

Universidad Nacional de Quilmes  
Programación con Objetos II  
1er cuatrimestre 2020

# Trabajo Práctico Integrador: A la caza de las vinchucas

Natalia Caporale  
Cintia García  
Fernando Gauna

En el siguiente informe se presentan las decisiones tomadas con respecto al diseño del enunciado del trabajo grupal sobre la creación una aplicación web que maneje los datos de muestras de vinchucas que son tomadas por personas a todo lo largo del país. Para realizarlo se tomaron en cuenta conceptos de programación con objetos y patrones vistos en clase y tomados de la bibliografía proporcionada por la cátedra, más la búsqueda de otros materiales de referencia en línea para resolver problemas específicos del lenguaje Java. En verde, se encuentran las modificaciones y agregados realizadas al programa en base a las correcciones realizadas a la primera versión.

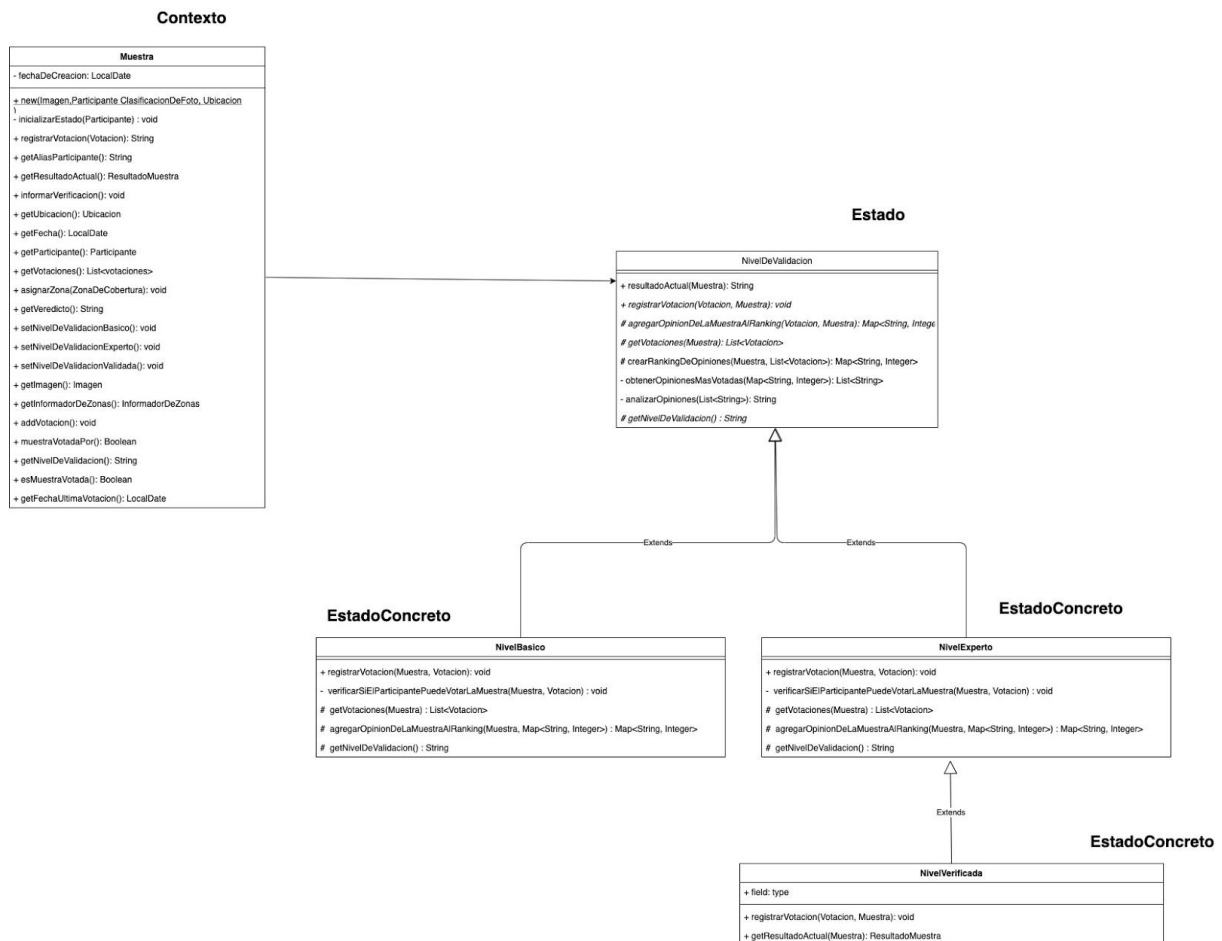
Las publicaciones de los participantes son representadas por la clase Muestra, que contiene una imagen (para la cual se decidió también generar una clase aparte), el/la participante que le dio origen, la fecha de creación. Almacena además el estado del participante al momento de crear la muestra en la variable nivelDeConocimientoDeCreacion y por último contiene la opinión del/de la participante sobre la imagen que subió.

