Trabajo Práctico Integrador
Programación con Objetos II

– 2025 –

1er Semestre

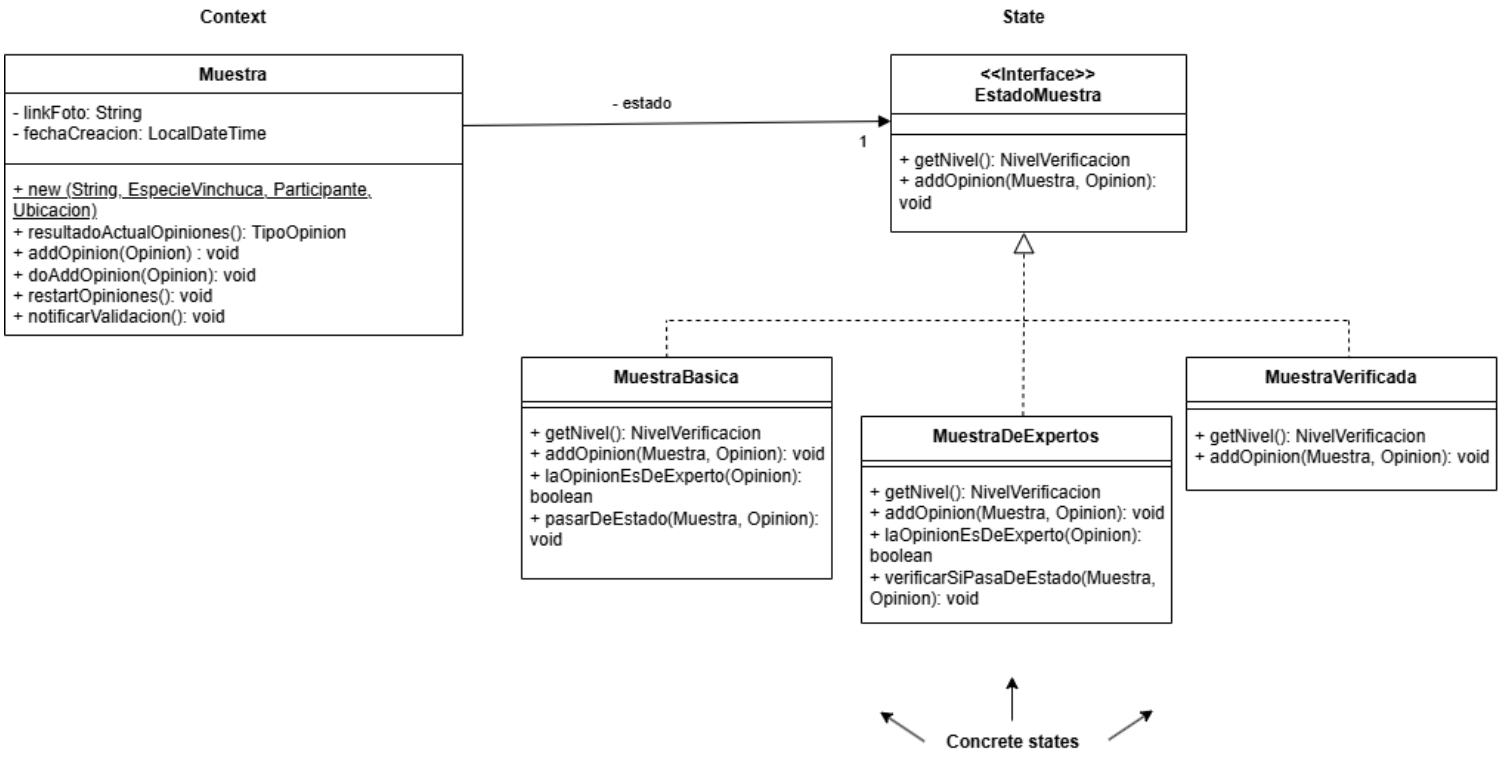
“A la caza de las vinchucas”

