Informe Practica Diseño de Software

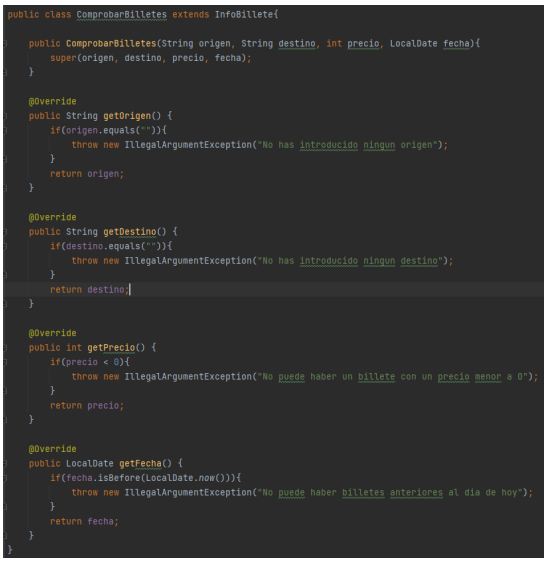
lucas.campos@udc.es
ruben.fernandez.farelo@udc.es

- ## Ejercicio 1.

  Como principios de SOLID, hemos utilizado Responsabilidad Única e Inversión de la Dependencia:

  -Principio de Responsabilidad Única: donde cada objeto tiene una única responsabilidad, la cual está enteramente encapsulada en la clase. Todos los servicios que provee el objeto están estrechamente alineados con dicha responsabilidad. Con este principio se busca evitar las clases Dios, donde una única clase realiza todo el trabajo de un programa, dejando pocas responsabilidades al resto de clases. Esto se ve reflejado en nuestras 2 clases, donde una se encarga de imprimir por pantalla la información de los billetes que coinciden con los criterios de búsqueda que se han indicado en los tests (clase Billete), y la otra se encarga de comprobar que esos criterios que se han pasado están correctos (clase ComprobarBilletes), como por ejemplo que un precio no sea menor que 0, que a la hora de buscar un billete este no sea anterior a la fecha actual…

```
public class Billete extends InfoBillete {

    private final ComprobarBilletes billete;

    public Billete(String origen, String destino, int precio, LocalDate fecha){
        super(origen, destino, precio, fecha);
        billete = new ComprobarBilletes(origen, destino, precio, fecha);
    }

    @Override
    public String getOrigen() { return origen; }

    @Override
    public String getDestino() { return destino; }

    @Override
    public int getPrecio() { return precio; }

    @Override
    public LocalDate getFecha() { return fecha; }

    public String info(List<Billete> listaBillete){
        StringBuilder aux = new StringBuilder();
        int i;

        if(listaBillete.size() == 0){
            aux.append("");//Lista vacia
        }
        else{
            if(!billete.getOrigen().equals("") && !billete.getDestino().equals("") && billete.getPrecio() > 0
                for(i = 0; i < listaBillete.size(); i++){
                    aux.append("Informacion del billete ").append(i+1).append(":\n");
                    aux.append("Ciudad de origen: ").append(listaBillete.get(i).getOrigen()).append("\n");
                    aux.append("Ciudad de destino: ").append(listaBillete.get(i).getDestino()).append("\n");
                    aux.append("Precio del viaje: ").append(listaBillete.get(i).getPrecio()).append("\n");
                    aux.append("Fecha del viaje: ").append(listaBillete.get(i).getFecha());
                    if((i+1)<listaBillete.size()-1){
                        aux.append("\n\n");
                    }
                }
            }
```



```
public class ComprobarBilletes extends InfoBillete{

    public ComprobarBilletes(String origen, String destino, int precio, LocalDate fecha){
        super(origen, destino, precio, fecha);
    }

    @Override
    public String getOrigen() {
        if(origen.equals("")){
            throw new IllegalArgumentException("No has introducido ningun origen");
        }
        return origen;
    }

    @Override
    public String getDestino() {
        if(destino.equals("")){
            throw new IllegalArgumentException("No has introducido ningun destino");
        }
        return destino;
    }

    @Override
    public int getPrecio() {
        if(precio < 0){
            throw new IllegalArgumentException("No puede haber un billete con un precio menor a 0");
        }
        return precio;
    }

    @Override
    public LocalDate getFecha() {
        if(fecha.isBefore(LocalDate.now())){
            throw new IllegalArgumentException("No puede haber billetes anteriores al dia de hoy");
        }
        return fecha;
    }
}
```

