Programación Orientada a Objetos 2

A la caza de las vinchucas

Trabajo Práctico Integrador

Integrantes:
Cristofanetti, Lucio Ariel. - luciocristofanetti@gmail.com
Martinez, Eduardo Martín - mm7735608@gmail.com
Martinez, Quimey - quimeyadonai@gmail.com

Patrones de Diseño:

STATE: Se implementó este patrón para determinar el estado de verificación de las muestras. La Muestra toma el rol de ***Contexto*** y el ***Estado*** de Verificación puede ser No Verificada (en subclases OpinionBasicos y OpinionExpertos) y Verificada.

Cuando a una muestra se le pide el Resultado Actual, ésta envía la petición a su estado de verificación para que la resuelva y dé la respuesta correspondiente. Cuando un usuario pregunta si puede opinar en una muestra, esta consulta se le hace al estado de la muestra y es este último quien determina si corresponde o no.

El mensaje verificar() de las subclases de Verificación, encapsula la lógica con la cual se pasa de un estado al siguiente. Cada vez que se agrega una opinión en la muestra, la muestra solicita al estado verificar con el fin de cambiar el estado en caso de ser pertinente.

COMPOSITE: Para el buscador de la web se utilizó este patrón con el fin de poder utilizar y combinar las diferentes instrucciones de búsqueda solicitadas. La clase Filtro toma el rol de ***Componente,*** la cual cuenta con dos subclases principales: Simple y Compuesto**.** Todas las subclases de Simple son lo que en el patrón se conoce como ***Hoja*****,** mientras que las subclases de ***Compuesto*** refieren a su rol homónimo. Al momento de crear un Compuesto, se deben pasar por parametro dos instancias de un Componente (ya sean hojas o compuestos). Cuando una hoja recibe el mensaje buscar(List<Muestra>), resuelve de acuerdo a la lógica propia de su instancia, mientras que cuando un Compuesto recibe el mismo mensaje, este deriva la petición a sus componentes y filtra según sea And u Or.

ADAPTER: Si bien la aplicación de este patrón no es literal a la que propone el libro de Gamma et. al., se basó en la implementación del adapter para combinar TipoOpinion con TipoVinchuca. Cuando un usuario crea una muestra, solo puede indicar un TipoVinchuca, más cuando quiere opinar una muestra existen otras alternativas como alguna chinche o imagen poco clara. La ***Interfaz*** IOpinable (que cumple ese rol) es implementada por los dos tipos mencionados previamente. No existen clases con un rol de ***Adaptador*** como tal, ya que tanto TipoOpinion como TipoVinchuca son polimórficas entre si y cumplen ya con el protocolo de la interfaz.

OBSERVER: Existe una **cadena de observadores** en nuestra solución del TP. Las Organizaciones ***observan*** a las ZonaCobertura. A su vez las Zonas de cobertura, ***observan*** (con el GestorDeUbicaciones como **gestor de eventos** de por medio) a dos sucesos en la Web: la creación de Muestras y su verificación. Cuando un Usuario crea una Muestra, la Web la agrega a su lista de muestras. La Web por su parte, ***observa*** a las Muestras, que pueden ser Validadas en determinado momento. Cuando el estado de una Muestra cambia a Verificada, envía una notificación a la

Web. La Web recibe estas notificaciones y en consecuencia notifica ambos eventos a un gestor de ubicaciones encargado de gestionar las zonas y las ubicaciones (con mensajes independientes). Estos mensajes llegan al GestorDeUbicaciones el cual cuenta con un List<ZonaCobertura> y un Map<Ubicacion,ZonaCobertura>. Cuando un Usuario crea una Muestra, envía la Ubicación al Gestor (por medio de la Web) y el este último asocia a todas las zonas de cobertura que la contengan en su area. De esta manera, evitamos notificar a zonas de eventos que no les interesa. Las zonas de cobertura se pueden registrar y quitar del gestor para recibir (o dejar de recibir) las notificaciones que el Gestor envía. Finalmente, cada zona notifica a las Organizaciones suscritas, las cuales también pueden suscribirse y desuscribirse de cada zona.

