Trabajo Final: A la caza de las vinchucas

Integrantes

- Caceres Jaramillo Rodrigo Daniel - rodicom10@gamail.com
- Garcia Juan Ignacio - juan.ignacio.garcia.02@gmail.com
- Mendoza Ignacio Martin - nachommendoza@hotmail.com

Diseño:

Administrador de muestras

Para poder manejar la lógica de las muestras de forma separada se creó la clase Administrador de muestras. Este conoce la colección de muestras en todo el sistema, es el encargado de iterar sobre las muestras y ejecutar los métodos adecuados sobre las muestras de forma individual.

Administrador de zonas

Para poder manejar la lógica de las zonas de forma separada se creó la clase Administrador de zonas. Este conoce la colección de zonas en todo el sistema;
Es el encargado de iterar sobre las zonas y ejecutar los métodos adecuados sobre las zonas de forma individual.

El Sitio Web

Ya que el Sitio web reúne demasiadas tareas y responsabilidades diferentes decidimos separarlo en partes(Administrador de zonas, Administrador de muestras). Esto nos facilita la comunicación entre dichas partes, permite que el sistema sea más flexible y abierto. Permitiendo mantener el principio de Single responsability.
El sitio web interactúa con ambos administrados para poder dividir las responsabilidades adecuadamente, siendo la cara principal del sistema la cual será
la que siempre interactúa con el cliente. Este distribuirá la información proporcionada por los administradores.

Usuario

El usuario interactúa con el sitio web de diferentes maneras dependiendo el tipo de usuario del mismo. Este es el encargado de subir muestras y opinar sobre las mismas al sitio web. A su vez, el usuario contiene un registro de fechas en las cuales subió muestras y otro que registra las fechas de las opiniones hechas por el mismo.

Patrones

Composite: Criterio

A la hora de hacer la búsqueda de muestras debemos usar filtros estos pueden ser unos simples o un conjunto de ellos. A partir de esa premisa decidimos utilizar el patrón Composite, para poder tratar los simples y el compuesto de forma uniforme.

- cliente -> AdministradorDeMuestras
- component -> Criterio
- leaf -> CriterioFechaDeCreacion, CriterioFechaUltimaVotacion, CriterioTipoDeInsecto, CriterioNivelDeVerificacion (casos no recursivos)
- composite -> CriterioCompuesto

Strategy: ConectorLogico

En el caso del criterio compuesto nos encontramos que dependiendo el conector (Or o And) se debe proseguir de distinta forma.
Por esa razón usamos Strategy para poder usar el ConectorLogicoAnd y el ConectorLogicoOr

- Stategy -> ConectorLogico
- ConcreteStrategy -> ConectorLogicoOr , ConectorLogicoAnd
- Context -> CriterioCompuesto

Strategy: OrganizacionNoGubernamental

Las organizaciones a la hora de registrar una muestra o validarse en una zona de interés para ellas hacen una función externa(que puede ser la misma o diferente en caso de la razón), el patrón Strategy permite cambiar la función externa.

- Stategy -> FuncionExterna
- ConcreteStrategy -> el trabajo menciona que hay pero no especifica cuáles
- Context -> OrganizacionNoGubernamental

Singleton: CalculadorDeDistancia

Decidimos utilizar el patrón Singleton para el CalculadorDeDistancia ya que decidimos separar la lógica matemática de calcular la distancia entre dos ubicaciones y la lógica de ubicaciones. Para ello decidimos tener la clase CalculadorDeDistancia.
Al comienzo instanciamos un CalculadorDeDistancia por cada mensaje esto hacía que una una ubicación pudiera instanciar x cantidad de CalculadorDeDistancia, al ver esto optamos ponerlo como contribuidor pero esto mantiene el problema de tener múltiples instancia (se redujo a 1 por ubicación). Esto nos llevó al patrón Singleton, esto nos permite tener una o ninguna instancia de CalculadorDeDistancia en total.

