Trabajo Final
“A la caza de las vinchucas”

Materia: Programación Orientada a Objetos II

Docentes: Matias Butti, Diego Cano, Diego Torres

Integrantes del grupo

Castelanich Joaquín
joaquincastelanich@outlook.com

Vidal Franco
franconicolasvidal@gmail.com

Villalba Damián
damianalumno@gmail.com

Contenido del documento

- Decisiones de diseño
- Detalles de implementación relevantes
- Principios SOLID aplicados
- Patrones de diseño utilizados

Junio de 2025

Decisiones de diseño:

Estructura Proyecto: Organizamos el proyecto en paquetes según los subdominios del enunciado: *Muestra*, *Usuario*, *ZonaCobertura*, *Organización*, *FiltroBusqueda* y *AppWeb* para el sistema central.
Esta división respeta la separación de responsabilidades y mejora la mantenibilidad del código, permitiendo ubicar fácilmente cada clase según su rol funcional.

AplicacionWeb: Creamos la clase *AplicacionWeb* como coordinador central del sistema. Es la encargada de las operaciones clave como registrar muestras, usuarios, zonas y coordinar la recategorización o notificaciones. Esto facilita el acceso a la información global del sistema sin que las clases del dominio se acoplen entre sí. Si bien no fue impuesta por el enunciado, su presencia simplifica la orquestación de responsabilidades distribuidas sin romper el principio de responsabilidad única en los objetos de dominio.

Opiniones: ¿Por qué se almacenan en las Muestras y no en los Usuarios?
Decidimos almacenar las opiniones únicamente en las muestras, ya que son ellas quienes validan y conservan su historial. Cuando un usuario opina desde la app, lo hace sobre una muestra específica, que se encarga de procesar y aceptar (o no) esa opinión según el estado de evaluación de la misma y las reglas básicas propuestas por el enunciado.

Guardar las opiniones en los usuarios implicaría asumir que toda opinión es válida, lo que rompe con el control de integridad del

dominio. En su defecto, si quisiéramos que la muestra valide y luego el usuario almacene sólo las opiniones aceptadas, se requeriría un doble dispatch innecesariamente complejo que entorpece el diseño y la claridad de responsabilidades. Además, requeriría que cada *Opinión* contenga una referencia a la muestra sobre la que se emitió, acoplando innecesariamente ambas entidades y complicando la trazabilidad del resultado final.

Si bien podría haber una leve mejora de eficiencia en el acceso a las opiniones del usuario (por ejemplo, en la recategorización), esto implicaría lógica redundante o referencias cruzadas que no aportan claridad. Por el contrario, dejar el historial en la muestra permite construir el resultado actual sin ambigüedades, manteniendo una única fuente de verdad.

Este enfoque respeta el principio de responsabilidad única (SRP), mantiene bajo acoplamiento entre entidades y evita duplicación de datos. Consideramos que es una solución coherente con el dominio y el enunciado, privilegiando simplicidad y consistencia por sobre optimizaciones prematuras.

Detalles de implementación relevantes

Usuario: Creamos al usuario para representar a los participantes junto a su evolución dentro del sistema, centralizando su existencia en base a su conocimiento, lo que aporta para las opiniones y recategorización del mismo.
En relación a los usuarios “generales” y los validados externamente, optamos por cambiar el comportamiento del conocimiento en lugar de crear distintos tipos de usuario, porque nos parece más simple y flexible (OCP).

