Trabajo final programación con objetos 2

- Juan Cruz Ozores (juanozores97@gmail.com)
- Juan Manuel Sanchez Díaz (jmsanchezdiaz02@gmail.com)
- Ezequiel Gonzalez (ezegonzalez912@gmail.com)

1er semestre de 2022

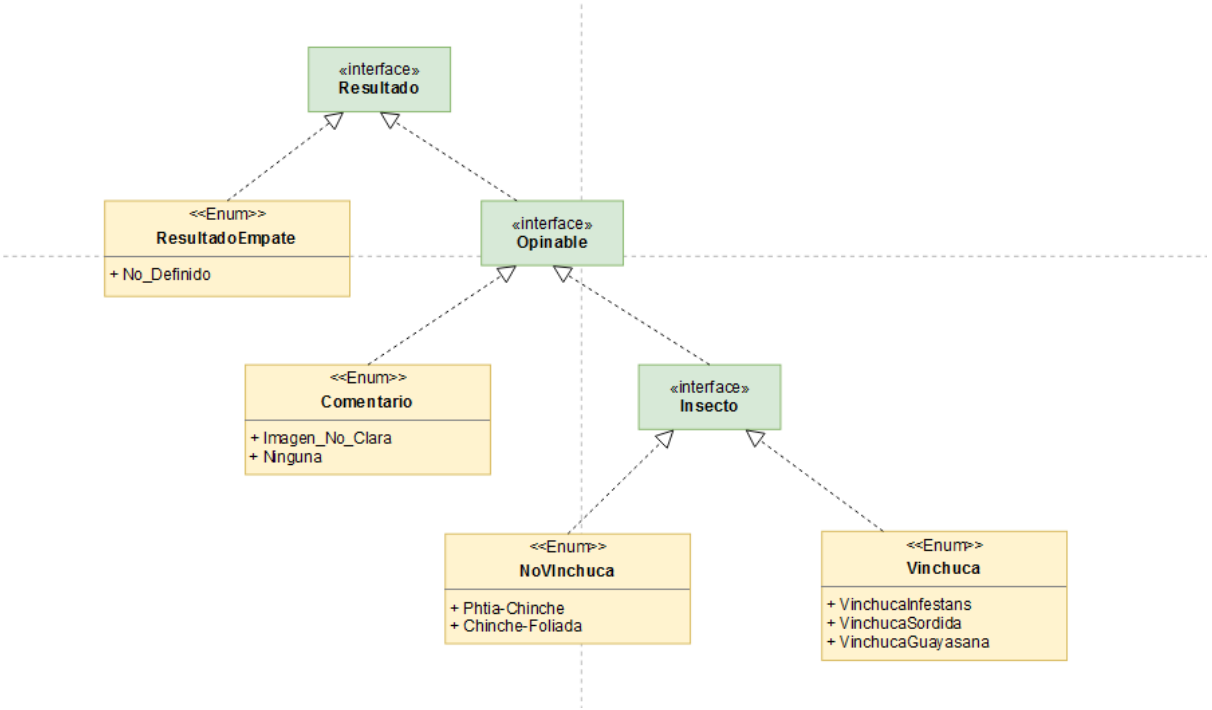
En el siguiente informe se expondrán los distintos detalles implementativos y decisiones de diseño presentes en la solución propuesta a la consigna del trabajo. Dichos elementos se dividen de acuerdo a las problemáticas particulares identificadas y/o clases intervinientes en la resolución de las mismas.

Resultado, Opinable, Insecto.

Cuando pensamos la lógica de las opiniones, resultados y verificaciones. Llegamos a la conclusión de que estos eran valores fijos y categóricos, por lo cual optamos por usar Enumerativos para representarlos, los cuales fueron NivelDeVerificación, ResultadoEmpate, Comentario, NoVinchuca, Vinchuca.

Con esto nos surgió un problema, ¿cómo podríamos usarlos como resultado sin romper el dominio?. Para eso diseñamos una jerarquía de interfaces que nos permiten agrupar Enumerativos.

Como se muestra en el siguiente diseño:



Aplicación Web

Para crear la lógica de la Aplicación Web, decidimos crear una clase **AplicaciónWeb**, la cual tiene como variables de instancia, una instancia de sí misma, una lista de zonas de cobertura y una lista de muestras. Decidimos hacerlo de esta manera porque si no teníamos un lugar central donde se almacenarán las muestras y zonas, habría un montón de listas duplicadas por el modelo. Además decidimos utilizar el patrón **Singleton** en esta clase, porque llegamos a la conclusión de que no tendría sentido tener más de una instancia **AplicaciónWeb** circulando por nuestro diseño.

Participantes Del Patrón.

- **AplicaciónWeb:** Un Singleton.

Usuarios

Para crear la lógica de **Usuario**, decidimos crear una clase Usuario, la cual tiene como variables de instancia, el id, nombre, y **NivelDeConocimiento**. Además los usuarios pueden enviar muestras, opinar sobre muestras, subir y bajar de nivel de conocimiento y saber responder si su nivel de conocimiento es experto o básico.

En el enunciado se dice que hay 2 tipos de usuario, el que puede subir y bajar de nivel y el que es un científico o una persona con estudios en el tema, cuyo nivel permanece fijo. Para implementar eso, utilizamos el patrón **State**, porque nos permite realizar distintos comportamientos para un protocolo común dependiendo del estado del usuario (NivelDeConocimiento, una interfaz).

Por ejemplo cuando un Usuario tiene NivelDeConocimiento Básico, solamente puede subir de nivel y no bajarlo.

