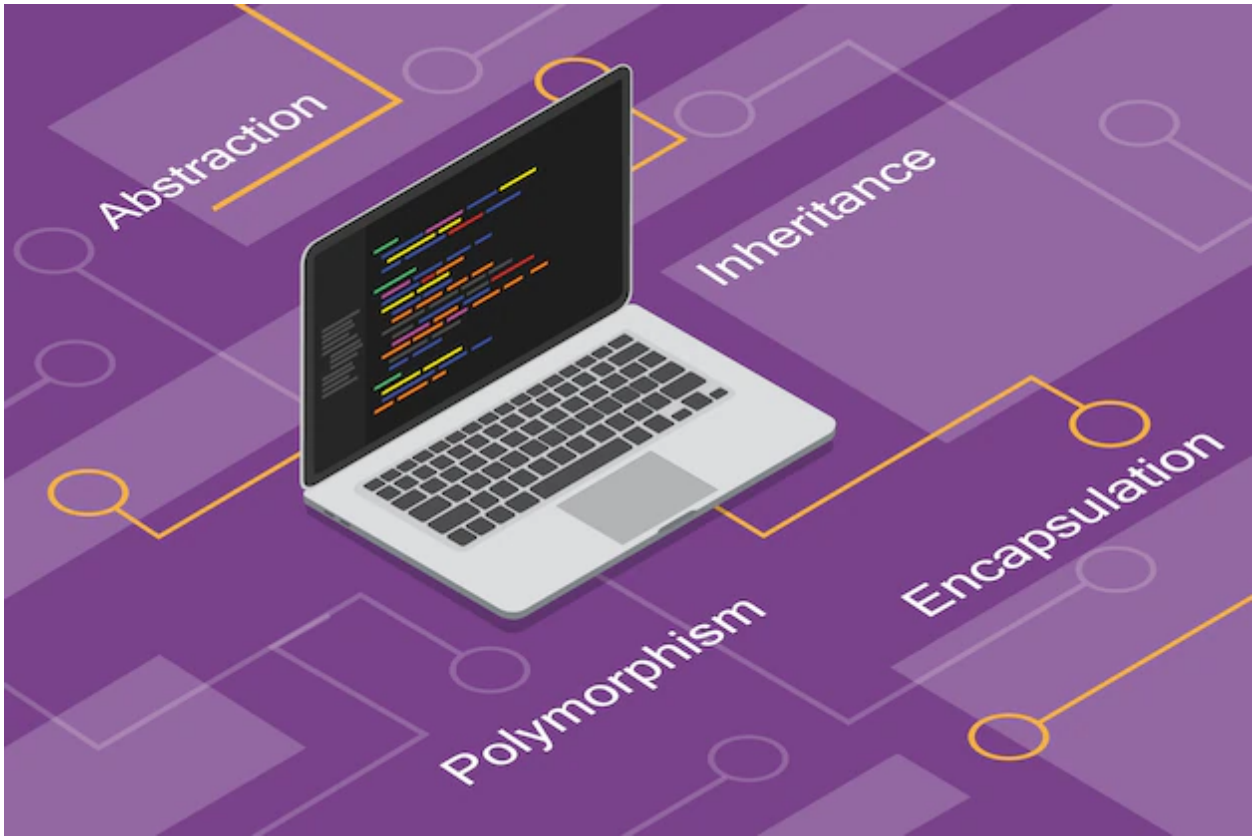

Ciencia Participativa y juegos

Programación con Objetos 2

—

Pezzola Antonella
antopezzola@gmail.com

Jara Julieta
jarajuli13@gmail.com

Decisiones de diseño

En principio, modelamos las clases principales del enunciado Proyecto, Usuario, Muestra y Desafío. Luego, modelamos las clases que se necesitaban para la comunicación entre las clases principales, como DesafioDeUsuario. El comportamiento de esta clase se basa en actualizar y registrar cuando un usuario agrega una muestra. Es decir, cada vez que el usuario agrega una muestra, el DesafioDeUsuario se encarga de verificar si es una muestra válida para el desafío, en caso de que lo sea agregarla a la lista de muestrasObtenidas y actualizar el Estado si corresponde.

Seguidamente, decidimos hacer la clase abstracta Restricciones. El comportamiento de esta clase con sus subclases es verificar si la muestra dada es válida, de acuerdo con las restricciones que poseía el desafío dado. Por ejemplo, la clase RestriccionArea, verifica si la muestra se realizó en la misma área que el desafío permitía. Asimismo, modelamos la clase Área, la responsabilidad de esta clase es realizar el cálculo para ver si la coordenada geográfica de la muestra coincide con la RestricciónÁrea del desafío.

