Trabajo Práctico Integrador

Número de grupo: 14
Alumnos: Oreskovic, Franco Juan – Reynoso, Elías
Materia: Programación con Objetos 2
Universidad: Universidad Nacional de Quilmes
Año: 2025 (primer cuatrimestre)
Comisión: 2

Información de contacto:
-Oreskovic, Franco: kitofran15@gmail.com
-Reynoso, Elías: ereynoso2040@gmail.com

1. Decisiones de diseño

1.1 Bases del modelo lógico

Como parte de la base del modelo lógico implementado, tenemos una clase **App** destinada al almacenamiento de Regions. También, haciendo uso de estas Regions, es capaz de enunciar la totalidad de Samples creadas en el marco de la aplicación representada.
Tanto la lista de Regions como la lista de Samples son pasadas por parámetro a los diferentes métodos de distintos objetos que requieren de dicha información para llevar a cabo su funcionalidad.
Se opta por hacer esto en vez de hacer uso de una clase Singleton y acceder a los objetos almacenados de manera global, ya que, en este nivel de capa de diseño, el acceso a instancias globales puede ser evitado.
Por tanto, a cualquier funcionalidad que requiera utilizar la totalidad de Regions o Samples del sistema, recibirá las mismas por parámetro tras un pedido a la clase de almacenamiento App.
Ya con las listas de objetos a su disposición, las distintas clases hacen los procesamientos necesarios para obtener los resultados que buscan

1.2 Usuarios y estado de usuarios

Los usuarios de este sistema, quienes cargan y reseñan muestras, están modelados mediante la clase **User**.
Estos tienen 2 funcionalidades principales que son la de cargar muestras al sistema y la de reseñar una muestra específica del sistema.
Al momento de reseñar una muestra, se chequeará si el usuario en cuestión puede o no realizar la reseña sobre esa muestra en base a si aún no opinó sobre esta y a su expertise.
Devolver el expertise es una tarea que se delega al objeto que representa al estado del usuario, ya que se hace uso del patron State sobre User y parte de la funcionalidad de este es delegada a su estado.
Dependiendo de cuál sea el expertise del usuario (lo cual le pregunta a su estado), este podrá o no agregar su reseña a una cierta muestra, lo cual dependerá también de en qué estado se encuentre dicha muestra.

El estado del usuario calculará con el método statCheck() si debe llevar a cabo un cambio de estado a otro de los estados posibles para el usuario, esto en base a la cantidad de muestras cargadas y de reviews realizadas en los últimos 30 días por este.
Los usuarios comenzarán con un estado que representa al usuario básico y podrán variar entre ese y otro que representa al usuario experto según sus aportaciones a la aplicación.
Sin embargo, habrá usuarios especiales que tienen un estado que representa a los expertos permanentes, el cual jamás cambia.

1.3 Muestras, estado de muestras y reseñas/opiniones

Nuestras muestras son representadas por la clase **Sample**. Una muestra, la cual es creada por un usuario, se conforma de distintos datos como el usuario que la creó, la posición geográfica en donde tuvo lugar la foto, la fecha en que fue subida y el tipo de opinión que, en base a las distintas reseñas de los usuarios sobre la muestra, se toma como el resultado para la misma en ese momento.
El resultado, si bien se calcula en base a las reseñas realizadas para esa muestra, va a calcularse de forma diferente dependiendo del estado en el que se encuentre esta.

Acá nuevamente se hace uso del patrón State y se delega parte de las responsabilidades de la muestra a su estado, como, por ejemplo, el cálculo del resultado actual.

El estado abierto de la muestra realizará el cálculo teniendo en cuenta solo las opiniones de usuarios básicos, el estado “solo expertos” solo tendrá en cuenta la opinión de expertos, y el estado cerrado simplemente devolverá el resultado que ya se tiene, el cual es definitivo.

Además, el estado de la muestra es el que se encarga de revisar si es válido que un cierto usuario con un determinado expertise pueda reseñar la muestra a la que está ligado.
En este sentido, el estado abierto no restringirá las reseñas de nadie, el “solo expertos” permitirá solamente opiniones expertas, y el cerrado directamente no permitirá opiniones.

Por último, los estados de la muestra también llevarán a cabo cambios de estado si corresponde. El estado abierto pasará a “solo expertos” si se añade una reseña experta, el “solo expertos” pasará a cerrado si se añade una reseña experta que coincida en el tipo de opinión con otra reseña experta preexistente para la muestra, y el cerrado nunca lleva a cabo cambios de estado.

