sueldoBasico);
    }

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

Iteración 2

Tras extraer la superclase, existe otro mal olor:

i. Mal olor
    i. Código duplicado en el cálculo del impuesto (sueldoBasico * 0.13) repetido en EmpleadoPlanta y Pasante
ii. Refactoring:
    i. Extract Method: Mover la parte esa del cálculo a un método que sea común llamado calcular impuesto en la superclase.
iii. Código refactorizado:

```
public abstract class Empleado {
    protected String nombre;
    protected String apellido;
    protected double sueldoBasico;

    public Empleado(String nombre, String apellido, double sueldoBasico) {
        this.nombre       = nombre;
        this.apellido     = apellido;
        this.sueldoBasico = sueldoBasico;
    }

    // plantilla de cálculo: cada subclase define solo sus diferencias
    public abstract double sueldo();

    // extraído aquí para reutilizar
    protected double calcularImpuesto() {
        return this.sueldoBasico * 0.13;
    }
}

public class EmpleadoPlanta extends Empleado {
    private int cantidadHijos;
    public EmpleadoPlanta(/*...*/) { /* igual */ }
```

```
    @Override
    public double sueldo() {
        double base = this.sueldoBasico + (this.cantidadHijos * 2000);
        return base - calcularImpuesto();
    }
}

public class EmpleadoPasante extends Empleado {
    public EmpleadoPasante(/*...*/) { /* igual */ }

    @Override
    public double sueldo() {
        return this.sueldoBasico - calcularImpuesto();
    }
}
```

Iteración 3

i. Aparecen todavía ”Números mágicos” aunque no se mencione ese mal olor en el material, es un mal olor bastante común

ii. Refactoring:

- Replace Magic Number with Symbolic Constant: Declarar constantes que sean descriptivas en sus nombres

iii. Código final:

```
public abstract class Empleado {
    protected static final double TASA_IMPUESTO = 0.13;

    protected String nombre;
    protected String apellido;
    protected double sueldoBasico;
    // ...
```

```
    protected double calcularImpuesto() {
        return this.sueldoBasico * TASA_IMPUESTO;
    }
}

public class EmpleadoTemporario extends Empleado {
    private static final double TASA_HORA_EXTRA    = 500;
    private static final double TASA_HIJO_EXTRA    = 1000;
    // ...

    @Override
    public double sueldo() {
        return this.sueldoBasico
            + (this.horasTrabajadas * TASA_HORA_EXTRA)
            + (this.cantidadHijos   * TASA_HIJO_EXTRA)
            + (this.sueldoBasico    * TASA_IMPUESTO); // si es extra
    }
}
```

2.2 Juego

i. Bad Smell:

- Envidia de atributos (Feature Envy): La clase Juego posee métodos que puede hacer directamente la clase Jugador por si mismo. Por ejemplo el método incrementar y decrementar.

ii. Refactoring:

- Extract Method: Mover esa parte del cálculo incrementar y decrementar e implementar esos dos métodos dentro de la clase Jugador.

iii. Código final:

```
public class Juego {
    public void incrementar(Jugador j) {
        j.incrementarPuntuacion();
    }

    public void decrementar(Jugador j) {
        j.decrementarPuntuacion();
    }
}
public class Jugador {
    public String nombre;
    public String apellido;
    public int puntuacion = 0;

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

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

}
```

Iteración 2

Despues de haber movido los métodos, encontramos otro mal olor:

(i) Mal Olor:

- Data Class: Jugador expone de manera directa puntuación y otros datos en campos públicos

(ii) Refactoring:

- Encapsulate Field: Convertir el campo público en privado y exponer sólo la operación necesaria.

(iii) Código resultante:

```
public class Juego {
    public void incrementar(Jugador j) {
        j.incrementarPuntuacion();
    }

    public void decrementar(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;
    }

    public int getPuntuacion() {
        return this.puntuacion;
    }

}
```

Iteración 3

Ultimo mal olor:

(i) Mal olor:

- Magic Numbers: Otra vez se encuentran los numeros mágicos, pero en este caso son 100 y 50, no posee ninguna explicación de estos números

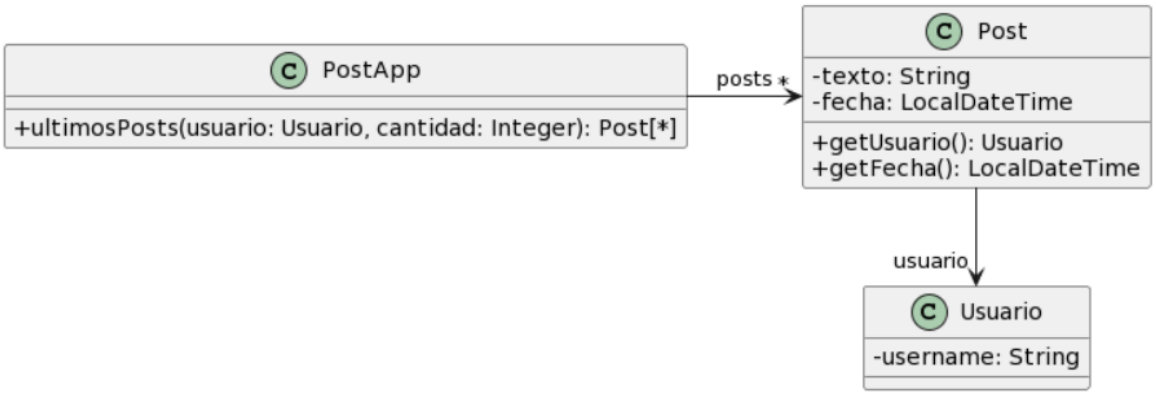
(ii) Refactoring:

