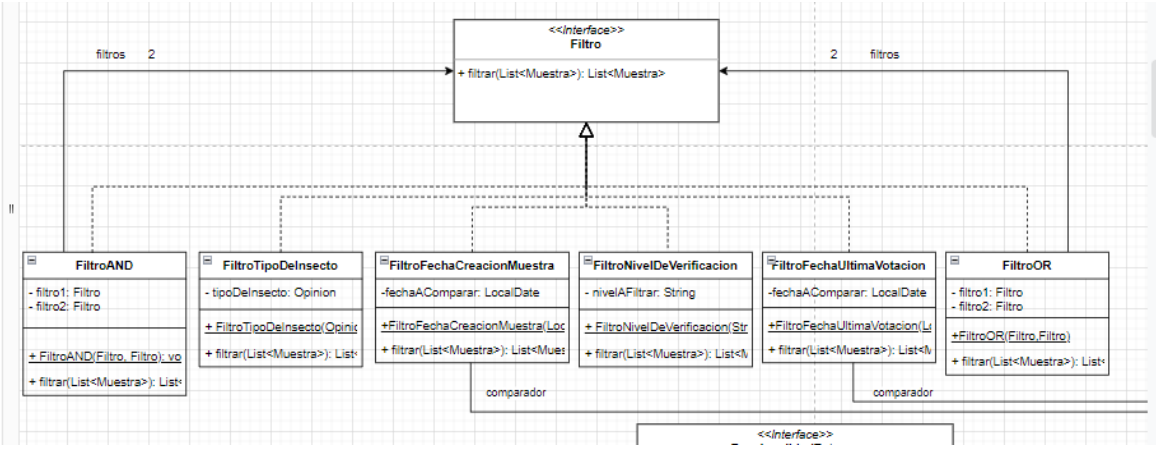
Proyecto Grupal: POO2
A la caza de vinchucas

Integrantes:
Agustin Lascar - aguslascar@hotmail.com.ar
Franco Martín Marengo - tenamarengo.11@gmail.com
Roberto Leonardo Medici - leomedici06@gmail.com