1.4 Zonas de cobertura y ubicaciones

Las zonas de cobertura son zonas geográficas con un epicentro (que es una ubicación geográfica) y un radio expresado en kilómetros como atributos. Las mismas son modeladas mediante la clase **Region**. Esta almacena la lista de muestras que tuvieron lugar en una ubicación correspondiente al rango geográfico de la zona de cobertura y también es capaz de enunciarlas.
Esta clase tiene un método para saber si una ubicacion, las cuales son modeladas con la clase **Position**, pertenece al rango geográfico de la zona de cobertura, el cual es utilizado por App al momento de almacenar una muestra cargada en las listas de muestras de las regiones correspondientes.
Además, también posee la funcionalidad de, recibiendo una lista de Regions, enunciar aquellas con las que se da un solapamiento geográfico.

Tanto para el cálculo del solapamiento (la distancia entre centro A y centro B debe ser menor a la suma de los radios) como para el cálculo para saber si una ubicación pertenece a la zona de cobertura (la distancia entre el centro de la zona de cobertura y la ubicación es menor al radio de la primera) se recurre al método getDistanceTo() de la ubicacion que representa al epicentro de la zona de cobertura. Este método expresa la distancia entre la ubicación que lo ejecuta y la otra ubicación recibida como parámetro.

Además de esa funcionalidad, las ubicaciones también son capaces de recibir una lista de ubicaciones y un double que representa un radio y enunciar la lista de ubicaciones que se encuentran a una distancia menor o igual a la del radio. También son capaces de hacer lo propio con una lista de muestras, para las cuales utilizan el dato de su ubicación.

Para todas estas funcionalidades de Position, se recibe por parámetro también un objeto que representa la unidad de medida que debe ser devuelta como resultado. Puede ser o bien kilómetros o bien metros. En getDistanceTo() se le delega a esta unidad de medida la tarea de convertir el resultado obtenido en kilómetros a la unidad correspondiente (a metros o dejarla en kilómetros).
En los otros 2 métodos, se envía por parámetro la unidad de medida a los getDistanceTo() utilizados.

1.5 Zonas de cobertura, manejador de eventos, organizaciones y funcionalidades externas

En las zonas de cobertura también se implementa un patrón Observer, ya que existen organizaciones representadas mediante la clase **Organizacion** (que cumple el rol de ConcreteObserver) las cuales quieren enterarse de eventos relativos a ciertas zonas de cobertura que les interesan como cuando se

carga una muestra para una de estas o cuando una muestra de una de estas es verificada (o sea, pasa a estar en estado cerrado).
Para tener métodos destinados a reaccionar de una cierta forma ante tales avisos de eventos, la clase Organizacion implementa la interfaz **Iobserver** (que cumple el rol de Observer).

Para enterarse del acontecimiento de estos eventos, las zonas de cobertura (que cumplen el rol de Subject) tienen como atributo un objeto de tipo **EventManager** (que cumple el rol de EventManager) el cual maneja una lista de suscritos al evento de carga y una lista de suscritos al evento de validación.
Se decidió usar 2 listas de tipo Iobserver en vez de un Map<String, List<Iobserver>> ya que tenemos certeza de que no se agregarán más eventos y, por tanto, no se agregarán más listas de suscritos para esos eventos, por lo que podemos utilizar la cantidad exacta de listas (lo cual no estaría bien si no fuera así, ya que estaríamos rompiendo el principio Open-Closed).

La notificación de carga se origina del addSample() de App, en el que la clase de almacenamiento recorre las diferentes zonas de cobertura almacenadas revisando si la nueva muestra corresponde a estas o no. De ser así, les envía a estas zonas de cobertura el mensaje addSample(sample), con la muestra en cuestión como parámetro.
Ya en la clase Region, la muestra se añade a la lista de muestras almacenadas que corresponden a esa zona de cobertura y, además, se envia el mensaje notify("upload", sample) a sí misma, pasando a la muestra en cuestión por parámetro.
El método de Region notify() recibe un String y una Sample como parámetros, siendo el String lo que identifica el tipo de evento del cual se va a realizar un anuncio.
El trabajo de notificar a las organizaciones correspondientes es delegado al objeto de tipo EventManager que se tiene como atributo. Se le envía el mensaje notify() con el mencionado String que identifica al tipo de evento y la Sample, además de la propia Region, ya que es información que van a necesitar las organizaciones notificadas.

