



# Trabajo Integrador de P002

1ra Entrega(26/06/2025)

---

Joaquin Doares

Aisha Cassano

Thomas Sandoval

Grupo: Número 19, “Los polimórficos”

Programación con Objetos 2

## Tareas a Realizar

1. Un diseño de la solución completa utilizando diagramas de clases UML.
2. La documentación de las decisiones de diseño, detalles de implementación que merezcan ser explicados, patrones de diseño utilizados y los roles según la definición de Gamma et. al.
3. Implementación completa en lenguaje JAVA que incluya test de unidad con un 95% de cobertura.


## Objetivo y Decisiones de Diseño

Este proyecto tiene como propósito desarrollar un sistema que permita mapear la presencia de vinchucas en distintas zonas de Argentina. A partir de esa información, se busca establecer vínculos entre la aparición del insecto, los registros de personas diagnosticadas con chagas, la actividad de organizaciones locales, y las posibilidades de reducir la propagación de esta enfermedad.

La propuesta incluye dos aplicaciones interconectadas: una app móvil, encargada de recolectar los datos en campo, y una plataforma web que recibe esas muestras y se ocupa de su análisis.

En esta instancia, el enfoque está puesto exclusivamente en la lógica de negocio correspondiente a la aplicación web.

---




A la hora de decidir el diseño, empezamos por el UML. En este planteamos tener los siguientes objetos que serán los principales: Muestra, Usuario, AppWeb, ZonaDeCobertura, Organizacion, Ubicacion, Opinion y BuscadorDeMuestra. En base a estos que son los “pilares” de éste proyecto, tuvimos la capacidad de hacer el resto del trabajo en el lenguaje de JAVA.

Por el lado de Usuario, habíamos asumido que era una clase que tenía un comportamiento relacionado con la creación de la muestra, cosa que interpretamos mal, ya que el enunciado mismo nos dice que **“...El proyecto tiene dos aplicaciones conectadas, una móvil donde se realiza la toma de la información y una web donde se reciben los datos de las muestras y se hace el procesamiento...”**, al hablarlo con los profesores y ayudantes, y recibir el feedback, supimos darnos cuenta de el error que teníamos e hicimos la clase AppWeb que era la que realmente se encargaba del procesamiento de las muestras.

Durante el desarrollo del proyecto, se adoptaron ciertos comportamientos clave para mejorar la comunicación entre los diferentes componentes del sistema, siguiendo una lógica inspirada en el patrón Observer y aplicando distintos niveles de notificación.

Primero que nada, debemos tener en cuenta que cada vez que se crea o se verifica una nueva muestra, esto desencadena eventos en las zonas de cobertura. Esta relación se establece de manera indirecta: las zonas de cobertura actúan como observadores de las muestras, respondiendo a su creación o validación para determinar si se encuentran dentro de su área geográfica.

Cada área de cobertura cuenta con sus propios observadores, que son las organizaciones interesadas en recibir información sobre los



eventos que suceden dentro de su ámbito. Cuando estas organizaciones son notificadas por la zona, se encargan de comunicar estos eventos a sus funciones externas, las cuales están diseñadas para responder de manera personalizada según las necesidades de cada organización.

Además, asumimos que cada objeto dentro del dominio está registrado en la instancia del Sistema de la aplicación Web, que actúa como el núcleo de una administración centralizada y se encarga de gestionar usuarios, muestras, zonas de cobertura y organizaciones.

Hablando de los usuarios, se identificaron tres categorías: básicos, expertos y validados. Los usuarios validados son un caso particular, ya que siempre tienen un nivel experto y su estado permanece constante. Por otro lado, los usuarios básicos y expertos pueden cambiar de estado según su actividad en el sistema, evaluando factores como la cantidad de muestras que han enviado o las opiniones que han dado en los últimos 30 días.

Estas decisiones hicieron posible crear una arquitectura modular y flexible, donde cada componente del sistema actúa de manera independiente y conectada ante eventos importantes, garantizando que el comportamiento se pueda extender sin crear dependencias rígidas.

---

Ahora sí, pasamos a ver cómo implementamos cada clase.

