Trabajo Práctico Final
Informe de Proyecto

Ciencia Participativa y Juegos

UNIVERSIDAD NACIONAL DE QUILMES.

DEPARTAMENTO DE CIENCIA Y TECNOLOGIA.

TECNICATURA EN PROGRAMACIÓN INFORMÁTICA.

PROGRAMACIÓN ORIENTADA A OBJETOS II.

PROFESORES: DIEGO TORRES, DIEGO CANO, MATIAS BUTTI.

ESTUDIANTES:

Sebastian Quaglia: sherlock-ezequiel@hotmail.com - **Nro. de Legajo: 45544.**

Ivan Gomez: Ivangonezunq@gmail.com - **Nro. de Legajo: 32355.**

Federico Slavinchins: FedericoSlavinchins@gmail.com - **Nro. de Legajo: 51504.**

FECHA LIMITE DE ENTREGA: 06/11/2022, 23:59hs.

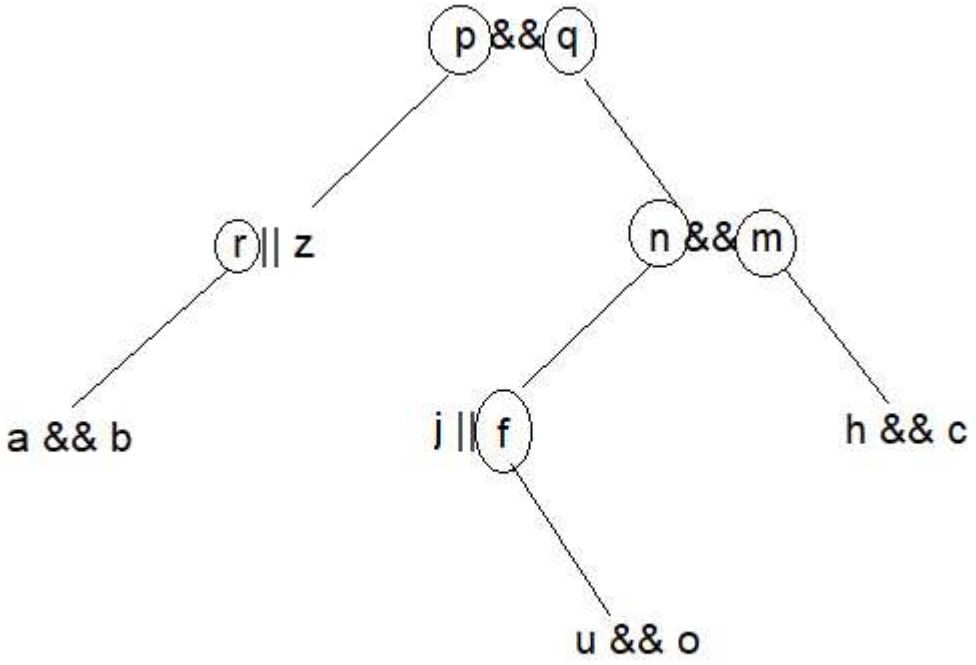
Decisiones de Diseño

Funcionalidad de Buscador de proyectos

Para desarrollar la funcionalidad de Buscador de Proyectos, se realizó una jerarquía de clases. La super clase **BuscadorDeProyectos**, implementa la interface *FiltroDeBúsqueda*, que le provee el protocolo que debe conocer para cumplir su propósito, el cuál es filtrar sobre una lista de proyectos. En esta funcionalidad se detectó que se podía aprovechar el *patrón de diseño composite* ya que el desarrollo de esta funcionalidad que se planteó posee una estructura recursiva. La clase BuscadorDeProyectos es el elemento Component del patrón.

De ella extienden BuscadorSimple, Negacion y BuscadorCompuesto:

BuscadorCompuesto es el elemento Composite de esta funcionalidad. De ella extienden BuscadorAnd y BuscadorOr, que aplican los criterios de filtro de acuerdo a la lógica de cada uno. BuscadorCompuesto posee la estructura recursiva ya que opera sobre dos términos (también llamados buscadores en este caso), siendo un operador binario. La recursividad radica en que cada uno de esos términos/buscadores se extienden de la superclase BuscadorDeProyectos, por lo que pueden ser tanto BuscadorSimple, Negación o nuevamente BuscadorCompuesto, lo que siendo este último caso, continúa la recursividad.