Participantes Del Patrón.

- **Usuario**: Contexto, es la clase que tiene el estado interno.

- **NivelDeConocimiento**: State, es la interfaz que declara el protocolo común entre los estados concretos.

- **Básico, Experto y Experto Permanente**: ConcreteStates, que implementan la interfaz NivelDeConocimiento.

Filtros

Para crear la lógica de **Filtros**, decidimos implementar el patrón **Composite** el cual nos permite agrupar de forma uniforme objetos primitivos (Leaf) y complejos (Composite). Llegamos a la conclusión de que deberíamos usar este patrón porque aparte de tener filtros básicos, también existen filtros compuestos por las operaciones lógicas AND y OR.

Participantes Del Patrón.

- **IFiltro**: Componente, es la interfaz que declara el protocolo común.

- **FiltroTipoInsecto, FiltroFechaCreacion, FiltroFechaUltimaVotacion, FiltroTipoVerificacion:** son los objetos primitivos del patrón (Leaf's).

- **FiltroCompuesto**: Composite, este es la clase abstracta que abstrae el comportamiento común de los filtros compuestos, de este extienden **FiltroAND** y **FiltroOR**.

Ubicación

En lo que concierne a las ubicaciones hay que mencionar el detalle de implementación en la construcción de las mismas, ya que se decidió por la limitación de las latitudes y longitudes, permitiendo así que solo se generen instancias de ubicaciones con representaciones válidas en el sistema de coordenadas terrestres. Una de las motivaciones en la decisión fue la posibilidad de que en un futuro con esa información se quiera representar visualmente en un mapa las coordenadas definidas. Además, no se debería permitir a los usuarios definir ubicaciones no válidas porque generaría inconsistencias y errores en los datos que este proyecto apunta a obtener.

Aviso a las organizaciones

Respecto a esta problemática, se optó por diseñar un patrón Observer donde los roles son cumplidos por los siguientes elementos.

Participantes Del Patrón.

- **Observador**: interfaz que cumple el rol de observador abstracto, definiendo un protocolo para los objetos que deban ser notificados de eventos.
- **ZonaDeCobertura:** sujeto concreto, este conoce a sus observadores y provee para ellos una serie de mensajes para la suscripción o cancelación de suscripciones a distintos aspectos - eventos. Además, tiene un mensaje para la notificación a los suscriptores cuando sucedan los eventos mencionados previamente.
- **Organización:** observador concreto que conoce a las distintas zonas de cobertura a las que se suscribe según aspecto de interés e implementa la interfaz Observador.

Una peculiaridad en cuanto al diseño de este patrón en nuestra problemática fue la de utilizar un enumerativo para describir los distintos aspectos-eventos a los cuales se puede suscribir un observador, de esta manera se posibilita que una organización se suscriba a un evento utilizando un mismo mensaje pero pasando otro aspecto como parámetro. Una de las ventajas es la simplificación del protocolo de los elementos que cumplen roles en el patrón mediante la reducción de la cantidad de mensajes tanto para suscripción, notificación, y acción consecuente frente a los eventos suscritos (expresada en el protocolo del observador), favoreciendo la fácil extensión del patrón (sin extender protocolos) ante nuevos posibles aspectos - eventos a los que se quiera suscribir un observador. Además, poniendo foco en el protocolo de la interfaz observador, si se incorporan nuevos observadores a la zona de cobertura por algún motivo que lo requiera, estos no tendrían que implementar sin comportamiento alguno a ninguno de los mensajes que tendría la interfaz en el caso de que el observador se quiera suscribir solamente a un aspecto. Por ejemplo, teniendo en cuenta los eventos que expone la problemática actualmente, si el protocolo de observador no estuviera parametrizado, tendríamos un mensaje que se podría llamar muestraVerficada() y otro llamado muestraEnviada(), y si un observador solo se quiere suscribir a las muestras enviadas igualmente se vería obligado a implementar

muestraVerificada() aunque sin comportamiento. Con la parametrización del aspecto se suprime esa necesidad.

Otro detalle en la implementación de este patrón fue la utilización de mapas en Zona de cobertura para diferenciar por aspecto a los observadores, de esta manera se logra eficacia en el sentido de sólo notificar a los observadores que se suscribieron a un evento - aspecto específico. La contraparte de la utilización de estos mapas es la pérdida de performance por el aumento en el costo de acceso a los observadores dentro de la Zona de cobertura tanto para cuando exista una notificación,suscripción o cancelación de suscripción. Esta estructura también fue aprovechada para el vínculo de la organización como observador concreto con la acción correspondiente a un evento expresada en la Funcionalidad externa.

Muestra

Muestra es una de las clases más importantes, para crear la misma se decidió que debe recibir un *Usuario*, una *foto,* la *especie* según el usuario que la sube y la *ubicación*. De esta podemos obtener su fecha de creación, y todas las opiniones que recibe.

Entre los métodos más importantes se encuentran:

-agregarOpinion: El cual agrega opiniones pero antes realiza ciertas verificaciones, como que el usuario no haya opinado antes.

-getResultadoActual: Este es uno de los métodos principales, el mismo nos dice la opinión más votada. La lógica es un poco compleja, decidimos agrupar las opiniones, esto nos permite tener un mapa con la opinión y su cantidad de votos, luego bastaba con la utilización de un forEach para encontrar a la más votada.

-getVerificacionActual: Gracias a este método podemos saber si una muestra está verificada o no, en base a su resultado.