Por consiguiente, creamos una clase Sistema en la que se da información requerida a las demás clases, como recomendarDesafíos al Usuario, seleccionar Proyectos de interés, entre otros. Para ello, creamos una clase abstracta Filtro con sus subclases particulares de filtro. El comportamiento que le dimos, fue filtrar todos aquellos proyectos que cumplían con ciertas preferencias. Asimismo, también filtrar por conectores lógicos (and, or y negación). Asimismo, para recomendar desafíos al usuario, el usuario debe seleccionar la estrategia que desea para la búsqueda de desafíos. Por esa razón, creamos la clase abstracta Recomendación con sus subclases Favoritos y PreferenciaDeJuego. Cada subclase busca desafíos de preferencia para un usuario dado.

Detalles de implementación

Es importante destacar que, al momento de realizar la implementación de la clase abstracta EstrategiaDeSeleccion decidimos crear una clase llamada Pares, la cual su comportamiento consiste en guardar un Desafio y una coincidencia. Este atributo coincidencia es, por decir de alguna manera, el puntaje que se le da al Desafío para el usuario asignado, es decir si su coincidencia es 1, es probable que

ese desafío sea recomendado al usuario en cuestión. En caso de que sea un número mayor significa que la preferencia de usuario posee mayor diferencia con el Desafio.

Asimismo, otro detalle de implementación fue la resolución de la clase abstracta Filtro. Esta clase posee tres subclases, la cual, una de ellas es FiltroCompuesto, esta clase es el composite del Patrón Composite que utilizamos para dar solución a este problema. Pero, a su vez, FiltroCompuesto es una clase abstracta y sus subclases pertenecen todas al participante Composite.

Patrones de diseño y roles

En primer lugar, el Patrón de diseño que utilizamos fue el Patrón Composite. Este patrón lo utilizamos en dos clases. Por un lado, la clase Restricciones. Esta clase posee las subclases RestriccionArea, RestriccionDia, RestriccionFecha o Restricción Compuesta. Está última, es el composite en esta estructura, en donde, dentro de ella tiene una lista de Restricciones.

Los roles que cumplen son:

- ➔ **Component:** Restricciones.
- ➔ **Leaf:** RestriccionArea, RestriccionFecha, RestriccionDia.
- ➔ **Composite**: RestriccionCompuesta.
- ➔ **Client:** Desafío.

Por otro lado, se volvió a utilizar este patrón en la clase Filtro. Asimismo, la diferencia que posee con la clase mencionada anteriormente es que la clase composite (FiltroCompuesto) es una clase abstracta y dentro de ella se encuentran tres subclases composite (FiltroAnd, FiltroOr, FiltroNegacion).

Los roles que cumplen:

- ➔ **Component:** Filtro.
- ➔ **Leaf:** FiltroTitulo, FiltroCategoria.
- ➔ **Composite**: FiltroCompuesto.
- ➔ **Client:** Sistema.

En segundo lugar, se utilizó el Patrón State. Este patrón se utilizó en la clase DesafioDeUsuario. Esta clase se comporta diferente dependiendo del estado en el que se encuentre. Para ello, modelamos la

clase Estado con sus subclases finitas de estado y sus diferentes comportamientos tal como es la solución del patrón State.

Los roles que cumplen:

- **Context:** DesafioDeUsuario.
- **State:** Estado.
- **ConcreteState**: EstadoPendiente, EstadoAceptado y EstadoCompleto.

En tercer lugar, se utilizó en Patrón Strategy. Este patrón se utilizó al momento de modelar la recomendación de desafíos a un usuario . El usuario posee un atributo con la estrategia que quiere utilizar para dicha recomendación.

Los participantes en este patrón son:

- **Strategy:** Recomendación.
- **ConcreteStrategy:** PreferenciaDeJuego y Favoritos.
- **Context:** Usuario.

Finalmente, utilizamos el Patrón Template Method. Este patrón lo utilizamos en varias clases. Por un lado, lo utilizamos en la clase FiltroCompuesto.

Los participantes en este patrón son:

- **AbstractClass:** FiltroCompuesto.
- **ConcreteClass:** FiltroAnd, FiltroOr y FiltroNegación.

Asimismo, lo utilizamos en la clase Estado definiendo por default los métodos de sus subclases con el fin de que la subclase correspondiente redefina el método en caso de ser necesario.

Los participantes en este patrón son:

- **AbstractClass:** Estado.
- **ConcreteClass:** EstadoPendiente, EstadoAceptado y EstadoCompleto.

De igual manera, fue utilizado en la clase Restricciones definiendo por default el método agregarRestriccion(Restricciones) con el propósito de que se redefina la clase composite.

Los participantes en este patrón son:

- **AbstractClass:** Restricciones.
- **ConcreteClass:** RestriccionArea, RestriciconFecha, RestriccionDia y RestriccionCompuesta.