Como se puede observar en la imagen los términos/buscadores p, q, r, n, m, y f, son buscadores compuestos, que a su vez, en el caso de r, n , m y f, están dentro de otros buscadores compuestos. Gracias al polimorfismo que nos provee la programación orientada a objetos, todos comprenden el método filtrar , el cual retorna los resultados de búsqueda, un *ArrayList* de la clase Proyecto.
Las subclases **BuscadorTítulo, IncluidorDeCategorias, ExcluidorDeCategorias y Negacion**, son los buscadores que detienen la recursividad, por lo que son el elemento Leaf de este patrón. En el dibujo mostrado anteriormente serían los buscadores z, a, b, j, h, c, u , y o.
Para la clase BuscadorSimple se añade el *patrón de diseño Template Method* ya que las subclases del mismo repetían código, y en lo único que cambiaba era en la forma en que aplicaban el criterio de selección de proyectos (dependiendo de la naturaleza de selección de cada BuscadorSimple [BuscadorTítulo, ExcluidorDeCategorias, IncluidorDeCategorias]), por lo que el método "filtrar" sobre una lista de proyectos del BuscadorSimple es el template method en este caso, y el método concreto que es aplicado y que le brinda comportamiento las subclases es el de "aplicarCriterioDeFiltro" sobre un proyecto.

La aplicación del Template Method proporciona una ventaja y es que deja abierto al sistema y a nosotros como desarrolladores del mismo, en que si el día de mañana se desea agregar un buscador nuevo, solo se debe 'codear' el método "aplicarCriterioDeFiltro", y no todo el método "filtrar" de la forma en que se hacía anteriormente a aplicar el Template Method.

Por otro lado, el buscador de Negación no forma parte del BuscadorSimple y del template method, ya que posee una estructura un poco diferente, y debe recibir una lista de proyectos a filtrar y otra a sacar (resultado del buscador a negar), mediante un foreach que quita elementos deseados, se obtiene el resultado de búsqueda de la negación, lo cual es la funcionalidad de negación de términos solicitada en el trabajo.

Dicho buscador, opera sobre otro buscador, el cual es el que va a negar, por lo que al igual que el buscador compuesto emula los operadores lógicos and y or que son operadores binarios, el buscador de Negación funciona de manera similar al "not" de lógica que es un operador unario (opera sobre un término/buscador que va a negar, el cual está contenido en su variable interna desde el momento en que se instancia el buscador de Negación).

El elemento Contexto del patrón y que utiliza el buscador de proyectos, es la clase Sistema.

Uso de la Restricción Temporal

A la hora de realizar la restricción temporal, nos surgió la duda acerca del diseño que debíamos implementar. Decidimos plantearlo con una jerarquía de clases debido a que:

Si la **RestriccionTemporal** era una clase única, teníamos la desventaja de que en el caso de que la restricción sea solo un bloque del día de la semana, debíamos instanciar como *null* las dos fechas de **EntreFechas**.

De la manera en que lo realizamos, con la jerarquía de clase, tenemos la ventaja de que si hay que instanciar solo el bloque de día la semana, no es necesario poner las dos fechas en *null*.

Luego creamos una clase que implementa RestricciónTemporal al igual que las otras hermanas, que representa la combinación entre bloque semanal (restricción fin de semana o día de semana) junto con restricción entre fechas.

Aplicación de los Estados del DesafioDeUsuario

Hicimos uso del patrón State al ver la necesidad de los cambios de estado que debian tener los desafíos del usuario, ya que estos pueden estar NoAceptado, Aceptado o Completado. Estos estados son parte de las ConcreteState subclasses en el patrón.

La primer implementación fue una Interface **EstadoDesafio** que contenía los métodos *aceptar()* y *completar()* y cumple la función de State dentro del patrón.

Pero la desventaja que vimos con esta construcción es que según el estado en el cual estuviera el desafió, algunos métodos no tendrían comportamiento.