- Singleton -> CalculadorDeDistancia

State: TipoDeUsuario

Usamos el patrón State para el estado de Usuario entendiendo que algunos métodos tiene comportamiento distinto a partir del estado del mismo, este se puede actualizar cuando opine sobre alguna muestra o publique una muestra (el experto validado puede “transformarse” en cualquier momento y nunca se cambia).

- Context -> Usuario.
- State -> TipoDeUsuario.
- ConcreteState -> Basico, Experto, ExpertoValidado.

State: TipoDeMuestra

Usamos el patrón State para el estado de Muestra entendiendo que algunos métodos tiene comportamiento distinto a partir del estado de la muestra y este se puede actualizar cada vez que recibe una opinión.

- Context -> Muestra.
- State -> TipoDeMuestra.
- ConcreteState -> Verificada, SiendoVerificada, NoVerificada.

Observer: ZonasDeCobertura

Decidimos que era necesario el uso del patrón Observer en el momento que detectamos que las OrganizacionesNoGubernamentales requieren conocer cuando se valida o se agrega una nueva muestra a una de sus zonas de interés para poder hacer una función externa.

(No cumple el patrón de diseño estándar)

- Subject -> OrganizacionNoGubernamental
- ConcreteSubject -> OrganizacionNoGubernamental
- Observer -> ZonasDeCobertura
- ConcreteObserver -> ZonasDeCobertura

Aclaraciones

- Decidimos separar la lógica matemática de la lógica de Ubicación, por esa razón se creó la clase CalculadorDeDistancia que es la encargada de los cálculos.

- CalculadorDeDistancia implementa una lógica irreal a al hora de saber la distancia entre dos ubicaciones en la tierra, ya que la fórmula utilizada no tiene en cuenta la curvatura de la misma. Se le comento este caso al profesor Diego Cano y se le preguntó si se tendría que usar otra fórmula que sí tenga en cuenta la curvatura (*) y nos comentó que no había problema al usar la fórmula implementada.

(*) Esta es la fórmula que se le presentó:

```
public static double distanciaCoord(double lat1, double lng1, double lat2, double lng2) {
        //double radioTierra = 3958.75;//en millas
        double radioTierra = 6371;//en kilómetros
        double dLat = Math.toRadians(lat2 - lat1);
        double dLng = Math.toRadians(lng2 - lng1);
        double sindLat = Math.sin(dLat / 2);
        double sindLng = Math.sin(dLng / 2);
        double va1 = Math.pow(sindLat, 2) + Math.pow(sindLng, 2)
        * Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2));
        double va2 = 2 * Math.atan2(Math.sqrt(va1), Math.sqrt(1 - va1));
        double distancia = radioTierra * va2;
        return distancia;
}
```

-OrganizacionNoGubernamental tiene dos atributos internos que son del tipo FuncionExterna, se modelo de esta forma ya que en el enunciado del trabajo se menciona que la función externa que debe utilizarse cuando se valida una muestra o se agregar una muestra (en una zona de interés de la org) puede ser la misma o no.

- Si bien el Usuario no implementa una interfaz, a la hora de cranear el diseño del proyecto, se tuvo en consideración, ya que el enunciado menciona un usuario móvil.

- Decidimos representar la Foto con un String. Porque, la representación y funcionamiento de la Foto está fuera del alcance del proyecto.

- Decidimos modelar una clase Opinion, la cual no tiene más funcionamiento que gets y sets. Porque, nativamente java no tiene tripletes y decidimos no utilizar una librería externa para utilizar esa funcionalidad. Entendemos que la clase Opinion no respeta las buenas prácticas de programación orientada a objetos.

- Elegimos utilizar el enum TipoDeOpinion, para definir todos los casos de opiniones posibles. Comprendemos que esta implementación en ciertos casos, permitirá la utilización de un tipoDeOpinion incorrecta, pero por la forma que están implementados los enum en Java no se pueden incluir enums dentro de otro o crear una jerarquía de los mismos.