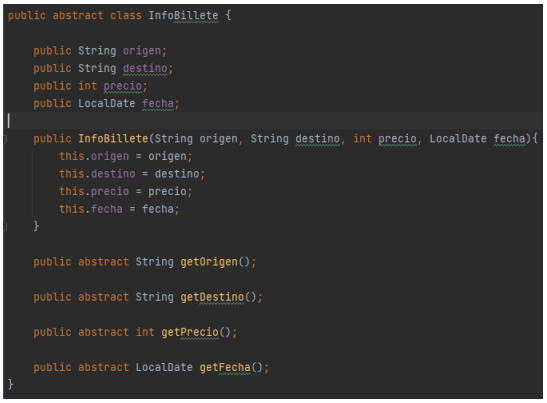
  -Principio de Inversión de la Dependencia: este principio se basa en la dependencia de abstracciones y no de concreciones. Con depender de abstracciones nos referimos a que nuestras clases están ligadas con clases, interfaces o funciones abstractas. En nuestro caso, hemos utilizado la interfaz List<> en la clase Billete, donde nos estamos restringiendo a usar solo los métodos declarados en esta interfaz, por lo que cambiar una implementación por otra no tendría efecto en nuestro código, y la clase InfoBillete, donde cada clase hereda los métodos getOrigen(), getDestino(), getPrecio() y getFecha().

```
public String info(List<Billete> listaBillete){
```

```
public abstract class InfoBillete {

    public String origen;
    public String destino;
    public int precio;
    public LocalDate fecha;

    public InfoBillete(String origen, String destino, int precio, LocalDate fecha){
        this.origen = origen;
        this.destino = destino;
        this.precio = precio;
        this.fecha = fecha;
    }

    public abstract String getOrigen();
    public abstract String getDestino();
    public abstract int getPrecio();
    public abstract LocalDate getFecha();
}
```

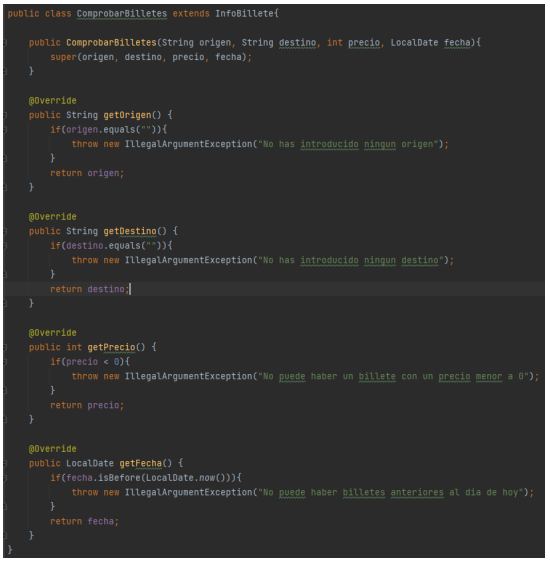
Y como patrón de diseño hemos utilizado Composición:

