Practica 2 - Refactoring

Dudas

~~6 - Introduce Null Object ¿como se cuales son los metoods que debe sobreescribir el null?~~
→ Los que chequean por null

5 - ~~Revisar~~ → Ta joya

4 - ~~Revisar la herencia Replace Conditional with Polymorphism (FormasDePago) y resultado final~~
~~¿Hasta que punto debemos hacer refactoring?~~
→ Esta bien

~~¿Es correcto que marque al final aquellos Bad Smells que no solucione pq me falta contexto?~~ → No es necesario

2.6 - ~~Revisar~~ → Joya

2.1 - ~~¿es necesario aplicar otra herencia? ya que EmpleadoPlanta y EmpleadoTemporal tiene una~~
~~variables en común que es cantidad de hijos, y seria código duplicado~~
→ Tenemos 2 opciones optar segun el contexto

~~¿es correcto que el método calculoAdicional del Empleado Pasante retorne 0?~~ → Si

~~¿Utilizar un getter es bad smell~~? → No

~~¿Es un bad smell no escribir camelCase?~~ → Si

~~¿Loop Abusers == Reinventar la rueda?~~ → ¿ Se soluciona con Replace Loop with Pipeline? si ambos

~~¿Una posible division por 0 es un mal olor?~~ → No, ya que los malos olores son problemas de diseño y de estructuración mientras que esto solo es un error de lógica o de ejecución, por lo tanto funcional pero no estructural

1.3 ~~Bad Naming o Long method?~~ → Long method

2.1 ~~¿Es valido aplicar 2 refactoring juntos por que están estrechamente relacionados?~~ → Pull up field y pull up method si

2.1 ~~¿Cuando tenemos el mal olor de Data Class significa que solo tiene atributos públicos o getters/setters sin ninguna lógica y los expone directamente~~? → Sin logica

Ejercicio 1 ✅

1.1 ✅

Problema con nombres de metodos (Bad Naming) → Los nombres de los metodos son poco explicativos de su objetivo

Puede haber un problema si se le piden datos a un cliente y los metodos estan en private o protected

```
/**
- Retorna el límite de crédito del cliente
*/
public double obtenerLimiteCliente() {...
/*
- Retorna el monto facturado al cliente entre dos fechas
*/
public double montoFacturadoEntre(LocalDate fechaInicio, LocalDate fechaFi
/**
- Retorna el monto cobrado al cliente entre dos fechas
*/
public double montoCobradoEntre(LocalDate fechaInicio, LocalDate fechaFin)
```

1.2 ✅

Feature envy y Data Class→ Se realizo un Move Method del metodo "participaEnProyecto" de la clase Persona a la clase Proyecto, ya que Persona utilizaba un getter para obtener el contenido de una variable de Proyecto y operaba con ella, ademas la clase Proyecto era una Data Class ya que solo poseia 1 getter y la variable participantes. Esto debe ser responsabilidad de Proyecto, ya que es quien contiene la variable.

1.3 ✅

Long Method→ El método "imprimirValores" hace demasiadas cosas, calcula valores iterando sobre una colección "personal", resume información en un String y lo imprime. Se podrían dividir en métodos mas específicos que tengan una única responsabilidad (Extract Method)

```
public double calcularTotalEdades(){
    return (double) personal.stream().mapToInt(personal → personal.getEdad())
}

public double calcularTotalSalarios(){
    return (double) personal.stream().mapToInt(personal → personal.getSalario
}

public void imprimirValores() {
    int totalEdades = this.calcularTotalEdades();
    double totalSalarios = this.calcularTotalSalarios();
    double promedioEdades = totalEdades / personal.size();
    String message = String.format("El promedio de las edades es %s y el total
    System.out.println(message);
}
```

Variables temporales (Temporary Variables) → Ahora vemos con facilidad que las variables “promedioEdad”, “totalSalarios” y “totalEdades” son innecesarias por lo tanto aplicamos Replace Temp with Query

```
public double calcularPromedioEdades(){
    return (double) personal.stream().mapToInt(personal → personal.getEdad())
}

public double calcularTotalSalarios(){
    return (double) personal.stream().mapToInt(personal → personal.getSalario
}

public void imprimirValores() {
    String message = String.format("El promedio de las edades es %s y el total
    System.out.println(message);
}
```

Ejercicio 2 ✅

2.1 ✅

Código Duplicado → En las clases EmpleadoTemporario, EmpleadoPlanta y EmpleadoPasante vemos variables de instancia que son iguales y tienen el mismo proposito, "nombre","apellido"y "sueldoBasico" ademas vemos que todos calculan el sueldo de maneras diferentes pero siempre se resta al sueldo basico el 13%.
Es necesario aplicar una herencia con clase abstracta y que las sub-clases implementen su version del metodo calcularAdicional, esto se suma en un comportamiento comun en el metodo sueldo(). Hay que aplicar un Pull Up Method y un Pull Up Field

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

    public abstract double calculoAdicional();

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

    public double sueldo(){
        return this.sueldoBasico - this.getDescuentoDelSueldo();
    }
}

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

    public double montoHoraTrabajada(){
        return 500;
    }
```

```
    public double montoPorHijo(){
        return 1000;
    }

    public double calculoAdicional(){
        return (this.horasTrabajadas * this.montoHoraTrabajada())
                + (this.cantidadHijos * this.montoPorHijo());
    }

    @Override
    public double sueldo(){
        return super.sueldo() + this.calculoAdicional();
    }
}

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

    public double montoPorHijo(){
        return 2000;
    }

    public double calculoAdicional(){
        return (this.cantidadHijos * this.montoPorHijo());
    }

