Unidad 12. Manejo Avanzado de Clases. Patrones de comportamiento

Contenido

1 Introducción 1
2 Patrones estructurales 2
2.1 Patrón Composite 3
2.1.1 Ejemplo Composite Equipos deportivos. Practica guiada Composite 5
2.2 Practica independiente del patron Composite 19
3 Behavioral patterns. Patrones de comportamiento 19
3.1 Iterator 21
3.1.1 Ejemplo patron Iterator . Practica guiada patrón iterator 21
3.2 Practica independiente patron iterator. 29
3.3 Observer 30
3.3.1 Ejemplo implementado en Java de Observer. Practica guiada Observer. 34
3.4 Practica independiente del patrón observer 51
4 Patrones de diseño en programación funcional 51
4.1 Patrón Strategy. Pratica guiada patrón Strategy 51
4.2 Practica Independiente del Patrón Strategy 57
5 Bibliografía y referencias web 58

1 Introducción

En este **capítulo vamos continuamos explicando los diferentes patrones** o acercamientos que **usaremos para resolver nuestros problemas de programación en programación orientada a objetos**. En este caso **patrones relacionados con colecciones. Ampliamos el uso de funciones como parámetros** y el operador ::. Todo esto **nos llevará a ver el patrón Strategy en versión funcional.**

Recordamos **el capítulo anterior donde repasábamos los patrones de diseño** de orientación a objetos. **Usamos las estructuras y herramientas del tema anterior** como base para **mejorar nuestro diseño de clases**. Las **clases y los interfaces son unas estructuras de control** más en programación orientada objetos. Nuestra labor es saberlas, conocerlas y utilizarlas apropiadamente.

Recordamos que tenemos cuatro tipos de patrones en orientación a objetos.

Patrones básicos: hacemos **manejo básico de encapsulación, herencia,**

polimorfismo y abstracción para resolver problemas de orientación a objetos. Son la base o los cimientos sobre los que vamos a construir el resto de patrones de diseño.

Patrones de creación o creacionales: nos ayudarnos con la **creación de objetos de un framework** o conjunto de clases y subclases para hacerlo más transparente. En la mayoría de las ocasiones **enmascararemos los detalles internos** de esas clases. En otras sólo nos interesará usar una o dos operaciones comunes a todo ese tipo de clases.
Patrones estructurales nos **permiten resolver problemas de orientación a objetos creando nuevas estructuras** en nuestras **clases para resolver múltiples problemas de incompatibilidades entre subclases** que deben **colaborar** y mostrar un comportamiento común.
Patrones de comportamiento: Los patrones de comportamiento **tratan con los algoritmos** y como asignar de manera correcta las responsabilidades entre objetos y a los diferentes objetos. Son **fundamentales para que nuestras aplicaciones sean dinámicas**. Además, **nos van a permitir usar y aplicar** la **nueva programación funcional**.
Pero empezaremos introduciendo los principios, fundamentos de diseño orientado a objetos

2 Patrones estructurales

Tratar con **objetos que delegan responsabilidades** a otros objetos. Esto da como resultado en **una arquitectura en capas de componentes** con bajo grado de acoplamiento. Facilita **la comunicación entre objetos** cuando un **objeto no es accesible** para el otro por medios normales o cuando un **objeto no es utilizable debido a su interfaz incompatible**. Proporciona **formas de estructurar un objeto agregado para que se cree en su totalidad y recuperar los recursos del sistema** de manera oportuna.
Visto de otra manera, **los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes**, **manteniendo estas estructuras flexibles y eficientes**. Cada **patrón describe un problema que se produce en orientación a objetos** una y **otra vez en nuestro entorno**, y luego **describe el núcleo de la solución a ese problem**a, de tal **manera que puede usar esta solución un millón de veces más**, sin hacerlo de la misma manera dos veces.

Tenemos diez patrones estructurales:
Adapter: Permite que **los objetos con interfaces incompatibles colaboren**. Interfaz como ya sabeis, es el **conjunto de métodos que ofrece ese objeto para ser utilizado**.
Bridge: Permite **dividir una clase grande o un conjunto de clases estrechamente relacionadas** en **dos jerarquías separadas —abstracción e implementación**— que se pueden **desarrollar independientemente una de la otra**.

Composite: es un **patrón de diseño estructural que te permite componer objetos en estructuras de árboles** y luego **trabajar con estas estructuras como si fueran objetos individuales**.
Decorator: es **un patrón de diseño estructural que te permite conectar nuevos comportamientos** a **los objetos colocando estos objetos dentro objetos contenedor** especiales que **contienen dichos comportamientos.**
Low Coupling: Es un patron que **indica como asignar responsabilidades** entre clases de manera que **disminuyamos las dependencias entre ellas al mínimo**. De esta manera **un cambio en una clase afecte lo mínimo al resto**. No lo vamos a implementar pues mas o menos lo aplicamos en otros patrones.
Flyweight: es un **patrón de diseño estructural que te permite encajar más objetos en la cantidad disponible de memoria RAM** compartiendo partes de estado entre varios objetos en lugar de mantener todos los de los datos de cada objeto. Con **nuestras memorias RAM actuales está en desuso.**

Façade: es un **patrón de diseño estructural que proporciona un patrón simplificado de interfaz a una biblioteca**, un **framework** o cualquier otro **conjunto complejo de clases**. Adapter intenta adaptar tu interfaz actual. **Facade añade uno nuevo**. Es una clase que nos da acceso y uso a una **biblioteca de objetos entera.**
Proxy: es un **patrón de diseño estructural** que le permite **proporcionar una sustituto o referencia para otro objeto**. Un **proxy controla el acceso al objeto original**, lo que le **permite realizar algo antes o después de que la solicitud llegue a el objeto original**.
Veremos algunos de los más importantes a lo largo de este tema. **Los primeros cuatro el ABCD** de los **patrones estructurales, plantearemos tres con ejemplos, el cuarto, Composite**, lo dejaremos **para temas posteriores**. A estos patrones se lo conoce como **The gang of Four**.
Vamos a introducir conceptos relacionados con patrones
Intent, intención. En patrones, se **define el intent como el objetivo** o el problema para el que **están diseñados**.
Problema: cual es **el problema de programación que se nos plantea** y como resolverlo.
Solución: es **la solución orientada a objetos** que **se adopta** para el problema.

2.1 Patrón Composite

El patrón Composite nos permite mantener en un mismo diseño objetos simples y objetos compuestos. Estos objetos compuestos estarán formados por objetos simples y otros objetos compuestos y así sucesivamente. Lo estudiaremos en el tema 8, con listas.

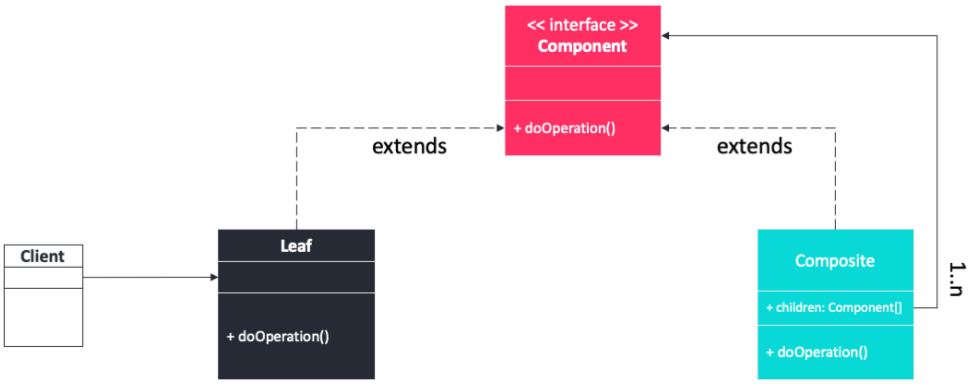
Intent

El patrón compuesto **permite crear estructuras jerárquicas de árbol de complejidad variable** al tiempo que permite que cada elemento de la

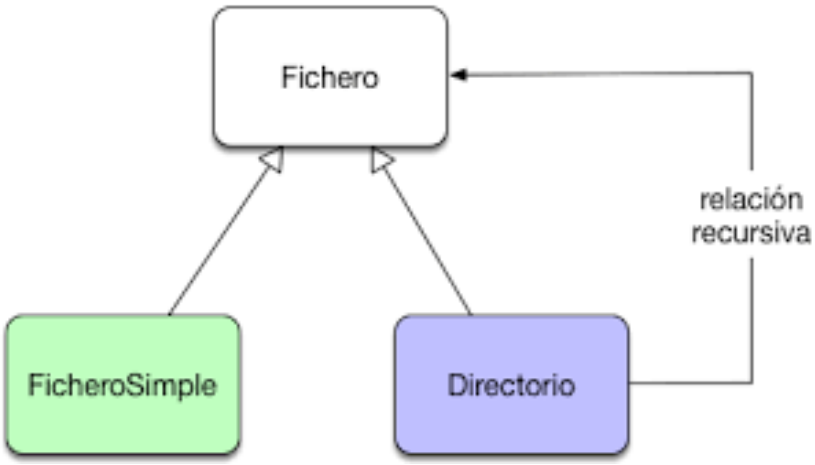
estructura funcione con una interfaz uniforme.

Solución

El **patrón Compuesto combina objetos en estructuras de árbol** para representar toda la **jerarquía o una parte de la jerarquía**.