## Detalles de Implementación

### Usuario:

Las clases que manejan a los usuarios del sistema, teniendo en cuenta su nombre, estado actual (básico, experto o verificado) y las funcionalidades para calcular su nivel de actividad en los últimos 30 días, como la cantidad de muestras enviadas y las opiniones emitidas. El diseño se basa en un enfoque similar al patrón State (Usuario / UsuarioVerificado) del libro de Gamma.

### AppWeb:


Agrupar a la clase AppWeb, que se encarga de coordinar y orquestar los diferentes elementos de la aplicación. Esta clase centraliza la gestión de usuarios, muestras, zonas de cobertura y organizaciones dentro del sistema.

### Muestra:

Tiene una serie de respuestas a un formulario diseñado para informar sobre la posible presencia de vinchucas, y viene acompañado de una imagen. Esta clase se encarga de centralizar el proceso de identificación del insecto, reuniendo opiniones tanto de expertos como de personas comunes. Además, se definen los diferentes estados que puede tener una muestra: MuestraAbierta, MuestraValidada, MuestraSoloExpertos, etc, que indican si la muestra está en espera de validación, en proceso de análisis o ya confirmada.

### Ubicación:

Describe un punto geográfico a través de la latitud y longitud. Ofrece funciones para calcular la distancia entre dos lugares usando la fórmula de



Haversine, y también permite filtrar ubicaciones o muestras que estén dentro de un radio específico.

### BuscadorDeMuestra(CriterioDeBusqueda):

Contiene la clase `BuscadorMuestra`, que se encarga de aplicar diferentes filtros a las muestras, basándose en criterios de búsqueda específicos.

### ZonaDeCobertura:

La `zonaDeCobertura` se compone de clases e interfaces que representan regiones geográficas mediante un epicentro y un radio de acción. Estas entidades son útiles para identificar muestras que se encuentren dentro de su área y para gestionar la notificación a los observadores cuando se cargan o validan nuevas muestras en esa zona.

### Organización:

La clase `Organización` se encarga de representar a las entidades que están interesadas en los eventos que ocurren dentro de una o más áreas de cobertura. Estas organizaciones tienen la capacidad de recibir notificaciones cuando se cargan o validan muestras, y pueden decidir cómo reaccionar ante esos eventos mediante funcionalidades externas.

### Opinión:

Las opiniones que los usuarios pueden emitir sobre las muestras son modeladas de manera cuidadosa. Cada opinión incluye detalles sobre el autor, el tipo de opinión y la fecha en que se emitió.