    @Override
    public double sueldo(){
        return super.sueldo() + this.calculoAdicional();
    }
}

public class EmpleadoPasante extends Empleado{
    public double calculoAdicional(){ return 0; }
}
```

Data class → Todas las clases rompen el encapsulamiento al tener sus variables de instancia en public, de esta forma se comportan como una Data

class la cual solo tiene atributos públicos o getters/setters sin ninguna lógica y expone directamente. Hay que aplicar un Encapsulate Field de forma de preservar la integridad de los datos

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

    public abstract double calculoAdicional();

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

    public double sueldo(){
        return this.sueldoBasico - this.getDescuentoDelSueldo();
    }
}

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

    public double montoHoraTrabajada(){
        return 500;
    }

    public double montoPorHijo(){
        return 1000;
    }

    public double calculoAdicional(){
        return (this.horasTrabajadas * this.montoHoraTrabajada())
                + (this.cantidadHijos * this.montoPorHijo());
    }

    @Override
```

```
    public double sueldo(){
        return super.sueldo() + this.calculoAdicional();
    }
}

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

    public double montoPorHijo(){
        return 2000;
    }

    public double calculoAdicional(){
        return (this.cantidadHijos * this.montoPorHijo());
    }

    @Override
    public double sueldo(){
        return super.sueldo() + this.calculoAdicional();
    }
}

public class EmpleadoPasante extends Empleado{
    public double calculoAdicional(){ return 0; }
}
```

2.2 ✅

Envidia de atributos (Feature Envy) → La clase Juego pide y utiliza los datos públicos de la clase Jugador, esto es incorrecto ya que la modificación de sus variables de instancia debería ser responsabilidad de la clase que las contiene. Hay que hacer un Move Method de la clase Juego a Jugador

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

```
    public void decrementar(Jugador j){
        j.decrementar();
    }
}

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

    public void incrementar(){
        puntuacion += 100;
    }

    public void decrementar(){
        puntuacion += 50;
    }
}
```

Data class → Ahora despues de haber hecho el Move Method vemos que es totalmente innecesario tener las variables de instancia de la clase Jugador en publicos ya que previamente se comportaba con una Data class. Hay que aplicar un Encapsulate Field

Malos olores no tratados → Data class Jugador y Juego podria funcionar como un Middle Man

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

public class Jugador {
    private String nombre;
```

```
    private String apellido;
    private int puntuacion = 0;

    public void incrementar(){
        puntuacion += 100;
    }

    public void decrementar(){
        puntuacion += 50;
    }
}
```

2.3 ✅

Long Method (Metodo largo) → En la clase el metodo "ultimosPost" realizar multiples operaciones ademas de retornar los ultimos post que no pertezcan a un usuario. Esto se soluciona con extraer en metodos mas chicos sus funciones (Extract Method)

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

public List<Post> ordenarPostPorFecha (List<Post> postsOtrosUsuarios){
    // 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(masNuev
        postsOtrosUsuarios.set(masNuevo, unPost);
    }
}

public List<Post> ultimosPosts(Usuario user, int cantidad) {
    List<Post> postsOtrosUsuarios = obtenerPostQueNoCorrespondanAUnUsu
    List<Post> postsOrdenados = ordenarPostPorFecha(postsOtrosUsuarios);
    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;
}
```

Envidia de atributos (Feature Envy) → La clase PostApp pide datos a la clase Post para realizar una evaluacion, esto deberia ser responsabilidad de la clase la cual tiene las variables de instancia. Por lo tanto aplicamos un Move Method.

```
public boolean noEsDelUsuario (Usuario user) {
    return !this.usuario.equal(user)
}

public List<Post> obtenerPostQueNoCorrespondanAUnUsuario(Usuario user)
    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    for (Post post : this.posts) {
        if (post.noEsDelUsuario(user)) {
            postsOtrosUsuarios.add(post);
        }
    }
    return postOtrosUsuarios;
}

public List<Post> ordenarPostPorFecha (List<Post> postsOtrosUsuarios){
```

```
// 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(masNuev
        postsOtrosUsuarios.set(masNuevo, unPost);
    }
}

public List<Post> ultimosPosts(Usuario user, int cantidad) {
    List<Post> postsOtrosUsuarios = obtenerPostQueNoCorrespondanAUnUsu
    List<Post> postsOrdenados = ordenarPostPorFecha(postsOtrosUsuarios);
    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;
}
```

Abuso bucles → El metodo "ultimosPost" hace uso de varios for para recorrer las colecciones, esto podria ser mejorado con el fin de reducir su complejidad y achicar las lineas de codigo utilizando Streams. Replace Loop with Pipeline

```
public List<Post> obtenerPostQueNoCorrespondanAUnUsuario(Usuario user)
    return posts.stream().filter(post → post.noEsDelUsuario(user).collect(Collec
}

//CREO QUE ORDENA AL REVES !!
public List<Post> ordenarPostPorFecha (List<Post> postsAOrdenar){
    return postsAOrdenar.stream().sorted((p1,p2) → p1.getFecha().compareTo(p
}
```

```
public List<Post> ultimosPosts(Usuario user, int cantidad) {
    List<Post> postsOtrosUsuarios = obtenerPostQueNoCorrespondanAUnUsu
    List<Post> postsOrdenados = ordenarPostPorFecha(postsOtrosUsuarios);
    return postOrdenados.stream().limit(cantidad).collect(Collectors.toList());
}
```

Variables locales (Temporary Variable) → Vemos que las variables “postOtrosUsuarios” y “postOrdenados” son innecesarias ya que podemos utilizar directamente las llamadas a los metodos. Replace Temp with Query

```
public List<Post> ultimosPosts(Usuario user, int cantidad) {
    List<Post> postsOtrosUsuarios = obtenerPostQueNoCorrespondanAUnUsu
    ordenarPostPorFecha(postsOtrosUsuarios);
    return postsOtrosUsuarios.stream().limit(cantidad).collect(Collectors.toList(
}
```