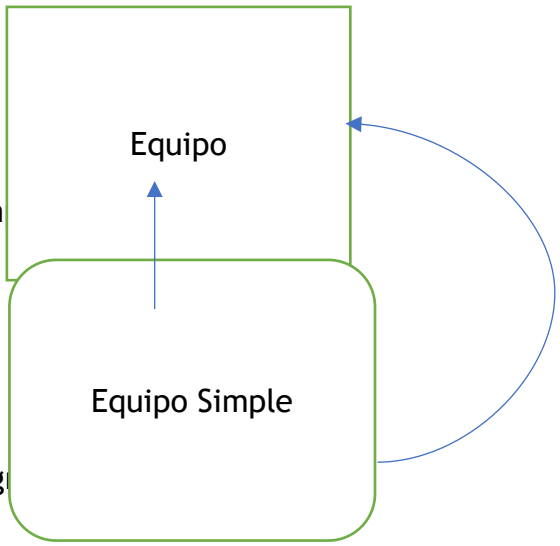
Un ejemplo de **patrón Composite sería el manejo de ficheros**. Tenemos ficheros **simples que son las hojas, y directorios que a su vez están compuestos de otros ficheros**. Como veis hay una relación de recursividad en este patrón. Debemos aplicar nuestra lógica de programación a Ficheros simples y a directorios**. Pero los directorios contienen ficheros y más directorios**. Con lo que **habrá que aplicar nuestra lógica de programación a los ficheros y directorios** que contiene el directorio recursivamente. El proceso acabara cuando hayamos tratado todos los archivos y ficheros que contiene un directorio.



2.1.1 Ejemplo Composite Equipos deportivos. Practica guiada Composite

Como bien sabéis los equipos deportivos pueden tener secciones o ser un equipo único. Por ejemplo, **el Real Madrid tiene varias secciones**, el **Granada sólo tendría una sección.** La idea **con el patron Composite es que podamos tratar de igual manera a un equipo con múltiples secciones** que a **un equipo con una sección única.**

Si **nos fijamos bien una sección va a ser exactamente igual a un equipo**, por lo que **podemos definir equipos sin secciones** y **equipos con secciones exactamente igual**, de **manera que una sección es un equipo per se**, en si mismo. Es decir, **podemos tener equipos que a su vez contengan equipos**, esta es la clave del Patrón Composite.



Com

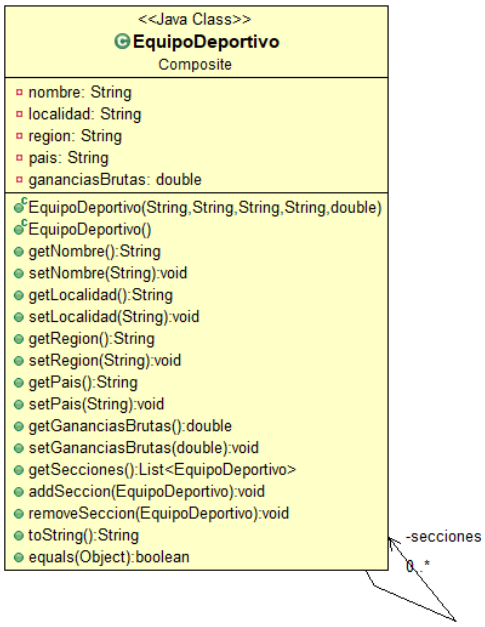
Relación recursiva

Prog

Un equipo compuesto, que tiene secciones, está compuesto de varios equipos simples. Un equipo simple no tiene composición. Pero ambos **en nuestro programa van a ser Equipos, y los trataremos igual.**

Para conseguir esto **nos basamos en dos propiedades de la orientación a objetos que estudiamos a principio de curso. Composición y agregación**. Recordamos que se produce **agregación cuando un objeto o clase tiene una colección de otro objeto** como **una de sus propiedades**. En este caso **tenemos la agregación en EquipoDeportivo** de un objeto **de si mismo**, EquipoDeportivo que **serán sus secciones**. Guardaremos las **secciones del EquipoDeportivo compuesto** como una colección de tipo **EquipoDeportivo**.

Si **os fijais en el diagrama de clases Equipo esta compuesto de O o N secciones**, que es **un ArrayList de equipos**. Podíamos haber usado un Set on un Map igualmente. Los **equipos simples sin secciones tendrán el ArrayList vacio,** porque **no tienen secciones**.



Y vamos al código. Para este ejemplo tenemos **tres clases**. El **programa principal** donde creamos los equipos **AplicacionEquipos.java**. **EquipoDeportivo.java,** la clase de nuestro modelo. Y una clase para auditar los impuestos del equipo **AuditarImpuestos.java**. Vamos a explicar las tres. Mi consejo que montéis el **ejemplo antes de seguir con los apuntes.**

En **EquipoDeportivo añadimos** las secciones como **un arraylist de EquipoDeportivo** secciones, **para que un EquipoDeportivo pueda contener equipos**.

```
        private List<EquipoDeportivo> secciones;
```

Con **los métodos addSeccion y removeSeccion permitimos eliminar o añadir secciones**, otros **EquipoDeportivo a un EquipoDeportivo compuesto**. Por ejemplo, el **Real Madrid ha añadido este año la sección de futbol femenino**.

```
public void addSeccion(EquipoDeportivo eq) {

                this.secciones.add(eq);

        }
```

```
        public void removeSeccion(EquipoDeportivo eq) {

                if (this.secciones.stream().anyMatch((e)-> e.equals(eq)))
                                this.secciones.remove(eq);




        }
```

Nuestro **objetivo es calcular los impuestos de los equipos**, y **poder tratarlos todos igual, sean simples o compuestos de secciones**. Para ello, **en el método getGananciasBrutas, calculamos las ganancias** de un **equipo simple devolviendo sus ganacias brutas directamente**, y si **tiene secciones, las ganancias serán la suma de las ganancias de sus secciones**, **recorriendo el ArrayList** y sumando todas. La solución que os doy es con un Iterator para que veáis us uso. Intentad resolver o realizar una versión de la función `getGananciasBrutas()` con la API Stream.

EquipoDeportivo.java

```
import java.util.ArrayList;
import java.util.List;

public class EquipoDeportivo {



        private String nombre;
        private String localidad;
        private String region;
        private String pais;
        private double gananciasBrutas=0.0;
        private List<EquipoDeportivo> secciones;
```

```
        public EquipoDeportivo(String nombre, String localidad, String region
, String pais, double gananciasBrutas) {
                super();
                this.nombre = nombre;
                this.localidad = localidad;
                this.region = region;
                this.pais = pais;
                this.gananciasBrutas = gananciasBrutas;
                secciones = new ArrayList<EquipoDeportivo>();



        }




        public String getNombre() {
                return nombre;
        }

        public void setNombre(String nombre) {
                this.nombre = nombre;
        }

        public String getLocalidad() {
                return localidad;
        }

        public void setLocalidad(String localidad) {
                this.localidad = localidad;
        }

        public String getRegion() {
                return region;
        }

        public void setRegion(String region) {
                this.region = region;
```

```
	}

	public String getPais() {
		return pais;
	}

	public void setPais(String pais) {
		this.pais = pais;
	}

	public double getGananciasBrutas() {


		Double ganancias =0.0;

		if (secciones.size()>0) {


			for (EquipoDeportivo seccion: secciones) {

				ganancias = ganancias + seccion.getGanancias
Brutas();

			}
		} else {

			ganancias = this.gananciasBrutas;
		}




		return ganancias;
	}
```

```
	public List<EquipoDeportivo> getSecciones() {

		return this.secciones;
	}

	public void addSeccion(EquipoDeportivo eq) {

		this.secciones.add(eq);

	}

	public void removeSeccion(EquipoDeportivo eq) {

		if (this.secciones.stream().anyMatch((e)-> e.equals(eq)))
				this.secciones.remove(eq);




	}



	@Override
	public String toString() {
		return "EquipoDeportivo [nombre=" + nombre + ", localidad=" +
localidad + ", region=" + region + ", pais="
				+ pais + ", gananciasBrutas=" + gananciasBru
tas + ", secciones=" + secciones + "]";
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
```

```
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                EquipoDeportivo other = (EquipoDeportivo) obj;
                if (Double.doubleToLongBits(gananciasBrutas) != Double.double
ToLongBits(other.gananciasBrutas))
                        return false;
                if (localidad == null) {
                        if (other.localidad != null)
                                return false;
                } else if (!localidad.equals(other.localidad))
                        return false;
                if (nombre == null) {
                        if (other.nombre != null)
                                return false;
                } else if (!nombre.equals(other.nombre))
                        return false;
                if (pais == null) {
                        if (other.pais != null)
                                return false;
                } else if (!pais.equals(other.pais))
                        return false;
                if (region == null) {
                        if (other.region != null)
                                return false;
                } else if (!region.equals(other.region))
                        return false;
                if (secciones == null) {
                        if (other.secciones != null)
                                return false;
                } else if (!secciones.equals(other.secciones))
                        return false;
                return true;
        }
```

```
}
```

En **AuditarImpuestos definimos un HashMap** con **los diferentes tramos de los impuestos**. Nos devolverá un porcentaje

```
tablaImpuestos.put(100000L, 0.5);
		tablaImpuestos.put(500000L, 0.10);
		tablaImpuestos.put(1000000L, 0.15);
		tablaImpuestos.put(10000000L, 0.20);
		tablaImpuestos.put(50000000L, 0.25);
		tablaImpuestos.put(100000000L, 0.3);
		tablaImpuestos.put(1000000000L, 0.4);
```

DevuelveImpuestos nos devuelve los impuestos en función de las ganancias del equipo y **nuestro Map con los tramos. Recorremos el HashMap con un iterator hasta que encontramos el tramo adecuado** y **usamos ese porcentaje para calcular los impuestos con ganancias**. Si veis la tabla, **cuando las ganancias del HashMap son mayores que las ganancias del equipo** me he **pasado de tramo**. Por eso nos quedamos con la claveanterior a pasarnos de tramo.

```
	private static Double devuelveImpuestos(Double ganancias) {

	Iterator<Long> it = tablaImpuestos.keySet().iterator();

	Long clave = 0L, claveanterior=0L;



	while (it.hasNext()) {
		claveanterior=clave;
		clave = it.next();

		if (clave>ganancias)
			break;
	}
```

```
        return tablaImpuestos.get(clave)*ganancias;

        }
```

