Trabajo Práctico Final

Programación con Objetos 2.

Callegaris, Belén. belen.alumine@gmail.com

Rivero, Carlos. carlos\_rivero7410@hotmail.com

Para la organización de las clases creamos los paquetes accionesDeProyecto, accionesGenerales, elementosDelSistema, y sistema.

Proyecto:

La clase Proyecto define un proyecto, se lo construye con dos parámetros String, para nombre y descripción, y otro de AreaGeografica, para el área del mismo.

Contiene participantes (List<Usuario>), muestras (List<Muestra>), desafios (List<Desafio>) y categorías (List<String>).

Usuario:

El usuario posee una lista de desafíos en la que se encontrarán tanto los desafíos que vaya aceptando como los que vaya completando, pudiendo acceder a la lista de todos los desafíos aceptados, y todos los desafíos completados de forma aparte.

La manera de obtener nuevos desafíos será mediante un método que los solicitará a una instancia de clase ‘RecomendacionDeDesafio’ teniendo en cuenta que el usuario solo aceptará desafíos hasta tener cinco activos a la vez. Sin embargo, los desafíos a recibir varían según el tipo de Recomendación que indique el usuario mediante su mensaje que se aplicará a la instancia de ‘RecomendacionDeDesafio’.

También podrá consultar por el porcentaje de completitud de algún desafío que brinda como parámetro, como también el porcentaje de completitud aplicado a todos los desafíos que posea el usuario. Referente a los desafíos, el usuario podrá indicar una valoración asociada a cada uno de ellos que haya aceptado con una valoración del 1 al 5, pudiendo acceder al desafío que más le gusto, o en otras palabras su desafío favorito, en base al desafío con mayor valoración.

Además usuario posee un atributo que lo asocia con una instancia de la clase ‘PerfilUsuario’ que denotará sus preferencias respecto a los desafíos que desea.

PerfilUsuario:

El perfil de usuario posee los atributos que indicarán las preferencias a buscar en los desafíos para el usuario mediante coincidencias y estos serían: ‘Cantidad de muestras a recolectar’ en el desafío, ‘dificultad’ deseada del desafío y la ‘recompensa’ que se busca a obtener de este mismo. Mediante getters se obtendrán sus valores para realizar los cálculos correspondientes para la obtención de las características buscadas en los desafíos mediante la instancia ‘RecomendaciónDeDesafio’.

Muestra:

Una Muestra es una clase que se contruye a partir de un usuario (Usuario), y la fecha y la hora actual(LocalDate, LocalTime), para definir el momento de creación de la misma.

Las muestras son creadas mediante la clase GeneradorDeMuestra, que es invocada por un usuario.

Desafio:

Un Desafio es una clase que utiliza en su constructor los parámetros muestrasARecolectar y dificultad (int) para determinar la cantidad de muestras que debe recolectar un usuario, y la dificultad que representa el mismo, restriccion (RestriccionTemporal) para determinar qué tipo de restricción define cuándo pueden contabilizarse muestras en el desafío y areaGeografica (AreaGeografica), que determina la locación del desafío.

Además tiene atributos que se calculan o settean por otras clases, recompensa (int) que se incrementa cada vez que se genera una Muestra (siempre de acuerdo a la restricción temporal), estadoDelDesafio (EstadoDelDesafio) que es determinado por el state del desafío, fechaActual (LocalDate), se genera de forma automática, es la fecha de creación del desafío, desafioRestrigido (boolean) lo settea la RestriccionTemporal y es para saber si el desafío actualmente se encuentra restrigido para la carga de muestras o no, y votacionDeUsuario (int), que es un valor que settea el usuario.

ValoraciónDesafio:

La clase se asocia a cada desafío y mediante el se le podrá indicar una valoración que solo puede estar entre 0 y 5. De intentar dar otra valoración que no cumpla esta condición se tira una excepción informando del incumplimiento del valor posible para una valoración.

GeneradorDeMuestra:

La clase GeneradorDeMuestra se encarga de crear las muestras y mediante la clase ActualizarDesafio, actualizar los desafíos.

Esta clase es invocada por un usuario, dentro del método crearMuestra, la agrega en el Proyecto dónde participa el usuario, y si el state del desafío es EstadoEnCurso, mediante la clase ActualizarDesafio, actualiza el desafío actual si corresponde.

ActualizarDesafio:

Esta clase es usada por el GeneradorDeMuestra para actualizar los desafíos. Actualizar un desafío consiste en incrementar en uno la cantidad de muestras cargadas del desafío.

Con el método actualizarDesafio corrobora el estado del desafío, y si es en curso, con el método actualizarMuestrasRecolectadas chequea que no se encuentre restringido, y si no lo está, incrementa.