STRATEGY: Las funcionalidades externas que implementan la interfaz dada en la consigna del TP se utilizan para aplicar este patrón de diseño. Las Organizaciones son el **Contexto** y cuentan con dos colaboradores internos, carga y validación (ambos objetos que implementan FuncionalidadExterna), los cuales toman el rol de estrategias Concretas y pueden ser usados por cualquier otra organización o clase que implemente dicha interfaz. La lógica de dicho algoritmo puede ser cambiada desde afuera de la Organización con una clase Cliente llamando a los métodos set de ambos colaboradores que implementan FuncionalidadExterna en la clase.

Requerimientos del TP:

- **De las Muestras:**

"En principio de cada muestra se debe poder responder cuál es la especie de vinchuca que han fotografiado (Infestans, Sordida, Guasayana), la foto de la vinchuca, la ubicación y la identificación de la persona que recolectó esa muestra"

Muestra.getVinchuca() devuelve la especie de vinchuca fotografiada con un enum IOpinable

Muestra.getFoto() devuelve un objeto Foto (creado meramente para darle una representación en el modelo)

Muestra.getUbicacion() devuelve un objeto Ubicacion con los datos de localización geográfica.

Muestra.getPersona() devuelve un objeto Usuario que identifica a la persona que recolectó la muestra.

"Los usuarios básicos pueden opinar sobre la foto y los expertos también con el fin de verificarla"

Usuario.opinar(Muestra, IOpinable) consulta si el usuario puede opinar sobre la muestra dada y en caso afirmativo, agrega en la muestra un objeto Opinion que contiene el IOpinable dado.

"En todo momento se puede pedir a una muestra su resultado actual, dado por la opinión que tenga mayoría de votos."

Muestra.resultadoActual() devuelve un String con el valor correspondiente a la opinion mayoritaria de la muestra. Este mensaje obtiene el string de la Verificacion de la muestra con el mensaje Verificacion.resultadoActual(), donde se encapsula la logica para determinar el resultado.

"Es importante obtener el historial de votaciones realizado"

Muestra.getOpiniones() devuelve una lista con todas las opiniones que se hicieron en la muestra.

- **De la verificación de las muestras y las opiniones:**

"La opinión de quien subió la foto cuenta"

Al momento de crear la muestra el constructor de Muestra(Web, Usuario, TipoVinchuca, Foto, Ubicacion) genera una Opinion con el TipoVinchuca dado y la almacena en una lista de opiniones.

"La opinión puede ser: Vinchuca (y en tal caso la especie), Chinche Foliada, Phtia-Chinche, Ninguna, Imagen poco clara."

Cuando se crea una opinión el constructor Opinion(Nivel, IOpinable) recibe el nivel del Usuario que la crea y el valor de IOpinable el cual se tipa con dicha interfaz que implementan los enums TipoVinchuca y TipoOpinion

Si una foto subida como Vinchuca Infestans pero luego opinada por dos usuarios básicos como Imagen "poco clara", el resultado actual es "Imagen poco clara". Si luego 3 usuarios básicos nuevos opinan que es una "Chince Foliada", el resultado actual será "Chince Foliada"

Cuando se crea una muestra, se genera un objeto OpinionBasicos (subclase de NoVerificada, subclase de Verificacion) que contempla las opiniones de todos los usuarios para evaluar el resultado actual.

"Cuando opina un experto, los usuarios básicos ya no pueden opinar. Solo pueden opinar expertos."

Cada vez que se ejecuta Usuario.opinar(Muestra, IOpinable) y el usuario puede opinar, esto ejecuta en la Muestra.agregarOpinion(Opinion). En consecuencia, la muestra almacena la opinión y hace que su estado de objeto Verificacion verifique las opiniones. OpinionBasicos.verificar() revisa el nivel de los usuarios que dejaron opinion, y si hay una opinion de un experto, cambia el estado de la muestra a un nuevo estado OpinionExpertos. Para saber si el usuario puede opinar, el estado de la muestra determina qué niveles pueden votar (como lo indican sus nombres)

"Para que la muestra quede verificada, deben coincidir dos expertos en su opinión"

OpinionExpertos.verificar() consulta las opiniones de los expertos y si dos coinciden, cambia el estado de la Muestra a un nuevo objeto Verificada.

"Nadie puede opinar sobre muestras verificadas"

El mensaje de Muestra.puedeOpinar() consulta a su estado y recibe un booleano. Verificada.puedeVotar() devuelve false.