NivelConocimiento: Clase abstracta la cual define la base del nivel de conocimiento para un *Usuario*, delegando la responsabilidad a este a la hora de recategorizarse (SRP).
Cada nivel conoce su lógica para decidir si debe cambiar de nivel (de básico a experto, o viceversa). Esto se logra mediante un método común que permite al *Recategorizador* actuar sin conocer detalles internos (DIP). También incluimos un *NivelExpertoValidado* que representa a los usuarios validados que no deberían recategorizarse, tal como lo indica el enunciado.
Elegimos este enfoque por las siguientes razones:

- Permite validar a cualquier usuario en cualquier momento, sin necesidad de reemplazarlo por otro objeto. Esto es útil si un usuario “general” es reconocido externamente como experto más adelante.
- Evita crear muchas clases innecesarias. Al mantener una única clase Usuario, se simplifica el modelo y se conserva toda su información e historial.
- Es más fácil de mantener y extender, respetando a su vez el LSP. Si en el futuro aparecen otros niveles (como “moderador” o “especialista”), podemos agregar más variantes del nivel sin cambiar la estructura del usuario.

Recategorizador: La lógica de evolución de usuarios está contenida en esta clase (SRP), que utiliza una interfaz IDatosUsuario para acceder a los usuarios y sus muestras. Esto permite que el Recategorizador no dependa de AplicacionWeb, sino solo de lo que necesita para operar (DIP). Gracias a esto, el sistema es más modular, testeable y extensible.

Muestra: Contiene la información principal de la *Muestra* y deriva la verificación de la misma en EvaluacionMuestra, que encapsula las reglas sobre cómo se procesan las opiniones (y las aloja) en cada etapa de votación y cuándo debe cambiar. Además de esto agregamos en el constructor de cada Muestra a una instancia de *RegistroDeValidaciones*, la cual va a ser la encargada de enviar la información de la validación de esa muestra hacia la *AplicacionWeb*. Por ultimo, esta clase cuenta con un *ManejadorDeMuestras* propio el cual se encarga de suscribir y desuscribir a las zonas las cuales quieren recibir la información de cuando esa muestra sea validada.

ManejadorDeMuestras: Propio de cada muestra, esta clase es la encargada de llevar la lista de *IObserverMuestra* en la cual cada sujeto que quiera recibir información de la validación de una muestra (En este caso las zonasDeCobertura), previamente registradas en la aplicación, pueda ser notificado.

IObserverMuestra: Es la interfaz la cual implementan las *zonasDeCobertura* para recibir la información de cuando una muestra, previamente registrada en la app web, se valida.

Verificación de muestras (EvaluacionMuestra, EstadoEvaluacionMuestra): Gestionamos la evolución de una muestra con clases que representan las etapas según la interpretación del enunciado: *VotacionGeneral*, *VotacionExperto* y *MuestraVerificada*. Esto permite encapsular

las reglas de validación en cada etapa sin necesidad de condicionales extensos. Así, el comportamiento de una muestra cambia dinámicamente según su estado de evaluación.

Opinión: Representa la revisión que da un *Usuario* sobre una *Muestra*, capturando el nivel de conocimiento del usuario en ese momento y un *Resultado* que lo acompaña. De este modo, aunque luego el usuario cambie de nivel, la opinión mantiene su validez como experta o no, lo que permite consistencia en el historial de votaciones de dicha muestra.

Resultado: Representa los valores posibles en la *Opinion* y determina el resultado de la votación de una *Muestra*.
Decidimos utilizar un enum porque garantiza seguridad y expresividad. No tiene lógica adicional que no le corresponda y evitar margen de errores ante un String.

Ubicación: Luego de investigar, decidimos utilizar la implementación a través del cálculo "**Haversine**". Esto se debe a que decidimos implementar la ubicación con coordenadas para representar tanto la Latitud como la Longitud y a raíz de esto la decisión, ya que, este cálculo nos permite modelar las distancias con mayor exactitud y realismo que si lo hiciéramos con un plano 2D.
A la hora de calcular si una muestra se encuentra o no dentro de la zona de cobertura, calculamos la distancia con este método entre la ubicación de la muestra y el epicentro de la zona de cobertura y la comparamos con el radio en KM.