- Replace Magic Numbers With Symbolic Constant

(iii) Código Final:

```
public class Juego {
    public void incrementar(Jugador j) {
        j.incrementarPuntuacion();
    }

    public void decrementar(Jugador j) {
        j.decrementarPuntuacion();
    }
}
public class Jugador {
    private static final int PUNTOS_INCREMENTO = 100;
    private static final int PUNTOS_DECREMENTO = 50;
    private String nombre;
    private String apellido;
    private int puntuacion = 0;

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

    public void decrementarPuntuacion() {
        this.puntuacion-= PUNTOS_DECREMENTO;
    }

    public int getPuntuacion() {
        return this.puntuacion;
    }
}
```

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;
}
```

(i) Bad Smell:

- Long Method: Este método abarca muchas tareas, como por ejemplo el filtrar, ordenar y recortad cant
- Manual Sorting Algorithm: Implementa de manera clandestina un “selection sort” cuando Java ya ofrece funcionalidades para ordenar que son más eficientes.

(ii) Refactoring:

- Extract Method: Mover el método si queremos separar cada paso en su propio método.
- Replace Loop with Pipeline: Usar Streams de Java para hacer el filtrado, ordenado y limite en una sola ida.

(iii) Código Final:

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

    return posts.stream()
    // 1. Se filtra
    .filter(post → post.getUsuario.equals(user))
```

```
    // 2. Ahora se ordena por fecha de manera descendente
    .sorted(Comparator.comparing(Post::fetFecha).reversed())
    // 3. Se toma solo los primeros N posts
    .limit(cantidad)
    // 4. Por ultimo se recolecta la lista
    .collect(Collectors.ToList());
}
```

2.4 Carrito de compras

(i) Mal olor:

- Feature Envy: ItemCarrito tiene un método llamado getProducto(), esto se deberia hacer dentro de la clase Producto, no en el itemCarrito. ItemCarrito piensa saber más que Producto en si.
- Data Class: ItemCarrito parece ser simplemente una clase que posee getters/setters.

(ii) Refactoring:

- Extract Method y Move Method: Mover el método getProducto a su respectiva clase Producto o eliminarlo directamente y crear un Método que obtenga el subtotal de la cantidad de dinero que hay juntando la cantidad de items de ese producto y el precio de cada uno de ellos.

(iii) Código final:

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

    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
```

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

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

    public double getSubtotal() {
        return cantidad * producto.getPrecio();
    }

}

public class Carrito {
    private List<ItemCarrito> items;

    public Carrito(List<ItemCarrito> items) {
        this.items = items;
    }

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

2.5 Envío de pedidos

(i) Bad Smell:

- Feature Envy: El método getFireccionFormateada() está en cliente, pero acede a los datos que están dentro de dirección para formatearlo (direccion.getLocalidad() por ejemplo).
- Esto muestra que la lógica deberia estar dentro de la clase Direccion, la cual sabe exactamente como es una dirección.

(ii) Refactoring:

- Move Method: mover getDireccionFormateada() desde Cliente a Direccion.

(iii) Código refactorizado:

```
public class Supermercado {
  public void notificarPedido(long nroPedido, Cliente cliente) {
    String notificacion =
    MessageFormat.format(
    "Estimado cliente, se le informa que hemos recibido su pedido con número {0
    nroPedido, cliente.getDireccionFormateada()
    );

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

public class Cliente {
   private Direccion direccion;

   public Cliente(Direccion direccion) {
      this.direccion = direccion;
   }

  public String getDireccionFormateada() {
      return direccion.getDireccionFormateada();
  }
```

```
}

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

    public Direccion(String localidad, String calle, String numero, String departame
        this.localidad = localidad;
        this.calle = calle;
        this.numero = numero;
        this.departamento = departamento;
    }

    public String getDireccionFormateada() {
        return localidad + ", " + calle + ", " + numero + ", " + departamento;
    }
}
```

2.6 Peliculas

(i) Bad Smell:

- Switch Statements: En Usuario.calcularCostoPelicula(), hay varios if/else los cuales dependen del valor de tipoSubscripcion.
- Esto hace que si quiero agregar otro tipo de subscripción, tenga que modificar este método.

(ii) Refactoring:

- Replace Conditional with Polymorfism

(iii) Código final:

```
public class Usuario {
    private TipoSubscripcion tipoSubscripcion;
    // ...