2.4 ✅

Envidia de atributos (Feature Envy) → El metodo “total()” de la clase Carrito solicita datos a la clase ItemCarrito y Producto para hacer un calculo, esto no deberia ser asi ya que la responsabilidad de hacer este calculo recae en IteamCarrito que es quien posee las variables de instancia

Malos olores no tratados → Item Carrito podria funcionar como un Middle Man

```
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 double total(){
      return producto.getPrecio() * cantidad;
    }
}

public class Carrito {
    private List<ItemCarrito> items;

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

2.5 ✅

Clase de datos (Data class) → La clase Dirección funciona como una clase de datos ya que posee sus variables de instancia expuestas rompiendo el encapsulamiento, debemos aplicar un Encapsulate Class para mantener la integridad de los datos.

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

Envidia de atributos (Feature Envy) → La clase Cliente con el metodo “getDireccionFormateada()” solicita y opera con datos de la clase Direccion,

esto podria mejorarse aplicando un Move Method y formateando los datos directamente en la clase Dirección que es la que tiene los datos.

Malos olores no tratados → Data class Direccion y Cliente.

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

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

public class Cliente {
    private Direccion direccion;
    public String getDireccionFormateada() {
        return this.direccion.toString()
    }
}

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

    public String toString(){
        return localidad + “, ” + calle + “, ” + numero + “, ” + departamento;
    }
}
```

2.6 ✅

Switch Statements (o if’s) → Utilizar ifs anidados para determinar el costo de la película basado en el tipo de suscripción y tener las descripciones mediante nombre es un problema ya que podría resolverse aplicando polimorfismo (Replace Conditional with Polymorphism)

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

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

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

public abstract class TipoSubscripcion{

    public abstract double porcentajeExtra()

    public double getPrecio(Pelicula peli){
        return peli.getCosto() * this.porcentajeExtra();
    }
}

public class Basica extends TipoSubscripcion{
    public double porcentajeExtra(){
        return 1;
    }
    @Override
    public double getPrecio(Pelicula peli){
        return peli.getCosto() + peli.calcularCargoExtraPorEstreno();
    }
}

public class Familia extends TipoSubscripcion{
    public double porcentajeExtra(){
        return 0.9;
    }
    @Override
    public double getPrecio(Pelicula peli){
        return super.getPrecio() + pelicula.calcularCargoExtraPorEstreno());
```

```
    }
}

public class Plus extends TipoSubscripcion{
    public double porcentajeExtra(){
        return 1;
    }
    @Override
    public double getPrecio(Pelicula peli){
        return pelicula.getCosto();
    }
}

public class Premium extends  TipoSubscripcion{
    public double porcentajeExtra(){
        return 0.75;
    }
    @Override
    public double getPrecio(Pelicula peli){
        return pelicula.getCosto()  * 0.75;
    }
}

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 c
        return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) )
  }
}
```

Ejercicio 3 ✅

Variable temporal (Temporary Variable) → En el método characterCount() y calculateAvg() de la clase Documente se esta usando una variable que es innecesaria a la hora de hacer el calculo, debemos aplicar un Remplace Temp with Query para que directamente se retorne el resultado.

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

   public long characterCount() {
         return this.words.stream()
                     .mapToLong(w → w.length())
                     .sum();
      }
   public long calculateAvg() {
      return this.words.stream()
               .mapToLong(w → w.length())
               .sum() / this.words.size();
         }
// Resto del código que no importa
}
```

Hay un error ya que en el caso que this.words.size() == 0 se puede producir un error en tiempo de ejecución, debería agregarse un if que evalua que si this.words.size() es 0 no se haga el calculo.

Si hay un error en la lógica mediante la cual se efectúa un calculo no es tarea del Refactoring corregir esto, ya que el refactoring esta enfocado en solucionar malos olores de diseño y no logicos y propios de la programación

Ejercicio 4 ✅

Abuso de loops (Loop Abusers) → En las líneas 16 hasta la 19 de la clase Pedido se utiliza un for para realizar la sumatoria de los precios de los productos, esto puede ser solucionado aplicando un Replace Loop with Pipeline, es decir aplicando un stream.

```
double costoProductos = this.productos.stream().mapToDouble(Producto::get
```

Switch Statements (Exceso de condicionales) → En las lineas 21 a 27 se utilizan if’s para determinar el costo final de los productos segun el metodo de pago, lo cual es un gran problema ya que en el caso de agregar mas metodos de pago hay que agregar mas if’s, para solucionar esto aplicamos Replacer Conditional with Polymorphism.

Este refactoring se realiza en tres partes:

- Crear la jerarquia de clases, de acuerdo al tipado que se usa en el switch. Esto es, crear una clase abstracta FormaDePago, y tres subclases Efectivo, SeisCuotas y DoceCuotas
- Crear un método polimotfico en la clase FormaDePago que contenga la lógica que varía segun el switch en cada subclase
- Reemplazar el switch por la invocación al método polimorfico

[Hasta ahi. Despues si tenes que hacer mas refactorings porque te quedó un choclo en el metodo polimorfico, poné aparte otro bad smell y refactorizas eso]

```
public abstract class FormaDePago {
    public extraFormaDePago(double costoProductos){
        return this.porcentaje() * costoProductos
    }
    public abstract porcentaje();
}

public class Efectivo extends FormaDePago{
    public double porcentaje(){
        return 0;
    }
}

public class SeisCuotas extends FormaDePago{
    public double porcentaje(){
        return 0.2;
    }
}

public class DoceCuotas extends FormaDePago{
```

```
    public double porcentaje(){
        return 0.5;
    }
}

//Clase Pedido
//Variable de instancia
private FormaDePago formaPago;
//... metodo getCostoTotal() clase Pedido
double extraFormaPago = this.formaPago.extraFormaDePago(costoProductos
```