- Estudiantes:
  - Di Franco Leonardo: leonardodanieldifranco@gmail.com
  - Fabeiro Leandro: *l.fabeiro07@gmail.com*
  - Vildoza Lautaro: *lautarovildoza1@gmail.com*

- **Decisiones de diseño:**

MUESTRA

Las muestras tienen tres estados, según la cantidad y el tipo de opiniones que reciben. Esto nos lleva a utilizar el patrón de diseño **State**:



Cada muestra cuenta con un colaborador de tipo EstadoMuestra, que se encarga de permitir o impedir el agregado de nuevas opiniones y también del propio cambio de estado.

De esta forma, el estado de la muestra varía de forma dinámica a medida que recibe opiniones siendo la Muestra siempre la misma.

PARTICIPANTE

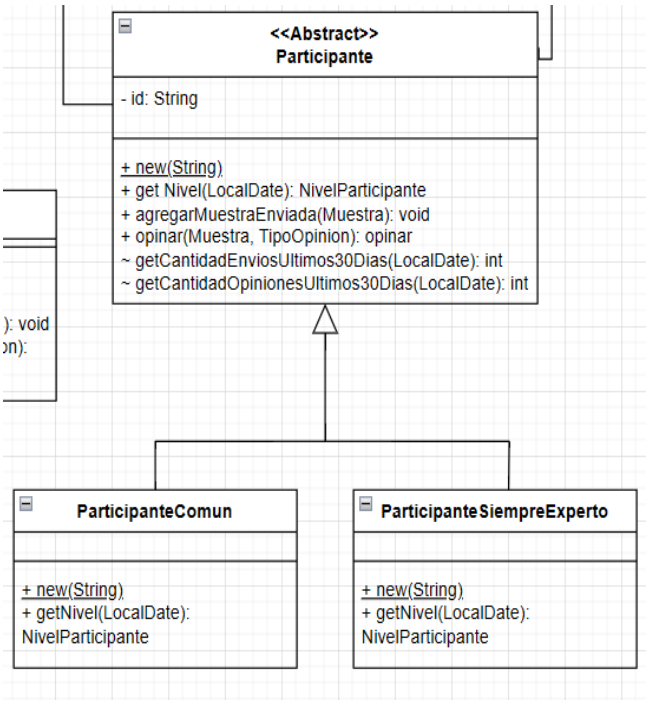
En nuestro sistema de "Caza de Vinchucas", existen diferentes tipos de participantes:

ParticipanteComun: Su nivel (BÁSICO o EXPERTO) se determina por su actividad (cantidad de envíos y opiniones en los últimos 30 días).

ParticipanteSiempreExperto: Siempre mantiene el nivel EXPERTO, independientemente de su actividad.

A pesar de esta diferencia en cómo se calcula el nivel, ambos tipos de participantes comparten muchas características y comportamientos comunes (tienen un ID, envían muestras, emiten opiniones, y guardan un historial de estas). La forma de interactuar con ellos debe ser uniforme para el resto del sistema.

Para modelar esta relación de "es un" (ParticipanteComun es un Participante, ParticipanteSiempreExperto es un Participante) y manejar las similitudes y diferencias de forma eficiente, se ha utilizado la Herencia, a través de una clase base abstracta Participante.

