TRABAJO PRÁCTICO INTEGRADOR

VINCHUCAS

Programación con Objetos II

Tomás Federico Acosta: tomas23058@gmail.com

Martin Alejandro Boidi : boidi.martinalejandro@gmail.com

Alessandro Giordanella: giordanellalessandro@gmail.com

Decisiones de diseño

El sistema

En nuestro proyecto decidimos tener una clase llamada **Sistema** que contiene todas las muestras del proyecto y todas las zonas de cobertura. Esto permite hacer cálculos tanto con las muestras, como con las zonas de cobertura. Además, tiene la capacidad de avisar a las zonas de cobertura, tanto de una nueva muestra cómo de una muestra verificada.

Al conocer las muestras y las zonas de cobertura, le delegamos la responsabilidad de decir que zonas están solapadas con respecto a una zona en particular. Esta responsabilidad originalmente era de **ZonaDeCobertura,** ahora la zona puede responder cuáles son sus zonas solapadas a partir de pasarle el sistema y que la zona se encargue de preguntarle.
El sistema calcula esto a partir de un cálculo simple siendo x la distancia, r1 y r2 los radios de dos zonas: si x > r1 + r2 entonces no se solapan. En caso de x <= r1 + r2 las zonas se solapan.

Por otro lado, hicimos algo parecido con el requerimiento de **Ubicación** el de “Dado una muestra, conocer todas las muestras obtenidas a menos de x metros o kilómetros.” ya que el sistema se encarga de hacer el cálculo a partir de una **Distancia** y una muestra.

Las muestras

Para las muestras del sistema implementamos una clase llamada **Muestra** que permite ser opinada, conoce al usuario autor, tiene una fecha de creación y de última votación, puede decir su resultado actual y enviarse al sistema. Para las verificaciones usamos el patrón State, ya que la muestra se comporta de manera diferente dependiendo si es una muestra verificada o no.
Un estado es **MuestraNoVerificada**, con el que inicia, este permite agregar opiniones, mientras que el estado **MuestraVerificada** no. El estado de la muestra cambia a verificada cuando coinciden dos opiniones de experto, momento en que ya no se puede opinar. También, cuando se verifica se le avisa al sistema.

La **MuestraNoVerificada** se encarga de validar que, cuando opina un experto, los usuarios básicos ya no puedan opinar. También de que un usuario no pueda opinar sobre muestras que hayan enviado ellos mismos, como así tampoco opinar más de una vez para una muestra.

Para calcular el resultado actual, la muestra hace un conteo de las opiniones y ve cual es la que aparece más veces. En caso de haber empate el resultado es **NoDefinido**.

También, en este cálculo se ve si hay alguna opinión de algún experto, en caso de ser así, se filtra a los usuarios básicos y el cálculo se hace solo con los expertos. Para obtener el historial de opiniones se le puede pedir las opiniones a la muestra.

Para las opiniones hicimos una clase **Opinion** que contiene el estado de Usuario que la creó, una fecha en la que se creó y el tipo de opinión. Esta opinión guarda al usuario autor y al estado del usuario al momento de opinar, para que si cambia el estado del usuario en algún momento, se mantenga el estado original. A partir de esto, puede responder si es una opinión de experto o no.

Para los tipos de opiniones usamos un enum, que contiene tanto los tipos de vinchuca como los tipos de insectos y los diferentes tipos de opinión. Esto lo hicimos así porque el enum más general (los tipos de opinión), a su vez contenían a los otros. Asumimos que no se van a usar tipos incorrectos en donde no deban. Por ejemplo, la opinión inicial de una muestra debería ser un tipo de vinchucas

El Usuario

Para el usuario implementamos la clase **Usuario** que puede agregar y enviar muestras, las cuales se agregan a una colección de muestras propias que tiene el usuario y se manda al sistema para que otros usuarios opinen sobre ella. También, tiene la función de opinar sobre las muestras que subieron otros usuarios para poder validarlas. Estas opiniones se guardan en una lista de opiniones enviadas del usuario.

En la implementación decidimos utilizar el patrón de diseño state, ya que tenemos dos tipos de usuarios diferentes los cuales pueden ir cambiando de estado.
Tenemos el usuario básico, es el estado en cual arranca un usuario nuevo, este va cambiar si supera las diez muestras enviadas y si opina en más de veinte muestras durante los últimos treinta días. Esto se puede calcular en cualquier momento llamando al mensaje *calcularCategoria()* del usuario.
También existen los usuarios de tipo Especialista los cuales al iniciar ya son expertos. Estos los implementamos de manera que sean subclase de Usuario y no cambien su estado.

Ubicación

Para las ubicaciones implementamos una clase **Ubicación** la cual posee una latitud y una longitud y sabe responder si está a menos de una distancia dada entre ella y otra ubicación. También es capaz de filtrar ubicaciones que se encuentren a menos de una distancia dada y calcular la distancia, tanto en metros y kilómetros, entre sí misma y una ubicación dada.

El sistema se encarga de conocer todas las muestras obtenidas a menos de x metros o kilómetros, esto lo hace a partir de una **Distancia** y una muestra. A la ubicación se puede

preguntarle si está a menos de una distancia de otra ubicación y preguntarle a partir de unas ubicaciones dadas, cuales están a menos de una distancia dada.