Metodo largo (Long method) → En la linea 28 de la clase Pedido dentro del metodo getCostoTotal() vemos que se calculan los años de antigüedad del cliente, no tiene sentido que este calculo se haga dentro del metodo. Aplicamos un Extract Method.

```
public int obtenerAntiguedadCliente(){
    return Period.between(this.cliente.getFechaAlta(), LocalDate.now()).getYea
}
```

Envidia de atributos (Feature Envy) → Una vez que extraemos el metodo podemos ver de mejor manera que se estan pidiendo datos del cliente, como es su fecha de alta, para realizar un calculo en una clase distinta a la del cliente. Aplicamos un Move method y movemos la responsabilidad del calculo a la clase Cliente quien debe ser la encargada de hacer este calculo ya que posee las variables.

```
public class Cliente {
    private LocalDate fechaAlta;

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

    public int obtenerAntiguedadCliente(){
        return Period.between(this.getFechaAlta(), LocalDate.now()).getYears();
    }
}
```

```
//Clase Pedido
//... metodo getCostoTotal() clase Pedido
    int añosDesdeFechaAlta = cliente.obtenerAntiguedadCliente()
    if (añosDesdeFechaAlta > 5) {
        return (costoProductos + extraFormaPago) * 0.9;
    }
        return costoProductos + extraFormaPago;
}
```

Metodo largo (Long method) → Volvemos a ver ahora que el calculo de la antigüedad del cliente en el metodo getCostoTotal() de la clase pedido se podria extraer de forma tal que se quede en un metodo, para esto aplicamos Extract Method

```
public double descuentoSiClienteMayorACincoAnos(){
    int añosDesdeFechaAlta = cliente.obtenerAntiguedad()
    if (añosDesdeFechaAlta > 5) {
        return (costoProductos + extraFormaPago) * 0.9;
    }
        return costoProductos + extraFormaPago;
}
```

Feature Envy (Envidia de atributos) → Podemos observar que se están solicitando los años de antigüedad a la clase Cliente para después en base al valor determinar si se aplica un descuento o no, esto debería ser responsabilidad del Cliente, y devolver un booleano indicando si esta apto o no. Aplicamos un Move Method

```
public class Cliente {
    private LocalDate fechaAlta;

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

    public int obtenerAntiguedad(){
        return Period.between(this.getFechaAlta(), LocalDate.now()).getYears();
```

```
    }

    public boolean antiguedadMayorACinco(){
        return this.obtenerAntiguedad() > 5;
    }
}

public double descuentoSiClienteMayorACincoAnos(){
    boolean cumple = cliente.antiguedadMayorACinco();
    if (cumple) {
        return (costoProductos + extraFormaPago) * 0.9;
    }
        return costoProductos + extraFormaPago;
}
```

Variables temporal (Temporary Variable) → Ahora podemos ver con mas facilidad que la variable “cumple” es innecesaria para el calculo y podemos utilizar directamente el metodo del cliente “antiguedadMayorACinco()”. Aplicamos un Replace Temp with Query.

```
public double descuentoSiClienteMayorACincoAnos(double costoProductos, d
    if (cliente.antiguedadMayorACinco()){
        return (costoProductos + extraFormaPago) * 0.9;
    }
    return costoProductos + extraFormaPago;
}
```

Numeros magicos (Magic numbers) → Los valores del descuento estan hardcodeados y sin explicacion, comentario o documentacion es imposible saber a que se refiere con ese valor, por lo tanto optamos por aplicar un Replace Magic Number with Method

```
public double descuentoSiClienteMayorACincoAnos(double costoProductos, d
    if (cliente.antiguedadMayorACinco()){
        return (costoProductos + extraFormaPago) * this.obtenerDescuento();
    }
    return costoProductos + extraFormaPago;
}
```

```
public double obtenerDescuento(){
   return 0.9
}
```