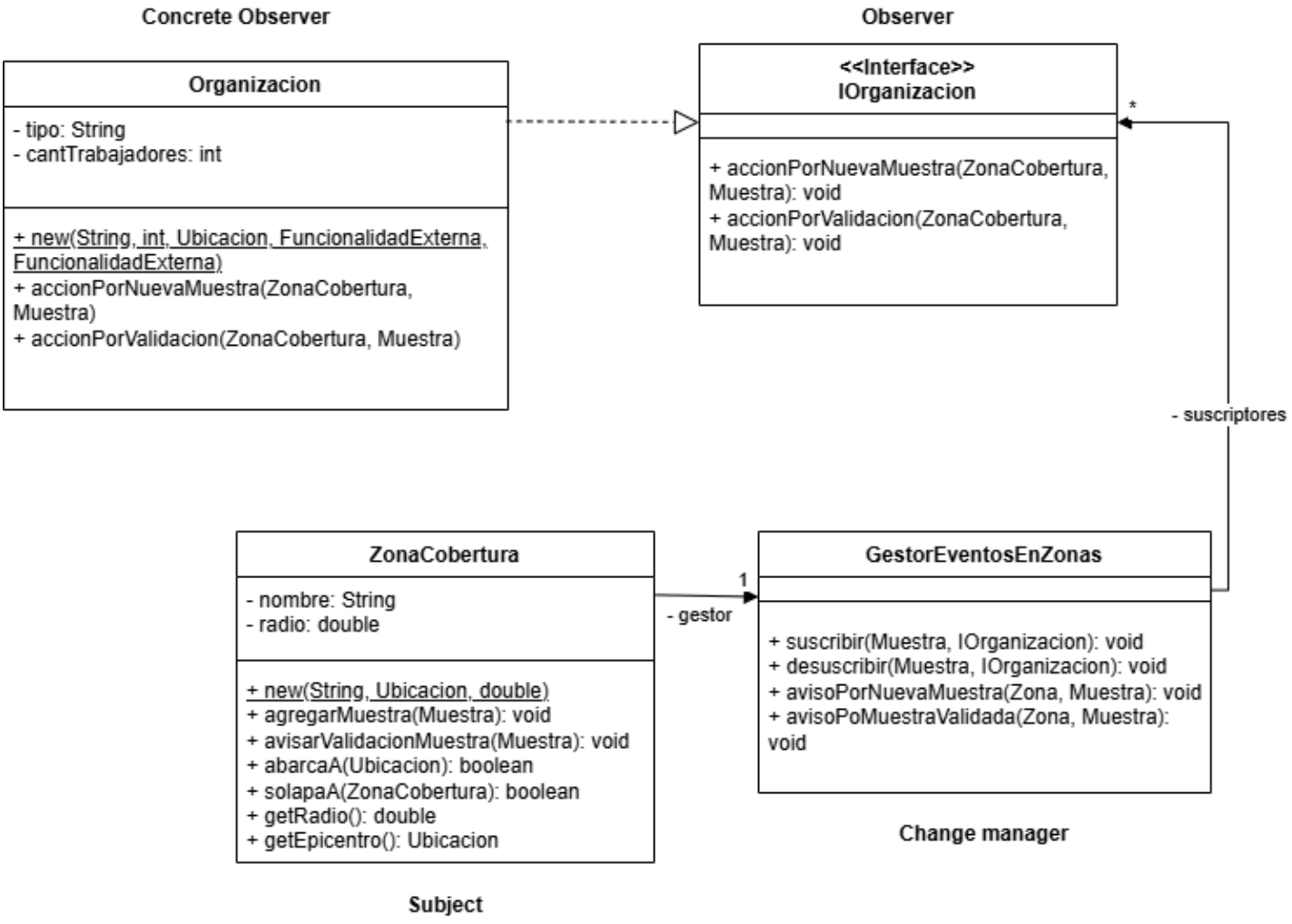


Clase Abstracta Participante: Contiene toda la lógica y los datos que son comunes a todos los tipos de participantes (como id, muestrasEnviadas, opinionesEmitidas, y los métodos para agregar muestras, opinar, y calcular cantidades de envíos/opiniones recientes). Lo más importante es que declara el método getNivel(LocalDate fechaActual) como abstracto, forzando a las subclases a proporcionar su propia implementación para esta parte variable del comportamiento.