Patrones:  
Para el diseño del trabajo incluimos 3 patrones de diseño distintos, en 4 partes. Estos son Strategy, State y Composite.

En el caso del State se tomó esta decisión de diseño porque se identificó que un desafío puede pasar por distintos estados a partir de la actividad que realice un usuario en torno a él.

En el caso de los composite, en ambos caso notamos la recursividad al combinar los distintos criterios, tanto de búsqueda cómo de restricción.

Y la desición del Strategy fue tomada por el hecho de que un usuario puede decidir un tipo de recomendación, y cambiarla cuándo le parezca.

**State. EstadoDelDesafío.**

Este patrón está compuesto por tres estados, EstadoNoAceptado, EstadoEnCurso, EstadoFinalizado.

EstadoDelDesafio:

EstadoNoAceptado: Es el estado inicial del desafío, este cambia cuándo el desafío es aceptado por el usuario.

EstadoEnCurso: Es el segundo estado del desafío, inicia cuándo el desafío es aceptado y finaliza cuándo se recolectan las muestras necesarias. No puede regresarse al estado anterior una vez iniciado.

EstadoFinalizado: Es el último estado del desafío, inicia cuando se recolectan las muestras necesarias, y no puede regresarse a estados anteriores.

**Composite. Buscador.**

Este patrón está compuesto por la clase component Busqueda, las clases leaf BuscadorSimple, que a su vez tiene las clases hijas BusquedaIncluyeCategoria, BusquedaExcluyeCategoria, BusquedaTituloConTexto, que definen los tipos de busqueda que pueden realizarse, y el composite BuscadorCompuesta, que a su vez tiene las clases hijas OperadorAnd y OperadorOr, que definen los operadores que pueden utilizarse para combinar los criterios de busqueda.

FiltroDeBusqueda: Interfase. Define el método buscar(List<Proyecto>), la cual es implementada por el Sistema, (este le carga toda la lista de proyectos contenidos en el sistema).

*Busqueda*: Es una clase abstracta que implementa el FiltroDeBusqueda, y hereda a sus hijas, BusquedaSimple y BusquedaCompuesta.

*BuscadorSimple*: Es una clase abstracta que define la estructura de los tipos de busqueda. Tiene un atributo valorBuscado, dónde se settea el valor de tipo String a buscar, y sus respectivos getter y setter. También el método abstracto buscar(List<Proyecto>), que es redefinido en cada clase hija.

BusquedaIncluyeCategoria: Utiliza los métodos y la variable de la clase padre para obtener el valor que se está buscando, y redefine el método de buscar, y añade otro proyectoIncluyeValor(Proyecto, String), que es utilzado para buscar dentro de la lista de categorías de un proyecto el valor buscado, e incluirlo en la lista de resultados si lo incluye.

BusquedaExcluyeCategoria: Utiliza los métodos y la variable de la clase padre para obtener el valor que se está buscando, y redefine el método de buscar, y añade otro proyectoExcluyeValor(Proyecto, String), que es utilzado para buscar dentro del proyecto el valor buscado, si el valor buscado no se encuentra entre las categorías del proyecto, se lo incluye en la lista de resultados.

BusquedaTituloConTexto: Utiliza los métodos y la variable de la clase padre para obtener el valor que se está buscando, y redefine el método de buscar, y añade otro proyectoConValorBuscadoEnTitulo(Proyecto, String), que es utilzado para buscar en el título de un proyecto el valor buscado. Si lo encuentra, lo incluye en la lista de resultados.

*BuscadorCompuesta*: Es una clase abstracta que define la estructura de los operadores de busqueda. Utiliza los atributos busqueda 1 y 2 para settear los tipos de busqueda que deben seguirse, con los que deberán operar las hijas, y las listas primer y segundo filtrado, que guardan los proyectos que se van revisando. Además define los getters y setters de cada uno de los atributos.

OperadorAND: Redefine el método buscar(List<Proyecto>), le carga en los atributos de la clase padre las listas de proyectos de dónde debe revisar, una vez la lista de proyectos siguiendo el primer criterio de busqueda setteado, y luego revisa la lista según el segundo criterio. Con el método agregarProyecto(Proyecto) agrega el proyecto actual a la lista si corresponde.

OperadorOR: Esta clase funciona igual al operador and, pero bajo la lógica de un Or.

**Composite. RestriccionTemporal.**

En este caso el composite es RestriccionTemporal, las leaf RestriccionPorFecha,

RestriccionPorSemana, RestriccionPorFinDeSemana, y el composite RestriccionCombinada. Un desafío puede estar restringido, o no estarlo, esto se settea en un boolean del desafío, mediante el método setDesafioRestringido(boolean) del Desafio.