Para la distancia implementamos la clase **Distancia**, la cual tiene una **Unidad** y una cantidad. La Unidad está implementada como un enum que tiene metros y kilómetros. Esta distancia puede calcularse entre ubicaciones con el teorema de pitágoras para calcular la distancia entre dos puntos de un plano.

Distancia = √((Longitud2 - Longitud1)² + (latitud2 - latitud1)²)

Zona De Cobertura

Para la zona de cobertura implementamos una clase que se llama **ZonaDeCobertura**, la cual tiene un epicentro el cual es su **Ubicacion** y un radio que es una **Distancia** que abarca la zona y una lista de muestras, las cuales se encuentran dentro de su radio. Cuando se agrega una nueva muestra al sistema, este mira a qué zonas de cobertura pertenece y las agrega a esa zona en particular.
También, una zona de cobertura a partir de pasarle el sistema, puede preguntarle a este sus zonas solapadas.

Para que las organizaciones puedan saber cuando se agregó o verificó una muestra implementamos el patrón Observer. La zona de cobertura sería el ConcreteSubject , teniendo una lista de **ObservadorDeZona** a los cuales notifica. El **observadorDeZona** es el observer, puede actualizar tanto si es una muestra nueva o verificada. El ConcreteObserver sería la organización y por cada actualización ejecuta una funcionalidad concreta.

Organizaciones

Para las organizaciones implementamos la clase **Organizacion**, la cual tiene una cantidad de trabajadores, un tipo de organización que es enum, una **Ubicacion**, y dos funcionalidades externas que ejecutan dependiendo si se agregó o verificó una nueva muestra en las zonas a las que está suscrito.
Las organizaciones implementan observadorDeZona para poder suscribirse a las zonas de cobertura de interés. Cuando una zona actualiza una muestra, la organización realiza una funcionalidad dependiendo si es nueva muestra o muestra verificada. Las funcionalidades externas se implementan como una interfaz y pueden ser cambiadas dependiendo de lo que se requiera. La funcionalidad que esté seteada para un aviso particular en la organización será la que ejecute el nuevo evento.

Búsqueda de muestras

Los filtros de búsqueda se extienden de la clase abstracta **Filtro** y permiten buscar en las muestras que son dadas por parámetro aquellas que cumplan con un determinado criterio de búsqueda. Este último es implementado por cada uno de los filtros.

Esta implementación es modelada por medio del patrón de diseño Composite que nos permite unir distintos filtros mediante filtros recursivos. El mismo nos permite fácilmente realizar búsquedas ya que incorpora la posibilidad de componer búsquedas con distintos filtros mediante conectores lógicos.

Para implementar el patrón, la clase abstracta **Filtro** cumple el rol de Component. La clase abstracta **OperadorDeFiltros** es el Composite e incorpora la estructura que deben cumplir los operadores, ya que todos deben operar filtros de acuerdo a un criterio específico. Los filtros simples como por: fecha de creación de la muestra, fecha de la última votación, tipo de insecto detectado en la muestra o nivel de verificación, serían los Leaf.

Por otro lado, las clases **ConjuncionDeFiltros** y **DisyuncionDeFiltros** extienden **OperadorDeFiltros**, ya que incorporan componentes para ejecutar la búsqueda de manera recursiva, permitiendo unir a sus componentes de acuerdo al comportamiento de las compuertas lógicas. Encontramos innecesario en este caso implementar un comportamiento que permitiera agregar más de dos filtros a los compuestos, ya que entendemos que las formas lógicas de unir filtros solo actúan siempre a partir de dos, pudiendo ser cada uno de los mismos también compuestos para realizar búsquedas aún más complejas, pero siempre cumpliendo con este mismo criterio básico.

Clases

Paquete filtro:

- **Filtro**
- ConjuncionDeFiltros
- DisyuncionDeFiltros
- FiltroPorFecha
- FiltroPorFechaDeCreacion
- FiltroPorFechaDeUltimaVotacion
- FiltroPorNivelDeVerificacion
- FiltroPorTipoInsecto
- OperadorDeFiltros

Paquete muestra:

- **Muestra**
- EstadoMuestra
- MuestraNoVerficada
- MuestraVerificada
- Opinion
- TipoOpinion

Paquete organizacion:

- **Organizacion**
- FuncionalidadExterna
- TipoOrganizacion

Paquete sistema:

- **Sistema**

Paquete usuario:

- **Usuario**
- EstadoUsuario
- UsuarioBasico
- UsuarioEspecialista
- UsuarioExperto

Paquete zona de cobertura:

- **ZonaDeCobertura**
- Distancia
- ObservadorZona
- Ubicación
- Unidad

Patrones de diseño

Primer Patron State

Context: Muestra
State: EstadoMuestra
ConcreteState: MuestraNoVerificada
ConcreteState: MuestraVerificada

Segundo Patrón State

Context : Usuario
State: EstadoUsuario
ConcreteState: UsuarioBasico
ConcreteState: UsuaioExperto

Tercer Patrón Observer

ConcreteSubject: ZonaDeCobertura
Observer: ObservadorZona
ConcreteObserver: Organización

Cuarto Patrón Composite

Component: Filtro
Composite :

- OperadorDeFiltros
- DisyuncionDeFiltros
- ConjuncionDeFiltros

Leaf:

- FiltroPorFecha
- FiltroPorFechaDeCreacion
- FiltroPorFechaDeUltimaVotacion
- FiltroPorNivelDeVerificacion
- FiltroPorTipoInsecto