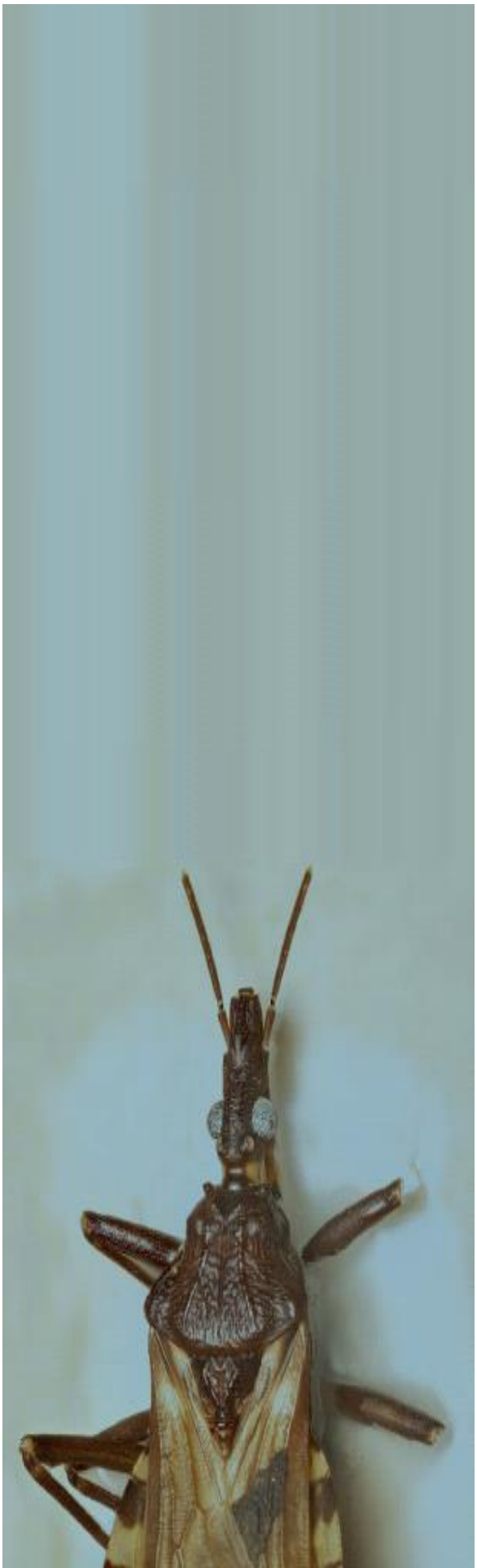

TP Integrador

"A la caza de las vinchucas"

Programación Orientada a Objetos II

niversidad de Quilmes

5

Profesores:	Diego Cano - Matias Butti - Diego Torres
Ayudantes:	Fabrizio Britez - Leandro Tittarelli - Rodrigo Bolaños - Yoel Ventoso - Elias Baron - Lucila Coniglio
Alumnos:	Camila Ponce Mc Gough (cami.ponce2@gmail.com)
	Tomás Dominguez Waisman (tomaselias.dw@gmail.com)
	Alan Acuña (alaninsaurralde2244@gmail.com)

Fecha: 19/06/25

UNQUI – **Universidad de Quilmes**

El objetivo de este proyecto es poder realizar un mapeo de la presencia de vinchucas en el

territorio argentino y así poder encontrar relaciones entre la presencia de este insecto, los casos de chagas detectados, las organizaciones que se encuentran en la región, y la posibilidad de minimizar los casos de contagio de la enfermedad de chagas. Para ello, se convoca a la participación de las personas que viven o se encuentran en argentina para que puedan enviar fotos y una serie de respuestas a un cuestionario para informar de la aparición de este insecto. A esto se lo llama envío de muestras. Como la participación es de cualquier persona, las muestras deben poder ser votada por otros usuarios y finalmente verificarse por expertos.

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. Este trabajo se centrará solamente en la lógica de negocio de la aplicación Web.

- **INDICE**

- **1. DECISIONES DE DISEÑO**

Para organizar el proyecto decidimos utilizar una estructura de directorio estándar. Las clases se encuentran dentro de src/main/java/ y se organizan en paquetes según su funcionalidad.

