

# Trabajo Integrador: Vinchucas

## Integrantes:

Carnevale Facundo  
([carnevalefacundo@gmail.com](mailto:carnevalefacundo@gmail.com)).  
Soares Nicole  
([nicolesoares918@gmail.com](mailto:nicolesoares918@gmail.com)).  
Yegro Tobias Agustín  
([tobiyeegro@gmail.com](mailto:tobiyeegro@gmail.com)).

## Materia:

Programación Orientada A  
Objetos 2

## Fecha de Entrega:

19/06/2025

## Profesores:

- Butti Matias
- Cano Diego
- Torres Diego



## ÍNDICE

<b>Patrones de diseño utilizados y los roles según la definición de Gamma et. al.....</b>	<b>1</b>
Patrón observer.....	1.1
Patrón State.....	1.2
Patrón Composite.....	1.3
<b>Decisiones de diseño y detalles de implementación.....</b>	<b>2</b>
Aplicación web.....	2.1
Muestras.....	2.2
Verificación de las muestras.....	2.3
Opiniones.....	2.4
Usuarios.....	2.5
Ubicaciones.....	2.6
Las organizaciones.....	2.7
Zonas de cobertura.....	2.8
Filtros.....	2.9
Filtro Compuesto.....	2.10

## Patrones de diseño utilizados y los roles según la definición de Gamma et. al.

### Patrón observer

#### Roles:

- Subject: Sujeto
- Observer: Observador
- Concrete Subject: ZonaCobertura
- Concrete Observer: Organización

#### Roles:

- Subject: ManejadorMuestraVerificada (conoce a los observadores)
- Concrete Subject: Muestra (Almacena el estado de interes)
- Concrete Observer: ZonaCobertura

### Patrón State

#### Roles:

Context: Muestra

State: Estado

Concrete states: EstadoBasico, EstadoExperto, EstadoVerificado

#### Roles:

Context: Usuario

State: CategoriaUsuario

Concrete States: CategoriaBasico, CategoriaExperto

### Patrón Composite

#### Roles:

Component: TipoFiltro

Composite: FiltroCompuesto ( OR y AND )

Leafs: FiltroFechaMuestra, UltimaVotacion, TipoInsecto, NivelDeVerificacion

## Decisiones de diseño y detalles de implementación.

### Aplicación web

La representación de la aplicación está definida en tener cuatro manejadores los cuales sirven para delegar la tarea de crear y registrar muestras, zonas, opiniones y usuarios. En él vamos a tener el historial de todas esas cosas, además de avisarle al manejador de zona la creación de nuevas muestras y poder utilizar los filtros.

### Muestras

La representación de las muestras está definida por una especie de vinchuca, una foto, una fecha, una ubicación, lista de opiniones, un estado, autor y un manejador.

El comportamiento esperado es que sea capaz de entender los mensajes de, cargar opiniones, dar un resultado final, suscribirse, desuscribirse, fecha de la última opinión recibida y los getters de sus colaboradores internos.

En cuanto a los métodos de cargar opinión y resultado final, decidimos delegarlo en el estado y que él nos responda utilizando un double dispatch, ya que utilizamos el patrón state porque los estados conocen al siguiente y hay un cambio que no depende de una decisión externa sino que depende de una lógica interna.

#### Para cargar la opinión:

En Estado Básico: la opinión de cualquiera es cargada en la muestra y si la opinión es de experto pasa a cambiar su estado a Estado experto.

En Estado Experto: si ya hay una opinion de algun experto del mismo tipo\*, entonces la muestra pasa a cambiar a estado verificado y notifica a sus suscriptores\*

En Estado verificado: ya no se admiten opiniones.

#### Para saber el resultado final:

En Estado Básico: se devuelve la primera opinión que sea del tipo más repetido. Se crea un map vacío el cual mientras se va recorriendo la lista de opiniones y se va creando como clave el tipo de la opinión y el valor en uno, si ya el tipo existe en el map, se le aumenta en uno el valor de ese tipo.

En Estado Experto: se devuelve la primera opinión de un experto, ya que si está en este estado es porque no hubo dos expertos que coincidan.

En Estado Verificado: se devuelve la primera opinión de experto que sea del tipo que esté igual o más de dos veces repetida. Se crea un map el cual la clave es el tipo y el valor las veces repetidas, pero solamente de opiniones de expertos. Después adquirimos el tipo (la clave) que tiene como valor dos o más y buscamos en la lista de opiniones cual coincide con ese tipo y que sea de experto para retornarla.

\*

#### Suscribirse y desuscribirse:

Los métodos delegan a `ManejadorMuestraVerificada` para no tener tantas responsabilidades.

### Verificación de las muestras

Creamos una clase llamada `ManejadorMuestraVerificada` la cual su responsabilidad es tener un registro de todas las zonas que estén interesadas en ser avisadas cuando la muestra pase a estar verificada. Se encarga de suscribir, desuscribir y notificar.