Subclases Concretas (ParticipanteComun, ParticipanteSiempreExperto): Extienden Participante y cada una implementa el método getNivel() de la forma específica que le corresponde.

AVISO A ORGANIZACIONES

Como era necesario que las organizaciones se suscriban a distintas zonas de cobertura, y que realicen acciones en base a ciertos eventos ocurridos en esas zonas, decidimos implementar el patrón de diseño **Observer**:



Cada vez que se agrega una nueva muestra, se realiza un aviso a las organizaciones suscritas a la o las zonas a la que pertenezca esta muestra. Lo mismo para las validaciones de muestras.

Decidimos implementar un gestor de eventos, para no asignarle funciones distintas a las que realiza y así cumplir con el principio SOLID ‘Single-Responsibility’.

BÚSQUEDA DE MUESTRAS

El enunciado del trabajo establece claramente la necesidad de buscar muestras utilizando múltiples criterios (Fecha de creación, Fecha de última votación, Tipo de insecto, Nivel de verificación). Lo más importante es que estos criterios "pueden ser combinados de diversas formas con operadores lógicos OR y AND, para formar expresiones complejas". Se proporcionan ejemplos como:

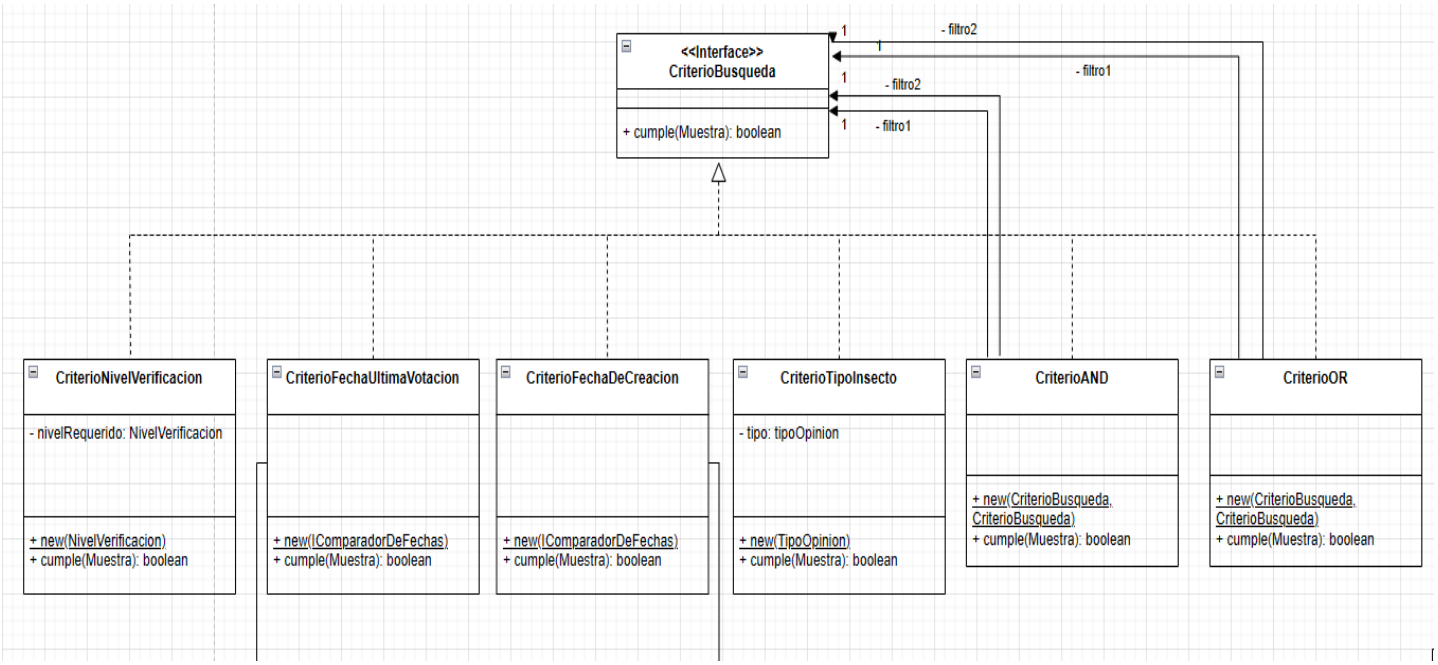
Fecha de la última votación > '20/04/2019'