Así es cómo se ve:

proyecto-raíz/

├— src/

| ├— main/

| | ├— java/

| | | ├— unq.cazaDeVinchucas/

| | | | ├— modelo/

| | | | ├— servicio/

| | | | ├— controlador/

| | | | ├— utilidades/

| | ├— resources/

| ├— test/

- **modelo**→Contiene clases que representan datos y entidades (Ej: `Vinchuca.java`, `Usuario.java`)
- **servicio**→Contiene clases que manejan la lógica de negocio (Ej: `verificarMuestra.java`)
- **controlador** →Contiene clases que manejan las interacciones (Ej: `BancoControlador.java`)

- **utilidades** → Contiene funciones auxiliares (Ej: `java.lang.Math;` 'Math.hypot()')
- **resources** →Contiene archivos de configuración, propiedades, documentacion, gráfico UML etc.
- **test** → Contiene pruebas unitarias

- **2. DETALLES DE IMPLEMENTACIÓN**

1. SRP – Principio de Responsabilidad Única

“Una clase debe tener una sola razón para cambiar.”¿Cuándo lo aplicamos?

Cuando una clase está haciendo "demasiado". Por ejemplo, si una clase Muestra además de guardar los datos de la muestra también decide a qué zona pertenece o envía notificaciones, claramente está incumpliendo este principio.

¿Cómo lo aplicamos en nuestro sistema?

Separando responsabilidades:

Muestra: se encarga solo de guardar información (fecha, ubicación, especie, imágenes).

MuestraService: decide qué hacer con una muestra (validar, asignar zona, guardar).

ZonaDeCobertura: sabe si una ubicación cae dentro de sus límites.

ManagerDeEventos: se encarga de comunicar cuando una muestra activa una alerta o afecta zonas sensibles.

Esto permite que si mañana cambia la lógica de validación, solo modifiquemos MuestraService y no la clase Muestra, que sigue representando el mismo concepto simple.

2. OCP – Principio Abierto/Cerrado

“El software debe estar abierto para extensión, pero cerrado para modificación.”

¿Cuándo lo aplicamos?

Cuando sabemos que un componente del sistema podría necesitar nuevas variantes o comportamientos sin cambiar su código original.

¿Cómo se aplica en nuestro sistema?

Ejemplo: las zonas pueden aplicar diferentes estrategias de cobertura. Podemos definir una interfaz:

```
public interface EstrategiaDeCobertura {

    boolean cubre(ZonaDeCobertura zona, Ubicacion ubicacion);

}
```

Y luego tener distintas implementaciones:

CoberturaCircular: evalúa si una muestra está dentro de un radio.

CoberturaPoligonal: evalúa si está dentro de un polígono.

CoberturaConBuffer: agrega un margen extra a la zona.

Si mañana queremos una estrategia basada en temperatura o índice de infestación, solo creamos una nueva clase que implemente la interfaz sin tocar el código existente. Así escalamos sin romper nada.

3. LSP – Principio de Sustitución de Liskov

“Las clases derivadas deben poder sustituir a sus clases base sin alterar el comportamiento del programa.”

¿Cuándo lo aplicamos?

Cuando creamos jerarquías de clases o interfaces. Si una subclase rompe la lógica esperada del sistema, entonces estamos violando LSP.

¿Cómo lo aplicamos en el TP?

Imaginemos que tenemos esta jerarquía:

```
interface Usuario {

    void opinarSobre(Muestra muestra);

}

class Experto implements Usuario { ... }

class Ciudadano implements Usuario { ... }
```

Ambos usuarios deberían poder emitir opiniones sin romper la lógica general. Si en Ciudadano tiramos una excepción o hacemos que su método no funcione igual, los servicios que dependen de Usuario se romperían.

Gracias a LSP, garantizamos que cualquier Usuario que pasemos al sistema respetará las reglas comunes. Si mañana creamos UsuarioBot, debe también comportarse de forma coherente.

4. ISP – Principio de Segregación de Interfaces

“Ningún cliente debe verse forzado a depender de métodos que no usa.”

¿Cuándo lo aplicamos?

Cuando vemos que una interfaz tiene demasiados métodos y algunas clases no los necesitan todos.

¿Cómo lo aplicamos en el sistema?