*RestriccionTemporal*: Es una clase abstracta que define la estructura de las restricciones. Tiene métodos abstractos cómo restringir(Desafio), restringido(Desafio), agregarRestriccion(RestriccionTemporal) que serán definidos en cada restricción, y métodos generales que utilizan algunas restricciones para saber si el día actual es fin de semana.

RestriccionPorFecha: Esta clase utiliza un constructor que le provee dos fechas, una de inicio y otra de fin de la restricción. Define los métodos abstractos de la clase padre y agrega otros para saber si el día actual es anterior o posterior a las fechas de la restricción.

RestriccionPorSemana: Esta clase define los métodos abstractos de la padre y mediante los métodos heredados para saber si es fin de semana restringe la clase si lo es.

RestriccionPorFinDeSemana: Esta clase define los métodos abstractos de la padre y mediante los métodos heredados para saber si es fin de semana restringe la clase si no lo es.

RestriccionCombinada: Esta clase tiene un atributo List<RestriccionTemporal>, restricciones, dónde mediante el método de agregarRestriccion, agrega las restricciones que se deben cumplir.

**Strategy. RecomendacionDeDesafio.**

Este patrón se compone de las dos estrategias concretas, Favorito y PreferenciasDeJuego, la clase TipoDeRecomendación que indica la interfaz a cumplir por las estrategias concretas y la clase RecomendaciónDeDesafio que hace uso de las dos estrategias según cual se opte como estrategia a emplear.

PreferenciasDeJuego: Esta estrategia devuelve cinco desafíos los cuales serán los desafíos con mayor coincidencia respecto a los valores definidos entre los desafíos y el perfilUsuario en los atributos ‘cantidadDeMuestrasARecolectar’, ‘dificultad’ y ‘recompensa’.

Favorito: Esta estrategia devuelve cinco desafíos los cuales se consiguen tras obtener los 20 desafíos con mayor coincidencia para un ‘perfilUsuario’ determinado y procesar a dichos desafíos para obtener a los cinco desafíos con mayor similitud entre ellos según los valores de ‘cantidadDeMuestrasARecolectar’, ‘dificultad’ y ‘recompensa’ posea un desafío determinado que en este contexto será el favorito del usuario.

RecomendaciónDeDesafio:

Esta clase tomaría el rol de ‘Contexto’ en el patrón Strategy aplicado para la obtención de desafíos que serán destinados a una instancia de clase Usuario.

La clase tendrá como atributo a una instancia de ‘TipoDeRecomendación’, mas concretamente a una de las posibles subclases que hereden de dicha clase, cuyo tipo denotará la forma o preferencias a tener en cuenta para obtener los desafíos a entregar a los usuarios.

También posee un atributo destinado al sistema al que se asocia, del cual obtendrá todos los desafíos disponibles a procesar según el tipo de recomendación indicada por un setter.

El método que se encarga de brindar una lista de desafíos debe recibir un valor de tipo Usuario como argumento del cual se obtiene los valores en los que basarse para optar por los desafíos correspondientes. También se utiliza a la instancia Usuario para descontar de los desafíos traídos por sistema, todos los desafíos que ya posea el usuario al consultar por sus desafíos aceptados para evitar la posibilidad de brindar un desafío repetido para el usuario.

TipoDeRecomendación. Clase Abstracta:

Esta clase tomaría el rol de ‘Estrategia’ en el patrón Strategy implementado, poseyendo la interfaz que deben poseer los distintos algoritmos a implementarse en base a esta clase.

Como método obligatorio a poseer para ser una clase que implementa a ‘TipoDeRecomendación’ este sería el mensaje ‘desafiosRecomendados’ que debe recibir como parámetro al usuario en que se basa para procesar a los desafíos según sus preferencias y una Lista de desafíos que provendrá del sistema del que se obtienen todos los desafíos a procesar.

Dentro de sus métodos para procesar a los desafíos que son utilizados por las dos estrategias concretas definidas son:

-**desafiosConCoincidencias** – Recibe un perfilUsuario y una lista de desafíos, utilizando los valores almacenados de ‘Cantidad de muestras a recolectar’, ‘dificultad’ y ‘recompensa’ que se encuentran tanto en desafío como en el perfil para calcular las coincidencias en un método aparte y así asociar dicho valor de coincidencia a cada uno de los desafíos conformando un LinkedHashMap donde la clave es el ‘desafio’ y el valor de coincidencias el valor asociado, siendo el mapa lo que se devuelve de resultado.