-Patrón Composición: este patrón es de tipo estructural, el cual se utiliza para componer objetos en estructuras de árbol y permite tratar uniformemente a los objetos y a las composiciones. La clave de este patrón es una clase abstracta, que en nuestro ejemplo es la clase InfoBillete, la cual representa al mismo tiempo a los elementos primitivos y a los contenedores. A parte es un buen ejemplo de utilización conjunta de herencia y composición.
En nuestro ejemplo, la clase InfoBillete, a parte de ser la clase abstracta, como hemos dicho anteriormente, juega el rol de **Componente**, siendo getOrigen, getDestino, getPrecio y getFecha las operaciones que se delegan en las sublcases. Luego tenemos la clase Billete, la cual juega el rol de **Componente concreto**; define el comportamiento de los objetos elementales de la composición. Y por último tenemos la clase ComprobarBilletes, que juega el rol de **composición** y define el comportamiento de los objetos compuestos.



```java
public abstract class InfoBillete {

    public String origen;
    public String destino;
    public int precio;
    public LocalDate fecha;

    public InfoBillete(String origen, String destino, int precio, LocalDate fecha){
        this.origen = origen;
        this.destino = destino;
        this.precio = precio;
        this.fecha = fecha;
    }

    public abstract String getOrigen();

    public abstract String getDestino();

    public abstract int getPrecio();

    public abstract LocalDate getFecha();
}
```

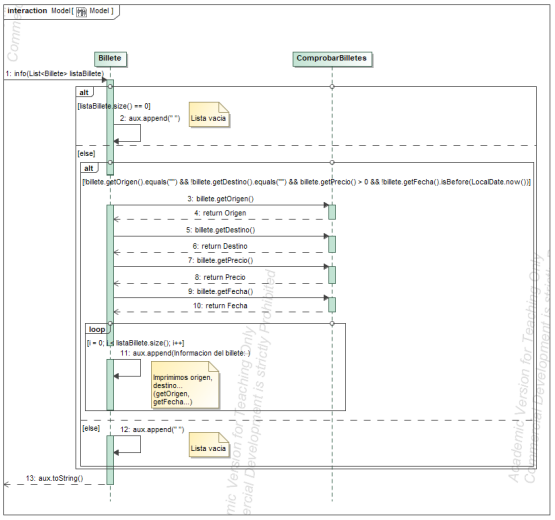
```java
public class Billete extends InfoBillete {

    private final ComprobarBilletes billete;

    public Billete(String origen, String destino, int precio, LocalDate fecha){
        super(origen, destino, precio, fecha);
        billete = new ComprobarBilletes(origen, destino, precio, fecha);
    }

    @Override
    public String getOrigen() { return origen; }

    @Override
    public String getDestino() { return destino; }

    @Override
    public int getPrecio() { return precio; }

    @Override
    public LocalDate getFecha() { return fecha; }

    public String info(List<Billete> listaBillete){
        StringBuilder aux = new StringBuilder();
        int i;

        if(listaBillete.size() == 0){
            aux.append("");//Lista vacia
        }
        else{
            if(!billete.getOrigen().equals("") && !billete.getDestino().equals("") && billete.getPrecio() > 0
                for(i = 0; i < listaBillete.size(); i++){
                    aux.append("Informacion del billete ").append(i+1).append(":\n");
                    aux.append("Ciudad de origen: ").append(listaBillete.get(i).getOrigen()).append("\n");
                    aux.append("Ciudad de destino: ").append(listaBillete.get(i).getDestino()).append("\n");
                    aux.append("Precio del viaje: ").append(listaBillete.get(i).getPrecio()).append("\n");
                    aux.append("Fecha del viaje: ").append(listaBillete.get(i).getFecha());
                    if(!(i==listaBillete.size()-1)){
                        aux.append("\n\n");
                    }
                }
            }
```