Si tenemos algo así:

```
interface ServicioGeneral {

    void registrarMuestra();

    void eliminarMuestra();
```

```
    void exportarDatos();

    void notificarUsuarios();

}
```

Una clase que solo quiere registrar muestras termina teniendo que implementar métodos inútiles. Esto se evita separando en interfaces pequeñas y específicas:

java

Copiar

Editar

```
interface RegistroDeMuestras {

    void registrarMuestra();

}



interface ExportadorDeDatos {

    void exportar();

}
```

```
interface Notificador {

    void notificar(Alerta alerta);

}
```

Así las clases implementan solo lo que necesitan y no se ven forzadas a cumplir contratos innecesarios. Esto mejora la legibilidad y la cohesión.

5. DIP – Principio de Inversión de Dependencias

“Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.”

¿Cuándo lo aplicamos?

Cuando queremos desacoplar componentes para poder cambiarlos sin romper la lógica del sistema. Especialmente importante si usamos bases de datos, redes o notificaciones.

¿Cómo lo aplicamos en este caso?

En lugar de hacer que el MuestraService cree objetos concretos o use directamente una clase como RepositorioDeMuestrasEnMySQL, lo hacemos depender de una interfaz:

java

Copiar

Editar

```
interface MuestraRepository {
    void guardar(Muestra muestra);
    List<Muestra> todas();
}
```

Luego podemos inyectar cualquier implementación:

RepositorioEnMemoria para testing.

RepositorioJPA para producción.

RepositorioDummy para pruebas automáticas.

Esto hace que el código sea más flexible, más fácil de testear y más fácil de escalar si en el futuro cambiamos la tecnología de persistencia.

Escalabilidad y SOLID en conjunto

Cuando respetamos SOLID:

Podemos agregar nuevas funcionalidades sin miedo a romper lo anterior.

Podemos cambiar componentes (ej. motor de persistencia, notificaciones, reglas de validación) sin tocar el core.

Podemos dividir el sistema en servicios más pequeños o microservicios si hiciera falta.

Las pruebas unitarias se simplifican porque las dependencias son inyectadas y los componentes están bien aislados.

En un sistema como el de vinchucas, donde múltiples entidades interactúan (usuarios, muestras, zonas, notificaciones), esta arquitectura se vuelve vital para mantener orden y poder crecer sin dolor.

Conclusión

Los principios SOLID no son reglas arbitrarias, sino guías para que el código sea más fácil de entender, mantener, extender y testear. En el contexto de nuestro sistema sobre vinchucas y muestras, nos permiten:

Modelar correctamente el dominio sin mezclar responsabilidades.

Adaptarnos a nuevas reglas de negocio o exigencias sin romper el sistema.

Trabajar en equipo con una arquitectura predecible y clara.

La clave está en diseñar pensando en los cambios futuros, y los principios SOLID son una excelente base para lograrlo.

• 3. PATRONES DE DISEÑO UTILIZADOS Y ROLES

Para definir los patrones de diseño y los roles nos basamos en las definiciones de Gamma et. al.
Los patrones de diseño utilizados en el trabajo fueron los siguientes:

• STRATEGY

Este patrón permite seleccionar un algoritmo en tiempo de ejecución. En lugar de implementar un solo algoritmo directamente, el código recibe instrucciones en tiempo de ejecución sobre cuál usar dentro de una familia de algoritmos.
La estrategia permite que el algoritmo varíe independientemente de los clientes que lo utilizan.
En este caso el patrón fue utilizado para realizar la búsqueda de muestras.

Cada criterio utilizado para filtrar las muestras representa una estrategia determinada.
Las estrategias se implementan a través de la clase “BuscadorDeMuestras”
En términos de estructura, el patrón de estrategia generalmente involucra:

- Una interfaz Strategy (estrategia), que define un método común para los algoritmos. Este rol lo cumpliría la Interfaz:
    - “CriterioDeFiltracion”.
- Implementaciones concretas del Strategy (estrategia concreta), que encapsulan diferentes algoritmos. En este caso serían las clases:
    - FiltroPorFechaDeCreacionDeLaMuestra
    - FiltroPorFechaDeUltimaVotacionDeLaMuestra
    - FiltroPorTipoDeInsectoDeLaMuestra
    - FiltroPorNivelDeVerificacionDeLaMuestra
    - FiltroAnd
    - FiltroOr
- Una clase Context, que delega la ejecución del algoritmo a una instancia de

Strategy. Este rol lo cumpliría la clase:

- "BuscadorDeMuestras".