"A partir del momento que vota un experto solo cuentan sus opiniones para responder al resultado actual."

OpinionExpertos.resultadoActual() no tiene en cuenta las opiniones de los usuarios con nivel basico.

"En caso de empate temporal el resultado actual es no definido. Es importante obtener el historial de votaciones realizado."

NoVerificada.resultadoActual() devuelve un String "no definido" en caso de que haya un empate de opiniones.

- **De los usuarios y sus niveles:**

"Los participantes pueden realizar dos actividades básicas. Enviar una muestra y opinar sobre las muestras que envió otra persona."

Usuario.enviarMuestra(TipoVinchuca,Double,Double) genera una muestra con el TipoVinchuca y la Ubicacion con los datos de Latitud y Longitud geografica dados, y la envía al objeto Web donde es almacenada en una lista.

Usuario.opinar(Muestra, IOpinable) consulta si el usuario puede opinar y en caso afirmativo: genera un objeto Opinion, se lo envia a la muestra dada con

Muestra.agregarOpinion(Opinion) y almacena la opinion generada en una lista interna.

"Nunca pueden opinar sobre muestras que hayan enviado ellos mismos, como así tampoco opinar más de una vez para una muestra".

Cuando se crea la muestra, se genera una opinion con el TipoVinchuca del usuario autor, y la misma se almacena internamente. Esto esta *hardcodeado* en el constructor de Muestra. Cada vez que un usuario quiere opinar, consulta a la muestra si puede opinar. Muestra.puedeOpinar(Usuario) coteja que el usuario no haya votado previamente, revisando la lista de opiniones que contiene y comparando el id del usuario con el de cada opinión. Esto restringe los dos escenarios planteados.

"El envío y la verificación de muestras les va dando diferentes niveles en el conocimiento. El sistema tiene una regla de promociones que funciona de la siguiente manera: 1) Básico: para aquellas personas que recién comienzan a participar. Un participante nuevo posee nivel básico"

Cuando se crea un usuario nuevo, el constructor de Usuario setea el valor de nivel como básico por defecto

"2) Experto: son personas que durante los últimos 30 días desde la fecha actual han realizado más de 10 envíos y más de 20 revisiones. Los algoritmos para subir o bajar de categoría de experto se pueden ejecutar en cualquier momento y considerarán siempre los 30 días anteriores al día de la fecha en que se lancen."

Web.actualizarNivel(List<Usuarios>) recorre la lista de Usuarios dada por parámetro y envía a cada usuario el mensaje actualizarNivel(). Este mensaje setea el nivel correspondiente según la lógica establecida, invocando mensajes propios de la clase Usuario.

"Existen algunos usuarios que poseen conocimiento validado en forma externa. [...] Estos usuarios siempre son expertos, sin importar el tiempo que llevan participando.

Usuario.convertirEnEspecialista() cambia el estado del usuario a especialista. Web.convertirEnEspecialista(List<Usuarios>) permite realizar esta accion para todos los usuarios de la lista dada por parámetro. La implementación de Usuario.actualizarNivel() contempla que si el usuario es Especialista, no modifica su nivel.

- **De las ubicaciones:**

“De una ubicación se quiere saber: La distancia entre dos ubicaciones”

Ubicacion.distanciaCon(Ubicacion) devuelve un double que indica o representa la distancia en kilómetros entre las dos ubicaciones.

“Conocer, a partir de una lista de ubicaciones, aquellas que se encuentran a menos de x metros, o kilómetros”

Ubicacion.ubicacionesCercanas(List<Ubicacion>,Double) devuelve una lista de ubicaciones en donde sólo figuran aquellas de la lista dada que se encuentran a una distancia menor que la dada. Para realizar ésto calcula las distancias que hay entre ambas ubicaciones y filtra por las que son menores o iguales a las dadas.

“Dado una muestra, conocer todas las muestras obtenidas a menos de x metros o kilómetros”

Muestra.muestrasCercanas(Web, double) devuelve una lista de muestras que se encuentran a una distancia menor a la dada por parámetro. La web se da por parámetro puesto que es el objeto que almacena todas las muestras.

- **De las Organizaciones y las Zonas de Cobertura:**

“De cada zona de cobertura se desea conocer el nombre que lo identifica, el epicentro y la distancia del mismo en kilómetros.”