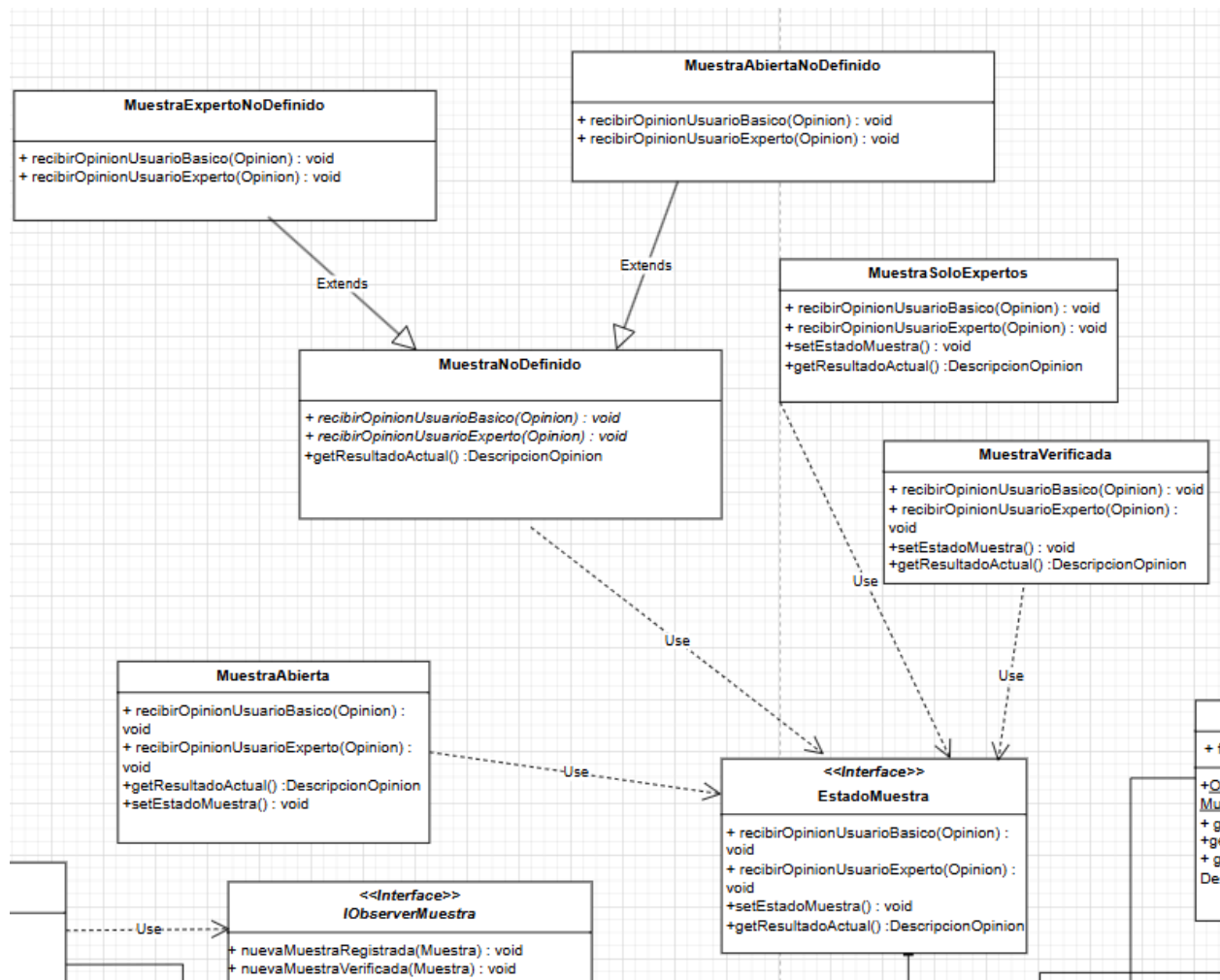
Además, se presenta un enum llamado `DescripcionOpinion`, que clasifica la opinión según el caso: ya sea una especie de vinchuca, chinche, o incluso observaciones especiales, como imágenes poco claras o que no se pueden identificar.

## Patrones de Diseños Utilizados

### 4. 1 Muestra - State

Participantes:

- Context : Muestra
- State : EstadoMuestra
- ConcreteStates : MuestraVerificada, MuestraSoloExpertos, MuestraAbiertaNoDefinido, MuestraExpertoNoDefinido, MuestraNoDefinido, etc