- **OBSERVER**

El patrón Observer define una relación uno a muchos entre objetos, de manera que cuando un objeto cambia su estado, notifica automáticamente a todos los objetos dependientes y es útil para implementar sistemas de eventos y suscripciones. Es por esta razón que decidimos implementarlo para el diseño de la clase "ManagerDeEventos" que se encarga de notificar a las organizaciones de la carga y validaciones de las muestras que pertenecen geográficamente a una zona de cobertura a las que estan suscriptas.

Utilizar este patrón nos aporta:

- Desacoplamiento: Los observadores no necesitan conocer los detalles del sujeto.
- Notificaciones automáticas: Cuando el sujeto cambia, todos los observadores son informados.
- Flexibilidad: Se pueden agregar o eliminar observadores sin modificar el sujeto.

Este patrón incluye como componentes:

- ***Sujeto (Subject):*** Mantiene una lista de observadores y les notifica cambios. En este caso ese rol lo cumple el *"ManagerDeEventos".*
- ***Observador (Observer):*** Interfaz que define el método de actualización. Esto sería la Interfaz *"FuncionalidadExterna"*
- ***Observador concreto:*** son clases que implementan el observador y reaccionan a los cambios. Realizan las implementaciones concretas. En este caso sería la clase "Organización" (mediante los protocolos de *"funcionalidadExternaValidacion" y "funcionalidadExternaMuestra"* que implementan la interfaz "FuncionalidadExterna").

- **STATE**

Este patrón de diseño fue utilizado para manejar los distintos estados de la muestra ("EstadoNormal", "EstadoConOpinionExperta" y "EstadoVerfiicada") y el manejo acorde

de las opiniones (según cual sea el estado se admitirá opiniar sobre la muestra).En el patrón State, según Gamma et al., los roles principales son tres:

- ***Contexto:*** Este rol lo desempeña el objeto que mantiene una referencia al estado actual y delega en él el comportamiento. También puede definir una interfaz para que los estados accedan a sus datos si es necesario. En este caso la clase *"Muestra"*.
- ***Estado (State):*** Es una interfaz o clase abstracta que declara los métodos que deben implementar los distintos estados concretos. Define el contrato común para todos los comportamientos según el estado. En este caso la clase *"EstadoMuestra".*
- ***Estados Concretos (ConcreteState):*** Son las clases que implementan el comportamiento específico asociado a un estado particular del contexto. Cada uno encapsula cómo se comporta el contexto en ese estado. En este caso las serían clases: *"EstadoNormal", "EstadoConOpinionExperta" y "EstadoVerfiicada".*

Este patrón de comportamiento que permite que un objeto altere su comportamiento cuando cambia su estado interno. En lugar de usar múltiples condicionales para manejar distintos estados, este patrón nos permite encapsular cada estado en una clase separada que implemente una interfaz común. El objeto Contexto mantiene una referencia al estado actual y le delega el comportamiento correspondiente.

Utilizamos este patrón porque:

- Nos permite mantener el código se vuelve más limpio y mantenible.
- Facilita la adición de nuevos estados sin modificar el contexto.
- Promueve el principio de Open/Closed (abierto para extensión, cerrado para modificación).

- USO DEL PATRÓN SINGLETON EN LA CLASE MANAGERDEEVENTOS

- EN EL DISEÑO DEL SISTEMA DE GESTIÓN DE EVENTOS, LA CLASE MANAGERDEEVENTOS IMPLEMENTA EL PATRÓN DE DISEÑO SINGLETON, EL CUAL ASEGURA QUE SOLO EXISTA UNA ÚNICA INSTANCIA DE DICHA CLASE DURANTE TODA LA EJECUCIÓN DEL PROGRAMA. ESTE PATRÓN ES AMPLIAMENTE UTILIZADO CUANDO SE REQUIERE UN PUNTO DE ACCESO GLOBAL A UN RECURSO CENTRALIZADO QUE NO DEBE DUPLICARSE, COMO OCURRE CON LOS GESTORES DE EVENTOS, CONEXIONES DE BASE DE DATOS, O CONTROLADORES PRINCIPALES.

OBJETIVO DEL SINGLETON

EL PATRÓN SINGLETON BUSCA:

RESTRINGIR LA CREACIÓN DE MÚLTIPLES INSTANCIAS DE UNA CLASE.