ZonaCobertura.getNombre() devuelve un String con el nombre de la zona.
ZonaCobertura.getEpicentro() devuelve una Ubicacion.
ZonaCobertura.getRadio() devuelve un double que indica la distancia del mismo.

“También es importante que se conozcan en todo momento las muestras que se han reportado en cada zona de cobertura.”

Para ello en zonaCobertura utiliza un método, getMuestras(), que lista todas las muestras que están cubiertas por esa zona.

En caso de no tener registrada la Ubicacion, el GestorDeUbicaciones devuelve la misma Ubicacion que recibió y consulta a todas las ZonaCobertura

“saber cuales son las zonas que la solapan.”

ZonaCobertura.intersecciones(List<ZonaCobertura>) devuelve una lista con las zonas de cobertura a partir de la lista dada donde solo figuran aquellas que se solapan.

“Lo que les interesa a las organizaciones es poder enterarse de la carga y validaciones de las muestras que pertenecen geográficamente a una zona de cobertura. Es por ello que las organizaciones deciden registrarse a estos eventos en diferentes zonas. Una organización puede registrarse a todas las zonas que les parezcan importantes, así como también dejar de estar registradas en las mismas. “

ZonaCobertura.suscribeOrganizacion(Organizacion) regista la organización dada por parámetro y ZonaCobertura.unsuscribeOrganizacion(Organizacion) quita la Organización dada por parámetro de la lista de organizaciones registradas en esa Zona. Este registro hará que las Organizaciones puedan recibir notificaciones de diversos eventos en las zonas que les interesan.

“Una organización se configura con una única funcionalidad externa para actuar cuando se carga una nueva muestra y una para cuando se valida alguna muestra, aunque en su vida útil la funcionalidad pueden cambiar o usar la misma funcionalidad para ambas cosas.”

La clase Organización almacena en dos atributos diferentes la funcionalidad externa para la carga y para la validación de muestras respectivamente. Estas se establecen en su constructor. Si utiliza la misma para ambas cosas, se pasa la misma funcionalidad externa en ambos parámetros.

Para cambiar la funcionalidad externa se puede cambiar utilizando los setters de carga y validación para setear nuevas funcionalidades externas.

- **De la Busqueda:**

“Se quieren buscar muestras por los siguientes criterios: ● Fecha de creación de la muestra. ● Fecha de la última votación. ● Tipo de insecto detectado en la muestra. ● Nivel de verificación (votada o verificada)”

Se creó una interfaz Filtro con el método buscar. Esta es implementada por una clase abstracta llamada Simple, la cual tiene una subclase para cada parámetro de Búsqueda. El constructor de todas las subclases de filtro, recibe a la web por parámetro. Las subclases son FechaDeCreacion, FechaUltimaVotacion, TipoDeInsecto, y PorVerificacion

FechaDeCreacion tiene 3 subclases que son CreadoAntesDe, CreadoEn y CreadoDespuesDe. FechaUltimaVotacion tiene 3 subclases que son UltimaVotacionAntesDe, UltimaVotacionEn y UltimaVotacionDespuesDe. El constructor de todas estas clases recibe por parámetro un dato tipo LocalDate.

TipoDeInsecto recibe en su constructor un objeto que implementa la interfaz IOpinable. Por verificacion tiene dos subclases SoloVotadas y SoloVerificadas

"Estos criterios pueden ser combinados de diversas formas con operadores lógicos OR y AND, para formar expresiones complejas"

Compuesto es una clase abstracta que implementa la interfaz Filtro, la cual tiene dos subclases: And y Or que cumplen con los operadores que representan. Cada compuesto recibe por su constructor dos instancias de Filtro independientes.

Ejemplos de busqueda:
Fecha de la última votación> '20/04/2019'

```
List<Muestra> listaDeMuestras = Web.todasLasMuestras();

Filtro query1 = new UltimaVotacionDespuesDe(LocalDate.of(2019, 4, 20));
query1.buscar(listaDeMuestras);
```

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

```
Filtro query2 = new SoloVerificadas();
Filtro query3 = new And(query2,query1);
query3.buscar(listaDeMuestras);
```

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

```
IOpinable insecto = IOpinable.VINCHUCA_INFESTANS;
Filtro query4 = new TipoDeInsecto(insecto);
Filtro query5 = new And(query4, new OR(query2,query3));
```