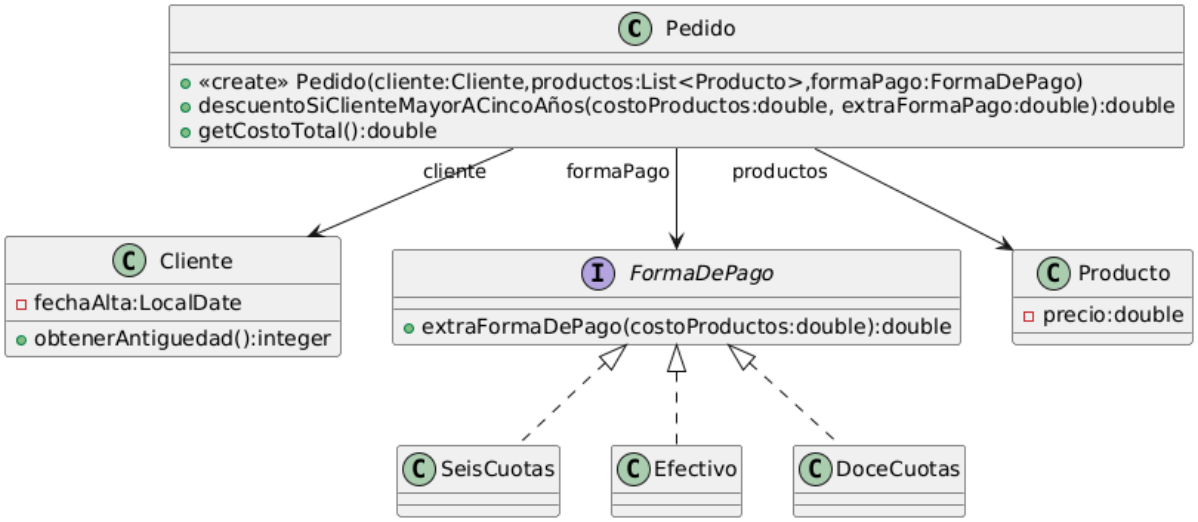
Clases luego del refactoring

Malos olores no tratados → Data class Producto.

```
public class Pedido {
   private Cliente cliente;
   private List<Producto> productos;
   private FormaDePago formaPago;

   public Pedido(Cliente cliente, List<Producto> productos, FormaDePago forr
      this.cliente = cliente;
      this.productos = productos;
      this.formaPago = formaPago;
   }

   public double descuentoSiClienteMayorACincoAnos(double costoProductos
      if (cliente.antiguedadMayorACinco()){
         return (costoProductos + extraFormaPago) * this.obtenerDescuento();
      }
      return costoProductos + extraFormaPago;
   }

   public double obtenerDescuento(){
      return 0.9
   }

   public double getCostoTotal() {
      double costoProductos = this.productos.stream().mapToDouble(Producto
      double extraFormaPago = this.formaPago.extraFormaDePago(costoProdu
      return this.descuentoSiClienteMayoraACincoAnos(costoProductos,extraF
}

public class Cliente {
   private LocalDate fechaAlta;
```

```
    public LocalDate getFechaAlta() {
     return this.fechaAlta;
    }

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

    public boolean antiguedadMayorACinco(){
        return this.obtenerAntiguedad() > 5;
    }
}

public abstract class FormaDePago {
    public extraFormaDePago(double costoProductos){
        return this.porcentaje() * costoProductos
    }
    public abstract porcentaje();
}

public class Efectivo extends FormaDePago{
    public double porcentaje(){
        return 0;
    }
}

public class SeisCuotas extends FormaDePago{
    public double porcentaje(){
        return 0.2;
    }
}

public class DoceCuotas extends FormaDePago{
    public double porcentaje(){
        return 0.5;
    }
}
```

```
public class Producto {
    private double precio;

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



Ejercicio 5 ✅

Envidia de atributos en la clase Empresa en el metodo agregarNumeroDeTelefono() a la hora de verificar si el numero ya esta registrado → Se debe aplicar un Move Method de la clase Empresa a GestorNumerosDisponibles

Antes

```
public boolean agregarNumeroTelefono(String str) {
        boolean encontre = guia.getLineas().contains(str);
        if (!encontre) {
            guia.getLineas().add(str);
            encontre= true;
            return encontre;
        }
        else {
            encontre= false;
```

```
            return encontre;
        }
    }
```

Despues

```
Clase GestorNumerosDisponibles
public boolean esta(String str){
    return this.lineas.contains(str);
}


Clase Empresa
public boolean agregarNumeroTelefono(String str) {
        boolean encontre = guia.esta(str)
        if (!encontre) {
            guia.getLineas().add(str);
            encontre= true;
            return encontre;
        }
        else {
            encontre= false;
            return encontre;
        }
    }
```

Variables temporales en la clase Empresa en el metodo agregarNumeroDeTelefono, la variable encontre es innecesaria → Aplicamos un Replace Temp with Query

```
Clase Empresa
public boolean agregarNumeroTelefono(String str) {
        if (!guia.esta(str)) {
            guia.getLineas().add(str);
            return true;
        }
        return false;
}
```

Envidia de atributos en la clase Empresa en el metodo agregarNumeroDeTelefono ya que se le esta pidiendo un valor de una variable a la clase GestorNumerosDisponibles cuando se lo puede delegar a esta ultima → Aplicamos un Move Method

```
Clase GestorNumerosDisponibles
public void agregarLinea(String lineaNueva){
   this.lineas.add(lineaNueva);
}

Clase Empresa
public boolean agregarNumeroTelefono(String str) {
      if (!guia.esta(str)) {
         guia.agregarLinea(str)
         return true;
      }
      return false;
}
```

Envidia de atributos en la clase Empresa en el metodo calcularMontoTotalLlamadas ya que se esta procesando la lista de llamadas del Cliente, esto debe ser responsabilidad de el mismo → Aplicamos un Move Method
Antes

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
      double c = 0;
      for (Llamada l : cliente.llamadas) {
         double auxc = 0;
         if (l.getTipoDeLlamada() == "nacional") {
            // el precio es de 3 pesos por segundo más IVA sin adicional por est
            auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
         } else if (l.getTipoDeLlamada() == "internacional") {
            // el precio es de 150 pesos por segundo más IVA más 50 pesos por
            auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
         }

         if (cliente.getTipo() == "fisica") {
```

```
            auxc -= auxc*descuentoFis;
        } else if(cliente.getTipo() == "juridica") {
            auxc -= auxc*descuentoJur;
        }
        c += auxc;
    }
    return c;
}
```

Despues

```
Clase Cliente
public double montoLlamadas() {
    double c = 0;
    for (Llamada l : llamadas) {
        double auxc = 0;
        if (l.getTipoDeLlamada() == "nacional") {
            // el precio es de 3 pesos por segundo más IVA sin adicional por est
            auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
        } else if (l.getTipoDeLlamada() == "internacional") {
            // el precio es de 150 pesos por segundo más IVA más 50 pesos por
            auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
        }

        if (this.getTipo() == "fisica") {
            auxc -= auxc*descuentoFis;
        } else if(this.getTipo() == "juridica") {
            auxc -= auxc*descuentoJur;
        }
        c += auxc;
    }
    return c;
}

Clase Empresa
public double calcularMontoTotalLlamadas(Cliente cliente) {
    return cliente.montoLlamadas();
```

Sentencias if else en la clase Cliente en el metodo montoLlamadas() ya que se verifica el tipo de las llamadas → aca debemos aplicar un Replace Conditional with Polymorphism para obtener el valor

//Aca se podria decir que tambien hay envidia de atributos?