$a = \sin^2(\Delta\varphi / 2) + \cos(\varphi 1) * \cos(\varphi 2) * \sin^2(\Delta\lambda / 2)$
$c = 2 * \text{atan2}(\sqrt{a}, \sqrt{(1-a)})$
$d = R * c$

φ = latitud (en radianes)
λ = longitud (en radianes)
R = radio de la Tierra (6371 km)

En relación al siguiente requerimiento de ubicación: “Dado una muestra, conocer todas las muestras obtenidas a menos de x metros o kilómetros.” Se decidió incluirlo en la AplicacionWeb por el siguiente motivo:

- Cohesión y bajo acoplamiento

  Ubicar esta funcionalidad en AplicacionWeb mantiene la cohesión de las clases (Ubicación sigue siendo responsable de representar una posición y calcular distancias), y reduce el acoplamiento entre entidades del dominio que no deberían conocerse.

- Acceso centralizado a muestras

  En nuestro desarrollo la encargada de administrar y almacenar las muestras es la AplicacionWeb. Por eso resultó natural ubicar en ella la funcionalidad que permite buscar las muestras cercanas ya que tiene acceso directo al conjunto completo.

El requerimiento: “Conocer, a partir de una lista de ubicaciones, aquellas que se encuentran a menos de x metros o kilómetros”, fue implementado en la clase Ubicación.

La clase Ubicación representa coordenadas geográficas y se encarga del manejo de la información asociada a ellas. Determinar qué ubicaciones están cerca de otra es una responsabilidad directamente relacionada con la naturaleza de la clase, ya que, solo requiere acceder a otras ubicaciones. No depende del conocimiento de entidades del dominio como Muestra, Organización u otras estructuras del sistema.

Además implementar este método en la clase Ubicación permite utilizarlo desde distintos contextos del sistema: tanto para zonas de cobertura, como para organizaciones, o incluso para validar proximidad entre puntos. No está acoplado a una entidad del dominio específica.

Organizaciones: Las Organizaciones cuentan con dos instancias de funcionalidades externas. Una para cuando una muestra se registra y se le notifica a la *organización* y otra para cuando se valida una muestra. En nuestro modelo, las organizaciones, implementan la interfaz de *notificadosPorMuestra* ya que necesitan realizar un nuevo evento con su funcionalidad externa en el momento en el que se registra o se valida una muestra. Estas funcionalidades las puede cambiar a sus requerimientos.

Funcionalidad Externa: Para la implementación de esta interfaz, decidimos que cada *Organización* tenga la posibilidad de tener dos instancias de funcionalidades intercambiables. Como no hay información concreta sobre qué objetos implementan un evento sobre esta interfaz, simplemente la dejamos abierta para que en un futuro se pueda seguir desarrollando distintos tipos de acciones que puedan realizar las organizaciones a la hora del registro o validación de una muestra. Como por ejemplo: Crear una clase *Fumigación* que simplemente al recibir alguna notificación, proceda a fumigar la zona de cobertura asignada a la muestra. Esto cumple con el principio “Open-Closed” de SOLID.

Zonas de cobertura: Para respetar el principio de responsabilidad única (SRP), la clase encapsula únicamente la lógica relacionada con su identidad geográfica y delega tareas específicas a otras clases: los cálculos de distancia se abstraen mediante el objeto *CalculadorDistancia*, y las notificaciones se manejan a través del *ManejadorDeNotificaciones*.

En relación al seguimiento de muestras, se decidió no almacenar referencias directas a las muestras en la zona. La consigna aclara que “es importante que se conozcan en todo momento las muestras que se han reportado en cada zona de cobertura”, pero esta necesidad se resuelve de forma dinámica: cuando se registra una nueva muestra, se consulta en qué zonas

se encuentra (según su ubicación), en lugar de mantener un listado estático en cada zona. Esta solución minimiza el riesgo de inconsistencias y promueve un diseño desacoplado, basado en el principio de inversión de dependencias (DIP), ya que las zonas no dependen directamente de los detalles internos del manejo de muestras. La principal desventaja es que en un sistema cargado de zonas de cobertura, se ve reflejado en la eficiencia, ya que, cada muestra registrada debería recorrer todas las zonas. Para cumplir con la funcionalidad pedida en el enunciado, permitimos que una zona consulte las zonas que la solapan utilizando una interfaz de acceso (*IDatosZonaCobertura* que implementa la AplicacionWeb) para mantener bajo acoplamiento.