En la **clase AuditarImpuestos** tenemos un **método calculaImpuestos**. Lo calcula **a partir de las gananciasBrutas del equipo**. `equipo.getGananciasBrutas()`. El concepto **de Composite, es que de cara al exterior el comportamiento de gananciasBrutas** es **idéntico** para **un equipo simple, que para un equipo con muchas secciones**. A AuditarImpuestos **lo único que le importa son las gananciasBrutas**, **no el tipo de Equipo**. Ese **calculo interno de GananciasBrutas es transparente al exterior**. Todos los equipos **se comportan igual en nuestro programa**. Objetivo conseguido.

```
public static double calculaImpuestos(EquipoDeportivo equipo) {




                Double impuestos=  devuelveImpuestos(equipo.getGananciasBrutas());

                System.out.println("El equipo de nombre " + equipo.getNombre() +
                                        " paga de impuestos: " + impuestos);


                return impuestos*ganancias;




        }
```

AuditarImpuestos.java

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;

public class AuditarImpuestos {

	private static HashMap<Long, Double > tablaImpuestos = new HashMap<Lo
ng, Double >();

	static {

		tablaImpuestos.put(100000L, 0.5);
		tablaImpuestos.put(500000L, 0.10);
		tablaImpuestos.put(1000000L, 0.15);
		tablaImpuestos.put(10000000L, 0.20);
		tablaImpuestos.put(50000000L, 0.25);
		tablaImpuestos.put(100000000L, 0.3);
		tablaImpuestos.put(1000000000L, 0.4);



	}

	private static Double devuelveImpuestos(Double ganancias) {

	Iterator<Long> it = tablaImpuestos.keySet().iterator();

	Long clave = 0L, claveanterior=0L;


	while (it.hasNext()) {
		claveanterior=clave;
		clave = it.next();

		if (clave>ganancias)
			break;
	}
```

```
        return tablaImpuestos.get(clave)*ganancias;

        }



        public static double calculaImpuestos(EquipoDeportivo equipo) {



                Double impuestos=  devuelveImpuestos(equipo.getGananciasBrutas()) ;
                System.out.println("El equipo de nombre " + equipo.getNombre() +
                                        " paga de impuestos: " + impuestos);


                return impuestos;




        }

}
```

Por ultimo **la creación de equipos en la clase AplicaciónEquipos**. Tenemos un método generaEquipos() Como veis **eq1, Real Madrid lo creamos como Equipo Simple**, y luego **le añadimos secciones**. Al Granada, **eq2, lo creamos como Equipo Simple sin secciones**

```
  eq1 = new EquipoDeportivo("Real Madrid","Madrid","Madrid","España",0);


             EquipoDeportivo eqSeccion =
                            new EquipoDeportivo("Real Madrid CF","Madrid","Madrid","España",500000000);
```

```
            eq1.addSeccion(eqSeccion);


            EquipoDeportivo eqSeccion2 =
                        new EquipoDeportivo("Real Madrid Baloncesto","Madr
id","Madrid","España",100000000);


            eq1.addSeccion(eqSeccion2);


            EquipoDeportivo eqSeccion3 =
                        new EquipoDeportivo("Real Madrid Femenino","Madrid
","Madrid","España",1000000);



            eq1.addSeccion(eqSeccion3);



             eq2 = new EquipoDeportivo("Granada CF","Granada","Granada","Es
paña",40000000);
```

Y **calculamos los impuestos para los dos exactamente igual. Dos equipos, uno simple, y otro compuesto de equipos reciben el mismo tratamiento,** como Equipos Deportivos. Cerrando la explicación del patrón Composite.

```
AuditarImpuestos.calculaImpuestos(realMadrid);
                AuditarImpuestos.calculaImpuestos(granada);
```

AplicacionEquipos.java

```
package Composite;


public class AplicacionEquipos {


        private static EquipoDeportivo eq1;
        private static EquipoDeportivo eq2;
```

```
        public static void generaEquipos() {

                    eq1 = new EquipoDeportivo("Real Madrid","Madrid","Madrid","
España",0);

                EquipoDeportivo eqSeccion =
                                    new EquipoDeportivo("Real Madrid CF","Madri
d","Madrid","España",500000000);

                eq1.addSeccion(eqSeccion);

                EquipoDeportivo eqSeccion2 =
                                    new EquipoDeportivo("Real Madrid Baloncesto
","Madrid","Madrid","España",100000000);

                eq1.addSeccion(eqSeccion2);

                EquipoDeportivo eqSeccion3 =
                                    new EquipoDeportivo("Real Madrid Femenino",
"Madrid","Madrid","España",1000000);


                eq1.addSeccion(eqSeccion3);


                    eq2 = new EquipoDeportivo("Granada CF","Granada","Granada",
"España",40000000);

        }


          public static void main(String[] args) {

                 AplicacionEquipos.generaEquipos();

                EquipoDeportivo realMadrid= AplicacionEquipos.eq1;
                        EquipoDeportivo granada = AplicacionEquipos.eq2;
```

```
                        AplicacionEquipos.generaEquipos();


                        AuditarImpuestos.calculaImpuestos(realMadrid);
                        AuditarImpuestos.calculaImpuestos(granada);



            }

}
```

2.2 Practica independiente del patron Composite

Los alumnos crearan una clase banco con:

Nombre
Código
Dirección
Ganancias netas

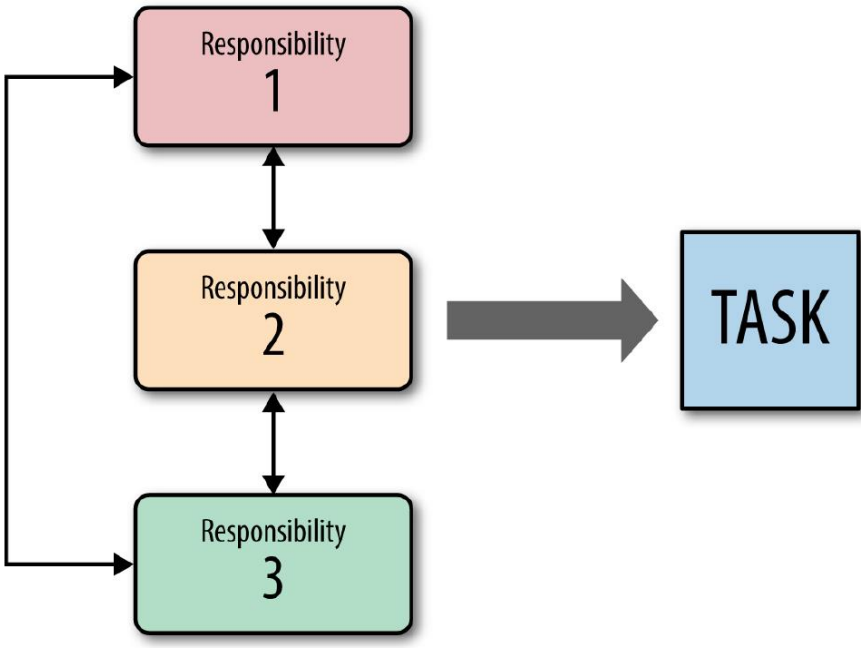
Teniendo en cuenta que un banco puede poseer otro banco, **crearan con el patrón composite un modelo de clases** para Banco teniendo en cuenta esta posibilidad.

3 Behavioral patterns. Patrones de comportamiento

Los patrones de comportamiento tratan con los algoritmos y el **reparto de responsabilidades entre objetos**. Los **patrones de comportamiento describen no solo patrones de objetos o clases**, sino también **los patrones de comunicación entre ellos**. Estos patrones ca**racterizan un flujo de control complejo que es difícil de seguir en tiempo de ejecución.** Alejan **su enfoque del flujo de control** para permitirle **concentrarse solo** en la **forma en que los objetos están interconectados**.
Los patrones de comportamiento a nivel de clase usan la herencia para **distribuir el comportamiento entre clases**. Ya hemos **visto dos de esos patrones**. El patrón template **es el más simple** y comun. **Consiste en dar una definición abstracta de un algoritmo**. Define **el algoritmo paso a paso**. Cada **paso invoca una operación abstracta o una operación primitiva**. Una **subclase concreta el algoritmo mediante la definición de las operaciones abstractas**. El **otro patrón de clase de comportamiento es interpreter**, que **representa una gramática como una jerarquía de clases** e implementa **un intérprete como una operación** entre **instancias** de estas clases.

Los **patrones de comportamiento a nivel de objeto utilizan la composición de objetos en lugar de la herencia**. Algunos **describen cómo un grupo de objetos del mismo nivel cooperan para realizar una tarea** que ningún objeto puede llevar a cabo por sí mismo. Un **problema importante en este punto** es cómo los objetos del mismo nivel se reconocen entre sí. Los **pares podían mantener referencias explícitas entre sí, pero eso aumentaría su acoplamiento**. En **un extremo, cada objeto sabría del otro**. El **patrón Mediator por ejemplo usa un objeto intermedio para la comunicación entre pares** y evitar dependencias directas entre pares.

En **total tenemos once patrones de comportamiento**. Como **ya hemos definido se encargan de controlar las comunicaciones entre objetos** y definir o repartir **responsabilidades.**



1. Chain of Responsibility
2. Command
3. Interpreter (de clase)
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method (de clase)
11. Visitor

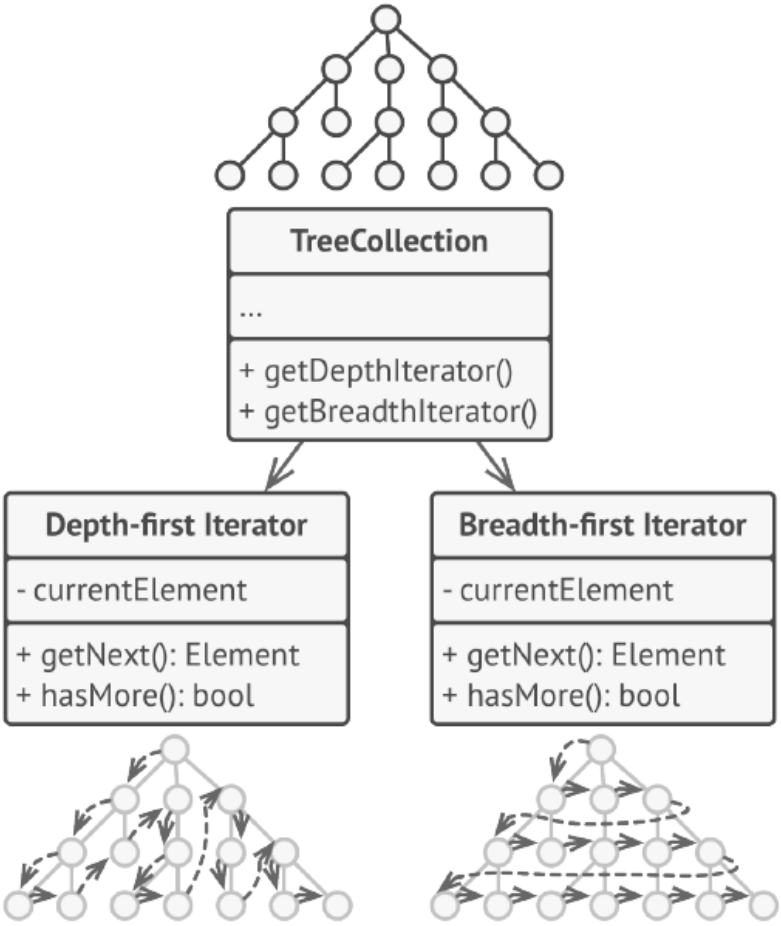
Vamos a definir y explicar brevemente los utilizados con colecciones en el siguiente apartado.

3.1 Iterator.

Proporciona una manera de tener acceso a los elementos de un colección o conjunto de objetos, **como una Array y se desea recorrer secuencialmente sin exponer cómo se representa internamente.**

Usar cuando: desea **acceder al contenido de una colección o conjunto de objeto**s sin **saber cómo se representa** internamente.

Ejemplo: tenemos un arbol de objetos y queremos recorrerlos de diferentes maneras. Podemos **añadir un iterator por cada manera de recorrerlo**, rama a rama, o **nivel a nivel del árbol.**



3.1.1 Ejemplo patron Iterator . Practica guiada patrón iterator

Vamos a realizar un ejemplo de iterator con la colección TreeSet de Java. En este **caso vamos a crearnos una clase MiTreeSet que extiende o hereda de TreeSet<String>.** El **iterator estándar de TreeSet recorre los elementos del conjunto de menor a mayor.** Para un String irá de la A a la Z, por ejemplo como veremos en la ejecución. Vamos a crear nuestro propio iterator, para que recorra el árbol en dirección contraria.

Hay **dos implementaciones, de Iterators nuevas**. Explicaremos la primera, intentad **entender la segunda**. Usamos dos clases para este ejemplo, **MiTreeSet.java y MainIterator.java.**
En la **función Main de MainIterator llenamos MiTreeSet** como sigue:

```
        tr.add("C");
                tr.add("B");
                tr.add("A");
                tr.add("F");
                tr.add("H");
                tr.add("E");
```

De manera que **cuando recorramos el árbol con el Iterator Standard del TreeSet nos** va a dar este resultado de ejecución, **en orden alfabético** que es el orden **natural de Comparable** para los Strings.

```
System.out.println("Creamos un set Tree, y lo recorremos normal con su iterat
or, nos dará orden alfabetico");


Iterator<String> it = tr.iterator();


                while (it.hasNext()) {


                        System.out.print(it.next() + ",");
                }
```

```
Creamos un set Tree, y lo recorremos normal con su iterator, nos dará orden
alfabetico
A,B,C,E,F,H,
```

Y ahora **hacemos el mismo recorrido pero usando el iterator que hemos creado nuevo**, y que vamos a explicar.

```
Iterator<String> itHaciaAtras = tr.getIteratorHaciaAtras();
```

```
                System.out.println("\nCreamos un set Tree, y lo recorremos en
direccion contraria con mi iterator creado ");
                while (itHaciaAtras.hasNext()) {


                        System.out.print(itHaciaAtras.next() + ",");
                }
```

Como vemos en la ejecución estamos recorriendo el Set de atrás adelante usando el **iterator que hemos implementado.** `tr.getIteratorHaciaAtras();`

```
Creamos un set Tree, y lo recorremos en direccion contraria con mi iterator
creado
H,F,E,C,B,A,
```

MainIterator.java

```
import java.util.Iterator;


public class MainIterator {


        public static void main(String[] args) {


                MiTreeSet tr = new MiTreeSet();


                System.out.println("Creamos un set Tree, y lo recorremos norm
al con su iterator, nos dará orden alfabetico");


                tr.add("C");
                tr.add("B");
                tr.add("A");
                tr.add("F");
                tr.add("H");
                tr.add("E");


                Iterator<String> it = tr.iterator();


                while (it.hasNext()) {
```

```
                        System.out.print(it.next() + ",");
                }

                Iterator<String> itHaciaAtras = tr.getIteratorHaciaAtras();

                System.out.println("\nCreamos un set Tree, y lo recorremos en
direccion contraria con mi iterator creado ");
                while (itHaciaAtras.hasNext()) {

                        System.out.print(itHaciaAtras.next() + ",");
                }

                Iterator<String> itHaciaAtrasComparator = tr.getIteratorHacia
AtrasConComparator();

                System.out.println("\nCreamos un set Tree, y lo recorremos en
direccion contraria con mi iterator creado con comparator ");
                while (itHaciaAtrasComparator.hasNext()) {

                        System.out.print(itHaciaAtrasComparator.next() + ","
);
                }

        }

}
```

Vamos a revisar ahora MiTreeSet. Observar como MiTreeSet, es un TreeSet porque extiende o hereda de TreeSet pero además ofrece nuevos Iterators.
Vamos a estudiar nuevo iterator que hemos introducido en la **función getIteratorHaciaAtras**. Y seguimos los siguientes pasos.

1. Copiamos el MiTreeSet, clonándolo con el constructor. (Ver el constructor)

```
treeset = new MiTreeSet(this);
```

2. Creamos **un Iterator nuevo de TipoString con clase anónima**, sobreescribiendo **los métodos del interfaz Iterator** que son dos.

    a. **hasNext() que devuelve true si hay siguiente. Habrá siguiente siempre que la longitud del treeset clonado sea mayor que cero**, sino devolvemos false y se termina de recorrer el Iterator.

```
@Override
                public boolean hasNext() {
// TODO Auto-generated method stub


                        if (treeset.size() <=0)
                                return false;
                        else
                                return true;
                }
```

    b. **El método Next()** nos **devolverá el siguiente elemento del Iterator**. En **este caso como estamos recorriendo de atrás a adelante,** devolvemos el **último elemento del Set en la copia**, y **lo borramos. Cuando ya no queden más elementos** en el **TreeSet, size()==0**, habremos **terminado de recorrer el TreeSet** con HasNext y Next.

```
@Override
                public String next() {


                        String resultado = treeset.last();
                        treeset.remove(resultado);
                        // TODO Auto-generated method stub
                        return resultado;
                }
```

3. La **función al final devuelve el Iterator creado** de **manera anónima.**

```
return miIterator;
```

```
public Iterator<String> getIteratorHaciaAtras() {
```

```
treeset = new MiTreeSet(this);

miIterator = new Iterator<String>() {



private String miElemento= treeset.last();

@Override
public boolean hasNext() {
        // TODO Auto-generated method stub

        if (treeset.size() <=0)
                return false;
        else
                return true;
}

@Override
public String next() {

        String resultado = treeset.last();
        treeset.remove(resultado);
        // TODO Auto-generated method stub
        return resultado;
}


};

return miIterator;
```

```
        }
```

Estudiad vosotros el otro método que funciona igualmente recorriendo de atrás adelante pero con un comparator, el método se llama `public Iterator<String>  getIteratorHaciaAtrasConComparator()` . Dará el mismo resultado.

```
Creamos un set Tree, y lo recorremos en direccion contraria con mi iterator
creado con comparator
H,F,E,C,B,A,
```

MiTreeSet.java

```
import java.util.Iterator;
import java.util.TreeSet;

public class MiTreeSet extends TreeSet<String> {

        private Iterator miIterator;
        private MiTreeSet treeset;


        public MiTreeSet() {


        }


        public MiTreeSet(TreeSet tr) {

                this.addAll(tr);
        }

        public Iterator<String> getIteratorHaciaAtras() {
```

```
            treeset = new MiTreeSet(this);

            miIterator = new Iterator<String>() {



            private String miElemento= treeset.last();

            @Override
            public boolean hasNext() {
                    // TODO Auto-generated method stub

                    if (treeset.size() <=0)
                            return false;
                    else
                            return true;
            }

            @Override
            public String next() {

                    String resultado = treeset.last();
                    treeset.remove(resultado);
                    // TODO Auto-generated method stub
                    return resultado;
            }


            };

            return miIterator;



    }
```

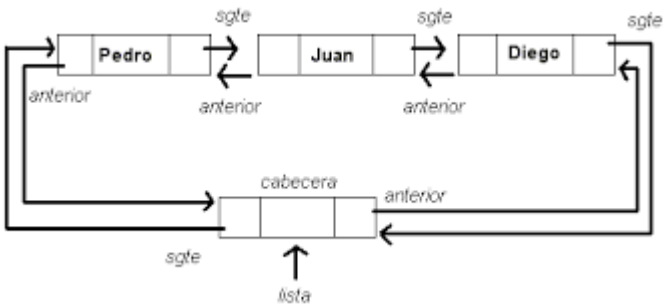
```
public Iterator<String> getIteratorHaciaAtrasConComparator() {

                TreeSet<String> trCopia = new TreeSet<String>((s1,s2)-> s1.co
mpareTo(s2)*-1);

                trCopia.addAll(this);

                return trCopia.iterator();