Nivel de validación = 'verificada' AND Fecha de la última votación > '20/04/2019'

Tipo de insecto detectado = 'Vinchuca' AND (Nivel de validación = 'verificada' OR Fecha de la última votación > '20/04/2019')

Una solución ingenua podría implicar una gran cantidad de condicionales if-else anidados en el método de búsqueda para cada combinación posible de filtros, lo que llevaría a un código difícil de leer, mantener y extender.

Para resolver este problema, se ha utilizado el patrón de diseño **Composite**.



Componente (CriterioBusqueda): Se define la interfaz CriterioBusqueda con un método boolean cumple(Muestra muestra). Este es el contrato común para todos los nodos del árbol.

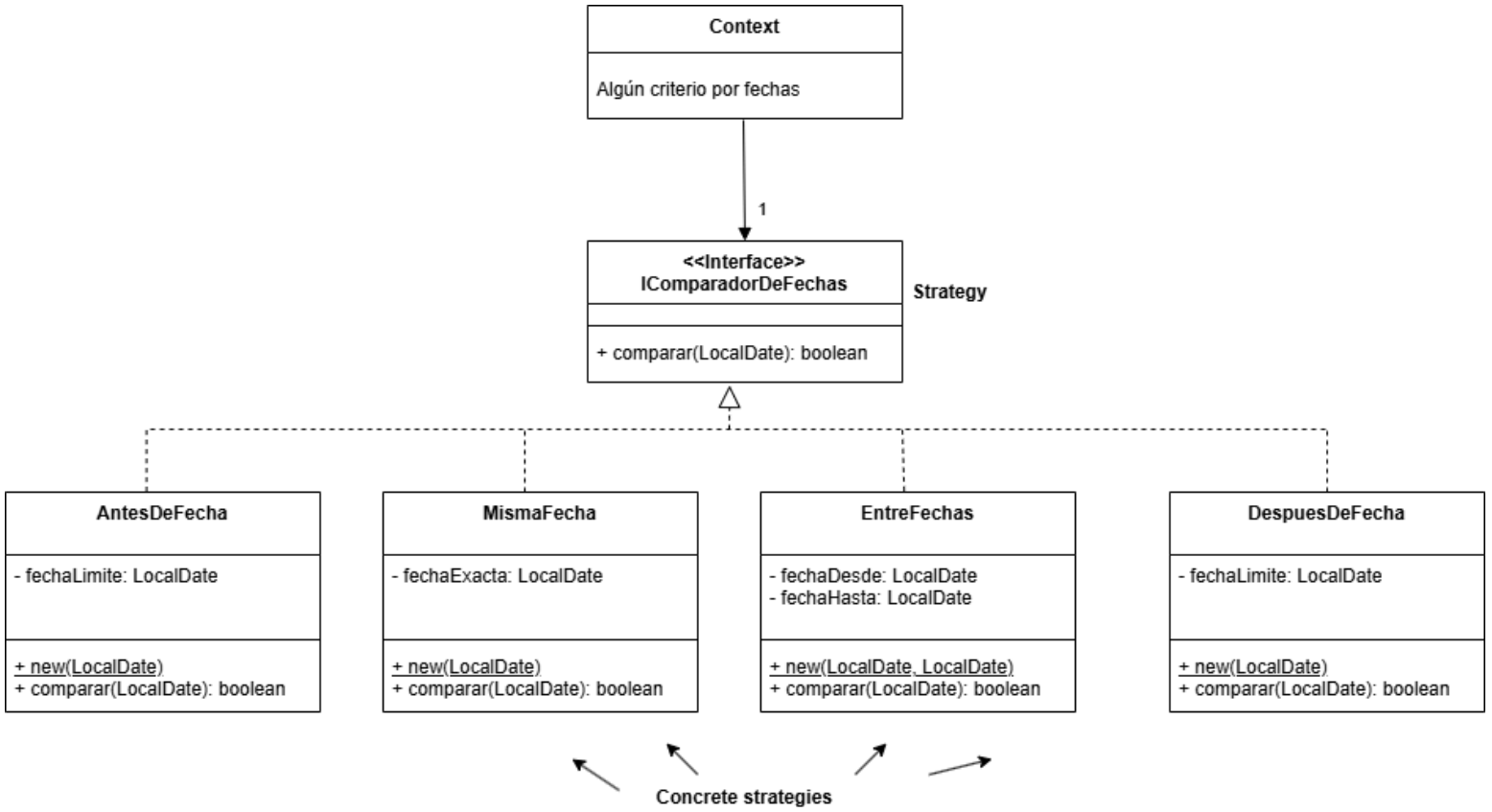
Hojas (Criterios Atómicos): Cada criterio de búsqueda individual (CriterioFechaCreacion, CriterioFechaUltimaVotacion, CriterioNivelVerificacion, CriterioTipoInsecto) implementa la interfaz CriterioBusqueda y evalúa una condición específica sobre la Muestra.