ManejadorDeNotificaciones: Es el encargado del manejo de las notificaciones a la hora del registro o validación de una muestra dentro de la zona correspondiente a él. También es el que tiene el registro de los notificados suscritos a las notificaciones. Puede agregar y eliminar notificados. Elegimos crear un manejador por *ZonaDeCobertura* ya que además de que una *Organización* puede estar registrada en varios manejadores de varias zonas, esto abstrae la información del registro de la *ZonaDeCobertura* y cumple con el principio de “Single responsability” de SOLID.

NotificadosPorMuestras: Esta interfaz tiene los métodos para recibir la información del registro o validación de muestra de la zona correspondiente. En este caso el único observador que le interesa esta información es la *Organización*. Cualquier otro tipo de observador distinto puede implementar esta interfaz en un futuro para poder recibir esta información y utilizarla a sus requerimientos sin necesidad de modificar el código ya hecho. Esto decidimos hacerlo así ya que cumple con el principio “Open-Closed” de SOLID.

FiltroDeBusqueda: Estos filtros los implementamos en el momento de instanciación de una *AplicacionWeb*. Decidimos que sea así ya que para cada aplicación que se crea estas puedan tener su propio filtro.

El filtro se encarga de filtrar las muestras recibidas de la *aplicacionWeb* segun un criterio dado por parámetro. Cada instancia de filtro cuenta con un criterio el cual conoce y filtra las muestras recibidas.

Criterio: Creamos una interfaz criterio la cual tiene un método llamado “cumpleMuestra”. Este método va a ser el mismo para cada criterio particular que se cree. Estos criterios deben implementar esta interfaz para sobreescribir el método en base a lo que se quiera filtrar.

CriterioBinario(CriterioAND, CriterioOR): Esta clase abstracta, la cual conoce dos criterios, implementa a la interfaz Criterio de tal manera que cuenta con dos subclases, *criterioAND* y *criterioOR*. Estas subclases extienden a la clase abstracta y sobreescribe el método de criterio de tal manera que dados dos criterios cualesquiera se los pueda combinar ya sea para compararlos mediante un OR o un AND. Esto cumple con el principio de “Open-Closed” ya que si en un futuro se quiere crear una nueva subclase, ejemplo: “CriterioXOR” simplemente tiene que extender la clase binaria y sobreescribir el método para que se cumpla la condición.

CriteriosUnarios(CriterioFechaCreacion, CriterioTipoDeInsecto, CriterioPorMuestraVerificada, etc.): Estos criterios a diferencia de los binarios no tienen una clase abstracta que los define. Ya que para muchos de ellos difieren en su comportamiento. Como por ejemplo es el caso del *CriterioPorTipoDeInsecto*, este criterio se construye a partir de un resultado que se le da por parámetro para que filtre las

muestras que cumplen con ese resultado. En cambio el *CriterioPorMuestraVerificada* simplemente va a filtrar, de todas las muestras, solo las que se encuentren verificadas. Este criterio no tiene ningún constructor ya que no lo necesita.

Al igual que los *CriteriosBinarios,* estos criterios, los creamos de tal forma que en un futuro se pueda crear otro criterio específico simplemente implementando la interfaz *Criterio* y sobreescribiendo el método “cumpleMuestra” dependiendo del criterio que se cree.

Principios SOLID aplicados

SRP (Principio de Responsabilidad Única)
Cada clase en el sistema tiene una única razón para cambiar, lo que mejora la cohesión y la mantenibilidad. Por ejemplo, Usuario delega la lógica de su nivel de conocimiento en NivelConocimiento, mientras que Recategorizador se encarga exclusivamente de evaluar si un usuario debe cambiar de nivel. Del mismo modo, ZonaDeCobertura se limita a representar una región geográfica, mientras que delega el envío de notificaciones en el ManejadorDeNotificaciones.