En el método notify del EventManager se pregunta con un if si el String recibido es igual a "upload" o "validation" para mandar el mensaje correspondiente a las organizaciones interesadas. Este if no se puede esquivar en el patrón Observer. Como mucho, puede hacerse dentro de la Organizacion en sí, pero nos parece más pertinente realizarlo en el método del EventManager.
Para el caso del aviso de carga, se recorre la lista de Iobserver (que son organizaciones, pues es la única clase que la implementa) en la que se almacenan los suscritos a eventos de carga. A cada una de estas regiones se le envia el mensaje notifyUpload() junto a la Sample y Region recibidas por parámetro, ya que las organizaciones requieren estos datos para reaccionar a los avisos.
Para el caso del aviso de validación, el procedimiento es el mismo, solo que se envia el mensaje notifyValidation() junto a la Sample y Region recibidas por parámetro a las organizaciones de la lista en la que se almacenan los suscritos a eventos de validación.
Acá el EventManager ya cumplió su tarea de avisar a las organizaciones interesadas en ciertos eventos de la zona de cobertura asociada de que ocurrieron los eventos de su interés.

Ya en la clase Organizacion, la misma implementa Iobserver y, por tanto, los métodos notifyValidation() y notifyUpload(), esto con la intención de tener una manera de recibir el aviso de que ocurrió un evento de una zona de cobertura que les interesa y poder reaccionar de una determinada manera a esto.
Lo que hacen las organizaciones en estos métodos es llamar a la **FuncionalidadExterna** que tienen como atributo para reaccionar al tipo de evento en cuestión.
Cada Organizacion tiene una FuncionalidadExterna para reaccionar al evento de carga y otra para reaccionar al evento de validación.

FuncionalidExterna es una interfaz que tiene como único método nuevoEvento(). Las clases concretas que la implementan se encargan de llevar a cabo la reacción a un evento de una forma particular, y hacen uso de los datos recibidos que son la Sample, la Region y la propia Organizacion para hacer esto.
Estas dependencias a las que se les delega parte del comportamiento de Organizacion pueden ser intercambiadas en cualquier momento por otra funcionalidad externa distinta mediante un setter, pero el cambio no lo llevan a cabo las propias funcionalidades externas, las cuales no se conocen entre ellas, sino la organizacion como tal, quien sí conoce la familia de algoritmos existentes.
Con todo esto dicho, queda más que claro que acá se implemento un patrón Strategy, donde Organizacion es Context y FuncionalidadExterna es Strategy.

Regresando a la otra notificación de las zonas de cobertura, que es la de validación, la misma se origina del addReview() de Sample.
En este método, una de las cosas que se hace es delegarle al estado de la muestra mediante checkStateChange() el chequear si se debe llevar a cabo un cambio de estado para dicha muestra.
Acá, para el estado “solo expertos”, lo que se va a chequear es si el OpinionValue de la nueva reseña que se va a añadir es igual al de una reseña ya existente para la muestra, lo cual significa que 2 expertos coincidieron en su opinión y se debe hacer un cambio de estado a Closed.
Este chequeo lo hace mediante un mensaje expertsCoinciden() a la propia muestra con el OpinionValue como parámetro. De devolver true, se lleva a cabo el cambio de estado al estado cerrado para la muestra.
En el método expertsCoinciden() de Sample, se toman unicamente las reseñas realizadas por expertos para esa muestra y se revisa si hay alguna que tenga el mismo OpinionValue recibido por parámetro (cabe aclarar que la reseña aún no se añade a la lista de reseñas, por lo que no va a pasar que se de una coincidencia con la propia reseña añadida).
De darse la coincidencia, se devuelve un valor true y, por tanto, se hace el cambio de estado a Closed mediante el método del estado changeState(), el cual no solo le setea a la Sample una instancia de Closed como nuevo estado, sino que también envia a la sample el mensaje notifyValidation().
En este método, se recorre la totalidad de Regions de la App ligada a la muestra y se recuperan unicamente aquellas que contienen a la misma.
A todas estas zonas de cobertura (si es que hay alguna) se les envía el mensaje notify(“validation”, this) para que den aviso a las organizaciones interesadas de que ocurrió un evento de validación en esa zona.
Luego, ocurre el proceso ya descrito anteriormente.