Clase Cliente

```
public double montoLlamadas() {
      double c = 0;
      for (Llamada l : llamadas) {
         double auxc = 0;
         auxc += l.costoLlamada()

         if (this.getTipo() == "fisica") {
            auxc -= auxc*descuentoFis;
         } else if(this.getTipo() == "juridica") {
            auxc -= auxc*descuentoJur;
         }
         c += auxc;
      }
      return c;
   }
```

Clase Abstracta Llamada

```
public double costoLlamada(){
   return this.getDuracion() * this.costoSegundo() +
            (this.getDuracion() * this.costoSegundo() * this.getIva()) +
            this.adicional()
}
protected abstract double costoSegundo()
protected abstract double adicional()
private double getIva(){
   return 0.21;
}
```

Clase LlamadaInternacional

```
protected double costoSegundo(){
   return 150;
```

```
}
protected double adicional(){
   return 50;
}

Clase LlamadaNacional
protected double costoSegundo(){
   return 3;
}
protected double adicional(){
   return 0
}
```

Sentencias if else en la clase Cliente en el metodo montoLlamadas() ya que se verifica el tipo del Cliente → aca debemos aplicar un Replace Conditional with Polymorphism para obtener el descuento segun corresponda

```
Clase Abstract Cliente
public double montoLlamadas() {
      double c = 0;
      for (Llamada l : llamadas) {
         double auxc = 0;
         auxc += l.costoLlamada()
         auxc -= auxc*this.descuentoCliente();
         c += auxc;
      }
      return c;
   }

public abstract double descuentoCliente();

Clase ClienteFisico
public double descuentoCliente(){
   return 0;
}

Clase ClienteJuridico
public double descuentoCliente(){
```

```
    return 0.15;
}
```

Metodo largo, el metodo montoLlamadas de la clase Cliente ademas de recorrer la lista de llamadas, obtiene el costo de la llamada y aplica un descuento, podemos dividir esta funcionalidad en metodo separados→ debemos aplicar un Extract Method

```
Clase Abstract Cliente
public double montoLlamadas() {
        double c = 0;
        for (Llamada l : llamadas) {
            c += this.calcularCostoConDescuento();
        }
        return c;
    }

public double calcularCostoConDescuento(){
    double auxc = 0;
    auxc += l.costoLlamada()
    auxc -= auxc*this.descuentoCliente();
    return auxc;
}
```

Iteracion con un for en el metodo montoLlamadas en la clase Cliente, podemos remplazar el uso de un for con un stream ademas quitamos la necesidad de variables temporales→ Aplicamos un reinventar la rueda

```
Clase Abstract Cliente
public double montoLlamadas() {
        return  this.llamadas.streams()
                            .mapToDouble(this -> this.calcularCostoConDescuento(l
                            .sum();
    }

private double calcularCostoConDescuento(Llamada llamada){
    return  llamada.costoLlamada()
```

```
                - ( llamada.costoLlamada() * this.descuentoCliente());
    }
```

Envidia de atributos en la clase Empresa en el metodo registrarLlamadaNacional y registrarLlamadaInternacional, ya que se agrega una llamada para el cliente origen, esto deberia ser responsabilidad de el mismo. → Aplico un Move Method

ANTES

```
public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int du
        Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.
        llamadas.add(llamada);
        origen.llamadas.add(llamada);
        return llamada;
    }
```

//EN ESTE PUNTO AL TENER LLAMADA ABSTRACTA DEBO IMPLEMENTAR 2 METODOS, UNO PARA LLAMADAS INTERNACIONALES Y OTRO PARA NACIONALES POR LO TANTO VOY A MODIFICAR LOS TEST, ¿ESTO ESTA BIEN? (por las dudas no lo hice) ¿TENER 2 METODOS UNO PARA LLAMADAS INTERNACIONALES Y OTRO PARA NACIONALES ES UN REFACTORING? ¿LO DEBO DOCUMENTAR?

```
public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int du
        if (t.equals("nacional")){
            LlamadaNacional llamada = new LlamadaNacional(origen.getNumeroTe
            llamadas.add(llamada);
            origen.agregarLlamada(llamada);
            return llamada;
        }
        else if (t.equals("internacional")) {
            LlamadaInternacional llamada = new LlamadaInternacional(origen.getN
            llamadas.add(llamada);
            origen.agregarLlamada(llamada);
            return llamada;
        }
        return null;
```

Codigo duplicado ahora en la clase Empresa en el metodo registrarLlamada vemos que tenemos codigo duplicado → Aplicamos un Extract Method

```
public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int d
        if (t.equals("nacional")){
            LlamadaNacional llamada = new LlamadaNacional(origen.getNumeroTe
            return agregarLlamada(llamada);
        }
        else if (t.equals("internacional")) {
            LlamadaInternacional llamada = new LlamadaInternacional(origen.getN
            return agregarLlamada(llamada);
        }
        return null;
    }

private Llamada agregarLlamada(Llamada llamada, Cliente origen){
        llamadas.add(llamada);
        origen.agregarLlamada(llamada);
        return llamada;
}
```