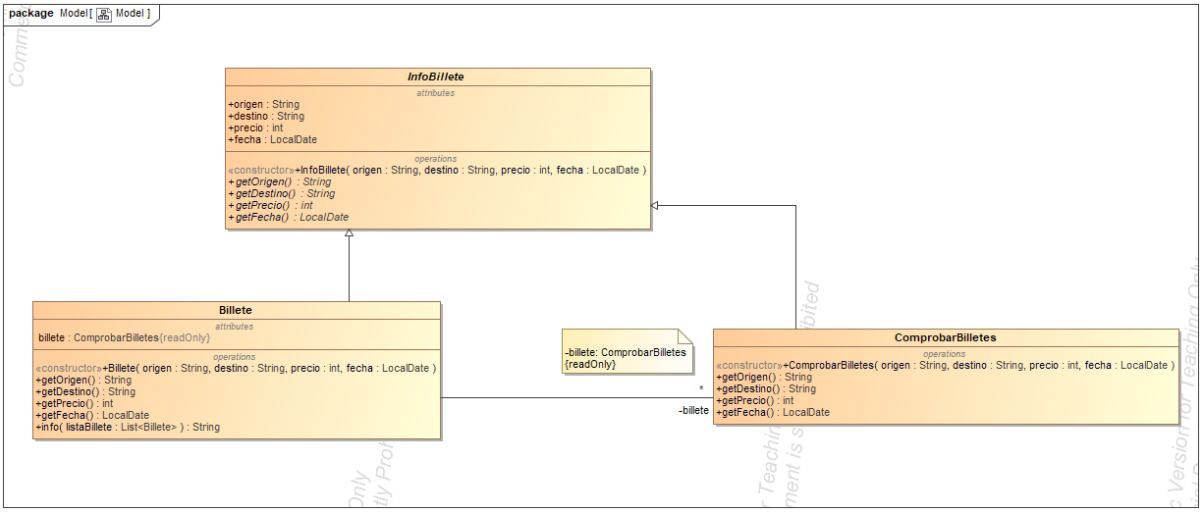


```java
public class ComprobarBilletes extends InfoBillete{

    public ComprobarBilletes(String origen, String destino, int precio, LocalDate fecha){
        super(origen, destino, precio, fecha);
    }

    @Override
    public String getOrigen() {
        if(origen.equals("")){
            throw new IllegalArgumentException("No has introducido ningun origen");
        }
        return origen;
    }

    @Override
    public String getDestino() {
        if(destino.equals("")){
            throw new IllegalArgumentException("No has introducido ningun destino");
        }
        return destino;
    }

    @Override
    public int getPrecio() {
        if(precio < 0){
            throw new IllegalArgumentException("No puede haber un billete con un precio menor a 0");
        }
        return precio;
    }

    @Override
    public LocalDate getFecha() {
        if(fecha.isBefore(LocalDate.now())){
            throw new IllegalArgumentException("No puede haber billetes anteriores al dia de hoy");
        }
        return fecha;
    }
}
```

Si en el futuro se añadieran nuevos criterios, como por ejemplo si hay transbordos, o cuanto dura el viaje como dice el enunciado, simplemente habría que añadir campos a los constructores de las clases con el criterio nuevo a añadir, y en el test, a la hora de crear los objetos billete, habría que rellenar ese campo.

Este ejercicio tiene el siguiente diagrama estático y dinámico:

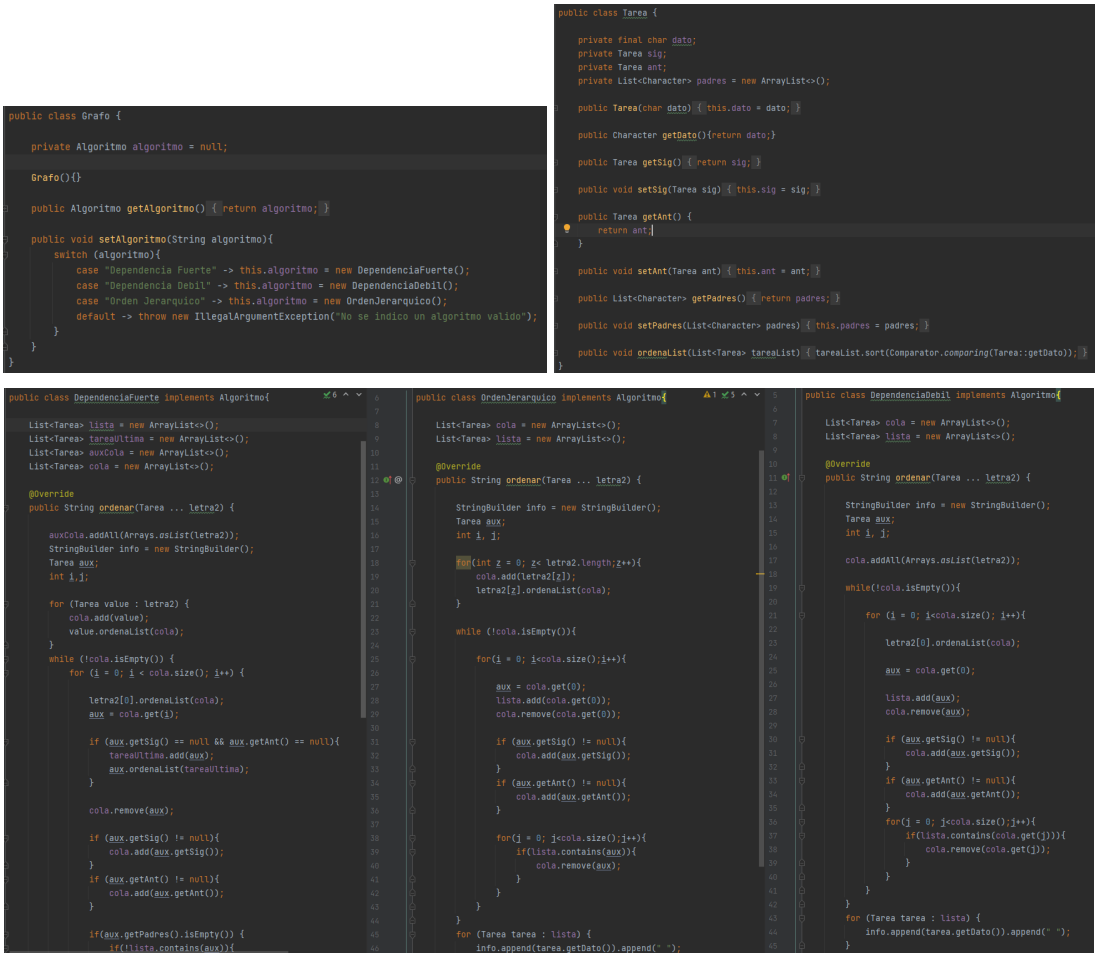




- **<u>Ejercicio 2.</u>**

  Como principios de SOLID, hemos utilizado Responsabilidad Única e Inversión de la Dependencia:

  -<u>Principio de Responsabilidad Única:</u> donde cada objeto tiene una única responsabilidad, la cual está enteramente encapsulada en la clase. Todos los servicios que provee el objeto están estrechamente alineados con dicha responsabilidad. Con este principio se busca evitar las clases Dios, donde una única clase realiza todo el trabajo de un programa, dejando pocas responsabilidades al resto de clases. Esto se ve reflejado en nuestras clases, donde la clase Tarea se encarga de facilitar toda la información sobre la Tarea actual, las clases OrdenJerarquico, DependenciaFuerte y DependenciaDebil se encargar de ordenar el grafo (cada clase ordena el grafo de forma diferente), y la clase Grafo, donde se hace la elección del algoritmo a usar(clases anteriores).



-Principio de Inversión de la Dependencia: este principio se basa en la dependencia de abstracciones y no de concreciones. Con depender de abstracciones nos referimos a que nuestras clases están ligadas con clases, interfaces o funciones abstractas. En nuestro caso, hemos utilizado la interfaz List<> en las clases encargadas de ordenar el grafo, donde nos estamos restringiendo a usar solo los métodos declarados en esta interfaz, por lo que cambiar una implementación por otra no tendría efecto en nuestro código.

```
List<Tarea> cola = new ArrayList<>();
List<Tarea> lista = new ArrayList<>();
```

