# Trabajo Práctico Objetos 2 - 2025 1° Cuatrimestre

## Integrantes:


- Lucarni, Lucas
- Pettinelli, Oriana
- Rincon, Tatiana


## Resumen


El sistema permite registrar y verificar la presencia de vinchucas a partir de muestras recolectadas por usuarios en ubicaciones determinadas. La clase `Sistema` centraliza la gestión de muestras, zonas de cobertura y organizaciones.
Esto favorece el desacoplamiento entre componentes, facilitando su mantenimiento, prueba y extensión futura.

Se emplean múltiples patrones de diseño:

- **Composite**, para construir filtros complejos sobre las muestras (`AND`, `OR`).

- **State**, para representar el **nivel de usuario** (básico o experto), el cual **cambia dinámicamente** en función de su participación activa en el sistema (opiniones y muestras cargadas en los últimos 30 días). Y para determinar el **estado de una muestra**, que va cambiando a partir de las opiniones que puede recibir, hasta llegar al estado de 'Verificada'.

- **Template Method**, utilizado en la lógica de actualización del nivel de usuario. La clase abstracta `NivelState` define un método plantilla (`updateNivel`) que encapsula los pasos del algoritmo de cambio de nivel, permitiendo que las subclases concreten la lógica según corresponda al tipo de usuario.

- **Observer**, para notificar automáticamente a organizaciones cuando se agregan o actualizan muestras. Las organizaciones se **suscriben dinámicamente** a zonas de cobertura y pueden **desuscribirse en cualquier momento**, permitiendo una arquitectura flexible y desacoplada ante los eventos del sistema.

# Requerimientos del sistema

- Los usuarios pueden registrarse en el sistema y se les asigna un nivel (básico o experto) que varía dinámicamente según su actividad en los últimos 30 días. En el caso de los usuarios del tipo **ExpertoExterno**, siempre serán considerados como de **Nivel Experto** por el sistema.

- Los usuarios pueden cargar muestras, proporcionando:

    - Una imagen (foto de la vinchuca).

    - Fecha y hora de la muestra.

    - Ubicación (latitud, longitud).

    - Una primera opinión, que se considera como la opinión del usuario creador.

- Las muestras pueden recibir nuevas opiniones por parte de otros usuarios, pero:

    - Cada usuario puede opinar **una sola vez** sobre una misma muestra.

    - El usuario que creó la muestra **no puede volver a opinar** sobre ella, ya que su opinión inicial fue registrada al momento de la creación.

- La muestra valida las opiniones recibidas, y cambia su estado según las opiniones recibidas hasta llegar a estar *Verificada*. En cualquier momento se le puede consultar por su resultado actual.

- Las zonas de cobertura están definidas por un epicentro y un radio, y observan las muestras que ocurren dentro de esa área.

- Las organizaciones pueden observar zonas de cobertura para ser notificadas ante eventos relevantes (nueva muestra o cambio de veredicto).

- El sistema permite que las organizaciones se suscriban y desuscriban de zonas de cobertura.

- Las muestras se notifican automáticamente a todas las zonas que las cubren, y estas a su vez notifican a sus organizaciones observadoras.

- Se pueden aplicar filtros sobre las muestras, combinando condiciones como:

    - Fecha de creación.

    - Estado de verificación.

    - Tipo de insecto detectado.
    - Fecha de última votación.

- Las consultas pueden ser refinadas mediante filtros compuestos (`AND`, `OR`).
- Se mantiene un registro histórico de opiniones por muestra.

# Modelado y decisiones de diseño

## Usuario y Nivel

La clase `Usuario` representa a cualquier persona que interactúa con el sistema, ya sea cargando una muestra o dando su opinión sobre otras muestras cargadas.
Cada usuario tiene asociado un **nivel**, que puede ser **Básico** o **Experto**, y que **varía dinámicamente** en función de la actividad reciente del usuario (cantidad de muestras creadas y opiniones emitidas en los últimos 30 días).

### Patrón State

Para modelar este comportamiento, se aplicó el **Patrón State**. El sistema define una jerarquía de clases que encapsulan los distintos niveles posibles:

- La clase abstracta `NivelState` representa el estado actual del usuario.
- Las subclases concretas `Basico` y `Experto` implementan los comportamientos correspondientes.