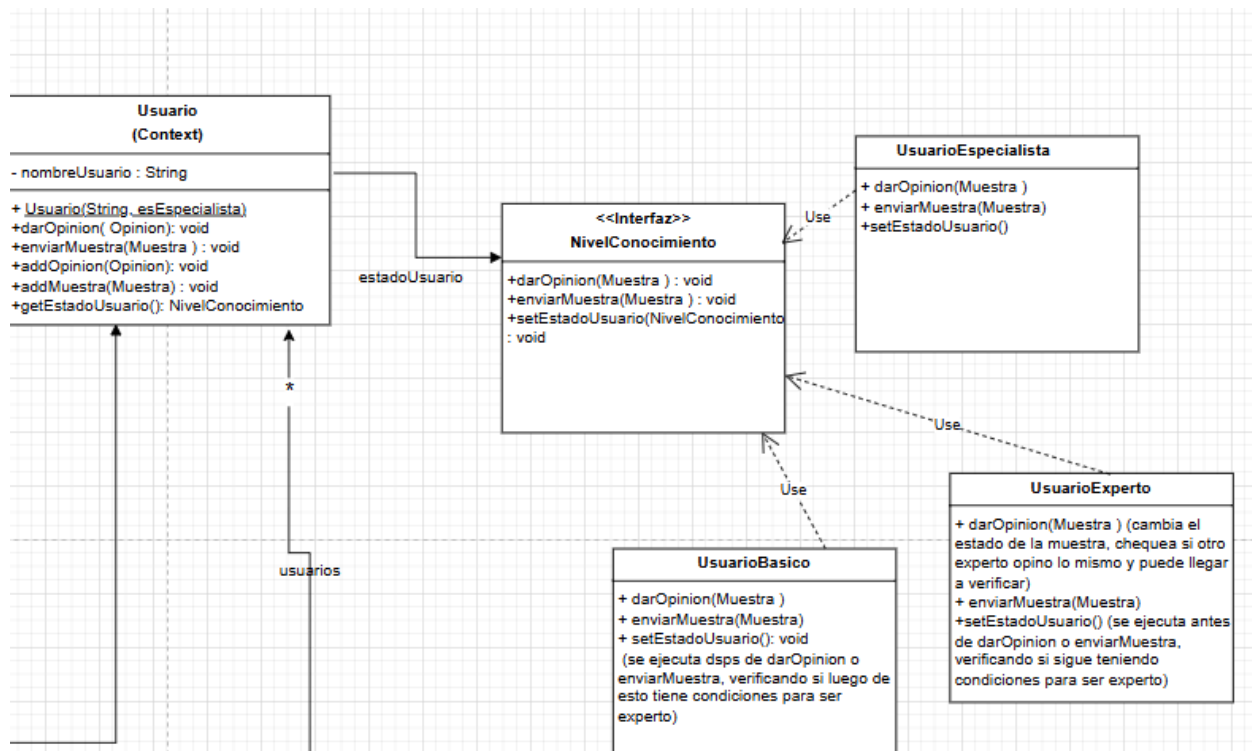


Clase Muestra (No alcanzaba la imagen)

El diagrama utiliza el patrón State, lo que significa que una muestra puede estar en diferentes estados, como abierta, verificada, solo para expertos, entre otros. Cada uno de estos estados cambia cómo se comporta la muestra al recibir una opinión, delegando esa lógica a objetos que implementan la interfaz EstadoMuestra. Esto permite que MuestraAbierta no tenga que preocuparse por los detalles de cada tipo de estado, lo que favorece el principio de abierto/cerrado y ayuda a reducir el uso de condicionales.

## 4.2 Usuario - State

- Context: Usuario.
- Estado: NivelConocimiento
- EstadoConcreto: UsuarioBasico, UsuarioExperto, UsuarioEspecialista son los EstadosConcretos.



Este diseño utiliza el patrón State, lo que permite que un objeto Usuario ajuste su comportamiento de manera dinámica según su nivel de



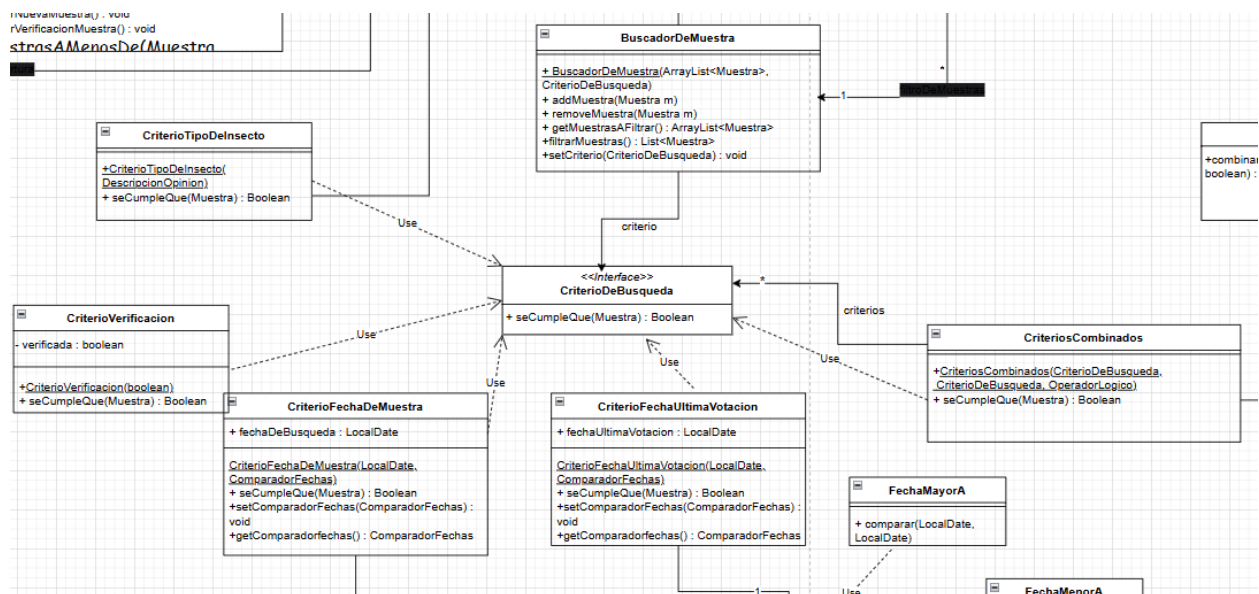
conocimiento actual. Cada nivel de conocimiento (UsuarioBásico, UsuarioExperto, UsuarioEspecialista) implementa la interfaz NivelConocimiento y establece lo que el usuario puede hacer: dar opiniones, enviar muestras o cambiar de estado.