Patrones de diseño

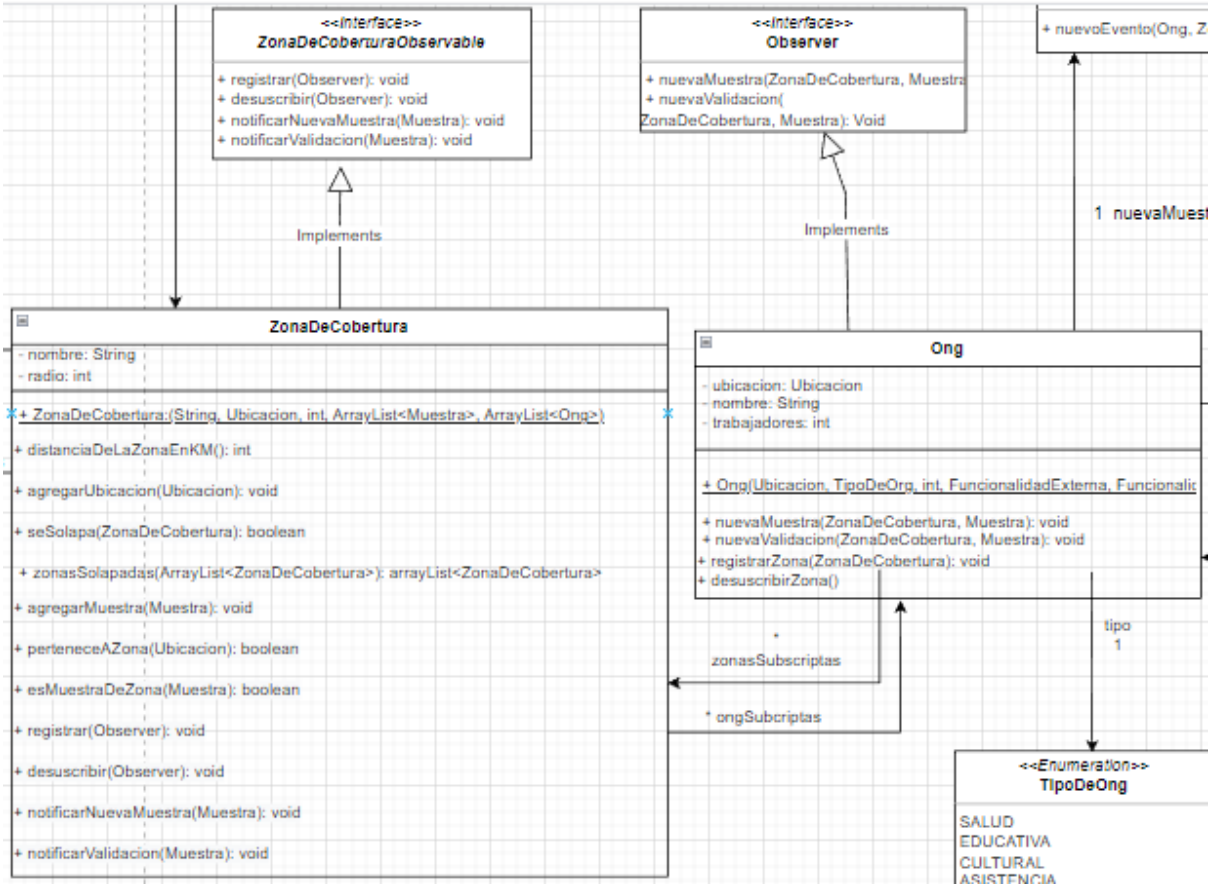


Utilizamos el patrón Composite para implementar los filtros de búsqueda. Esto se decidió luego de una charla entre los integrantes del grupo y el profesor Matías Butti, que nos recomendó utilizar un diseño de patrón Composite que nos iba a facilitar el diseño de los filtros AND y OR.

En este caso, tenemos un **Component** que es la interfaz genérica Filtro, la cual tiene como **leafs** los filtros de tipo de insecto, nivel de verificación y por fecha de creación y última votación. Y los filtros AND y OR, son los **composite,** los cuales están compuestos por otros dos Filtro.

Luego de recibir la corrección por parte de Axel Aguero, marcando el faltante del Client en este patrón Composite, se consultó con el profesor Diego Cano cuál sería el Client. Se llegó a la conclusión que estos filtros al ser "ensamblados" a la hora de realizar un programa, no tendrían un Client específico, sino es algo propio de utilización en un programa o test. Ya que ninguna clase tiene como variable de instancia ni guarda una lista de filtros predefinidos. En caso de que los filtros sean predefinidos(que no es este el caso), por ejemplo, el Client si sería una AplicacionWeb la cual tendría una lista de Filtro.

De esta manera, se puede tener un mensaje polimórfico "filtrar(List<Muestra>)" el cual van a saber responder ya sea una leaf o también un composite.



Otra patrón de diseño que utilizamos en nuestro trabajo fue el patrón Observer, ya que a las Ong les interesa saber cada vez que se valida o agrega una muestra en una cierta zona de cobertura, se creó a ZonaDeCobertura con una lista de Ongs suscriptas, que son las interesadas sobre las muestras de esa zona en específico, ZonaDeCoberturaObservable es una interface que es implementada por ZonaDeCobertura con los mensajes para registrar, desuscribir, notificar por una muestra nueva, y notificar por una validación de una muestra.

La clase ZonaDeCobertura actua como el ConcreteSubject al cual quieren observar las muestras que se ingresen, y las muestras que se validen.

La interfaz ZonaDeCoberturaObservable actua como el ObservableObject que es implementado por el Subject.

La clase Observer es la que va a manejar los mensajes de actualizar para que sean implementados por el ConcreteObserver; ese ConcreteObserver es la organización que depende si se agrega una muestra o si se valida, llamará a su propia funcionalidad externa correspondiente.