### Roles del patrón State en Usuario-Nivel:

- **Context**: `Usuario`
  Es quien mantiene una referencia al objeto `NivelState` actual, y delega en él los métodos que determinan si debe cambiar su estado.
- **State (Clase Abstracta)**: `NivelState`
  Define la interfaz común para todos los estadíos de los niveles de los usuarios.
- **ConcretState**:
    - `Basico`
    - `Experto`

Este enfoque permite que el comportamiento del usuario cambie dinámicamente sin necesidad de condicionales, y deja abierta la posibilidad de incorporar **nuevos niveles en el futuro** (por ejemplo, "Moderador", "Auditor", etc.) simplemente añadiendo nuevas subclases de `NivelState`.
Adicionalmente, la clase `Usuario` permite indicar mediante un booleano si se trata de un **experto externo**, lo cual puede configurarse desde su constructor, en ese caso, se lo inicializa directamente con un nivel experto y el mismo no se modifica nunca. Si dicha variable se encuentra en false, por defecto los usuarios inician como básicos y su nivel se ajusta automáticamente según su actividad.

## Patrón Template Method

Además del patrón State, en la clase `NivelState` se aplica el **Patrón Template Method** para estructurar la lógica de actualización de nivel.

- El método `updateNivel(Usuario)` define el esqueleto de la operación: primero verifica si corresponde un cambio de nivel mediante el método `debeCambiarNivel`, y en caso afirmativo, realiza la transición al nuevo estado utilizando `nuevoNivel`.

- Este método no se sobreescribe en las subclases, sino que delega en pasos (métodos abstractos o *hooks*) que pueden variar dependiendo del tipo de nivel.

Esta estructura permite que el proceso de actualización de nivel sea consistente para todos los niveles, a la vez que ofrece flexibilidad para personalizar partes específicas en las subclases, evitando duplicación de lógica y mejorando la mantenibilidad del código.

## Roles del patrón Template Method en NivelState:

- **Template Method:**
  - `updateNivel(Usuario)`
    Define la estructura del algoritmo para decidir y ejecutar el cambio de nivel.

- **Primitive Operations (pasos que varían según el nivel):**
  - `debeCambiarNivel(Usuario)`
  - `nuevoNivel()`

- **Hooks (métodos opcionales):**
  - `hook1()`
  - `hook2()`
    Estos permiten que las subclases agreguen comportamiento adicional sin necesidad de modificar la estructura general definida en el método plantilla.

# Muestra, Opiniones y Veredicto

La clase `Muestra` representa una posible detección de vinchuca y contiene la siguiente información:

- La `Especie de Vinchuca` que se ha fotografiado (Infestants, Sordida, Guasayana).
- Una `Foto`, que representa la imagen aportada por el usuario.
- Una `Ubicacion`, compuesta por latitud, longitud y dirección textual.
- La identificación del `Usuario` que recolectó la muestra.
- `Fecha y hora` de creación.
- Una lista de `Opiniones` emitidas por distintos usuarios.
- Un `veredicto actual` (puede estar sin definir, o validado).

**En todo momento se puede pedir a una muestra su resultado actual, dado por la opinión que tenga mayoría de votos.**

La primera opinión de una muestra se genera automáticamente al momento de su creación, y corresponde al usuario que la cargó.
Cada `Opinion` está compuesta por:

- Un `TipoDeOpinion` (Vinchuca, y en tal caso la especie, Chinche Foliada, Phtia-Chinche, Ninguna o Imagen Poco Clara).
- La `fecha` en la que fue emitida.
- El nivel del `Usuario` autor (Básico o Experto).

El sistema impone las siguientes reglas sobre las opiniones:

- Un usuario **no puede opinar más de una vez** sobre la misma muestra.
- El **usuario creador no puede volver a opinar**, ya que su opinión fue registrada al crear la muestra.
- Si bien el usuario puede cambiar de nivel a lo largo del tiempo, la opinión mantiene el nivel con el que fue cargada.

## Estrategia de validación y verificación (Patrón State)

En principio, no existe un límite para la cantidad de usuarios con nivel básico que pueden opinar sobre una muestra, pero al momento en que un usuario experto opina sobre una muestra, los de nivel básico ya no pueden opinar, a partir de ese momento solo los expertos pueden opinar. La muestra queda verificada cuando dos expertos coinciden en su opinión.
Para representar estas reglas de manera flexible, se aplicó el **Patrón State**. En nuestro diseño definimos distintos estados que irán cambiando de acuerdo al estadío en el que se encuentre la muestra de acuerdo a las opiniones recibidas.