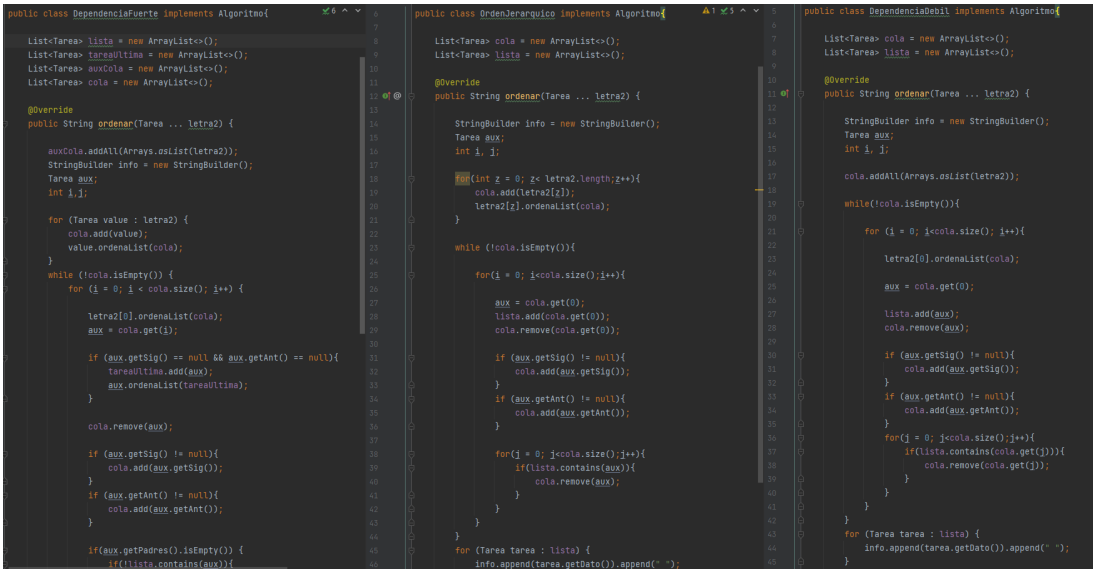
Y como patrón hemos utilizado Estrategia:

-Patrón Estrategia: es un patrón de comportamiento, el cual se utiliza para definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. En nuestro caso, tenemos una clase Grafo, la cual representa un grafo de Tareas, y a parte, en esta clase se hace la elección del algoritmo a utilizar para recorrer el grafo.
La clave de este patrón es la posibilidad de separar la parte que varía, los algoritmos de ordenación, del contexto, el grafo en nuestro caso, y permite cambiar dinámicamente el algoritmo de ordenación.

Nuestra clase Grafo juega el rol de **Contexto**, la cual delega en el objeto de la interfaz el cálculo del algoritmo. Nuestra interfaz Algoritmo, juega el rol de **Estrategia**, y se encarga de declarar una interfaz común para todos los algoritmos a soportar. Y por último, nuestros algoritmos juegan el rol de **Estrategia concreta**, y estos se encargan de implementar el algoritmo utilizando la interfaz anterior (cada algoritmo tiene una implementación diferente).

```
public class Grafo {

    private Algoritmo algoritmo = null;

    Grafo(){}

    public Algoritmo getAlgoritmo() { return algoritmo; }

    public void setAlgoritmo(String algoritmo){
        switch (algoritmo){
            case "Dependencia Fuerte" -> this.algoritmo = new DependenciaFuerte();
            case "Dependencia Debil" -> this.algoritmo = new DependenciaDebil();
            case "Orden Jerarquico" -> this.algoritmo = new OrdenJerarquico();
            default -> throw new IllegalArgumentException("No se indico un algoritmo valido");
        }
    }
}
```