        }


}
```

3.2 *Practica independiente patron iterator.*

Los alumnos crearán la clase ListaPropia. En esta clase contendrá tres propiedades:

Dato: de tipo String.
NodoAnterior de tipo ListaPropia
NodoSiguiente de tipo ListaPropia.

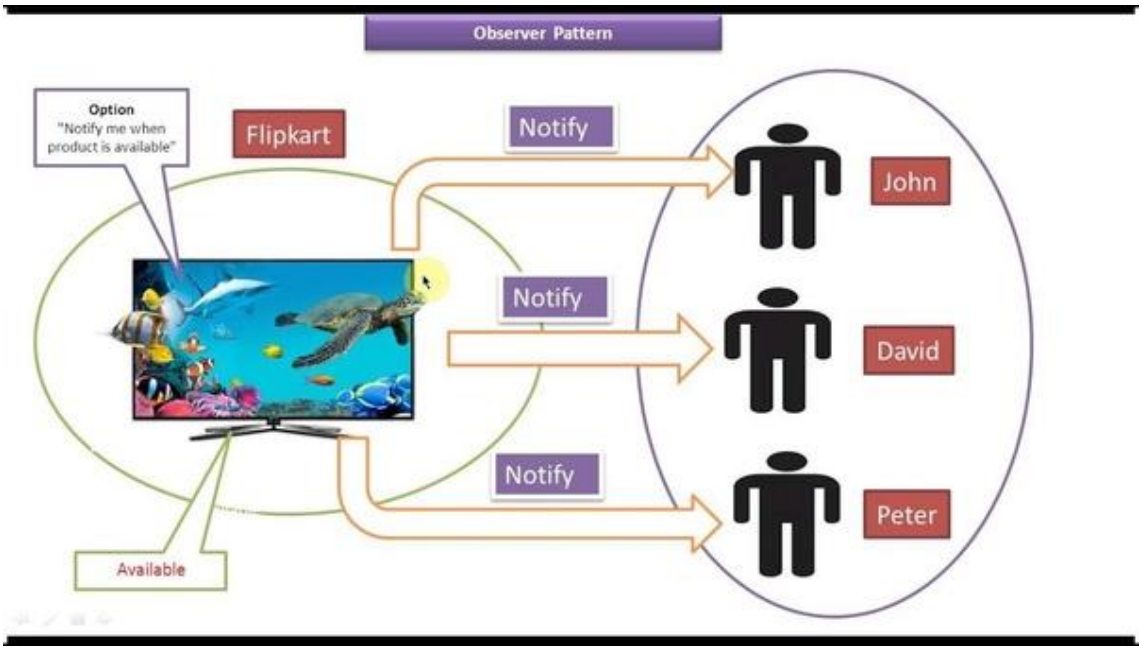
Aplicando el patrón iterator crearan un Iterador para recorrer la lista hacia atrás.



3.3 Observer

Observer es un **patrón de diseño de comportamiento que te permite definir** un mecanismo **de suscripción para notificar a varios objetos** sobre cualquier **evento o cambio de estado.**

Usar cuando: un **cambio en un objeto requiere cambiar otros**.
Ejemplo: se usa en cualquier sistema **de notificaciones en tu móvil**, cuando **te suscribes a una página web**, cuando te **suscribes a un canal de Youtube**, para **grupos de chat**, **comercio electrónico** etc. En el siguiente ejemplo se notifica al usuario cuando un nuevo producto está disponible.



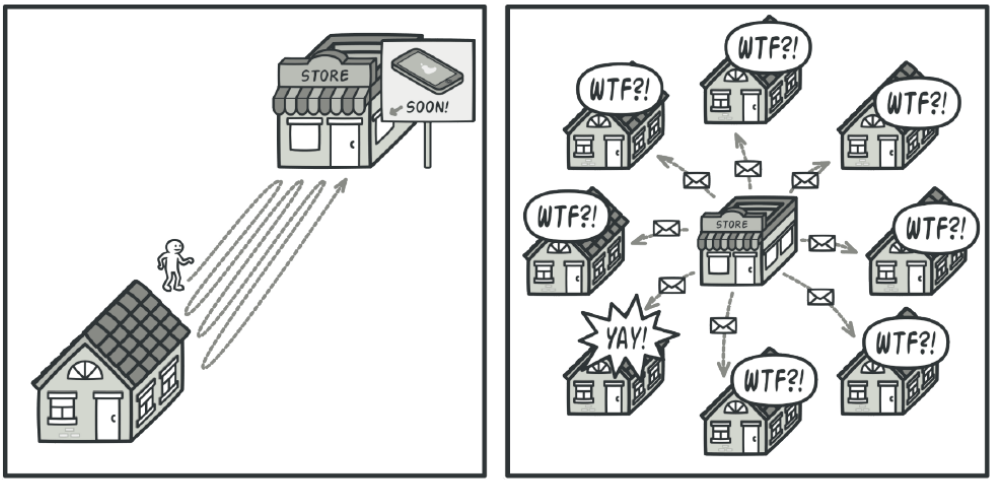
Se usa mucho en chats, **subscripciones a canales de redes sociales, de sitios web, para Streaming, web reactivas, robótica, sensores**, etc. En resumen, **es uno de los patrones con mucha aplicación** en la actualidad, conviene que **lo entendáis bien.**

Intent

- Defina **una dependencia uno a muchos entre objetos para que cuando un objeto cambie de estado**, todos **sus dependientes sean notificados y actualizados** automáticamente.
- **Encapsular los componentes principales** (o comunes o motor) en una **abstracción sujeto y los componentes variables** (un interfaz de usuario, una aplicación cliente que consume datos) en una **jerarquía de** observers.
- La parte "**Visual" de Model-View-Controller,** un patrón de diseño de **aplicaciones que veréis el año que viene**, sería **el observer** en este caso.

Problema

Imaginad que teneis dos tipos de objetos: un tipo cliente y un tipo Tienda. El cliente está muy interesado en una marca en particular de producto (por ejemplo, es un nuevo modelo del iPhone) que debe estar disponible en la tienda pronto.
El cliente podía visitar la tienda todos los días y comprobar el producto. Disponibilidad. Pero mientras el producto todavía está en camino, la mayoría de estos estos viajes no tienen sentido.



Visiting the store vs. sending spam

Por otro lado, **la tienda podría enviar muchísimos correos** electrónicos (que puede considerarse spam) a todos los clientes cada vez que un nuevo producto está disponible. Esto **ahorraría a algunos clientes desde interminables viajes** a la tienda. Al mismo tiempo, **habría molestado a otros clientes** que no están interesados en nuevos productos, **con correos SPAM.**

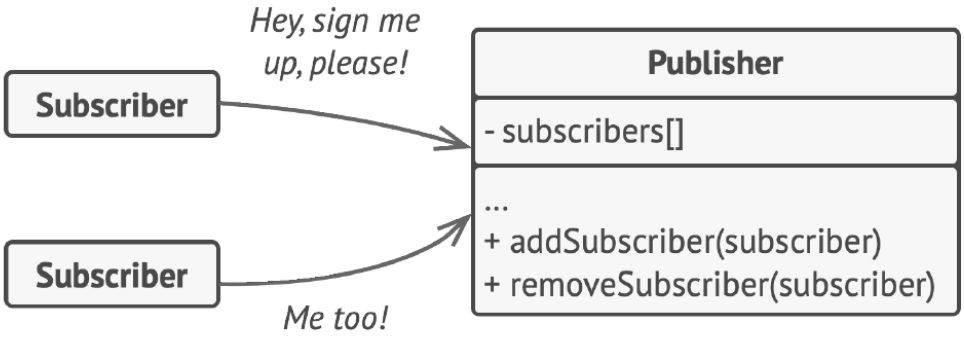
Solution

El objeto que tiene algún estado interesante a menudo se llama subject, *sujeto*, pero **ya que también va a notificar a otros objetos** sobre **los cambios en su estado**, lo **llamaremos *publisher***. Todos los **demás objetos que desea realizar un seguimiento de los cambios en el estado** del editor se **denominan *subscriber* *son los observers***.

El patrón Observer propone que se agregue una suscripción mecanismo a la clase de publisher para que **los objetos individuales puedan suscribirse o darse de baja** de una **corriente de eventos que viene** de ese **publisher**.
Resolverlo es bastante simple. Para implementar este mecanismo podemos

usar:

1) **Un array o colección campo para almacenar una lista de referencias** a objetos observers o subscriptores.
2) Varios métodos públicos que permiten agregar suscriptores a y eliminarlos de esa lista.



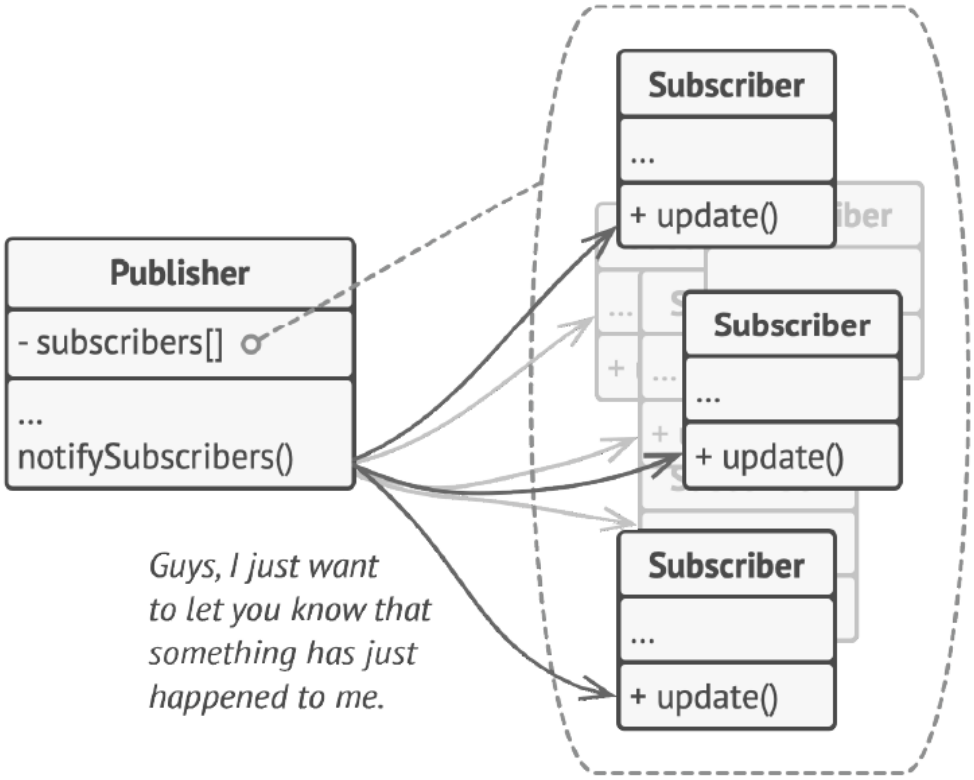
A subscription mechanism lets individual objects subscribe to event notifications.

Ahora, **cada vez que le sucede un evento importante al editor**, **examine el estado de sus suscriptores y llama a los metódos de notificación** específica **necesarios con respecto a ese cambio de estado**, por ejemplo **que el nuevo Iphone ha llegado** sería un cambio de estado .

Las **aplicaciones reales podrían tener decenas de clases de suscriptores** diferentes que están **interesados en rastrear eventos del mismo publisher**, por ejemplo, tienda, almacen, cliente anonimo, premium, etc.

No **queremos acoplar al editor todos esas Clases**, crearíamos **demasiadas subclases**. Además, es posible que ni **siquiera conozcamos algunas de esas clases** de antemano si tu **clase de publisher** se supone que **debe ser utilizado por otros desarrolladores**.

Es por eso **que es crucial que todos los suscriptores implementen lo mismo interfaz** y que el publisher se comunica con ellos sólo a través de esa interfaz. Esta interfaz debe declarar un método de notificación, con un conjunto de parámetros que el editor puede utilizar para pasar algunos datos contextuales junto con la notificación.

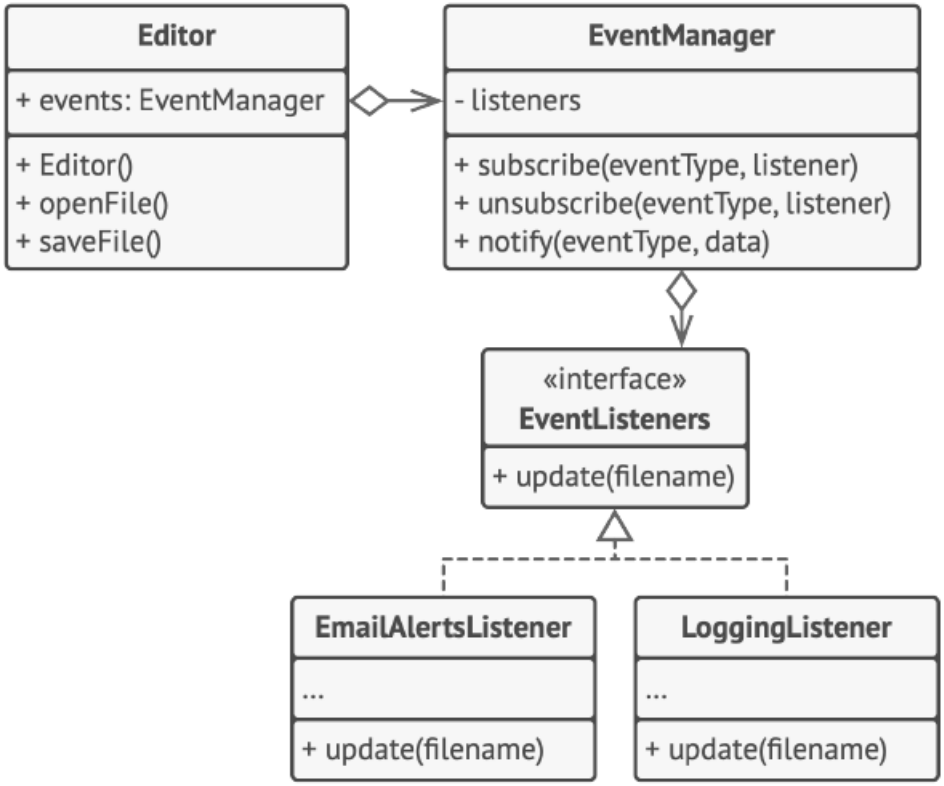


Si **la aplicación tiene varios tipos diferentes de publisher** y quieren **hacer que sus suscriptores sean compatibles con todos ellos**, puede ir aún más lejos y hacer que todos los publisher sigan la misma interfaz. Esta interfaz sólo tendría que describir algunos métodos de suscripción. La interfaz permitiría a los suscriptores para observar los estados de los editores sin acoplamiento a su clases concretas.

Ejemplo de problema resuelto con Observer

Tenemos un Editor de texto como OneDrive, en la que **reacciona a eventos, porque podemos compartir documentos con otros usuarios**, y que **estos otros usuarios lo modifiquen**. Cuando un **documento de OneDrive cambia de estado**, porque alguien lo ha modificado, **podemos notificar a los usuarios que comparten el documento**, vía email, o **en la propia aplicación si esta logeado**, el **usuario elige el método** de **suscripción.**

Fijaos **como EventListeners es el interfaz que implementan los Observers**, **Email, y Logging**. El **EventManager es el objeto Publisher**, el **que notificará a los Observer cada vez que haya un cambio de estado**. El Editor de Texto el Subject, es el que está sujeto al cambio de estado.



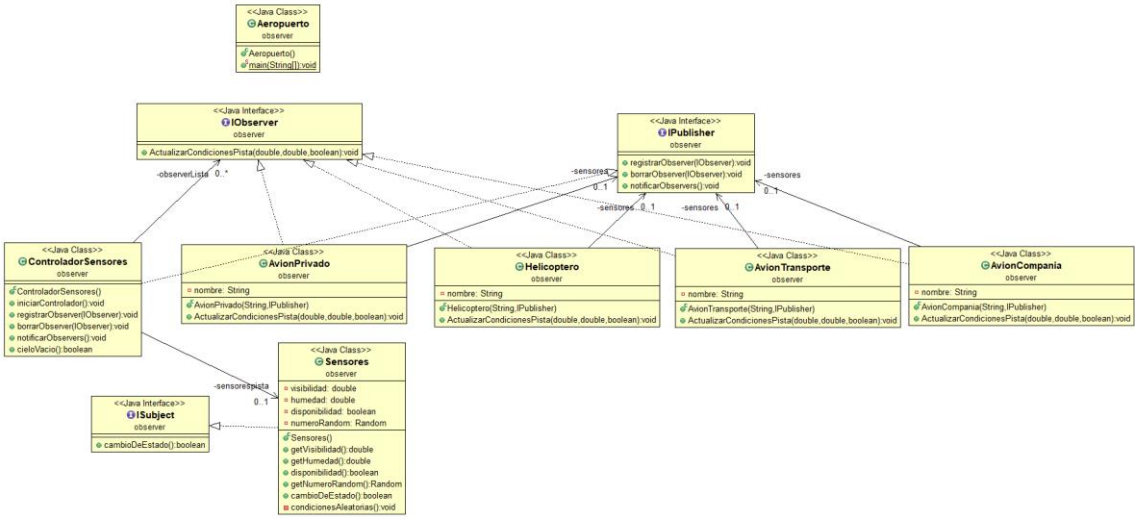
3.3.1 Ejemplo implementado en Java de Observer. Practica guiada Observer.

Tenemos **un aeropuerto que recibirá aparatos de distinto tipo para aterrizar**. Pero **los aparatos sólo pueden aterrizar si las condiciones climatológicas** de la pista son **correctas**. Componentes :

- **Nuestro Subject**, sujeto **serán los sensores de la pista**, que cambia de estado cada tiempo (perdemos el tiempo con un bucle for). Sería **nuestro Publisher**, es el que **publica los cambios de estado**
- **Nuestro Subscriber un programa que controla los sensores**, **permite el registro de observers** y **notifica a nuestros observer**.
- **Nuestros Observers, serán los aviones y helicópteros** que **quieren aterrizar**, y por tanto se subscriben al servicio. Serán **nuestros Subscriber**. Cuando **los aviones hayan aterrizado quitaran las subscripción** o registro al Servicio.

- El **programa principal que crea 100 aparatos e inicializa el controlador de sensores.** Cuando **no haya más aviones en el cielo**, el **programa se termina.**

Nuestro **modelo de clases para nuestro programa. Montad el ejemplo antes de leer los apuntes.**



Sensores, el subject.

Vamos **exponiendo las diferentes clases y explicando el programa.** Empezamos con el sensor, la clase Sensores **que implementa el Interfaz ISubject** con un método para conocer el **cambio de estado del sensor, cambioDeEstado().** Podemos **así obtener sus tres datos,** posteriormente, **desde nuestro controlador ControladorSensores.**
ISubject es el interfaz para nuestro **Subject sensores**, define :