A medida que los participantes voten, irá almacenando esas votaciones en una lista. Las votaciones se utilizan para brindar el estado de la muestra cuando este sea requerido, también dependiendo del estado de la persona que la vote, la muestra puede tener distintos estados, que van de básico a experto y de experto a validada, dependiendo del nivel de conocimiento del/de la participante votante.

Los estados de la muestra: Se utiliza el patrón state, ya que el comportamiento de la muestra puede variar dependiendo del estado interno que ella tenga en ese momento. sus distintos estados son: NivelBasico, NivelExperto y NivelValidada.

Se decide que la clase NivelExperto será padre de la clase NivelValidada ya que comparten el mismo comportamiento, solo que la muestra Validada ya no puede ser votada una vez que tiene este estado.

También informa si la muestra a recibido alguna votación o no y la fecha de la última votación; estos métodos serán utilizados internamente por la clase Filtro.



Se decidió que la lista de muestras creadas por lxs participantes esté en una clase aparte denominada AplicacionVinchuca. Esta clase es la que, como se explicará más adelante, esta clase es la que tendrá la posibilidad de buscar muestras que cumplan determinado criterio con un Buscador y múltiples filtros.

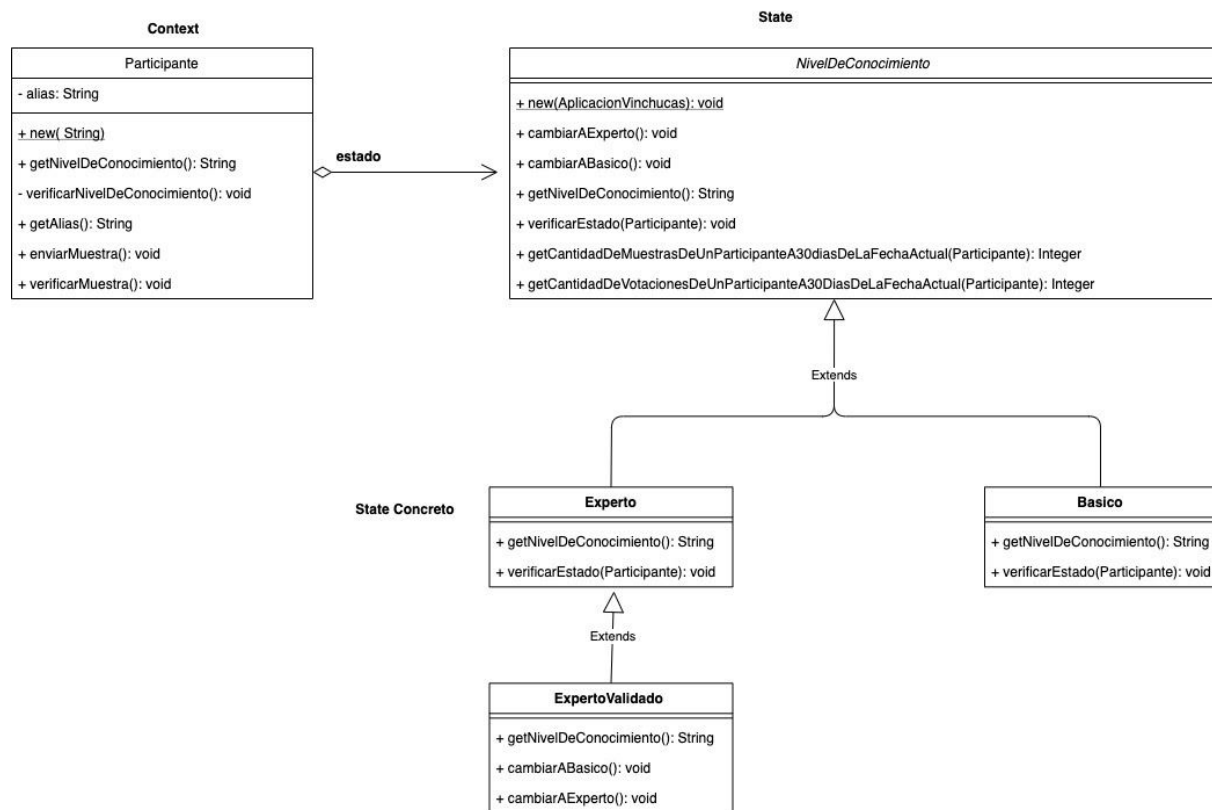
Para la validación de una muestra es necesario conocer el nivel de conocimiento del/de la participante que las creó o de lxs participantes que la fueron votando. Para ello, se crea la clase Participante, quien se encarga de informar su nivel de conocimiento.