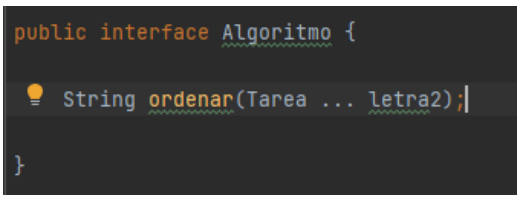
```
public class Tarea {

    private final char dato;
    private Tarea sig;
    private Tarea ant;
    private List<Character> padres = new ArrayList<>();

    public Tarea(char dato) { this.dato = dato; }

    public Character getDato(){return dato;}

    public Tarea getSig() { return sig; }

    public void setSig(Tarea sig) { this.sig = sig; }

    public Tarea getAnt() {
        return ant;
    }

    public void setAnt(Tarea ant) { this.ant = ant; }

    public List<Character> getPadres() { return padres; }

    public void setPadres(List<Character> padres) { this.padres = padres; }

    public void ordenaList(List<Tarea> tareaList) { tareaList.sort(Comparator.comparing(Tarea::getDato)); }
}
```



```
public class DependenciaFuerte implements Algoritmo{

    List<Tarea> lista = new ArrayList<>();
    List<Tarea> tareaUltima = new ArrayList<>();
    List<Tarea> auxCola = new ArrayList<>();
    List<Tarea> cola = new ArrayList<>();

    @Override
    public String ordenar(Tarea ... letra2) {

        auxCola.addAll(Arrays.asList(letra2));
        StringBuilder info = new StringBuilder();
        Tarea aux;
        int i,j;

        for (Tarea value : letra2) {
            cola.add(value);
            value.ordenaList(cola);
        }
        while (!cola.isEmpty()) {
            for (i = 0; i < cola.size(); i++) {

                letra2[0].ordenaList(cola);
                aux = cola.get(i);

                if (aux.getSig() == null && aux.getAnt() == null){
                    tareaUltima.add(aux);
                    aux.ordenaList(tareaUltima);
                }

                cola.remove(aux);

                if (aux.getSig() != null){
                    cola.add(aux.getSig());
                }
                if (aux.getAnt() != null){
                    cola.add(aux.getAnt());
                }

                if(aux.getPadres().isEmpty()) {
                    if(!lista.contains(aux)){
```

```
public class OrdenJerarquico implements Algoritmo{

    List<Tarea> cola = new ArrayList<>();
    List<Tarea> lista = new ArrayList<>();

    @Override
    public String ordenar(Tarea ... letra2) {

        StringBuilder info = new StringBuilder();
        Tarea aux;
        int i, j;

        for(int z = 0; z< letra2.length;z++){
            cola.add(letra2[z]);
            letra2[z].ordenaList(cola);
        }

        while (!cola.isEmpty()){

            for(i = 0; i<cola.size();i++){

                aux = cola.get(0);
                lista.add(cola.get(0));
                cola.remove(cola.get(0));

                if (aux.getSig() != null){
                    cola.add(aux.getSig());
                }
                if (aux.getAnt() != null){
                    cola.add(aux.getAnt());
                }

                for(j = 0; j<cola.size();j++){
                    if(lista.contains(aux)){
                        cola.remove(aux);
                    }
                }
            }
        }
        for (Tarea tarea : lista) {
            info.append(tarea.getDato()).append(" ");
```

```
public class DependenciaDebil implements Algoritmo{

    List<Tarea> cola = new ArrayList<>();
    List<Tarea> lista = new ArrayList<>();

    @Override
    public String ordenar(Tarea ... letra2) {

        StringBuilder info = new StringBuilder();
        Tarea aux;
        int i, j;

        cola.addAll(Arrays.asList(letra2));

        while(!cola.isEmpty()){

            for (i = 0; i<cola.size(); i++){

                letra2[0].ordenaList(cola);

                aux = cola.get(0);

                lista.add(aux);
                cola.remove(aux);

                if (aux.getSig() != null){
                    cola.add(aux.getSig());
                }
                if (aux.getAnt() != null){
                    cola.add(aux.getAnt());
                }
                for(j = 0; j<cola.size();j++){
                    if(lista.contains(cola.get(j))){
                        cola.remove(cola.get(j));
                    }
                }
            }
        }
        for (Tarea tarea : lista) {
            info.append(tarea.getDato()).append(" ");
        }
```



```
public interface Algoritmo {

    String ordenar(Tarea ... letra2);

}
```

Si en el futuro se añadieran nuevas órdenes de ejecución del grafo, simplemente habría que añadir una clase con el nombre del algoritmo y dentro de esta habria que hacer la implementación de este (heredando el método de la interfaz Algoritmo), y para poder ejecutar este, tendríamos un nuevo condicional en la clase Grafo, donde comprobamos si el algoritmo que mandamos ejecutar, es el nuevo que acabamos de añadir.

Y este ejercicio tiene el siguiente diagrama estático y el diagrama dinámico (el diagrama dinámico lo tenemos en 2 imagenes porque no nos cambia todo en 1, de

todos modos todas las imágenes de los diagramas están en la carpeta doc del git junto a esta documentación):