OCP (Principio Abierto/Cerrado)
Muchas partes del sistema están diseñadas para ser fácilmente extensibles sin modificar el código existente. Se puede agregar un nuevo tipo de NivelConocimiento o nuevos Criterios para el filtrado sin alterar las clases existentes. Esto se logra gracias al uso de herencia, composición e interfaces bien definidas.

LSP (Principio de Sustitución de Liskov)
Todas las subclases y variantes de comportamiento pueden ser utilizadas sin afectar el correcto funcionamiento del sistema. Por ejemplo, NivelExpertoValidado puede sustituir a cualquier otro NivelConocimiento sin romper la lógica. Esto permite que el comportamiento cambie dinámicamente en base al nivel del usuario sin tener que modificar los clientes.

ISP (Principio de Segregación de Interfaces)
Las interfaces están diseñadas para ser específicas y enfocadas. Por ejemplo, IDatosUsuario solo expone los métodos necesarios para que Recategorizador funcione, sin necesidad de que conozca toda la lógica de AplicacionWeb. Del mismo modo, NotificadosPorMuestras permite que las organizaciones solo

implementen lo relacionado con notificaciones sin acoplarse a otras responsabilidades del sistema.

DIP (Principio de Inversión de Dependencias)
El diseño evita que las clases de alto nivel dependan de clases concretas. En su lugar, dependen de abstracciones. Por ejemplo, EvaluacionMuestra delega el envío de muestras verificadas a través de RegistroDeValidaciones, una interfaz que puede ser implementada por cualquier entidad interesada (como AplicacionWeb).

Patrones utilizados y sus sujetos

- Patrón Observer: Utilizamos este patrón ya que cumple con las necesidades del envío de información hacia la organización y/o futuros observadores de las muestras.
    - Observador concreto: *Organización*.
    - Sujeto: *ManejadorDeNotificaciones*.
    - Observador: *NotificadosPorMuestra*.
    - Sujeto concreto: *ZonaDeCobertura*

- Template Method (CriterioBinario): En este caso este patrón fue útil para no repetir codigo en dos clases distintas (CriterioAND, CriterioOR)
    - Clase abstracta: *CriterioBinario*
    - Clases concretas: *CriterioOR*, *CriterioAND*

- Template Method (NivelConocimiento): Patrón detectado ante la necesidad de lógica distinta entre los niveles a la hora de saber si deben recategorizarse. Nuestra solución evita la repetición de código y facilita la variación de algoritmos sin alterar la lógica común definida en la clase base, favoreciendo la extensión.
    - Clase abstracta: *NivelConocimiento*
    - Clase concretas: *NivelBasico, NivelExperto y NivelExpertoValidado*

- Patrón State: Utilización del patrón para definir un estado de evaluación a cada muestra, donde cada estado tiene un comportamiento según la etapa de votación en la que se encuentra, determinando su siguiente y permitiendo avanzar en el proceso de votación de una muestra de manera uniforme y

controlada, cumpliendo las características solicitadas por el enunciado.
    - Contexto: *EvaluacionMuestra*
    - Estado: *EstadoEvaluacionMuestra*
    - Estados concretos: *VotacionGeneral*, *VotacionExperto y MuestraVerificada*

- Patrón Observer: Utilizamos este patrón para no forzar que todas las zonas de cobertura reciban la información de una muestra validada. Con esta implementación logramos que solo las zonas suscriptas puedan recibir esta información, una vez que la muestra sea registrada en esa zona.
    - Observador concreto: *ZonaDeCobertura*.
    - Sujeto: *ManejadorDeMuestras*.
    - Observador: *IObserverMuestra*.
    - Sujeto concreto: *Muestra*