```
public boolean cambioDeEstado();
```

Tened en cuenta que es una simulación de un sensor. Los **sensores tardan un tiempo en responder entre cambios**, lo **simulamos con un bucle for, que pierde tiempo**, en cambioDeEstado().
Después **generamos tres datos aleatorios para visibilidad, disponibilidad y humedad** en `condicionesAleatorias()`.

```
public boolean cambioDeEstado() {

```

```
                for (double i = 1; i < 1000000; i++) {

                    // Perdemos tiempo
            }

            condicionesAleatorias();
```

ISubject.java

```
public interface ISubject {


        public boolean cambioDeEstado();

}
```

Sensores.java

```
import java.util.Random;

public class Sensores implements ISubject{

        private double visibilidad, humedad = 0;

        private boolean disponibilidad = false;

        private Random numeroRandom = new Random();

        public double getVisibilidad() {
                return visibilidad;
        }
```

```
	public double getHumedad() {
		return humedad;
	}


	public boolean disponibilidad() {
		return disponibilidad;
	}



	public Random getNumeroRandom() {
		return numeroRandom;
	}



	public boolean cambioDeEstado() {

		for (double i = 1; i < 1000000; i++) {

			// Perdemos tiempo
		}

		condicionesAleatorias();

		return true;

	}

	private void condicionesAleatorias() {

		visibilidad = numeroRandom.nextDouble() * 100;
		humedad = numeroRandom.nextDouble() * 100;

		if (numeroRandom.nextInt(2) == 1)
```

```
                        disponibilidad = true;
                else
                        disponibilidad = false;

        }

}
```

La clase ControladorSensores, el Publisher

Para el Publisher, el encargado de controlar los sensores también definimos un interfaz, IPublisher, con tres métodos definidos para registrar, borrar el registro y notificar a nuestros observers.

IPublisher.java

```
public interface IPublisher {
        public void registrarObserver(IObserver o);
        public void borrarObserver(IObserver o);
        public void notificarObservers();
}
```

Y **seguimos con la clase ControladorSensores**, que **funcionará como un publisher**, recogiendo **los datos del sensor cuando haya un cambio y comunicándolo a todos los Observers**, los **aparatos que tienen que aterrizar**. Los **observers son guardados y borrados de un HashSet**, propiedad **privada de la clase**. Crearemos **unos sensores en el constructor de la clase**, guardados en sensorespista.