Se utiliza el patrón State para manejar el nivel de conocimiento de la clase Participante, ya que cambiará su comportamiento en base a su estado. Desde la AplicacionVinchuca se podrá llamar al método actualizarNivelDeConocimiento para que colecte a todxs lxs participantes que han interactuado con la aplicación y evalúe la necesidad de cambiar su estado de acuerdo a los parámetros establecidos en el enunciado. De su estado interno dependerá la validación de las muestras a la hora de generarlas, o al momento de votar una muestra, causarán distinto efecto en ellas el nivel de conocimiento que el/la participante tenga a la hora de realizarlas.

El proceso de registrar una votación se realiza en el nivel de validación actual de la muestra. Allí se verifica si el/la participante cumple con las condiciones para poder votar, y para el caso contrario se creó la excepción ErrorParticipanteNoPuedeVotarEstaMuestra, ahora

heredando de RuntimeException, ya que se decidió utilizar excepciones no manejadas. El cambio del nivel de validación en este proceso queda a cargo del nivel de conocimiento de la persona que efectúa la votación. De esta forma se reemplaza la consulta mediante string del nivel de conocimiento del participante (además del agregado del método esExperto).

Estructura:



En el caso de los niveles de conocimiento Experto y Experto Validado, como sus protocolos son iguales (la única diferencia entre ellos es que el/la participante que tiene nivel experto validado, no cambia nunca a básico, pase lo que pase, este nivel siempre se mantiene) es por eso que se decide tener como clase padre nivel experto y como clase hija nivel Experto validado.

Nivel de conocimiento conoce a la clase AplicacionVinchuca, a través de ella se comunica con el buscador y este le trae toda la lista de votaciones que tiene el participante desde hace 30 días hasta la fecha de hoy. El nivel de conocimiento se encarga de contar dichas votaciones. También se encarga de contar las Muestras ingresadas por el/la participante, mediante la utilización del filtroAnd, con el filtroParticioante y el filtroFecha, el buscador también le brinda la lista de muestras de esx participante desde hace 30 días a la fecha de hoy.

Con ambos resultados (cantidad de votaciones y cantidad de muestras) el nivel de conocimiento selecciona si el/la participante al día de la fecha es experto o básico. Ya mencionamos que si el/la participante se genera con un nivel experto validado este nunca cambia de estado, y se comporta igual que un experto. En caso de que por error

un/una participante se cree como nivel experto, a la hora de aplicar el protocolo `getNivelDeConocimiento()`, el sistema solo se encargara de setear su nivel a basico.

Las zonas de cobertura, representadas con una clase (`ZonaDeCobertura`), necesitan también saber qué muestras se encuentran en su área; decidimos que las zonas no deberían conocer a la aplicación, sino visceversa. Por lo tanto, `AplicacionVinchuca` también cuenta con la lista de zonas de cobertura.

Las ubicaciones fueron representadas en la clase `Ubicacion`, a la cual se le asignó la responsabilidad de calcular la distancia entre dos ubicaciones, las ubicaciones a menos de una distancia determinada, y las muestras que se encuentra a una determinada distancia a partir de una muestra. Todas las distancias fueron calculadas en kilómetros.

Consta de dos variables, latitud y longitud, con dichos valores expresados en decimales. Las mismas tienen valores límites que en caso de incumplirse lanzarán una excepción creada para tal propósito, `ErrorLaUbicacionNoExiste`. Al tener una clase específica para representarlas ubicaciones, esta excepción se lanzará en todos los casos en los que se utilice este objeto (en `Organizacion`, `Muestra` o `ZonaDeCobertura`, por ejemplo).

