Universidad Nacional de Quilmes
Programación Orientada a Objetos II
A la Caza de las Vinchucas

Integrantes:

- **Rosales, Braian**
- **Sandoval, Elías**
- **Spizzamiglio, Pablo**

Evolución del Diseño

En un principio, no consideramos modelar la AplicacionWeb pero al momento de diseñar la clase Usuario sospechamos que nos faltaba un lugar donde guardar todas las muestras enviadas del sistema para luego ser utilizadas por el resto de los objetos. Al implementar el buscador, confirmamos nuestras sospechas y decidimos crear una interfaz llamada Aplicacion que define el protocolo de una aplicación para luego implementar dicho protocolo mediante la clase AplicacionWeb.

De igual manera sucedió con la clase Foto, que primero la modelamos como un String, pero luego la vimos como una clase con un comportamiento que podría ser extensible a futuro.

Decidimos utilizar tipos enumerativos (Enum) para identificar los diferentes valores utilizados en la clasificación de una muestra, en el cálculo de nivel de verificación de una muestra y para el criterio de búsqueda por nivel de verificación y así no tener errores de inconsistencia en la información que representa las propiedades antes mencionadas.

Para el cálculo de solapamiento de zonas decidimos aprovechar que sus epicentros sabían calcular la distancia entre sí para luego aplicar una fórmula matemática que nos garantiza que dos circunferencias se solapan si la distancia entre sus epicentros es menor a la suma de sus radios.

La muestra precisa ser creada con una opinión que se pasa como parámetro al momento de creación para poder asegurar que la opinión de quién envía la muestra es considerada al momento de calcular el nivel de verificación o el resultado actual de la misma. Asimismo, nos ayudó a asegurarnos que la muestra y la opinión contienen la misma fecha de creación y la misma calificación.

La suscripción de Organizaciones a eventos de una Zona se realiza enviándole el mensaje *suscribir* a una Zona y como argumento la Organización interesada en recibir notificaciones de la misma. Lo mismo ocurre para el caso de dejar de recibir notificaciones, la única diferencia es que a la Zona se le envía el mensaje *desuscribir* con la Organización como argumento.

Patrones utilizados

Al modelar la interacción entre Zona y Organización nos dimos cuenta que las organizaciones precisaban estar al corriente de qué sucedía dentro de una zona de interés. Esto nos llevó a un diseño que violaba el principio *Open-Closed* de *SOLID* ya que si la Organización sufría un cambio nos podría obligar a realizar otro cambio en cascada en la Zona. Para ello decidimos utilizar el patrón *Observer* y así poder abstraer el protocolo de mensajes que deberia respetar una Organización para que la Zona pueda enviarle satisfactoriamente una notificación cada vez que suceda algún evento de interés para la misma.

En nuestro caso particular, los roles son representados por:

- Subject:
    - No está presente
- ConcreteSubject:
    - Zona
- Observer:
    - Suscriptor
- ConcreteObserver:
    - Organización

Al modelar el Usuario detectamos que cada estado presentaba un comportamiento diferente que se encargaba de decidir cómo evolucionaba el nivel del usuario dependiendo de la cantidad de muestras y opiniones enviadas. Para resolver éste problema utilizamos el patrón *State* para poder modelar los cambios de estado y así delegar las responsabilidades de cada estado en diferentes clases. Esto nos permitió agregar un estado Especialista (para los usuario validados fuera del sistema) que evita que un Usuario evolucione y así bloquea el cambio de nivel.
En nuestro caso particular, los roles son representados por:

- Context:
    - Usuario
- State:
    - Nivel
- ConcreteState:
    - Basico
    - Experto
    - Especialista

Un primer modelo de la Muestra puso en evidencia que la misma variaba su comportamiento dependiendo del estado en el que se encontraba en ese momento. Para resolver éste problema decidimos hacer uso nuevamente del patrón *State*. Este patrón nos ayudó a detectar diferentes estados de la muestra y delegar responsabilidad cuando se presentara la situación adecuada.
En nuestro caso particular, los roles son representados por:

- Context:
    - Muestra
- State:
    - MuestraEstado
- ConcreteState:
    - MuestraVotacionAbierta
    - MuestraVotacionExperto
    - MuestraVerificada

Una vez implementado el patrón *State* para la Muestra, nos encontramos con que cada estado tenían seguía una estructura de comportamiento similar pero que variaba levemente en puntos específicos. Para evitar este problema, decidimos hacer uso del patrón de diseño *Template Method.* Este patrón nos ayudó a definir la estructura de un algoritmo (el template method en sí) y dejar espacio para implementar de manera diferente los pasos que variaban en cada estado mediante la definición de operaciones primitivas y hook methods.
En nuestro caso particular, los roles son representados por:

- AbstractClass:
    - MuestraEstado
    - Operaciones Primitivas:
        - puedeCambiarEstado
    - Hook methods:
        - calcularVotosPorOpinion
        - calcularVotoMaximal
        - recolectarOpinionesEmpatadas
- ConcreteClass:
    - MuestraVotacionAbierta
    - MuestraVotacionExperto
    - MuestraVerificada

Se hizo uso del patrón composite para resolver la búsqueda de muestras. Lo que se buscaba era la aprovechar la estructura arbórea de dicho patrón para la personalización de las búsquedas con la opción de que puedan aparecer, como resultado, muestras que cumplieran con todos los criterios u opciones de un mismo criterio.
En nuestro caso particular, los roles son representados por:

- Component:
    - CriterioDeBusqueda
- Leaf:
    - CriterioFechaCreacion
    - CriterioNivel
    - CriterioTipoInsecto
    - CriterioFechaValidacion
- Composite:
    - CriterioCompuesto

Organización de las Tareas

El diseño inicial fue realizado de manera conjunto por todos los integrantes del grupo. En la primera reunión, luego de haber alcanzado un prototipo inicial, procedimos a la división de

tareas, para la cual cada integrante se encargó además de los colaboradores directos, tests y diseño del UML:

- Rosales, Braian: se le asignó la tarea de refinar e implementar la solución en Java del modelo de Usuario con sus estados (Nivel, Basico, Experto y Especialista).
- Sandoval, Elías: se le asignó la tarea de refinar e implementar la solución en Java del modelo de los CriterioDeBusqueda (CriterioFechaCreacion, CriterioNivel, CriterioTipoInsecto, CriterioFechaValidacion y CriterioCompuesto).
- Spizzamiglio, Pablo: se le asignó la tarea de refinar e implementar la solución en Java del modelo de la Muestra con sus estados (MuestraEstado, MuestraVotacionAbierta, MuestraVotacionExperto y MuestraVerificada), la Organizacion y la Zona.