### Opiniones

La representación de las opiniones está definida un tipo\* el cual decidimos hacerlo enum ya que es una cantidad limitada de opciones, una fecha de creación, un autor y una categoría, la cual se le asigna al momento de instanciar la misma que la del autor con la diferencia que esta no cambia en ningún momento.

El comportamiento esperado de una opinión es que entienda los mensajes de getters de sus atributos y sepa si es opinión de experto, el cual delega en su estado.

### Usuarios

La representación de los usuarios está definida por una lista de opiniones emitidas, muestras emitidas y una categoría, la cual es asignada como `categoriaBasica` al momento de instanciar un usuario. Usuario tiene una clase hija la cual es `UsuarioProfesional`, que la diferencia con la clase `Usuario` va a ser que su categoría es `categoriaExperta` y no va a cambiar en ningún momento.

El comportamiento esperado de un usuario es que entienda los mensajes de getters de sus atributos, opinar sobre una muestra y enviar muestra.

Cada vez que se opine sobre una muestra y/o se envíe una muestra, se va a verificar su categoría utilizando el patrón state, ya que el cambio de esta no depende de una decisión externa, sino que depende de la lógica implementada por la clase.

#### Opinar sobre:

Chequea si ya opino de esa muestra o si esa muestra es suya, en ese caso no podrá opinar. Si puede opinar se le agrega a su lista, se le avisa a muestra y se verifica la categoría.

#### Enviar muestra:

Lo agrega a su lista de muestras y verifica la categoría.

El usuario profesional sobreescribe el método de verificación y no hace nada.

## Ubicaciones

Para representar las ubicaciones se utiliza una latitud y una longitud (doubles).

-En cuanto al comportamiento que se espera de una ubicación, para poder resolver la distancia con respecto a otra se investigó en la web y se optó por utilizar la siguiente fórmula para calcularla:

The distance between two points  $(x, y)$  and  $(x_1, y_1)$  can be calculated by the following formula:

$$\text{distance} = \sqrt{(x_1 - x)^2 + (y_1 - y)^2}$$

De esta forma, en base a la latitud y longitud de las ubicaciones se puede calcular su distancia.

-Por otro lado, para conocer en base a una lista de ubicaciones aquellas que se encuentran a menos de cierta distancia, se le pasan las ubicaciones junto con la distancia máxima representada como un double, y se filtran aquellas cuya distancia con respecto a la ubicación actual es menor o igual a la distancia máxima.

-Por último, para conocer todas las muestras obtenidas a cierta distancia en base a una muestra, se le pasa una lista de muestras y una distancia máxima, para lo cual se compara la distancia entre la ubicación de la muestra que se busca comparar con el resto de muestras, filtrando aquellas cuya distancia es menor a la distancia máxima.

## Las organizaciones

Para la representación de las organizaciones, estas constan de una ubicación, su tipo, la cantidad de empleados que poseen y dos funcionalidades externas (interfaz).

-En cuanto a las notificaciones, las organizaciones son suscriptas a diversas zonas, y cuando se sube o valida una muestra reaccionan de determinada manera. Para esto tiene dos métodos que implementa de la interfaz Observador; cuando se sube una nueva muestra en una zona de interés, la organización es notificada y delega en su funcionalidad externa dedicada a actuar cuando se carga una nueva muestra. Por otro lado, cuando se valida una muestra tiene otro método para ser notificada, en el cuál delega a su funcionalidad externa dedicada a actuar cuando se valida una muestra.

-Las organizaciones tienen además otros dos métodos para cambiar cada una de las funcionalidades externas en caso de que lo necesiten.

## -¿Cómo se implementaría y usaría una funcionalidad externa en nuestro modelo?

Para implementar diferentes funcionalidades externas, se requeriría de clases que implementen la siguiente interfaz:

```
public interface FuncionalidadExterna{  
    public void nuevoEvento(<Organización>, <zonaDeCobertura>, <Muestra>);  
}
```

Entonces al momento de instanciar una organización, se le pasa dos de estas clases (o una única, según lo que sea necesario en el momento), y esta las guarda como variables de instancia.

En el momento donde una organización recibe el mensaje “serNotificadoNuevaMuestra(ZonaCobertura, Muestra)”, esta delega en su funcionalidad externa dedicada a reaccionar a las muestras nuevas con el mensaje “nuevoEvento(<Organizacion>, <ZonaDeCobertura>, <Muestra>)”, donde se pasa por parámetro a sí misma, la zona de cobertura y la muestra.

```
@Override  
public void serNotificadoNuevaMuestra(ZonaCobertura zona, Muestra muestra) {  
    this.funcionalidadNuevaMuestra.nuevoEvento(this, zona, muestra);  
}
```

Por otro lado, al recibir el mensaje de “serNotificadoMuestraVerificada(ZonaCobertura, Muestra)”, pasa lo mismo, pero en este caso delega en su funcionalidad externa dedicada a reaccionar ante el evento de una validación de muestra.

```
@Override
public void serNotificadoNuevaMuestra(ZonaCobertura zona, Muestra muestra) {
    this.funcionalidadNuevaMuestra.nuevoEvento(this, zona, muestra);
}
```

## Zonas de cobertura

Para representar las zonas de cobertura, estas constan de un nombre, una ubicación que representa su epicentro, un radio y una lista de muestras.