Envidia de atributos en la clase Empresa metodo registrarUsuario se esta instanciando un Cliente vacio y despues mediante sus setters se cargan los valores en su campo → Aplico un Move Method //ESTO ES CORRECTO YA QUE UTILIZO EL CONSTRUCTOR NOMAS

```
public Cliente registrarUsuario(String data, String nombre, String tipo) {
        if (tipo.equals("fisica")) {
            ClienteFisico cliente = new ClienteFisico(nombre,this.obtenerNumeroL
            clientes.add(cliente);
            return cliente;
        }
        else if (tipo.equals("juridica")) {
            ClienteJuridico cliente = new ClienteJuridico(nombre,this.obtenerNum
            clientes.add(cliente);
            return cliente;.
        }
```

```
        return null;
    }
```

Codigo duplicado, en la clase Empresa metodo registrarUsuario ya que las operaciones y agregar al cliente y devolverlo se hacen 2 veces → Aplicamos extract method

```
public Cliente registrarUsuario(String data, String nombre, String tipo) {
        if (tipo.equals("fisica")) {
            ClienteFisico cliente = new ClienteFisico(nombre,this.obtenerNumeroL
            return this.agregarCliente(cliente);
        }
        else if (tipo.equals("juridica")) {
            ClienteJuridico cliente = new ClienteJuridico(nombre,this.obtenerNum
            return this.agregarCliente(cliente);
        }
        return null;
    }
```

Switch Statement en la clase GestorNumerosDisponibles en el metodo obtenerNumeroLibre se utilizan diferentes formas de generarlo → Aplicamos un Replace Conditional with Strategy

Antes

```
public class GestorNumerosDisponibles {
    private SortedSet<String> lineas = new TreeSet<String>();
    private String tipoGenerador = "ultimo";

    public SortedSet<String> getLineas() {
        return lineas;
    }

    public String obtenerNumeroLibre() {
        String linea;
        switch (tipoGenerador) {
            case "ultimo":
                linea = lineas.last();
                lineas.remove(linea);
```

```
            return linea;
        case "primero":
            linea = lineas.first();
            lineas.remove(linea);
            return linea;
        case "random":
            linea = new ArrayList<String>(lineas)
                    .get(new Random().nextInt(lineas.size()));
            lineas.remove(linea);
            return linea;
      }
      return null;
   }

   public void cambiarTipoGenerador(String valor) {
      this.tipoGenerador = valor;
   }
}
```

Despues

```
public class GestorNumerosDisponibles {
   private SortedSet<String> lineas = new TreeSet<String>();
   private TipoGeneradorStrategy generadorStrategy = new UltimoStrategy();

   public SortedSet<String> getLineas() {
      return lineas;
   }

   public String obtenerNumeroLibre() {
      String linea = this.generadorStrategy.generar(this);
      lineas.remove(linea);
      return linea;
   }

   public void cambiarTipoGenerador(String valor) {
      if (valor.equals("primero"))this.generadorStrategy = new PrimeroStrategy
      else if (valor.equals("random"))this.generadorStrategy = new RandomStr
```

```
    }

Interface TipoGeneradorStrategy
public String generar(GestorNumerosDisponibles gestor);

Clase PrimeroStrategy
public String generar(GestorNumerosDisponibles gestor){
    return gestor.getLineas().first();
}

Clase RandomStrategy
public String generar(GestorNumerosDisponibles gestor){
    return new ArrayList<String>(lineas)
                .get(new Random().nextInt(lineas.size()));
}

Clase UltimoStrategy
public String generar(GestorNumerosDisponibles gestor){
    return gestor.getLineas().last();
}
```

Ejercicio 6 ✅

```java
public class ArbolBinario {
    private int valor;
    private ArbolBinario hijoIzquierdo;
    private ArbolBinario hijoDerecho;

    public ArbolBinario(int valor) {
        this.valor = valor;
        this.hijoIzquierdo = null;
        this.hijoDerecho = null;
    }

    public int getValor() {
        return valor;
    }

    public void setValor(int valor) {
        this.valor = valor;
    }

    public ArbolBinario getHijoIzquierdo() {
        return hijoIzquierdo;
    }

    public void setHijoIzquierdo(ArbolBinario hijoIzquierdo) {
        this.hijoIzquierdo = hijoIzquierdo;
    }

    public ArbolBinario getHijoDerecho() {
        return hijoDerecho;
    }

    public void setDerecha(ArbolBinario hijoDerecho) {
        this.hijoDerecho = hijoDerecho;
    }

   public String recorrerPreorden() {
        String resultado = valor + " - ";
        if (this.getHijoIzquierdo() != null) {
```

```
            resultado += this.getHijoIzquierdo().recorrerPreorden();
        }
        if (this.getHijoDerecho() != null) {
            resultado += this.getHijoDerecho().recorrerPreorden();
        }
        return resultado;
    }

    public String recorrerInorden() {
        String resultado = "";
        if (this.getHijoIzquierdo() != null) {
            resultado += this.getHijoIzquierdo().recorrerInorden();
        }
        resultado += valor + " - ";
        if (this.getHijoDerecho() != null) {
            resultado += this.getHijoDerecho().recorrerInorden();
        }
        return resultado;
    }

    public String recorrerPostorden() {
        String resultado = "";
        if (this.getHijoIzquierdo() != null) {
            resultado += this.getHijoIzquierdo().recorrerPostorden();
        }
        if (this.getHijoDerecho() != null) {
            resultado += this.getHijoDerecho().recorrerPostorden();
        }
        resultado += valor + " - ";
        return resultado;
    }
}
```

Null Check → Aplicamos Introduce Null Object ya que en la clase ArbolBinario en los metoodos recorrerPreorden, InOrden y PostOrden se realizan chequeos por null, primero hacemos un Extract Subclass llamada ArbolBinarioNull que extienda de ArbolBinario

```
public class ArbolBinarioNull extends ArbolBinario{
    public ArbolBinarioNull(int valor) {
        super(valor);
    }
}
```