-**obtenerCoincidencia** – Recibe un ‘desafío’ y un ‘perfilUsuario’ como parámetros con los que calcula el valor de coincidencia entre ambos. El cálculo consiste en una sumatoria entre los valores a obtener como resultado de la diferencia de los atributos que radican en ambas clases que son ‘Cantidad de muestras a recolectar’, ‘dificultad’ y ‘recompensa’. Se devuelve el resultado de dicha sumatoria que es utilizado en el método ‘desafiosConCoincidencias’.

-**ordenarDesafios** – Recibe un Map donde las llaves son de tipo ‘Desafío’ y el valor asociado es de tipo ‘float’. En este contexto el ‘float’ para cada desafío representaría el valor de coincidencia o el valor de similitud como es en el caso de uno algoritmos implementados.

El método se encarga de ordenar de menor a mayor según el valor asociado a cada uno de los desafíos siendo que en cuanto menor sea dicho valor, tendrá un mayor nivel de coincidencia/similitud con el usuario en el que se baso el cálculo.

La devolución será un LinkedHashMap que permite que al ordenar los valores se mantenga dicho orden deseado para cada par de llave-valor.

-**primerosDesafiosARecomendar** - Recibe un LinkedHashMap donde las llaves son de tipo ‘Desafío’ y el valor asociado es de tipo ‘float’, y por otro lado un int que indica una cantidad de desafíos a requerir del LinkedHashMap a procesar.

El método devuelve una lista de desafíos a obtener tras iterar por el LinkedHashMap recibido compuesto por la cantidad de desafíos indicada en el parámetro o en caso de no haber suficientes desafíos la lista contendrá todos los que se encontraban como llave en el Map.

PreferenciasDeJuego:

Esta clase tomaría el rol de Estrategia Concreta en el patrón Strategy implementado siendo una subclase de ‘TipoDeRecomendación’ y siendo uno de los algoritmos que podría emplear ‘RecomendaciónDeDesafio’ como clase con el rol de ‘Contexto’.

El método que debe emplear al ser subclase ‘desafiosRecomendados’ que recibe una instancia de Usuario y una lista de desafíos procesa a todos los desafíos de forma que asocia a todos los desafíos con su respectivo valor de coincidencia calculada en un LinkedHashMap. Tras esto se ordenan los desafíos con el método ‘ordenarDesafios’ para dejar a los desafíos con mayor coincidencia con el usuario de primeros y después se toma los 5 primeros desafíos que serán los que se entregarán al usuario cuando los solicite.

Favorito:

Esta clase toma el rol de Estrategia Concreta en el patrón Strategy implementado siendo una subclase de ‘TipoDeRecomendación’ y otro de los algoritmos posibles a emplear por ‘RecomendaciónDeDesafio’.

Este algoritmo aparte del método que hereda de la super clase posee el método ‘desafiosAsociadosASimilitud’ que en base a una instancia de ‘Usuario’ dado y una lista de desafíos devuelve un LinkedHashMap con cada uno de los desafíos asociados a un valor de similitud calculado. Este valor es obtenido mediante un cálculo entre el desafío a evaluar y el desafío que se considera el favorito del usuario, tomando los atributos ‘Cantidad de muestras a recolectar’, ‘dificultad’ y ‘recompensa’ de ambos y sumando sus diferencias para después dividir dicha sumatoria por tres para obtener el promedio. Debido a la posible similitud entre distintos desafíos al devolver un entero que redondea el resultado del cálculo, se optó por la devolución de un valor ‘float’ para una mayor exactitud, lo cual afecta al otro algoritmo implementado que no requiere de dicho ‘float’ para sus calculos pero no representa una alteración en su funcionamiento.

El método ‘desafiosRecomendados’ que hereda y recibe de parámetro un ‘Usuario’ y una lista de desafíos funciona de la siguiente manera:

-Primero obtiene los desafíos asociándolos a su coincidencia correspondiente, que se calcula en base a las preferencias en el perfil del usuario y los desafíos, en un LinkedHashedMap para proceder a ordenarlo para obtener a los de mayor coincidencia al comienzo para luego tomar los 20 primeros desafíos en una lista tras esto.

-De los 20 desafíos obtenidos se procede a asociarlos con su valor de similitud respecto al desafío que más le gusto al usuario iterando por toda la lista para crear un LinkedHashMap con cada desafío asociado a su valor correspondiente.

-Después de esto, se ordena los desafíos en el Map según cual posea el mayor nivel de similitud respecto al desafío favorito, para luego devolver en una lista solamente los cinco primeros desafíos que irían destinados al usuario.