## EstadoDeMuestra

La clase abstracta `EstadoDeMuestra` representa el estado actual de verificación de una muestra. Esta es una **abstracción** que encapsula la lógica que define:

- Cuándo una muestra puede ser considerada verificada.
- Qué tipo de opinión puede alterar el veredicto.
- Cuál es el resultado actual de la muestra.

Se definieron tres implementaciones concretas:

- **CualquierOpinion**: Permite que cualquier usuario opine sobre una muestra, hasta que se reciba una opinión de experto y ahí cambia de estado..
- **SoloOpinionDeExperto**: Solo considera válidas las opiniones de usuarios expertos. Verifica la muestra si hay al menos dos expertos que coinciden.
- **MuestraVerificada**: Representa una muestra ya verificada. No permite modificar el resultado, incluso si se agregan más opiniones.

## Roles del patrón State en Muestra:

- **Context**: `Muestra`
  Es quien mantiene una referencia al objeto `EstadoDeMuestra` actual, y delega en él los métodos que determinan si debe cambiar su estado, y define el resultado actual o final.

- **State (Clase Abstracta)**: `EstadoDeMuestra`
  Define la interfaz común para todos los estadíos de las muestras.

- **ConcretState**:
  - `CualquierOpinion`
  - `SoloOpinionDeExperto`
  - `MuestraVerificada`

Esta arquitectura permite que la lógica de verificación evolucione fácilmente. Por ejemplo, si se incorporan nuevas políticas que cambien el estado de la muestra, bastaría con agregar nuevas subclases de `EstadoDeMuestra`, sin modificar `Muestra`.

# Zonas de cobertura, Ubicación y Organizaciones

Se registran `Organizaciones` de las que se puede conocer:

- `Ubicación`
- `TipoDeOrganizacion` (Enum con tipos predefinidos: Salud, Educativa, Cultural y Asistencia).
- `Cantidad de personas` que trabajan allí.

La `Ubicación` está definida por dos elementos: latitud y longitud. A partir de una ubicación podremos saber la distancia que haya entre dos ubicaciones. También dada una lista de ubicaciones, podremos determinar aquellas que se encuentran a menos de x kilómetros. Y dada una muestra, conoceremos todas las muestras obtenidas a menos de x kilómetros.

Por su parte, las `ZonaDeCobertura` representan regiones geográficas circulares dentro de las cuales el sistema desea observar la aparición de vinchucas. Cada zona se define por:

- Un **nombre**.
- Un **epicentro** (`Ubicacion` con latitud y longitud).
- Un **radio** de cobertura.

El sistema permite registrar múltiples zonas, y cada una puede observar varias muestras que ocurren dentro de su área. Para determinar si una muestra se encuentra dentro del alcance, la zona utiliza el método `estaDentroDeLaZona`.

Además, las zonas de cobertura incluyen la funcionalidad que permite saber cuáles otras zonas la solapan.

Cuando se crea una nueva muestra y es cargada en el sistema, **es el `Sistema` quien se encarga de determinar a qué zonas corresponde dicha muestra**, evaluando su ubicación. Una vez identificadas las zonas correspondientes (pueden ser múltiples), el sistema **asocia la muestra a cada una de ellas**.

Posteriormente, son las propias `ZonaDeCobertura` quienes **notifican a sus organizaciones observadoras** sobre la incorporación de una nueva muestra o sobre cambios en las muestras que ya están bajo su vigilancia. Esto se implementa usando el **Patrón Observer**.

## Aplicación del patrón Observer:

- **Observable (Sujeto)**: `ZonaDeCobertura`
  Notifica a sus organizaciones cuando una muestra relevante es agregada o actualizada.

- **Observer**: `Organizacion`
  Recibe notificaciones de las zonas que observa y actúa en consecuencia.

Las organizaciones se pueden **suscribir o desuscribir dinámicamente** a las zonas de cobertura. Este mecanismo permite que múltiples organizaciones estén vinculadas a distintas zonas sin depender de la lógica interna del modelo `Muestra`.

Además, cada organización puede poseer una o más **funcionalidades externas**, modeladas mediante la interfaz `FuncionalidadExterna`. Esta interfaz permite ejecutar distintas acciones al ser notificada (como enviar alertas, recolectar estadísticas, etc.), facilitando la incorporación de nuevos tipos de funcionalidades sin modificar las organizaciones existentes.

Este diseño promueve una estructura **desacoplada y extensible**, donde las zonas y las organizaciones reaccionan de manera independiente a los eventos del sistema, sin necesidad de acoplarse directamente a las muestras ni al sistema central.