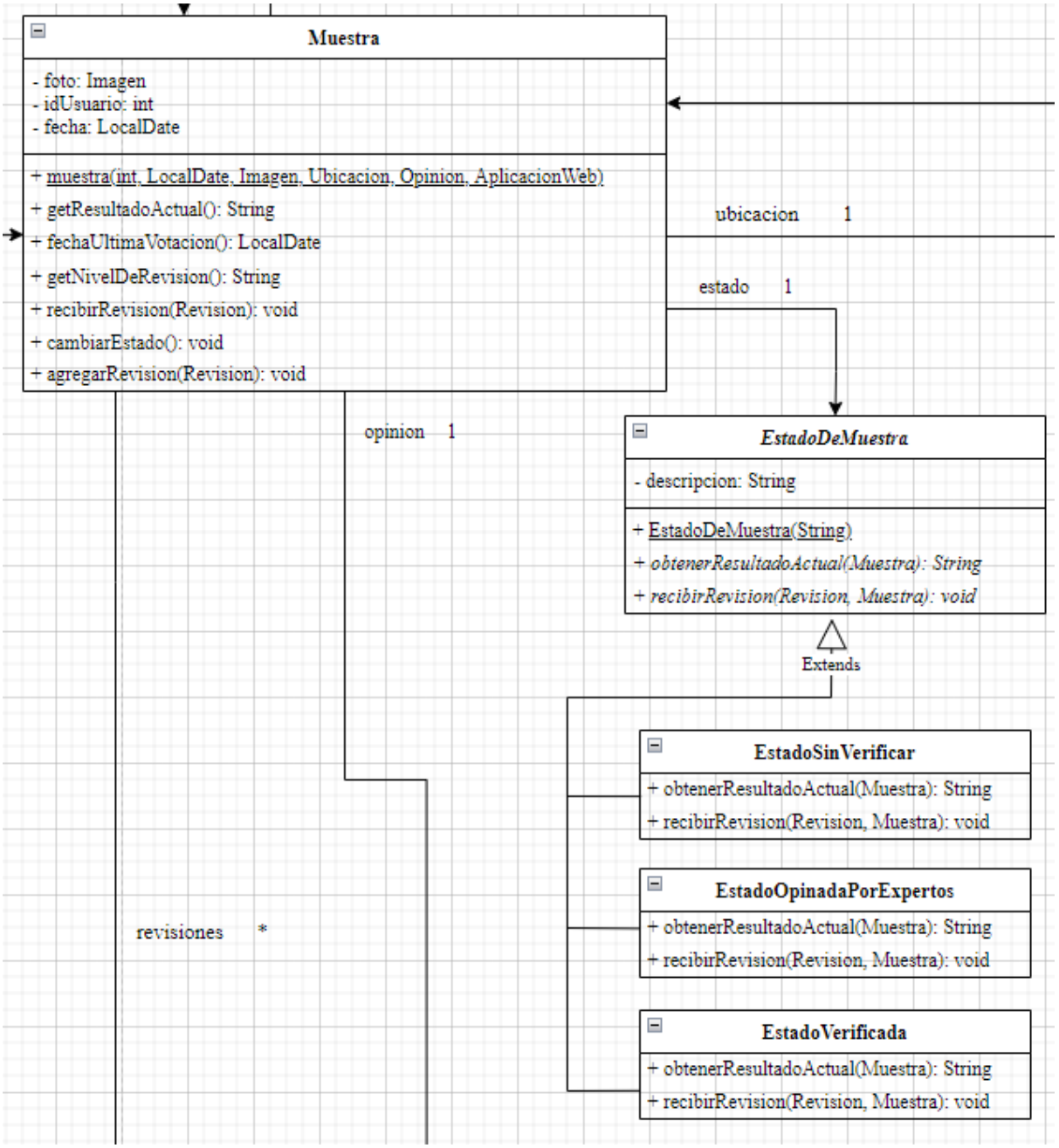
Por último utilizamos el patrón de diseño **State**, para que una muestra pueda cambiar entre distintos estados (y variar su comportamiento) a medida de que los usuarios opinaban sobre ésta.