1.6 Buscador

El buscador o motor de búsqueda es representado mediante la clase **SearchEngine**. Tiene un único método que es el de buscar(), el cual recibe una lista de Samples (que debería ser la totalidad de muestras del sistema, para que la búsqueda sea total), una lista de Criterios y una lista de LogicalOperators.
Criterio es una clase abstracta que representa un criterio de búsqueda utilizable en el buscador.
No tiene ningún atributo porque son las clases concretas que extienden a Criterio las que tendrán como atributo un parámetro de búsqueda del tipo que les corresponde para realizar el filtrado (o sea, el tipo de atributo es diferente en cada caso, por lo que cada criterio concreto debe tener su propio atributo de tipo particular).
La clase Criterio define en lasQueCumplenCriterio() el algoritmo para enunciar la lista de muestras que cumple dicho criterio, en el cual se utiliza un método abstracto en esta clase llamado cumpleCriterio(), el cual recibe una única muestra por parámetro y enuncia si la misma cumple el criterio o no.

Este método abstracto será implementado en las 6 clases concretas que extienden a Criterio, en donde utilizarán el parámetro de búsqueda con el que fueron instanciadas para determinar si la muestra pasada

por parámetro efectivamente cumple con el criterio o no (y, obviamente, la forma de evaluar eso variará dependiendo del criterio concreto).

Los 6 tipos de criterio serán si la fecha de creación de la muestra es posterior a x, si es anterior a x, si la fecha de la última reseña cargada es posterior a x, si es anterior a x, si su nivel de verificación es igual a x (“votada” o “verificada”), y si su tipo detectado es igual a x (siendo tipo detectado lo que se enuncia cuando se le pide el resultado actual a la muestra).

Volviendo nuevamente con SearchEngine y su método buscar(), se generan tantas listas de muestras como criterios se hayan pasado por parámetro, cada una de estas con la lista de muestras que cumple uno de los criterios en cuestión.
Tras esto, se instancia un Set de muestras al que se le agregarán las muestras de la primera de las listas obtenidas anteriormente. Esto se hace debido a que ahora se recorrerá la lista de operadores lógicos para encadenar los conjuntos de muestras obtenidos de la forma correspondiente, pero, en caso de que no haya ninguno, eso significa que no hay operaciones lógicas por hacer y las muestras que se deben devolver son las que cumplen ese único criterio analizado (lo cual es un caso posible).

Si ese no es el caso y sí hay operaciones lógicas por hacer, lo que se hace es recorrer la lista de LogicalOperators y, en cada caso, revisar si el procesado es un And o un Or.
En caso de ser And, se realiza un filtrado sobre el set de samples que cumplen la expresión lógica hasta el momento y se mantienen únicamente aquellas que también están contenidas en la próxima de las listas anteriormente obtenidas, que es la que toca procesar en este momento para llevar a cabo la operación lógica.
Si, en cambio, es un Or, simplemente se agrega al set de samples que cumplen la expresión lógica hasta el momento a todas las samples de la lista que toca procesar ahora. Al utilizar el addAll del set, no habrá problemas de repetidos, ya que los sets no los admiten.

Finalmente, ya recorridos todos los operadores lógicos, el ciclo termina y se devuelve el set convertido a lista, ya que el tipo de retorno es List<Sample>.

2. Detalles de la implementación que merecen ser explicados

*__Almacenamiento de las muestras__:
Tras que un usuario cargue una muestra, le delega el almacenamiento de la misma al objeto App que tiene como atributo, el cual se va a encargar de guardarla donde corresponde.
Si bien las muestras son almacenadas por la app en las listas de muestras de las regiones correspondientes, también pueden existir muestras que no correspondan a ninguna de las regiones conocidas por el sistema. Para estos casos, las muestras se almacenan en una lista del objeto App destinada a almacenar muestras sin región.
Cuando se le pide a App que enuncie la totalidad de las muestras del sistema, recuperará las muestras de cada una de las zonas de cobertura que tiene almacenadas exceptuando las repetidas entre distintas zonas de cobertura, y a ese conjunto de muestras le añadirá las muestras sin región, lo que conformará el resultado que finalmente devolverá.

3. Patrones de diseño utilizados y roles

*Patrones de diseño utilizados:

-Patron Observer:

Region => Subject
EventManager => EventManager (rol)
IObserver => Observer
Organizacion => ConcreteObserver

-Patron State (2 casos):

Usuario

User => Context
IUserState => State
Basic y Expert => ConcreteState

Sample

Sample => Context
ISampleState => State
Open, ExpertOnly, Closed => ConcreteState

-Patron Strategy:

Organizacion => Context
FuncionalidadExterna => Strategy
(las clases que implementen FuncionalidadExterna) => ConcreteStrategy