Por último, si bien no se implementa en el diseño y excede el alcance de los requerimientos pedidos en el trabajo, observamos que podría ser una buena oportunidad para implementar un Patrón Strategy, ya que, la Organización podría ser el Context, y las distintas Funcionalidades Externas serían las posibles estrategias a utilizar por cada Organización, que tendría la flexibilidad de cambiarlas de acuerdo a la funcionalidad que quiera usar.

# Filtros y consultas sobre muestras

El sistema permite aplicar **consultas complejas** sobre las muestras registradas mediante el uso de distintos filtros. Estos filtros permiten seleccionar subconjuntos de muestras en base a distintos criterios como:

- Fecha de creación de la muestra.
- Fecha de última votación.
- Tipo de insecto detectado en la muestra.
- Nivel de verificación (Votada o Verificada).

Para representar y combinar filtros de forma flexible y escalable, se utilizó el **Patrón Composite**.

## Estructura del patrón Composite:

- **Abstract Component**: `Consultable`
  Interfaz que define el método `filtrarLasMuestras(List<Muestra>)`, que debe implementar cualquier clase que desee actuar como filtro.

- **Leaf (filtros simples)**:

    - `FiltroCreacionDeMuestra`
    - `FiltroUltimaVotacion`
    - `FiltroTipoDeInsectoDetectado`
    - `FiltroNivelDeVerificacion`
      Cada uno de estos implementa `Consultable` y define una condición específica sobre las muestras.

- **Composite (filtros compuestos)**:
    - `AND`
    - `OR`
      Estas clases permiten **combinar lógicamente dos filtros**, ya sean simples o compuestos, lo que permite construir estructuras de filtrado complejas y reutilizables.

- **Client**: `Sistema`
  El sistema actúa como cliente del patrón Composite, ya que utiliza combinaciones de filtros para resolver búsquedas solicitadas por los usuarios u organizaciones.

Este diseño facilita:

- La reutilización de lógica de iteración y filtrado.
- La incorporación de nuevos tipos de filtros sin duplicar código.
- La combinación de condiciones sin generar una explosión de clases.

Por ejemplo, una consulta puede filtrar por un tipo de insecto, y que esté verificada o cuya fecha de última votación sea una en específica. Todo eso puede representarse con una combinación de `AND` `OR` y filtros simples adecuados.

# Conclusión

El sistema desarrollado permite modelar de forma robusta y extensible el proceso de monitoreo de vinchucas a partir de muestras geolocalizadas y validadas colaborativamente. Gracias al uso adecuado de **patrones de diseño**, se logró construir una arquitectura flexible que facilita la incorporación de nuevas funcionalidades sin comprometer la cohesión del sistema.

Se destacan los siguientes aspectos:

- El **Patrón State** se aplicó para representar dinámicamente el nivel de un usuario en función de su actividad reciente, favoreciendo una lógica fluida y extensible. Así como también, se utilizó para intercambiar los estados de las muestras a medida que iban recibiendo opiniones.

- El **Patrón Template Method** permitió estructurar el algoritmo de actualización del nivel de usuario, definiendo una lógica común en la clase abstracta `NivelState`.. Esto promueve la reutilización de lógica, evita duplicación de código y facilita la extensión del comportamiento según el tipo de usuario.

- El **Patrón Composite**, permitió una implementación limpia y escalable de filtros complejos sobre muestras.

- El **Patrón Observer** permitió implementar un modelo reactivo de notificaciones entre zonas de cobertura y organizaciones, con bajo acoplamiento y alta reutilización.

La responsabilidad de cada clase se mantuvo enfocada en un único propósito, siguiendo principios SOLID como el de Single Responsibility, lo cual facilita tanto la lectura como el mantenimiento del código. Además, en la implementación se tuvo en cuenta que las clases estén abiertas a escalar pero sin necesidad de que deban ser modificadas internamente, cómo es el caso de las muestras que podrían sumar nuevos estadíos, por su parte, las clases que tienen subclases siguen el principio de Liskov Substitution, y se cumple con los principios de Interface Segregation y Dependency Inversion, ya qué las clases dependen de abstracciones y no de implementaciones concretas, y dichas abstracciones son lo más particulares posible para evitar que las clases tengan métodos que no van a utilizar, como por ejemplo, `FuncionalidadExterna`, `NivelState` o `EstadoDeVerificacion`.

El sistema está preparado para escalar en cantidad de muestras, usuarios, zonas y criterios de validación, cumpliendo los objetivos del trabajo práctico tanto a nivel funcional como de diseño.

# UML de diseño de clases