Se tomó esta decisión, ya que muestra variaba el comportamiento según el estado en el que se encontraba, al no utilizar este patrón, se tenía que utilizar una gran rama de condicionales if-else.

Finalmente, esto nos ayudó a que esta parte del sistema sea extensible al momento de agregar nuevos estados de muestra de manera sencilla y cerrando el código implementado previamente, pensando que a futuro pueden haber nuevos tipos de usuario que pueden opinar sobre muestras y cambiar su estado a uno nuevo.

Mediante los estados se delegan todas las operaciones importantes de opinión como validar que un usuario pueda opinar sobre una muestra dependiendo su estado actual, o tomar la decisión de cuándo cambiar el estado a uno nuevo.

En el sistema, nuestro ‘Context’ sería la muestra, la clase abstracta que representa a ‘State’ sería 'EstadoDeMuestra' y los distintos 'ConcreteState' serían ‘EstadoSinVerificar’, 'EstadoVerificada', etc.


Muestra
- foto: Imagen
- idUsuario: int
- fecha: LocalDate
+ muestra(int, LocalDate, Imagen, Ubicacion, Opinion, AplicacionWeb)
+ getResultadoActual(): String
+ fechaUltimaVotacion(): LocalDate
+ getNivelDeRevision(): String
+ recibirRevision(Revision): void
+ cambiarEstado(): void
+ agregarRevision(Revision): void
ubicacion 1
estado 1
opinion 1
revisiones *
EstadoDeMuestra
- descripcion: String
+ EstadoDeMuestra(String)
+ obtenerResultadoActual(Muestra): String
+ recibirRevision(Revision, Muestra): void
Extends
EstadoSinVerificar
+ obtenerResultadoActual(Muestra): String
+ recibirRevision(Revision, Muestra): void
EstadoOpinadaPorExpertos
+ obtenerResultadoActual(Muestra): String
+ recibirRevision(Revision, Muestra): void
EstadoVerificada
+ obtenerResultadoActual(Muestra): String
+ recibirRevision(Revision, Muestra): void

Decisiones de diseño

Se decidió tener una clase AplicacionWeb, la cual se encarga de manejar los usuarios, las muestras, las opiniones de las muestras, el filtrado de resultados según los filtros que se quieran utilizar, las organizaciones no gubernamentales y las zonas de cobertura. Esto para que el modelo se asimile a la realidad de como funcionaria una aplicación, donde todo debe pasar por ella.

Esto evita problemas como que un usuario que no esté registrado en la aplicación, quiera hacer una revisión de una muestra, así como también quiera hacerla sobre una muestra que no existe en el sistema, entre otros casos.

Se decidió tener una clase Usuario que tiene un tipo de usuario(básico o experto) el cual puede cambiar según un algoritmo que evalúe los usuarios según sus conocimientos y participación.

Tomamos esta decisión ya que si Usuario fuera una clase abstracta y sus subclases fueran básico y experto, el algoritmo tendría que crear un nuevo usuario con la información del "viejo" usuario para cambiarlo de categoría.

La clase abstracta NivelDeUsuario tiene dos subclases Básico y Experto para representar los dos niveles de usuarios. Está abierto a que haya nuevos niveles.

Esta decisión se tomó para que se sepa responder el mensaje "esExperto()". Esto para que la muestra sepa qué nivel tiene el usuario que hace la revisión. Esto evita comparar tipos. Ejemplo: "Experto" == "Experto.

Los usuarios si bien se registran en la aplicación, decidimos que se instancien fuera de esta para poder “mockearlos” a la hora de hacer los test de la aplicación.

Los filtros fueron implementados con un patrón Composite que nos permite una sencilla implementación de los filtros compuestos, ya que contienen dos objetos de tipo Filtro, los cuales pueden ser una leaf o un composite, y dentro de ese composite tener otros dos composite y así sucesivamente hasta llegar a dos leaf.