Compuestos (Criterios Compuestos): Las clases CriterioAND y CriterioOR también implementan CriterioBusqueda y contienen referencias a uno o más objetos CriterioBusqueda (que pueden ser hojas o, a su vez, otros compuestos). Su método cumple() delega la evaluación a sus componentes hijos y combina los resultados con lógica booleana.

El patrón Composite es la elección ideal porque permite representar jerarquías de objetos complejos de manera flexible y polimórfica, encapsulando la lógica de combinación de criterios y facilitando la construcción de consultas dinámicas sin acoplar el código cliente a la complejidad de las expresiones de búsqueda.

COMPARADOR DE FECHAS

A la hora del filtrado por fecha, era necesario una forma de poder asignar el operador que se quería usar en la comparación. Por esto, aplicamos el patrón **Strategy** a una parte del Composite:



De esta manera, pueden buscarse fechas por distintos criterios, pero siempre haciendo uso de un ComparadorDeFechas, dándole flexibilidad a la implementación.

OTRAS DECISIONES

Método opinar() en Participante: Consideramos útil que el usuario guarde el historial de sus opiniones para mejorar la eficiencia en el cálculo de nivel de conocimiento. De esa manera, se ahorra el paso de tener que ir muestra por muestra buscando cuáles pertenecen al participante en cuestión. Es por esto que se hizo necesario el método opinar, como forma de guardar cada opinión del usuario.

Aviso por muestra validada: Se podría haber sumado implementado el patrón Observer nuevamente, donde el observable es la muestra, y el observer la zona de cobertura. Sin embargo, para no sobrecargar una implementación que ya contaba con muchas clases, decidimos delegar la responsabilidad del aviso al estado de las muestras; específicamente, al método donde se pasa a MuestraVerificada.

TipoOpinion.NINGUNA: Decidimos usar esta opción del enumerativo cuando el resultado actual es un empate, para no sumar otra opción como por ejemplo NO_DEFINIDO. Así, se impide la posibilidad de que los participantes dejararan opiniones de tipo no definido.

- **Conclusión:**

El trabajo integrador nos ayudó a implementar todos los conceptos vistos en la materia: patrones de diseño, principios SOLID, uso de Tests Unitarios y Tests Double. También refrescamos conceptos aprendidos en materias anteriores.
Además reforzamos el trabajo en equipo y la capacidad de delegar tareas, discutir e interactuar en grupo.