Ahora sobreescribimos los metodos, en la clase ArbolBinarioNull, donde se hacen null check en el ArbolBinario, agregando el comporamiento esperado para los null

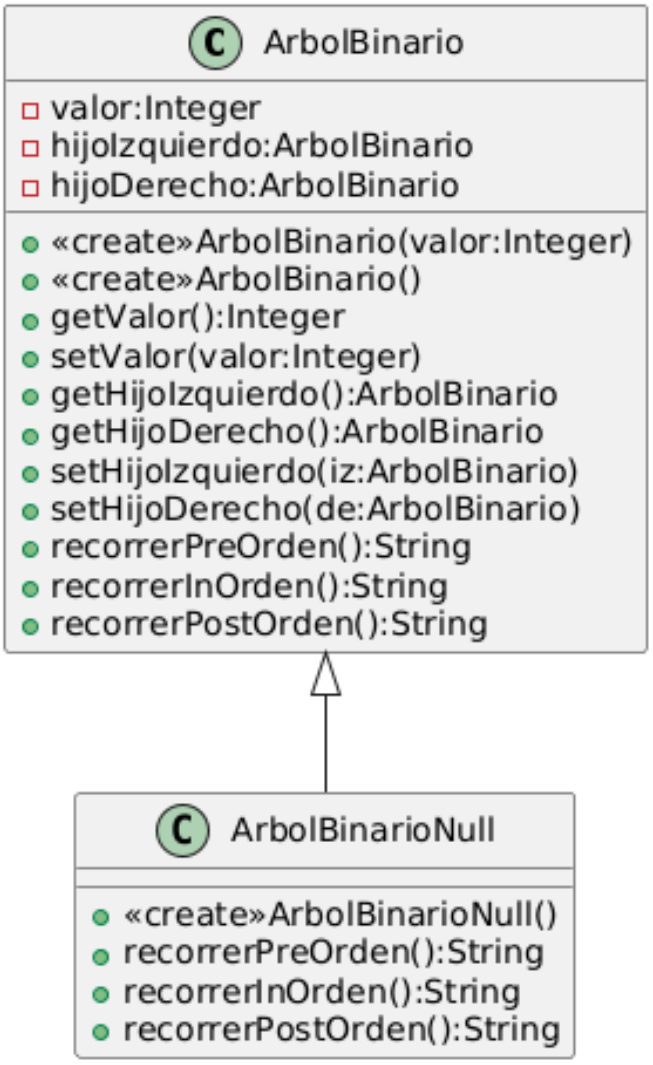
```
public class ArbolBinarioNull extends ArbolBinario{
    public ArbolBinarioNull() {

    }

    public String recorrerPreorden() {
        return "";
     }

    public String recorrerInorden() {
        return "";
    }

    public String recorrerPostorden() {
        return "";
    }
}
```

El ultimo paso es quitar los null checks de los metodos recorrerPreorden, InOrden y PostOrden de la clase ArbolBinario y realizar los cambios necesario para incluir instancias de ArbolBinarioNull, ademas para evitar recursiones infinitas es necesario definir otro constructor para ArbolBinario para que pueda ser utilizado por ArbolBinarioNull

```
public class ArbolBinario {
    private int valor;
    private ArbolBinario hijoIzquierdo;
    private ArbolBinario hijoDerecho;
```

```
public ArbolBinario(int valor) {
   hijoIzquierdo = new ArbolBinarioNull();
   hijoDerecho = new ArbolBinarioNull();
   this.valor = valor;
}

public ArbolBinario() {
}

public int getValor() {
   return valor;
}

public void setValor(int valor) {
   this.valor = valor;
}

public ArbolBinario getHijoIzquierdo() {
   return hijoIzquierdo;
}

public void setHijoIzquierdo(ArbolBinario hijoIzquierdo) {
   if (hijoIzquierdo == null) {
      this.hijoIzquierdo = new ArbolBinarioNull();
   }
   else this.hijoIzquierdo = hijoIzquierdo;
}

public ArbolBinario getHijoDerecho() {
   return hijoDerecho;
}

public void setDerecha(ArbolBinario hijoDerecho) {
   if (hijoDerecho == null) {
      this.hijoDerecho = new ArbolBinarioNull();
   }
   else this.hijoDerecho = hijoDerecho;
```

```
    }

    public String recorrerPreorden() {
        String resultado = valor + " - ";
        resultado += this.getHijoIzquierdo().recorrerPreorden();
        resultado += this.getHijoDerecho().recorrerPreorden();
        return resultado;
    }

    public String recorrerInorden() {
        String resultado = "";
        resultado += this.getHijoIzquierdo().recorrerInorden();
        resultado += valor + " - ";
        resultado += this.getHijoDerecho().recorrerInorden();
        return resultado;
    }

    public String recorrerPostorden() {
        String resultado = "";
        resultado += this.getHijoIzquierdo().recorrerPostorden();
        resultado += this.getHijoDerecho().recorrerPostorden();
        resultado += valor + " - ";
        return resultado;
    }

}
```


ArbolBinario
valor:Integer
hijoIzquierdo:ArbolBinario
hijoDerecho:ArbolBinario
«create»ArbolBinario(valor:Integer)
«create»ArbolBinario()
getValor():Integer
setValor(valor:Integer)
getHijoIzquierdo():ArbolBinario
getHijoDerecho():ArbolBinario
setHijoIzquierdo(iz:ArbolBinario)
setHijoDerecho(de:ArbolBinario)
recorrerPreOrden():String
recorrerInOrden():String
recorrerPostOrden():String
ArbolBinarioNull
«create»ArbolBinarioNull()
recorrerPreOrden():String
recorrerInOrden():String
recorrerPostOrden():String