Materia: Programación con Objetos II

Trabajo Practico Integrador

A LA CAZA DE LAS VINCHUCAS

Profesores: Butti Matias, Cano Diego, Torres Diego

Alumnos:

- Cirilli Ignacio
- Greco Guido

Mails:

- ignaciocirilli@hotmail.com
- guidogreco25@gmail.com

Fecha de entrega: 15/06/2023

Decisiones de diseño.

El mensaje que tiene la clase Muestra llamado *resultadoActual()* (que devuelve un TipoDeOpinion) no se realiza con State debido que se considera que no podría ser posible que escale. Este mensaje puede ser extendido y planteado como un patrón State pero en el código se resuelve con una declaración condicional (*if*) debido a que se considera que no puede llegar a haber más resultados que los planteados en el enunciado.
Si, en un futuro, se considera que esto se puede extender, esto puede llegar a quedar como una deuda técnica.

Las flechas de uso (*use*) son utilizadas ya que aportan legibilidad, permiten que las clases estén conectadas y así mismo, que una parte del diagrama UML no quede desconectado del resto.
Esto sucede, por ejemplo, de la clase Pagina hacia la clase Busqueda. La clase Pagina necesita a la clase Busqueda solamente para realizar el mensaje *filtrarMuestras()* ya que este recibe una Busqueda como parámetro.
Esta parte del diagrama puede verse resuelta sin la flecha de uso (*use*), no conectando dichas clases, que la clase Busqueda (y el patrón Composite de cual forma parte) quede desconectado del resto del diagrama y que la única manera que se conozcan sea cuando se invoca el mensaje mencionado.
En el caso de la flecha de uso utilizada en las clases EstadoVotada y EstadoVerificada, que se utiliza conectandolas con el tipo enumerativo ENivelDeVerificacion, tienen el mensaje *nivelDeVerificacion()* que retorna el tipo enumerativo mencionado y que no se la puede enlazar de otra manera porque no tienen otro tipo de relación. Nuevamente, este tipo enumerativo puede quedar aislado del diagrama pero por cuestiones de legibilidad y comprensión se hace uso de esta flecha.

Si bien el tipo enumerativo ENivelDeVerificacion parece no tener mucho sentido, aporta legibilidad y es usado por varias clases del sistema. Ésta es una de las principales razones por las que se mantuvo la misma en el sistema. Una observación a destacar es que este tipo enumerativo puede ser reemplazado por simplemente un tipo String pero, como se mencionó, se mantiene ya que aporta comprensión al diagrama y al código.

Detalles de implementación.

En ambos State distribuidos por el diseño, las clases que tienen el rol de State conocen a través de su constructor a quien es su Context. En el caso del State/Muestra la clase EstadoDeMuestra (State) conoce a Muestra (Context), y por el otro lado en State/Participante, la clase NivelDeConocimiento (State) conoce a Participante (Context) a través de lo mencionado anteriormente.

En caso de no realizarse esto, se debería pasar el Context en cada uno de los mensajes del State.

Para seguir con detalles de implementación de State/Participante, el mensaje *esExperto()* sería como un hook del template method, debido a que solo deben sobrescribir el código del mismo las subclases que deseen una implementación diferente al resto de las subclases.

Con respecto al patrón Composite, ambas clases que cumplen el rol de Composite (AND y OR) no traen consigo el típico mensaje *agregarChild()*, debido a que éstas tienen un límite de dos childs, ya que, al ser ambas operaciones de tipo booleanas, no se precisan más. De este modo basta con pasar, a través del constructor de cada clase Composite, las Busqueda que se desean realizar.

Por otro lado, se decidió realizar test para todos los tipos enumerativos del sistema. Si bien esto no es necesario, el no realizarlo bajaría el porcentaje de *coverage.* Este apartado en especial fue hablado con el Profesor Cano Diego, el cual especificó que se puede omitir aclarando que los tipos enumerativos no están contemplados en el porcentaje de *coverage*.

Para finalizar, en el diagrama pueden encontrarse varios constructores de clases que no reciben parámetros, esto se debe a que no se hace uso del constructor vacío que viene por defecto, sino que son constructores que dentro se realizan distintas instrucciones, como puede ser el caso de tener que crear una lista vacía de algún tipo específico.

Este último apartado podría colocarse también en la sección “Decisiones de diseño”.

Patrones de diseño utilizado y los roles según Gamma.

Se utilizaron 3 patrones de diseño, un Observer, un Composite y 2 State.

Con respecto al patrón de diseño Observer, el mismo se ve representado por las clases ZonaDeCobertura (Concrete Subject), Subject (Subject), IZonaDeCoberturaListeners (Observer) y Organizacion (Concrete Observer). Se identificó la presencia de este patrón de diseño debido a que a la Organizacion le interesa conocer una serie de acontecimientos que suceden dentro de una determinada ZonaDeCobertura.

En cuanto al patrón de diseño Composite, éste se ve representado en el dominio a través de las siguientes clases: Busqueda (Component), IFiltro (Leaf), AND (Composite) y OR (Composite). Éste patrón se reconoció a partir de las clases AND y OR, debido a que debe ser posible el realizar búsquedas de filtros complejas, lo que puede llegar a ser “traducido” como poder tratar a todas las búsquedas por igual y de esta manera poder componer objetos en estructura de árbol, para así poder representar una jerarquía part-whole.

Para finalizar, el primer patrón de diseño State (State/Muestra) se lo distinguió de la siguiente manera: Muestra (Context), EstadoDeMuestra (State), EstadoVotada (ConcreteStateA), y EstadoVerificada (ConcreteStateB); debido a que lo importante para la verificación de una Muestra es si dos participantes expertos coincidieron en sus votaciones. En el caso de que ésto suceda, la Muestra quedará en EstadoVerificada, por lo que ya no se le podrán realizar opiniones a la misma y esta no puede regresar a EstadoVotada. Se presupone que una muestra cuando es creada siempre tiene el EstadoVotada como estado inicial.
En relación al segundo patrón de diseño State (State/Participante), el mismo se vió representado a través de las siguientes clases: Participante (Context), NivelDeConociemiento (State), y NivelExpertoVerificado, NivelExperto y NivelBasico (ConcreteStateA, B y C respectivamente).

Con respecto a estos Concrete States, se sabe que un nivel de conocimiento NivelExpertoVerificado no puede cambiar de nivel ya que cuando un participante obtiene este nivel de conocimiento, no varía entre los otros estados.

Observación y comentarios acerca del dominio y de la cursada.

El trabajo está bien orientado ya que se utilizan todos los conceptos aprendidos en las clases dictadas. Hay varios patrones de diseño de los que se estuvieron viendo y se hacen uso de herramientas, como Mockito y los tipos enumerativos, que son temas importantes de la materia y que acoplan a los demás.

Además, el diagrama UML sirve para organizarse con el código de manera general y poder saber donde uno se encuentra parado al momento de realizar clases, hacer TDD o armar algún patrón de diseño.

Si bien al ser tan solo 2 integrantes, el desarrollo y comprensión del dominio dado, fue ameno, además que el tiempo para realizar el trabajo es el justo y necesario debido a la dificultad del trabajo. Desde ya muchas gracias por esta cursada, realmente nos dejaron con ganas de seguir aprendiendo y adentrarnos en el mundo de la programación.