Cada instancia de `ZonaDeCobertura` se crea con un radio (expresado en kilómetros), un epicentro (expresado como `Ubicacion`) y su nombre. Para este constructor, se generó una excepción que será lanzada en caso de que el radio sea menor o igual a cero (`ErrorElRadioDebeSerMayorACero`). **Un método para actualizar la lista de muestras fue agregado en ambos setters (ubicacion y epicentro); de esta forma, si alguna de las dos variables cambia, la zona actualiza la lista de muestras que pertenecen a su área. De esta forma, al crearse una zona también se agregan a la lista las muestras que hayan sido creadas antes de que la zona fuera definida**

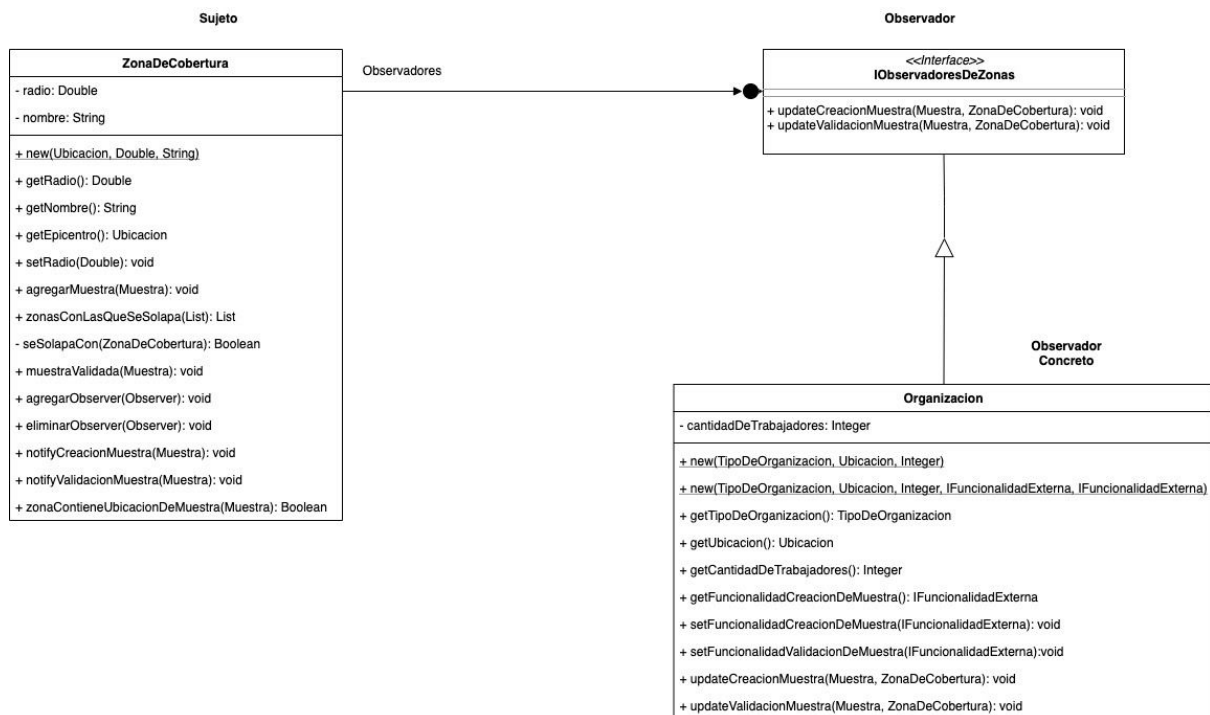
Como se decidió que `ZonaDeCobertura` no conociera a la aplicación, sino visceversa, y `ZonaDeCobertura` debe saber cuáles son las muestras que se encuentran ubicadas en su área, cuando se crea una muestra y `AplicacionVinchuca` la agrega a su lista, `AplicacionVinchuca` revisa su lista de zonas, se fija cuáles contienen la ubicación en la cual se encuentra la muestra, y la agrega a la lista de muestras de cada zona, dando de esta forma una respuesta rápida a este pedido.