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

    public double calcularCostoPelicula(Pelicula pelicula) {
     return tipoSubscripcion.calcularCosto(pelicula);
}

public class Pelicula {
    LocalDate fechaEstreno;
    private double costo;

    public Pelicula(LocalDate fechaEstreno, double costo) {
        this.fechaEstreno = fechaEstreno;
        this.costo = costo;
    }

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

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

}


public interface TipoSubscripcion {
```

```
    double calcularCosto(Pelicula pelicula);
}


public class Basico implements TipoSubscripcion {
    public double calcularCosto(Pelicula pelicula) {
        return pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
    }
}

public class Familiar implements TipoSubscripcion {
    public double calcularCosto(Pelicula pelicula) {
        return pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno() * 0.90;
    }
}

public class Plus implements TipoSubscripcion {
    public double calcularCosto(Pelicula pelicula) {
        return pelicula.getCosto();
    }
}

public class Premium implements TipoSubscripcion {
    public double calcularCosto(Pelicula pelicula) {
        return pelicula.getCosto() * 0.75;
    }
}
```

3.

1- Bad Smells: Duplicate Code, Variables temporales inutiles, Posible división por cero.

2-

- Para el código duplicado, aplico Extract Method:

```
public class Document {
    List<String> words;

    public long getTotalCharacters() {
        return this.words.stream().mapToLong(w -> 2.length()).sum();
    }

    public long characterCount() {
    long count = getTotalCharacters();
        return count;
    }
    public long calculateAvg() {
        long avgLength = getTotalCharacters() / this.words.size();
        return avgLength;
    }
// Resto del código que no importa
}
```

- Para las variables temporales inutiles: Replace Temp with Query:

```
public class Document {
    List<String> words;

    public long getTotalCharacters() {
        return this.words.stream().mapToLong(w -> 2.length()).sum();
    }

    public long characterCount() {
        return getTotalCharacters();
    }
    public long calculateAvg() {
        return getTotalCharacters() / this.words.size();
```

```
    }
// Resto del código que no importa
}
```

- Para la posible división por cero: Extract Method? o Reescribir el Método:

```
public class Document {
    List<String> words;

    public long getTotalCharacters() {
        return this.words.stream().mapToLong(w -> 2.length()).sum();
    }

    public long characterCount() {
        return getTotalCharacters();
    }
    public long calculateAvg() {
        if (words.isEmpty()) {
            return 0;
        }
        return getTotalCharacters() / this.words.size();
    }
// Resto del código que no importa
}
```

3. Si

Existe un problema lógico en el método `calculateAvg()` ya que si el size de la lista de Strings words es 0, entonces estaria dividiendo por cero, lo cual puede llevar a una excepción aritmética.

Si

Luego de reescribir el método, se ha resuelto el problema de la excepción:

```
if (words.isEmpty()) return 0;
```

4. Si

Se considera un refactoring completo ya que hemos cambiado la estructura del código, eliminamos la duplicación que este sufria, extrajimos algunos métodos para poder mejorar la legibilidad del código y aseguramos un método que podia lanzar una excepción en caso de que se dividiera por cero.

Sigue haciendo lo mismo el código, lo cual es escencial.

Ejercicio 4

```
public class Pedido {
    private static final double DESCUENTO = 0.9;
    private static final int EDAD_PARA_DESCUENTO = 5;
    private Cliente cliente;
    private List<Producto> productos;
    private FormaPago formaPago;


    public double aplicarDescuentoMayor5(double costoProductos, double extraF
        int añosDesdeFechaAlta = cliente.getAntiguedadAnios();
        if (añosDesdeFechaAlta > EDAD_PARA_DESCUENTO) {
            return (costoProductos + extraFormaPago) * DESCUENTO;
        }
        return costoProductos + extraFormaPago;
    }

    public Pedido(Cliente cliente, List<Producto> productos, FormaPago formaPag
        this.cliente = cliente;
        this.productos = productos;
```

```
        this.formaPago = formaPago;
    }

    public double getCostoTotal() {
        double costoProductos = 0;
        // for (Producto producto : this.productos) {
        //     costoProductos += producto.getPrecio();
        // }
        // Cambiado a:
        costoProductos += this.productos.stream().mapToDouble(x -> x.getPrecio())

        double extraFormaPago = formaPago.getExtraFormaPago(costoProductos);

        return aplicarDescuentoMayor5 (costoProductos, extraFormaPago);
    }
}

public interface FormaPago {

    public double getExtraFormaPago(double aux);
}

public class Efectivo implements FormaPago {
    public double getExtraFormaPago(double aux) {
        return 0;
    }
}

public class Cuotas_6 implements FormaPago {
    public double getExtraFormaPago(double aux) {
        return aux * 0.2;
    }
}

public class Cuotas_12 implements FormaPago {
    public double getExtraFormaPago(double aux) {
```

```
        return aux * 0.5;
    }
}

public class Cliente {
    private LocalDate fechaAlta;
    public LocalDate getFechaAlta() {
      return this.fechaAlta;
    }

    public int getAntiguedadAnios() {
        return Period.between(this.fechaAlta, LocalDate.now()).getYears();
    }
}

public class Producto {
  private double precio;
  public double getPrecio() {
    return this.precio;
  }
}
```