Trabajo Práctico Final
Informe de Proyecto
(Con correcciones de re-entrega)

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 LÍMITE DE ENTREGA: 06/11/2022, 23:59hs.
FECHA LÍMITE DE RE - ENTREGA: 4/12/2022, 23:59

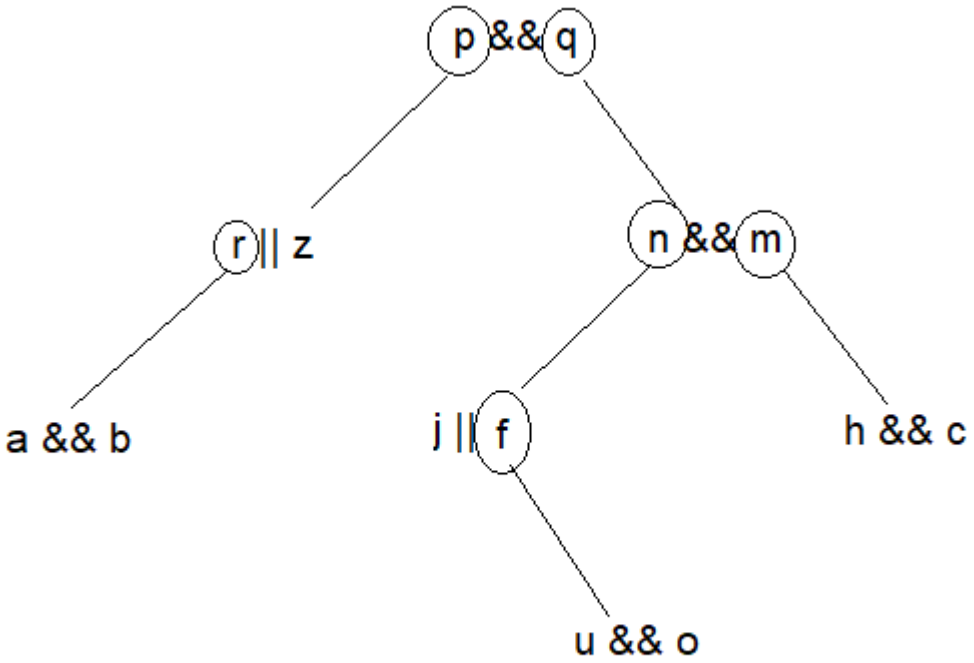
Decisiones de Diseño

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.

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. Esto fue cambiado en favor de una Clase Abstracta para hacer uso de la herencia en algunos metodos que se compartian entre si.

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

Interrupciones durante el transcurso del proyecto

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.

Durante el transcurso del trabajo nos encontramos con problemas para trabajar, debido a que a uno de los miembros se encontró sin luz por un intervalo de 4 días, y otro participante tomo la decision de abandonar el proyecto sin notificarnos.

Falta de aplicación de test con mockito

Por falta de tiempo a partir de las causas nombradas algunas clases de tests tuvieron que dejarse sin aplicar por completo mockito, ya que el retraso temporal que esto conllevaría era demasiado riesgoso. No obstante, se aplico mockito en muchas clases del proyecto sin inconvenientes.

CORRECCIONES REALIZADAS EN BASE A LA DEVOLUCIÓN DE LOS DOCENTES:

Correcciones de devolución:

1) **No corresponde realizar corrección**.

2) **Realizado**. El coverage de test del código en el 'src' ahora alcanza un 96,3%, superando el 95% solicitado.

3) **Realizado**: *Fue incluido en la entrega anterior* (está en la página anterior a esta).

4) **No corresponde realizar corrección**.

5) **Corrección realizada**. Se eliminaron muchos getters y setters innecesarios. A tener en cuenta que muchos de ellos eran de implementaciones antiguas del código y por ende

quedaron allí sin usar. Sin embargo se trato de eliminar lo mayor posible. Algunos getters y setters fueron creados para los tests.

6) **Corrección realizada**.

7) **Corrección realizada**.

8) No entendí bien como realizarlo, se consultó a los docentes pero aún así no me quedó bien en claro de qué manera se debía implementar. Aún así el código es funcional.

9) **Corrección realizada**. Se agregó el patrón composite para que se pueda restringir diferentes períodos de entre fechas. También modifiqué los nombres para que sean mas descriptivos.

Participantes del patrón composite:

RestriccionDeEntreFechas: es el elemento component.

RestriccionDeEntreFechasCompuesta: es el elemento composite.

RestricciónDeEntreFechasSimple: es el elemento leaf.

También se agregó otro patrón composite en la misma funcionalidad de restricciones para combinar diferentes tipos de restricciones (entre fechas y bloques semanales):

RestriccionTemporal: es el component.