Para las ONGs, para las cuales fue designada la clase `Organizacion`, se definieron dos constructores, para que sea posible crear una instancia de la misma sin o con sus funcionalidades externas definidas.

Para avisarles a las organizaciones sobre la creación o validación de muestras, fue utilizado el patrón observer. Se implementó en un principio la clase `Observable` y la interfaz `Observer` que forman parte de Java; sin embargo, el método `update` de dicha interface no resultó útil para el caso al tener sólo dos parámetros, ya que la organización necesita de tres elementos para poder actualizarse desde un sólo método: la muestra, la zona de cobertura donde se encuentra la muestra, y además algún tipo de aviso que indique si el evento ocurrido es una creación o una validación.

Por lo tanto, se resolvió utilizar una interfaz específica para el universo planteado, a la que se denominó `IObservadoresDeZonas`; así, cuando otro tipo de organización quiera observar a los eventos que ocurren en las zonas, sólo deben aplicar dicha interfaz y subscribirse a la lista de observers de las zonas (**agregado del método `interesEnZona` en la interfaz `IObservadoresDeZonas` para esta finalidad**). De esta manera, la estructura el patrón quedó conformada por los siguientes elementos:

- sujeto concreto: clase `ZonaDeCobertura`
- observador: interface `IObservadoresDZonas`
- observador concreto: `Organizacion`



Una zona de cobertura se entera de que una muestra ha sido creada cuando la aplicación la agrega a su lista. `AplicacionVinchuca`, sin embargo, no tiene forma de saber que una muestra ha sido validada para informarle a las zonas sobre dicho evento. Se pensó en un principio en que las muestras conocieran a sus zonas de cobertura; sin embargo para evitar una doble asociación, se decidió generar una clase cuya función sería conocer las zonas a las que pertenece una muestra, e informarles a las zonas cuando dicha muestra se valide. El nombre dado a dicho objeto fue `InformadorDeZonas`.

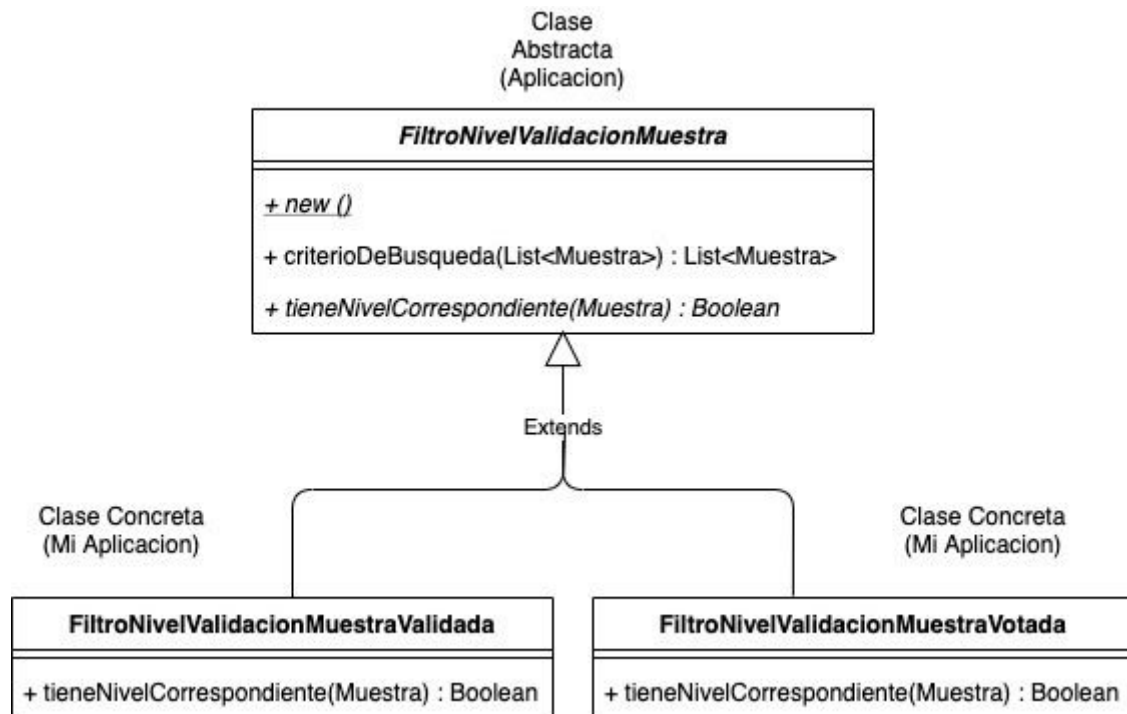
Por lo tanto, la creación de una muestra es informada a `ZonaDeCobertura` por la aplicación, y la validación de una muestra por `InformadorDeZonas`. Para cada uno de estos eventos, `ZonaDeCobertura` posee un método `notify`. A su vez, la interfaz `IObservadoresDZonas` posee dos métodos `update`, uno para cada evento. Por tanto, los `notify` de la zona que se activan con cada evento llaman a los observers, les pasan el mensaje `update` correspondiente, y cada observer (`Organizacion` en este caso), resuelve llamando a la funcionalidad externa que corresponda.