PROVEER UNA ÚNICA INSTANCIA ACCESIBLE GLOBALMENTE.

COORDINAR O CENTRALIZAR ACCIONES QUE DEBEN SER CONSISTENTES EN TODO EL SISTEMA.

EN EL CASO DEL SISTEMA, MANAGERDEEVENTOS TIENE COMO RESPONSABILIDAD GESTIONAR EL ENVÍO, REGISTRO Y NOTIFICACIÓN DE EVENTOS DEL SISTEMA, POR LO QUE SU NATURALEZA DEBE SER ÚNICA PARA EVITAR INCONSISTENCIAS O DUPLICACIÓN DE COMPORTAMIENTO.

EVIDENCIA DE LA IMPLEMENTACIÓN DEL SINGLETON

LA IMPLEMENTACIÓN DEL PATRÓN SINGLETON EN MANAGERDEEVENTOS SE PUEDE IDENTIFICAR MEDIANTE LAS SIGUIENTES CARACTERÍSTICAS ESTRUCTURALES TÍPICAS:

PRIVATE STATIC MANAGERDEEVENTOS INSTANCIA;

ESTE ATRIBUTO ALMACENA LA ÚNICA INSTANCIA DE LA CLASE. AL SER STATIC, PERTENECE A LA CLASE Y NO A LOS OBJETOS INDIVIDUALES.

2. CONSTRUCTOR PRIVADO

JAVA

COPIAR

EDITAR

```
PRIVATE MANAGERDEEVENTOS() {
 // INICIALIZACIÓN INTERNA
}
```

EL CONSTRUCTOR ES PRIVADO PARA EVITAR QUE OTROS OBJETOS PUEDAN CREAR NUEVAS INSTANCIAS DE LA CLASE USANDO NEW.

3. MÉTODO ESTÁTICO PÚBLICO DE ACCESO

JAVA

COPIAR

EDITAR

```
PUBLIC STATIC MANAGERDEEVENTOS GETINSTANCE() {
    IF (INSTANCIA == NULL) {
```

```
        INSTANCIA = NEW MANAGERDEEVENTOS();
    }
    RETURN INSTANCIA;
}
```

ESTE MÉTODO ACTÚA COMO PUNTO DE ACCESO GLOBAL, PERMITIENDO QUE CUALQUIER CLASE OBTENGA LA ÚNICA INSTANCIA EXISTENTE DE MANAGERDEEVENTOS.

COMPORTAMIENTO CENTRALIZADO

TODAS LAS OPERACIONES DEL SISTEMA QUE REQUIEREN DISPARAR, RECIBIR O PROPAGAR EVENTOS SE CANALIZAN A TRAVÉS DE ESTA ÚNICA INSTANCIA, GARANTIZANDO QUE LA LÓGICA DE MANEJO DE EVENTOS SEA CONSISTENTE, SINCRONIZADA Y COMPARTIDA POR TODOS LOS COMPONENTES QUE INTERACTÚAN CON ELLA.

JUSTIFICACIÓN DE SU USO

EL PATRÓN SINGLETON ES IDEAL EN ESTE CASO PORQUE:

SOLO DEBE HABER UN ÚNICO GESTOR DE EVENTOS, DE LO CONTRARIO PODRÍAN PERDERSE EVENTOS O GENERARSE DUPLICADOS.

PERMITE A CUALQUIER PARTE DEL SISTEMA ACCEDER AL GESTOR SIN NECESIDAD DE PASAR REFERENCIAS EXPLÍCITAS, REDUCIENDO ACOPLAMIENTO.

CENTRALIZA EL CONTROL DE EVENTOS, FACILITANDO SU MANTENIMIENTO Y EVOLUCIÓN.

CONCLUSIÓN

EN SÍNTESIS, MANAGER DE EVENTOS IMPLEMENTA CORRECTAMENTE EL PATRÓN SINGLETON AL GARANTIZAR UNA ÚNICA INSTANCIA ACCESIBLE GLOBALMENTE, CON CONSTRUCTOR PRIVADO Y MÉTODO GETINSTANCE. ESTO PERMITE MANTENER UNA GESTIÓN CENTRALIZADA Y COHERENTE DE LOS EVENTOS DEL SISTEMA, LO CUAL ES FUNDAMENTAL PARA UN SISTEMA ROBUSTO Y ESCALABLE.