```
private HashSet<IObserver> observerLista

private Sensores sensorespista;
```

El **método iniciaControlador**, nos **permite iniciar el controlador y dar respuesta a nuestros Observers registrados**. Mientras **en el cielo haya aviones el controlador no acaba**. Escucha cambios de estado del sensor, y cuando se produce notifica a los observers.

```
if (sensorespista.cambioDeEstado())
                        notificarObservers();
```

```
public void iniciarControlador() {

            while(!cieloVacio()) {

                  if (sensorespista.cambioDeEstado())
                        notificarObservers();
            }

}
```

Registrarobserver y borrarObserver, añade o quita el observer del HashSet. Son **sencillos**. Por último, **notificarObserver, recorre el HashSet con un Iterator, y actualiza las condiciones de pista** para cada **Observer** llamando **al método del Observer** actualiza condiciones de pista. **Clonamos la lista** para que no haya interferencias de observers que han aterrizado, y la recorremos con el Iterator.

```
HashSet<IObserver> listaObservers = (HashSet<IObserver> ) observerLista.clone
();

Iterator<IObserver> it = listaObservers.iterator();
                  IObserver ob;
                  while (it.hasNext()) {
```

Avisamos a cada Observer de que las **condiciones de la pista de aterrizaje han cambiado**. Y con esto **hemos terminado con el Publisher**. Vamos con los **Observers.**

```
ob.ActualizarCondicionesPista(
```

```
                                    sensorespista.getVisibilidad(),
        sensorespista.getHumedad(), sensorespista.disponibilidad());
```

ControladorSensores.java

```
import java.util.HashSet;
import java.util.Iterator;
public class ControladorSensores implements IPublisher{

        private HashSet<IObserver> observerLista = new HashSet<IObserver>();




        private Sensores sensorespista;

        public ControladorSensores() {

                sensorespista = new Sensores();


        }

        public void iniciarControlador() {

                while(!cieloVacio()) {

                        sensorespista.cambioDeEstado();
```

```
                        notificarObservers();
                }

        }

        @Override
        public void registrarObserver(IObserver o) {
                // TODO Auto-generated method stub
                observerLista.add(o);

        }

        @Override
        public void borrarObserver(IObserver o) {
                // TODO Auto-generated method stub
                observerLista.remove(o);

        }




        @Override
        public void notificarObservers() {
                // TODO Auto-generated method stub


                        HashSet<IObserver> listaObservers = (HashSet<IObserver> ) observerLista.clone();

                        Iterator<IObserver> it = listaObservers.iterator();
                        IObserver ob;
                        while (it.hasNext()) {

                        ob=it.next();
```

```
                        ob.ActualizarCondicionesPista(
                                sensorespista.getVisibilidad(),
                                                sensorespista.getHumedad(),
sensorespista.disponibilidad());

                        }




        }

        public boolean cieloVacio() {

                return (observerLista.size()<=0);

        }

}
```

Los Observers, Aviones y Helicopteros.

Nota: Podíamos **haber llamado Subscribers a los Observers**, pero como estamos haciendo un ejemplo de Sensores, **he preferido dejarlo como Observers**, ambos son válidos.

Como **ya hemos dicho los aparatos están en el aire esperando a ver que las condiciones sean favorables** para **aterrizar**, humedad, visibilidad, y que la pista este disponible. Todos los aparatos **implementan un interfaz común IObserver**, que define el **método actualizaCondicionesPista**. Cuando el publisher llama a este Método, el Observer reacciona a los cambios y comprueba si puede aterrizar.

IObserver.java

```
public interface IObserver {


        public void ActualizarCondicionesPista(double visibilidad, double hum
edad, boolean disponibilidad);


}
```

Tenemos **cuatro clases que implementan el Observer, AvionCompania, AvionPrivado, AvionTranporte y Helicóptero**. Todas se comportan igual en el ejemplo, pero podría haber diferencias en la manera de aterrizar o dependiendo de las condiciones de la pista.
Empezamos **a explicar AvionCompania**, el resto son iguales. Y lo primero **es ver su constructor**, que **recibe como parámetro un nombre y un parámetro de tipo IPublisher,** que en el programa principal será de tipo **ControladorSensores**, nuestra **implementación de Publisher**. Conforme **se crea un objeto de la clase se registra** en el **Publisher** . Eso quiere decir que **el avión quiere aterrizar**.

```
public AvionCompania(String nombre , IPublisher sensores)
this.sensores.registrarObserver(this);
```

Lo siguiente **es el método con el que el Subscriber** avisa **al AvionCompania** de que las **condiciones de la pista han cambiado** `public void ActualizarCondicionesPista`. El **avión** pierde algo de tiempo en aterrizar**, con el for**, y **comprueba que las condiciones para aterrizar,** hay disponibilidad y la visibilidad y humedad están dentro de los **umbrales razonables para el aterrizaje**. Si **aterriza al avión ya no le hace falta conocer datos** de los sensores, se borra de la subscripción con sensores.borrarObserver(this);.

```
if (disponibilidad && visibilidad > 0.5 && humedad >95 ) {


                    System.out.println("Avion de compañia Aterriza de nombre:"
+ nombre);
                    sensores.borrarObserver(this);
```

```
@Override
```

```
        public void ActualizarCondicionesPista(double visibilidad, double hum
edad, boolean disponibilidad) {

                for (double i=1; i<1000000 ;i++) {

                        //Perdemos tiempo
        }

                if (disponibilidad && visibilidad > 0.5 && humedad >95 ) {

                        System.out.println("Avion de compañia Aterriza de no
mbre:" + nombre);
                        sensores.borrarObserver(this);



                }




        }
```

El **resto de Observers son iguales, os dejo el código** pero no hay nada más que explicar. Importante,

Importante. Programamos para las abstracciones no para las implementaciones.
AvionCompania, no recibe un ControladorSensores el Publisher, recibe su interfaz IPublisher.
`public AvionCompania(String nombre , IPublisher sensores)`
Podríamos cambiar de Publisher y sería transparente para los Observers.

AvionCompania.java

```
public AvionCompania(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
```

```
                this.sensores.registrarObserver(this);

        }
```

AvionCompania.java

```
public class AvionCompania implements IObserver {

        private String nombre;;
        private IPublisher sensores;



        public AvionCompania(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
                 this.sensores.registrarObserver(this);

        }

        @Override
        public void ActualizarCondicionesPista(double visibilidad, double hum
edad, boolean disponibilidad) {

                for (double i=1; i<1000000 ;i++) {

                        //Perdemos tiempo
        }

                if (disponibilidad && visibilidad > 0.5 && humedad >95 ) {

                        System.out.println("Avion de compañia Aterriza de no
mbre:" + nombre);
                        sensores.borrarObserver(this);
```

```
                }




        }




}
```

AvionPrivado.java

```
public class AvionPrivado implements IObserver {

        private String nombre;;
        private IPublisher sensores;

        public AvionPrivado(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
                this.sensores.registrarObserver(this);

        }

        @Override
        public void ActualizarCondicionesPista(double visibilidad, double hum
edad, boolean disponibilidad) {

                for (double i=1; i<1000000 ;i++) {

                        //Perdemos tiempo
        }
```

```
                if (disponibilidad && visibilidad > 0.5 && humedad >95 ) {


                        System.out.println("Avion privado Aterriza de nombre
:" + nombre);


                        sensores.borrarObserver(this);
                }


        }







}
```

AvionTransporte.java

```
public class AvionTransporte implements IObserver {


        private String nombre;;
        private IPublisher sensores;


        public AvionTransporte(String nombre , IPublisher sensores) {


                this.nombre=nombre;
                this.sensores=sensores;
                this.sensores.registrarObserver(this);


        }


        @Override
        public void ActualizarCondicionesPista(double visibilidad, double hum
edad, boolean disponibilidad) {


                for (double i=1; i<1000000 ;i++) {


                        //Perdemos tiempo
```

```
        }

                if (disponibilidad && visibilidad > 0.1 && humedad <99 ) {

                        System.out.println("Avion de transporte Aterriza de n
ombre:" + nombre);

                        sensores.borrarObserver(this);
                }

        }

}
```

Helicóptero.java

```
public class Helicoptero implements IObserver {
        private String nombre;;
        private IPublisher sensors;



        public Helicoptero(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
                 this.sensores.registrarObserver(this);

        }

        @Override
        public void ActualizarCondicionesPista(double visibilidad, double hum
edad, boolean disponibilidad) {
                // TODO Auto-generated method stub

        }
}
```

El programa principal, Aeropuerto

Creamos el ControladorSensores contenido en una variable de tipo Interfaz IPublisher.