Como se mencionó previamente, las búsquedas pueden realizarse desde AplicacionVinchuca. Para cada criterio de búsqueda solicitado, se diseñó un filtro que busca entre la lista de muestras aquellas que cumplen con la condición mencionada. AplicacionVinchuca posee un Buscador, que es quien aplica los filtros sobre la lista de muestras.

Para la estructura general de los filtros, se consideraron dos opciones: realizar una superclase Filtro con los filtros específicos como subclases, o realizar una interfaz que las diferentes clases de filtro implementen. Debido a la diversidad de parámetros que necesitan los filtros para funcionar (fechas, niveles de validación o tipo de insecto), se decidió diseñar la última opción.

De esta forma, cada filtro incluye en los parámetros de su constructores los objetos que necesita para realizar su búsqueda específica, y para uniformar, se generó una interfaz Filtro que implementa el mensaje de búsqueda que todas las clases de filtros deben responder (criterioDeBusqueda(List<Muestra>)). Así, se recurre al polimorfismo para que cada filtro implemente dicho mensaje de acuerdo a su propio criterio de búsqueda.

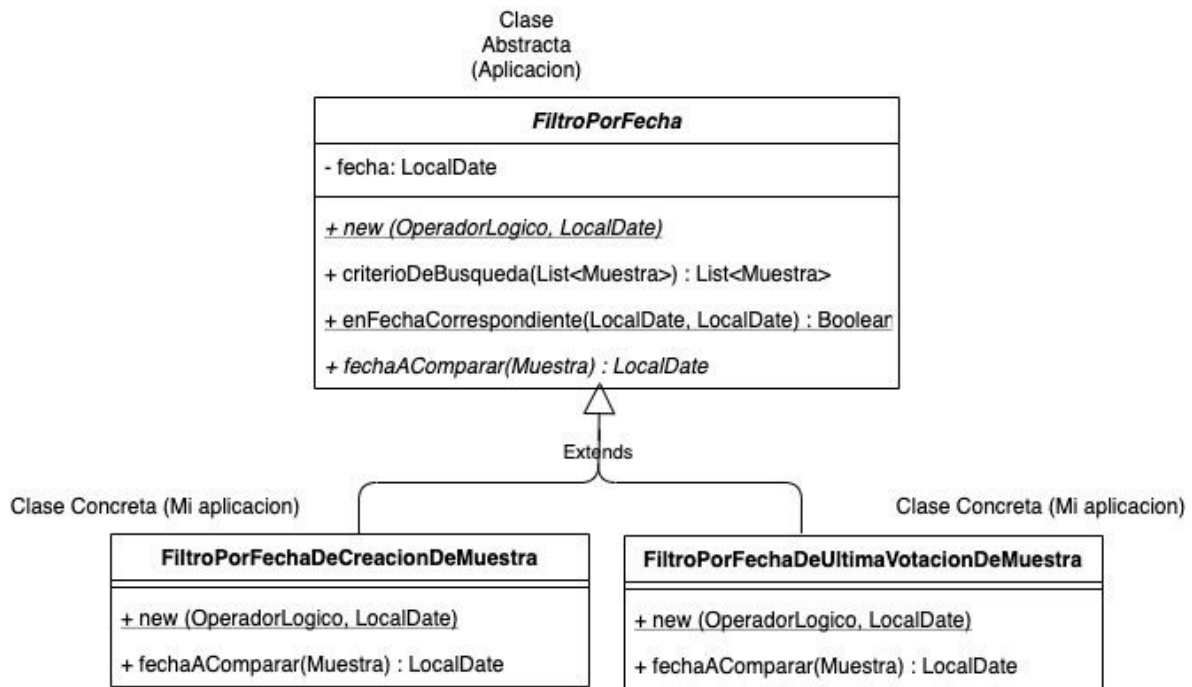
Para dos de los filtros, FiltroNivelValidacionMuestra y FiltroPorFecha, fue utilizado el patrón template method. En el primer caso, la super clase recorre la lista preguntando si se cumple el nivel de validación solicitado (votada o validada); dicha pregunta la responden las subclases mediante el mensaje tieneNivelCorrespondiente. Cada subclase implementa el método según les corresponda: Nivel Validado para una muestra validada, o Nivel Basico o Experto para una muestra votada.