Se decidió que haya una clase “ComparadorDeFechas”. Esta se compone de una interfaz que luego es implementada por dos clases ComparadorMenor y ComparadorMayor, que tienen un método comparar(fecha1, fecha2) que depende el caso devuelve un booleano según sea mayor o menor fecha1 de fecha2.

Los filtros según fechas, tienen una variable de instancia que es ComparadorDeFechas, que se declara si es por menor o por mayor al momento de instanciar el filtro.

En la clase Ubicación se decidió usar int en vez de doubles para simplificar los números; y en la clase ZonaDeCobertura se decidió pasar directamente las Ubicaciones a su lista como nos dijo el profesor Diego Cano para que no tengamos que calcular todos los puntos dentro de la zona.

Para representar la muestra se eligió una clase concreta que guarda información importante como opiniones de usuarios sobre ésta en una lista y también la opinión del autor, además de la fecha en que se tomó y otros datos requeridos. Se eligió que al momento de crear una muestra se tome el nivel del usuario del autor, permitiendo inicializar una muestra teniendo en cuenta la opinión del usuario autor y su nivel, no dejando opinar a usuarios básicos desde un primer momento, aunque lo más importante fue la manera en la que las muestras iban a cambiar su estado automáticamente y como iban a validar opiniones de distintos usuarios, ésta finalmente guarda un estado que va variando según opiniones de usuarios y otras condiciones.

La clase muestra tiene como variable de instancia a un objeto de la interfaz Imagen la cual se decidió realizar como interfaz para representar la foto tomada de la muestra. Esta interfaz no tiene métodos ya que solamente se utiliza para representación. Se decidió hacer esto para que la foto no sea simplemente un string o cualquier otro tipo de objeto sin sentido.

Se creó una clase abstracta EstadoDeMuestra, que permite extenderse hacia nuevos tipos de estados de muestra, modelando dentro de estos estados los algoritmos más importantes de validación y cambio de estado.

Se guardan revisiones de otros usuarios dentro de estas muestras, estas revisiones contienen datos importantes sobre una votación como el id del usuario que la realizó, además de una Opinión, representada como una clase enumerativa. Se hizo de esta manera para no realizar una jerarquía innecesaria de distintos tipos de votaciones, que apenas iban a tener comportamiento y estado propio, esto permite extender igualmente el programa con distintos tipos de votaciones sin ningún tipo de problema encontrado para el sistema hasta la fecha.

La Revisión contiene solo el id del usuario, ya que si contuviera el Usuario y quisieramos saber el tipo de usuario que hizo la revisión, si pasaron más de 30 días de esa Revisión quizá el usuario haya bajado o subido de categoría por lo que el resultado sería una categoría que no corresponde a la que tenía el usuario al hacer la revisión.

Problemas encontrados a la hora de diseñar

Uno de los problemas que apareció al momento del diseño, fue cómo implementar que los filtros de búsqueda por fecha, puedan ser antes o después de esa fecha. Luego de consultarlo con el profesor Diego Cano, nos recomendó utilizar una clase que nosotros denominamos “ComparadorDeFechas” explicada anteriormente.

Otro de los problemas fue a la hora de elegir cómo implementar las distintas opiniones de los usuarios, ya que no se encontró una decisión totalmente acertada entre utilizar enumerativos o una jerarquía de clases.

Por último, tras las correcciones, se intentó cambiar la manera en qué el estado de una muestra cambiaba, para no tener ramas if-else por cada condición de cambio de estado y no perjudicar la extensibilidad de nuevos niveles de usuario y estados de muestra. No se logró

implementar, fue recomendado un double dispatch (o patrón visitor) pero optamos por hacer este cambio, ya que para implementar un visitor nos debíamos asegurar de que alguna de las dos estructuras (los estados de muestra o niveles de usuario) no se fueran a extender en un futuro. Esto no nos permitió usar el patrón y no se halló una solución clara.