RestricciónTemporalCompuesta: es el composite.

RestriccionDeEntreFechasSimple, RestriccionDeFinDeSemana, RestriccionDeDíasDeSemana son los elementos leaf.

Importante: esta implementación tiene un error en la implementación, y es que si una RestricciónTemporalCompuesta conoce y contiene a las siguientes restricciones: RestriccionDeDíasDeSemana y RestriccionDeFinDeSemana, cuando se le pase como parámetro una fecha de finalización de un desafío de usuario, la misma siempre va a

retornar que no cumplió la restricción, ya que un día no puede ser día de semana o día de fin de semana al mismo tiempo. Queda como deuda técnica ya que no llego con el tiempo a resolverlo.

10) **Corrección realizada**. Se creó la clase solicitada para no tener todo acoplado en BloqueSemanal. BloqueSemanal fue eliminada y reemplazada por clases mas descriptivas y que aprovechan mas las ventajas de tener diferentes clases de objetos.

11) **Corrección realizada**. Ahora se utilizan los métodos para corroborar que un desafío cumpla con las condiciones de porcentaje de realizado para pasar de estado.

12) ***Realizado en la primer entrega***.

13) No llegamos con el tiempo a realizarlo. Sin embargo se entiende la necesidad de corregirlo y queda como deuda técnica. Ya que para hacerlo se deberían realizar muchos cambios y volver a testear, y no alcanzaría el tiempo.

14) **Corrección realizada**. Se detalla abajo los cambios realizados:

- Se creó la clase **MenuDeDesafios** que tiene como responsabilidad funcionar como "ventana" u organizador de desafíos del usuario, quitándole dicha responsabilidad. A la hora de organizar los desafios en disponibles, aceptados y completados, lo realiza esta clase que la conoce el Usuario como colaborador interno. Esta clase tiene la tarea de mover los desafíos en sus respectivos lugares determinados por la organización del mismo, y que son representados por los ArrayLists que antes tenía el Usuario: desafíos disponibles, aceptados y completados.
- Se creó la clase **MenuDeProgreso** para quitarle la responsabilidad al Usuario de consultar y verificar su progreso en desafíos. De esta manera el menu de progreso funciona de manera similar a una "ventana" o consultor de progreso del usuario en

sus desafíos. El Usuario conoce a una instancia de la clase MenuDeProgreso como colaborador interno.

- Se creó la clase **ValoradorDeDesafíos** para quitarle la responsabilidad al Usuario de votar los desafíos.
- Se eliminaron varios métodos que pasaron a otras clases, según sugerencias de los docentes, tales como aceptar y completar desafíos.

Correcciones de UML:

1) No aparece nada en la corrección, no hay nada para corregir aquí.

2) El UML de la primer entrega presentaba todas las cardinalidades. **Tal vez por error se hayan fijado en la carpeta Diseño UML que contiene versiones obsoletas del diagrama del proyecto que quedaron en el repositorio como muestra del proceso del trabajo realizado**. Por favor fijarse el UML que se encuentra en https://github.com/FedericoSlavinchins/unq-po2-tpFinal . Aún así subimos el de la re - entrega corregido según las correcciones , el mismo se va a llamar **UML Proyecto Final Entrega 4-12** (<--- Este es el UML correspondiente a la re entrega)

3) **Corrección realizada**.

4) **Corrección realizada**.

5) **Corrección realizada**. Fue un error involuntario.

6) **Corrección realizada**.

Correcciones de Implementación:

7) **Corrección realizada**.

8) **Corrección realizada**.

9) No se llegó con el tiempo a realizarlo.

10) **Corrección realizada**.

11) **Corrección realizada**, sin embargo luego se necesitó que el Usuario guarde la instancia del proyecto al que se agregó como participante, ya que es utilizado por el método Usuario.desafiosDeMisProyectos(). Aún así es el proyecto quien mantiene la responsabilidad de agregar al participante.

12) **Corrección realizada**.

13) **Corrección realizada**.

14) **Corrección realizada**. Ahora se utilizan excepciones.

15) **Corrección realizada**.

Correcciones de Tests:

16) **Corrección realizada**. Se borró esa clase tests porque estaba vacia y sin implementar y en un paquete incorrecto.

17) **Corrección realizada**.

18) **Corrección realizada**.

19) **Corrección realizada**.

Corrección en base a observaciones:

- Se utilizó la API de Stream en la corrección.
- De todos los tests solo falla uno, el restó corre de forma correcta.
- Se utilizó mockito y test doubles en todo lo que se pudo, sin embargo no se llegó con los tiempos para poder hacerlo en todos los tests. Queda como deuda técnica.
- Se arregló el UML.