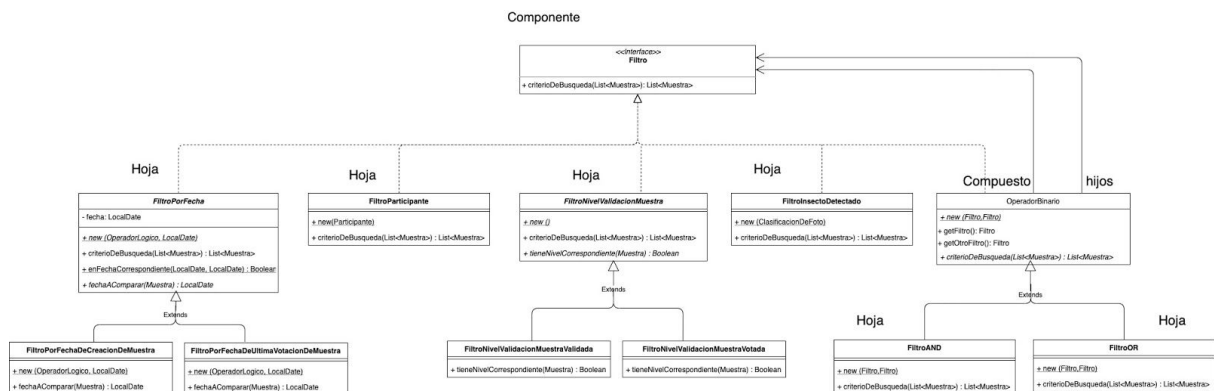
El segundo caso, FiltroPorFecha, en un principio fue pensado como un filtro para la fecha de creación de muestras, y se generó una clase para cada operador (muestras creadas antes, después o en determinada fecha). Al momento de crear el filtro para buscar las muestras por la última fecha de votación, se decidió unir ambos filtros en uno sólo que realice búsqueda por fecha. Como el único punto divergente entre ambos era la fecha a tomar de la muestra (la fecha de creación o la fecha de la votación), se decidió que el recorrido de las muestras lo realice la superclase, y la fecha por la cual son filtradas, sea un método aparte que cada subclase (FiltroPorFechaDeCreacionDeMuestra y FiltroPorFechaDeUltimaVotacionDeMuestra) desarrolla, quedando así conformado el patrón template method.

Para complementar este filtro, se creó una clase enumerador con las tres operaciones lógicas necesarias (mayores, menores e iguales a), con un método abstracto que cada una desarrolla para realizar su operación lógica específica comparando dos fechas. FiltroPorFecha, entonces, se crea con dos parámetros: el tipo de operador lógico, y una fecha.

En ambos casos de implementación de template method (FiltroPorFecha y FiltroNivelValidaciónMuestra), el mensaje a implementar por las subclases es abstracto; es decir, es obligatorio que las subclases lo desarrollen.



Finalmente, para generar la intersección y unión de los filtros de búsqueda, se decidió utilizar el patrón composite. De esta manera, la intersección y la unión también son filtros, dependientes de la clase abstracta OperadorBinario que implementa la interfaz Filtro, y responden también al mensaje criterioDeBusqueda de la interfaz Filtro trabajando recursivamente sobre otros filtros, que son las hojas del componente.



Otras correcciones realizadas al programa:

Se agregaron comentarios para generar una documentación uniforme a todo lo largo del programa.

Se corrigieron aquellos métodos donde se incorporaban variables sólo para llamarlas en el return.

Se organizaron las clases en paquetes.

Se modificó el método `crearRankingDeOpiniones` de la clase `NivelDeValidacion` utilizando el método `groupBy`.