-Para que se conozcan en todo momento las muestras que se registran en una zona de cobertura, tenemos una clase llamada AplicacionWeb, encargada de la delegación de la creación de muestras (entre otras cosas) en otra clase llamada ManejadorMuestras.

-Cuando se crea una muestra, la clase AplicacionWeb delega en su variable de instancia de la clase llamada ManejadorZonasCobertura la notificación de la nueva muestra a todas las zonas de cobertura que cubren la ubicación de la misma. Al recibir el aviso, cada zona a la que le corresponda se encarga de agregarla a su lista de muestras.

-Para filtrar aquellas zonas a las cuales le corresponde incluir a la nueva muestra de las que no, el manejador le pregunta a cada zona si contiene a la misma. Para que una zona contenga la ubicación de la muestra, compara si la distancia entre esta y el epicentro es menor que el radio.

```
public boolean contiene(Muestra muestra) {
    return this.epicentro.distanciaEntre(muestra.getUbicacion()) <= this.radio;
```

-Para saber si una zona la solapa, las zonas de cobertura tienen un método el cual compara si la distancia entre los epicentros es menor que la suma de los radios. Si no es así, entonces no solapan.

```
private double distanciaA(ZonaCobertura otraZona) {
    return this.epicentro.distanciaEntre(otraZona.getEpicentro());
}
public boolean seSolapaCon(ZonaCobertura otraZona) {
    //Si la distancia entre los epicentros es menor o igual que la suma de sus radios,
    //entonces las zonas de cobertura se solapan.
    return distanciaA(otraZona) <= (getRadio() + otraZona.getRadio());
}
```



Después, tienen otro método para saber todas las zonas con las que solapan, el cuál utiliza un `ManejadorDeZonasDeCobertura`, pasándose a sí misma como parámetro.

```
public List<ZonaCobertura> zonasQueSolapan(ManejadorZonasCobertura manejador){  
    return manejador.zonasQueSolapan(this);  
}
```

`ManejadorDeZonasDeCobertura` toma a la zona de cobertura que recibió como parámetro, y filtra aquellas zonas en las cuales se cumple que solapa, devolviendo las mismas.

```
public List<ZonaCobertura> zonasQueSolapan(ZonaCobertura zona){  
    return this.zonasDeCobertura.stream().filter(z -> z.seSolapaCon(zona)).toList();  
}
```

## Filtros

Para permitir a los usuarios ver solo las muestras que les interesan, el sistema cuenta con filtros. Estos filtros pueden ser simples (como filtrar por tipo de insecto o por fecha) o más complejos, combinando varios filtros a la vez con lógica *AND* o *OR*.

Para eso tenemos una clase llamada `Filtro`, que se encarga de aplicar un criterio de filtrado sobre una lista de muestras. Esta clase delega la lógica de qué se considera una muestra válida al filtro que tenga configurado, que es una implementación de la interfaz `TipoFiltro`.

```
public List<Muestra> aplicarFiltro() {  
    return muestras.stream()  
        .filter(m -> filtroAUsar.cumple(m))  
        .collect(Collectors.toList());  
}
```

La interfaz `TipoFiltro` define un único comportamiento: el método `cumple(Muestra)`, que responde si una muestra cumple o no con cierta condición. Cada clase que implementa esta interfaz define su propio criterio.

```
public interface TipoFiltro {  
    public boolean cumple(Muestra muestra);  
}
```

**FiltroFechaMuestra** evalúa si la fecha de la muestra cumple cierta condición.

**ÚltimaVotacion** analiza cuándo fue la última votación de una muestra.

**TipoInsecto** compara si el insecto que se ve en la muestra es del tipo buscado.

**NivelDeVerificacion** mira el estado de verificación que tiene la muestra.

## Filtro Compuesto

Se pueden combinar varios filtros usando lógica booleana. Para eso existe una clase abstracta llamada **FiltroCompuesto**, la cual, en el patrón de diseño, es el "Composite", el cual permite ingresar nuevos filtros para que estos se usen y se puedan anidar :

- **OR** es una clase que hereda de **FiltroCompuesto**, y considera que una muestra pasa el filtro si cumple **al menos uno** de los filtros que tiene adentro.
- **AND**, por el contrario, considera que una muestra cumple el filtro solo si pasa **todos** los filtros que contiene.

De esta forma, se puede generar filtros muy sencillos ( usando solamente una hoja del patrón ), o hacer un filtro mucho más extenso y complejo, usando el "FiltroCompuesto", esto también permite, que en un futuro, cuando se requiera agregar algún nuevo filtro, no modificar nada de lo antes hecho, y que siga cumpliendo correctamente.