Por ejemplo *completar()* en el estado de NoAceptado.

Actualmente lo modificamos y decidimos construir un único método llamado ***actualizarEstado(DesafioDeUsuario desafioDeUsuario)***. Esto trae la ventaja de mantener el polimorfismo y que no haya métodos sin comportamiento.

El Contexto del patrón y que aplica los cambios del estado es la clase DesafioDeUsuario.

Uso de los algoritmos de recomendación en Usuario.

Al encontrarnos con la necesidad de tener varios algoritmos de recomendación de desafíos para el usuario implementamos el patrón Strategy en nuestro sistema.

Si bien nuestra implementación actual del patrón se encuentra incompleta, nuestra idea para el mismo era implementar una clase llamada **RecomendadorDeLudificacion**, la cual cumpliría el rol de Contexto en el patrón e instanciaria los cambios de las estrategias de recomendación que el usuario le pidiera. La misma debe recibir datos básicos del usuario para operar y permitir que las estrategias usen esos datos para configurarse completamente.

Implementando la Interface **EstrategiaDeRecomendacion** que cumple la función de Strategy, las estrategias implementarían dos algoritmos de filtrado de desafíos distintos, para así darle variedad de elección al usuario.
Los algoritmos son instancias de dos clases distintas, una siendo **RecomendacionPorPreferencias**, la cual recibiría las preferencias del usuario y filtraría los desafíos para recomendarle al usuario aquellos mas acordes a su gusto. Luego esta la clase **RecomendaciónPorFavoritos**, que aplicaría un filtro de desafíos mas estricto, ya que ademas de encontrar los mas acordes a su gusto, también deberá filtrar estos por su parecido con el desafió mas favorito del usuario y así recomendar desafíos mas precisos para cada usuario. Estas dos clases ocupan el rol de ConcreteStrategyA y ConcreteStrategyB.

Uso de DesafioDeUsuario como clase de rastreo de progreso para los desafios

Para poder hacer el "tracking" o seguimiento de progreso del usuario en el desafio, decidimos realizar una clase aparte a Desafio, llamada DesafioDeUsuario. De esta forma:
Desafio: constituye la estructura y características del desafío.
DesafioDeUsuario: Realiza un seguimiento del progreso del desafío por parte del usuario.

Calculador de distancias
Dentro de la clase Ubicacion se implemento un medidor de distancias basado en metros. Este metodo se llama *distFrom()*, el cual toma las latitudes y longitudes de dos ubicaciones y coteja la distancia que existe entre ambas basada en metros.
Este metodo fue extraido de: https://www.iteramos.com/pregunta/26930/calcular-la-distancia-en-metros-cuando-se-conoce-la-longitud-y-latitud-en-java

Funcionalidad de validador de muestra:
Decidimos crear la clase ValidadorDeMuestra para cada DesafioDeUsuario, ya que descubrimos en un momento que la clase Usuario estaba absorbiendo la responsabilidad de decidir si contar la muestra (o no) para un desafío aceptado. Por lo cual descubrimos la idea de implementar un ValidadorDeMuestra que se encargue de indicar si la muestra contabiliza o no para cada desafío de usuario aceptado.

Aplicación de Sistema:

Implementamos una clase llamada Sistema, la cual serviría como Contexto para nuestro Buscador. Esta clase busca emular la aplicación del proyecto del trabajo y quien serviría como primer punto de contacto para el usuario. Esta clase conoce los proyectos de ciencia participativa y los usuarios de los mismos.

Problemas durante el desarrollo

Al modificar el archivo .classpath, nos fue imposible a todos los integrantes hacer un Push al repositorio debido a que el archivo modificado fue traido a los repositores locales de cada miembro. Esto nos provoco un retraso de tres dias durante la ultima semana. Este problema tuvo que ser consultado con los profesores (Matias Butti) y ayudantes de la materia (Fabrizio Britez), quienes nos pudieron guiar para encontrar la solución del problema. Esta solución fue agregar los archivos .classpath y .project al archivo .gitignore, con la idea de que de esta manera cada miembro pudiera trabajar con su propio archivo personal.