

|  |  |  |  |
| --- | --- | --- | --- |
|  | Trabajo Práctico Integrador | | |
|  | **“A la caza de las vinchucas”** | | |
|  | **Programación Orientada a Objetos II** | | |
|  | Universidad de Quilmes | | |
|  | **2025** | | |
|  | 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**

Introducción

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

Como nos organizamos

hablar del SOLID y de la posibilidad de escalar la aplicacion

# 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:

## COMPOSITE

El patrón Composite fue utilizado para la clase “ZonaDeCobertura”. Este patrón es útil cuando se necesita representar jerarquías parte-todo, donde los objetos pueden ser tratados de manera recursiva como componentes de una estructura más grande y facilita la recursividad, ya que los objetos compuestos pueden contener otros objetos compuestos. En este caso, la “ZonaDeCobertura”, puede incluir múltiples “regionesGeograficas” que sean a su vez “ZonaDeCobertura”.

[REVISAR]

En cuanto a los roles el patrón Composite incluye:

* Componente (Component): Interfaz común para todos los objetos.
* Hoja (Leaf): Representa objetos individuales sin hijos.
* Compuesto (Composite): Contiene otros objetos y delega operaciones a sus hijos.

## 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).