```
ControladorSensores controladorsensor = new ControladorSensores() ;
```

Creamos 400 aviones en el for

```
        for (int i=0; i<100; i++) {
              new AvionCompania("avioncomp" +i , controladorsensor);
              new AvionPrivado("avionpri" +i ,controladorsensor);
```

E inicializamos nuestro ControladorSensores

```
controladorsensor.iniciarControlador();
```

Aeropuerto.java

```
import java.util.Scanner;
public class Aeropuerto {


          public static void main(String[] args) {


          boolean finPrograma= false;
```

```
        ControladorSensores controladorsensor = new ControladorSensores() ;



        for (int i=0; i<100; i++) {

                new AvionCompania("avioncomp" +i , controladorsensor);
                new AvionPrivado("avionpri" +i ,controladorsensor);
                new AvionTransporte("aviontransporte" +i, controladorsensor);

                new AvionTransporte("helicoptero" +i, controladorsensor);


        }


        controladorsensor.iniciarControlador();




}


}
```

Un ejemplo de ejecución sería. Probadlo.

```
Avion de compañia Aterriza de nombre:avioncomp97
Avion privado Aterriza de nombre:avionpri51
Avion de compañia Aterriza de nombre:avioncomp66
Avion de compañia Aterriza de nombre:avioncomp80
Avion de compañia Aterriza de nombre:avioncomp0
Avion privado Aterriza de nombre:avionpri17
```

```
Avion privado Aterriza de nombre:avionpri83
```

3.4 Practica independiente del patrón observer

1. Añadir al modelo anterior CocheVolador que implementará el IObserver, como clase que pueda subscribirse al servicio de sensores
2. En el programa principal crear unos cuantos Coches Voladores que aterrizarán en el aeropuerto.

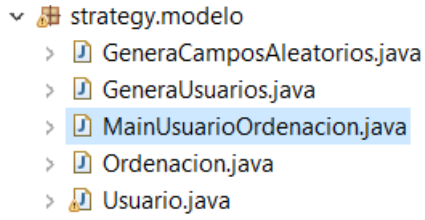
4 Patrones de diseño en programación funcional

Uno de **los problemas que vamos a intentar solucionar con programación funcional es reducir el tamaño de nuestros modelos de clases**. La resolución de problemas con **programación orientada a objetos produce la situación de que en programas complejos** el **modelo de clases crezca tanto que en ocasiones se hace inabarcable**. Aplicando clases anónimas y expresiones lambda podemos reducir esa complejidad.
Además, la **programación funcional va a hacer más flexibles y seguros el mantenimiento y seguridad de nuestros programas**. Igualmente **reducirá el número de errores.** Como ya hemos visto **ciertas características de la programación funcional** como la **inmutabilidad** y el **uso de Streams reducen el número de errores** en nuestro código. Empezaremos **realizando un ejemplo de patrón Strategy que vimos en el tema anterior pero Funcional.**

4.1 Patrón Strategy. Pratica guiada patrón Strategy

Como vimos en el tema anterior el patrón Strategy nos **permite modificar la algoritmia o el comportamiento de una clase en tiempo de ejecución**. Vamos a **aprovechar los aprendido en este tema del operador :: y el paso de funciones como parámetros** para aplicar el **patron Strategy de una manera funcional** con lambdas o funciones como parámetros.

Vamos **a explicar el patrón Strategy funcional sobre un ejemplo**. Copiamos el modelo de los apuntes de **Introducción a programación funcional 5,** que contenían la clase **Usuario.java, GeneraCamposAleatorios.java, GeneraUsuarios.java** y añadimos dos clases nuevas, Ordenación.java y MainUsuario.java Para que nos quede un proyecto de este tipo:



En **la clase ordenación definimos una función que nos devuelve una lista ordenada**, a **partir de un método de una estrategia de ordenación** que será un **Comparator**.

```
        public  List<Usuario> getListaOrdena(List<Usuario> listaUsuarios, Com
parator<Usuario> estrategiaMetodoOrdenacion) {
```

También **definimos cuatro estrategias de ordenación. Dos con funciones y dos con expresiones lambda**, para **que podáis ver que se puede hacer de las dos maneras**. Las **expresiones lambda las definimos como constantes estáticas.** Fijaos en la lamba de **ordenaHorasAscendente**, es un **Comparator** que ordena de **menor a mayor.**

Para **ordenaHorasDescendente**, usamos **ordenaHorasAscendente como base**, y **obtenemos su reverso** con `ordenaHorasAscendete.reversed();`.

```
public static Comparator<Usuario>  ordenaHorasAscendete =


                    (u1,u2) -> u1.getHorasDeUso()>u2.getHorasDeUso()?1
                                :( u1.getHorasDeUso()==u2.getHorasDeUso()?0:-
1);


                    public  static  Comparator<Usuario>   ordenaHorasDescendente
=
                                ordenaHorasAscendete.reversed();
```

Las **otras dos estrategias son definidas con funciones que implementan el Comparator**, reciben **dos usuarios como parámetro** y devuelve 1, 0 o -1, un entero.

```
public int ordenaNombreAscendente(Usuario usuario1,Usuario usuario2) {


            return usuario1.getNombre().compareTo(usuario2.getNombre());



      }
```

```
	public int ordenaNombreDescendente(Usuario usuario1,Usuario usuario2) {


		return ordenaNombreAscendente(usuario1,usuario2)*-1;
```

Ordenación.java

```
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class Ordenacion {


	public static Comparator<Usuario>  ordenaHorasAscendete =

			(u1,u2) -> u1.getHorasDeUso()>u2.getHorasDeUso()?1
					:(  u1.getHorasDeUso()==u2.getHorasD
eUso()?0:-1);

			public static Comparator<Usuario>  ordenaHorasDescen
dente =
					ordenaHorasAscendete.reversed();




	public int ordenaNombreAscendente(Usuario usuario1,Usuario usuario2)
{

		return usuario1.getNombre().compareTo(usuario2.getNombre());


	}
```

```
        public int ordenaNombreDescendente(Usuario usuario1,Usuario usuario2)
{



                return ordenaNombreAscendente(usuario1,usuario2)*-1;


        }


        public  List<Usuario> getListaOrdena(List<Usuario> listaUsuarios, Com
parator<Usuario> estrategiaMetodoOrdenacion) {







                return  listaUsuarios.stream().sorted(estrategiaMetodoOrdenac
ion).collect(Collectors.toList());



        }


}
```

En **MainUsuarioOrdenacion** definimos **una lista de veinte usuarios**, y una **objeto de tipo Ordenación.**

```
List<Usuario> listaUsuarios = GeneraUsuarios.devueveUsuariosLista(20);


            Ordenacion ordenaciones = new Ordenacion();
```

Posteriormente aplicamos las cuatro estrategias de ordenación definidas.

```
List<Usuario> listaNombres =


                        ordenaciones.getListaOrdena(listaUsuarios,  ordenaci
ones::ordenaNombreAscendente);
```

```
listaNombres= ordenaciones.getListaOrdena(listaUsuarios, ordenaciones::ordena
```

```
NombreDescendente);
```

```
List<Usuario> listaHoras = ordenaciones.getListaOrdena(listaUsuarios, Ordenac
ion.ordenaHorasAscendete);
```

```
listaHoras = ordenaciones.getListaOrdena(listaUsuarios, Ordenacion.ordenaHora
sDescendente);
```

Y **definimos una estrategia de ordenación extra en tiempo de ejecución**, con **una lambda para el número de conexiones ascendente.**

```
      List<Usuario> listaNumConexiones = ordenaciones.getListaOrdena(listaUsu
arios,
                           (u1,u2)->  u1.getNumConexiones()>u2.getNumConexiones
()?1:
                                      (u1.getNumConexiones()==u2.getNumConexiones()
?0:-1)
```

MainUsuarioOrdenacion.java

```
import java.util.List;


public class MainUsuarioOrdenacion {



        public static void main(String[] args) {


                List<Usuario> listaUsuarios = GeneraUsuarios.devueveUsuariosL
ista(20);
```

```
                Ordenacion ordenaciones = new Ordenacion();

                System.out.println("Lista sin ordenar");
                listaUsuarios.stream().forEach((u)-> System.out.println(u));



                System.out.println("Lista  ordenada sobre nombre ascendente")
;

                List<Usuario> listaNombres =

                                ordenaciones.getListaOrdena(listaUsuarios, o
rdenaciones::ordenaNombreAscendente);

                listaNombres.stream().forEach((u)-> System.out.println(u));

                System.out.println("Lista  ordenada sobre nombre descendente"
);



                listaNombres= ordenaciones.getListaOrdena(listaUsuarios, orde
naciones::ordenaNombreDescendente);







                System.out.println("Lista  ordenada sobre horas ascendente");

                List<Usuario> listaHoras = ordenaciones.getListaOrdena(listaU
suarios, Ordenacion.ordenaHorasAscendete);

                listaHoras.stream().forEach((u)-> System.out.println(u));
```

```
        System.out.println("Lista  ordenada sobre horas descendente");

                  listaHoras = ordenaciones.getListaOrdena(listaUsuarios, Orde
nacion.ordenaHorasDescendente);



                listaHoras.stream().forEach((u)-> System.out.println(u));


System.out.println("Lista  ordenada sobre numero conexiones  ascendente");

                List<Usuario> listaNumConexiones = ordenaciones.getListaOrden
a(listaUsuarios,
                                    (u1,u2)-> u1.getNumConexiones()>u2.getNumCon
exiones()?1:
                                                (u1.getNumConexiones()==u2.getNumCo
nexiones()?0:-1)

                                    );



                listaNumConexiones.stream().forEach((u)->  System.out.println
(u));
        }

}
```

4.2 Practica Independiente del Patrón Strategy

1. Crear una clase EstrategiaAgregación.java con los métodos

Double Suma (List<Integer> lista)
Double Media (List<Integer> lista)
Double Max (List<Integer> lista)
Double Min (List<Integer> lista)

Crear una clase principal Principal.java con programa principal que :

1. Genere una lista de Integer de manera aleatoria usando la función Math.Random y la API Stream

2. Un método estático Double AplicarStrategia (Function<Integer,Double> estrategia).
3. Aplicar las cuatro estrategias de la clase EstrategiaAgregación.java mostrando los resultados por pantalla.

5 Bibliografía y referencias web

Referencias web

Tutoriales de Java arquitectura Java

https://www.arquitecturajava.com/

Bibliografía

Dive Into DESIGN PATTERNS, Alexander Shvets, Refactoring.Guru, 2020