El patrón evita el uso de condicionales dentro de la clase Usuario, lo que apoya el principio de abierto/cerrado. Esto facilita la incorporación de nuevos comportamientos sin necesidad de alterar la lógica ya existente. Además, se centraliza la lógica de transición entre estados dentro de los propios estados, lo que permite que un UsuarioBasico se transforme automáticamente en UsuarioExperto según las reglas establecidas.

### 4.3 BuscadorDeMuestra - Strategy - Composite

Estrategia para la búsqueda de muestras

- Contexto: BuscadorDeMuestra
- Estrategia: CriterioDeBusqueda (interfaz)
- Estrategias específicas: CriterioTipoDeInsecto, CriterioVerificacion, CriterioFechaDeMuestra y CriterioFechaUltimaVotacion



## B. Estrategia para comparar fechas

- Contexto: CriterioFechaUltimaVotacion, CriterioFechaDeMuestra
- Estrategia: ComparadorFechas
- Estrategias específicas: FechaMayorA, FechaMenorA FechaIgualA

## C. Estrategia de combinación lógica

- Contexto: Criterios Combinados
- Estrategia: Operador Lógico
- Estrategias concretas: And y Or

Este sistema te permite definir diferentes maneras de buscar muestras, comparar fechas o combinar criterios lógicos, sin tener que limitar esas lógicas a una sola categoría. BuscadorDeMuestra se encarga de buscar muestras utilizando un CriterioDeBusqueda. Este criterio puede ser: por insecto, por estado, por fecha, y más.

Cada CriterioDeBusqueda se puede crear utilizando otros criterios, fechas o incluso combinaciones lógicas (And, Or).

Si se desea modificar la forma en que se buscan las muestras, no es necesario tocar BuscadorDeMuestra; simplemente implementa un nuevo criterio.

Este sistema te permite definir diferentes maneras de buscar muestras, comparar fechas o combinar criterios lógicos, sin tener que limitar esas lógicas a una sola categoría. BuscadorDeMuestra se encarga de buscar muestras utilizando un CriterioDeBusqueda. Este criterio puede ser: por insecto, por estado, por fecha, y más.

Cada `CriterioDeBusqueda` se puede crear utilizando otros criterios, fechas o incluso combinaciones lógicas (And, Or).

Si deseas modificar la forma en que se buscan las muestras, no es necesario tocar `BuscadorDeMuestra`; simplemente implementa un nuevo criterio.

#### 4. 4 Observer

##### A. Observadores de zona:

- **Subject:** `ZonaDeCobertura`
- **Observer:** `IObserverZona`
- **Concreto:** `Organizacion`

`ZonaDeCobertura` notifica eventos como nueva muestra registrada o verificación de muestra.

##### Roles:

- `ZonaDeCobertura` → **Subject**.
- `IObserverZona` → **Observer**.
- `Organizacion` → **Observer concreto**.

##### B. Observadores de muestra:

- **Subject:** Muestra
- **Observer:** IObserverMuestra
- **Concreto:** ObserverMuestra

#### **Roles:**

- Muestra → **Subject**.
- IObserverMuestra → **Observer**.
- ObserverMuestra → **Observer concreto**.

#### **A. En ZonaDeCobertura:**

Las organizaciones se registran como observadoras.

Cuando se verifica una muestra o se añade una nueva en esa área, la ZonaDeCobertura avisa a todas las organizaciones.

#### **B. En Muestra:**

Se puede notificar a un ObserverMuestra cuando su estado cambia o se recibe una nueva verificación.