Juan Ernesto Pinto

202021839

Taller 5 Patrones

Patrón elegido: Strategy Pattern

El patrón de estrategia es utilizado para implementar una familia de algoritmos. Este patrón sirve para que durante el tiempo de ejecución del programa se elija el algoritmo que se quiere usar y que no se use necesariamente el mismo algoritmo siempre. Los participantes principales de este patrón según la lectura son los objetos estratégicos, dentro de los cuales se encapsulan los diferentes algoritmos que se pueden utilizar.

Algunas ventajas que puede traer usar el patrón de estrategia son Flexibilidad y Variabilidad. Flexibilidad es una ventaja, ya que facilita que se incorporen nuevos algoritmos sin modificar el resto del código, de esta manera sigue el principio de abierto/cerrado, el cual dice que una clase debe estar abierta para extenderse, pero cerrada para modificarse. Variabilidad se convierte en una ventaja ya que permite que el objeto variar su comportamiento dependiendo de la situación que necesite solucionar. Se vuelve útil cuando a la hora de ejecución se necesita cambiar el comportamiento de la clase.

+---------------------------------------------------------------------------------------------------------------------------+

Proyecto elegido: <https://github.com/Sean-Bradley/Design-Patterns-In-Python/blob/master/strategy/client.py>

El proyecto elegido a analizar es un proyecto escrito en Python, pero con enfoque a la programación orientada a Objetos. El código implementa el patrón de Strategy Pattern ya que define una familia de algoritmos y los encapsula. Permitiendo que se elija el algoritmo apropiado en tiempo de ejecución.

El proyecto se divide en 5 clases, las cuales son:

* GameCharacter
* IMove
* Walking
* Running
* Crawling

La clase GameCharacter cuenta con un método move, en donde el cliente toma la decisión en el tiempo de ejecución de que tipo de movimiento quiere tomar. En el caso de este programa, el cliente puede elegir entre caminar correr y gatear. Lo importante acá es que en caso de querer añadir al cliente otra opción de movimiento, ejemplo saltar, se puede agregar sin afectar las otras clases y funciones, cumpliendo el principio de abierto/cerrado, pues la clase está abierta para extenderse en funciones que se le ofrecen al cliente, pero cerrada para modificar las ya existentes.

La clase IMove en este caso actúa como un contrato, ya que es la que establece el comportamiento o lo que deben seguir las estrategias concretas. Las clases Walking, Running y Crawling son las estrategias concretas y las que definen las opciones que el cliente puede tomar a la hora de la ejecución. Estas implementan la clase Interfaz de IMove y proporcionan los algoritmos específicos para cumplir con las funciones de caminar, correr y gatear.

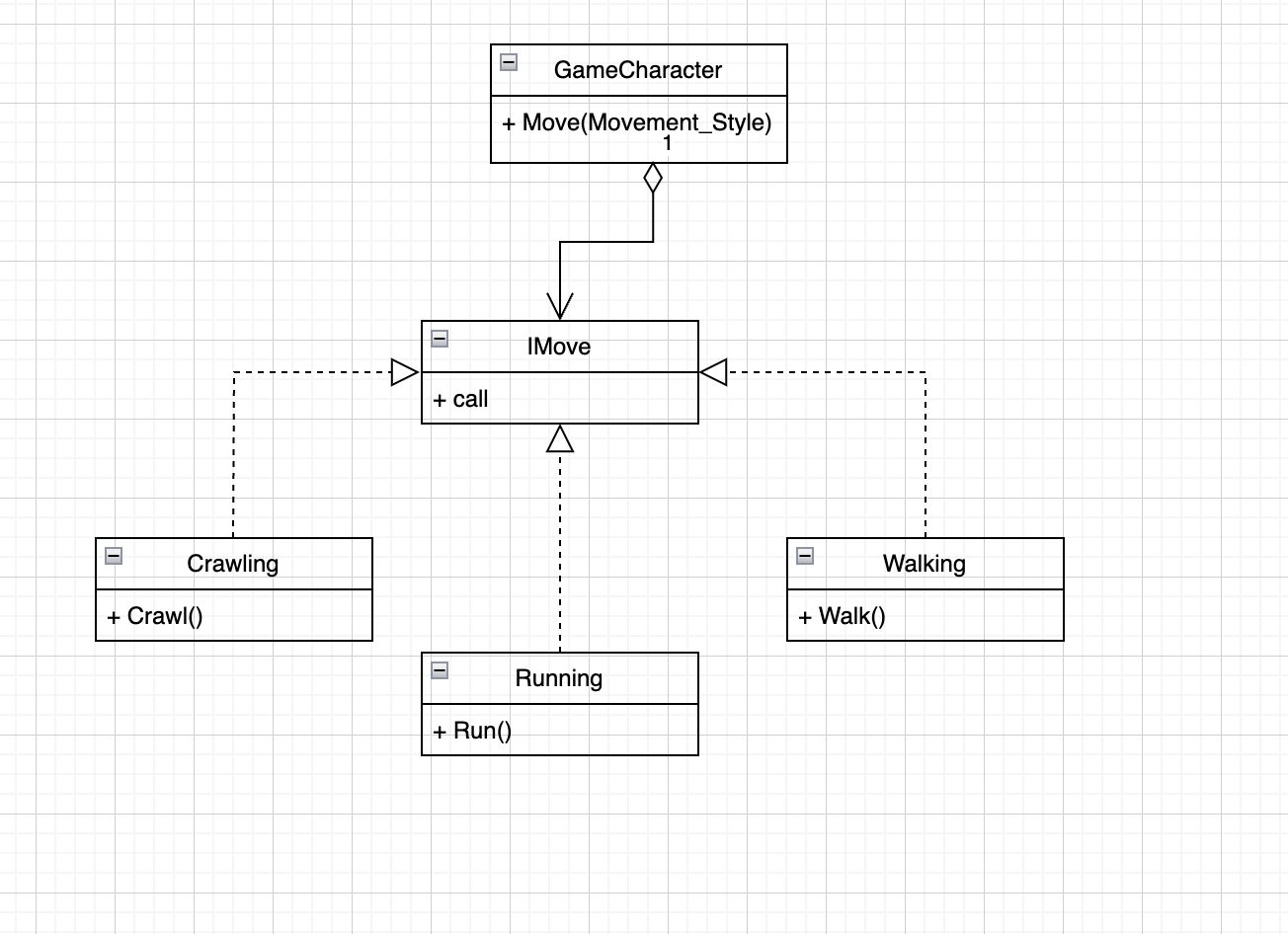
El código ejemplo tomado es una simulación de como seria el movimiento de un personaje de un videojuego, en el cual el usuario puede elegir en tiempo real el tipo de movimiento que desea que tome su personaje sin hacerle ninguna modificación a la clase personaje. Esto hace que gracias al patrón de diseño de Game Strategy.

Tiene sentido haber usado este patrón para el juego, ya que le da flexibilidad a el desarrollador de poder añadir futuras funciones a el personaje sin afectar las ya existentes. También es un código que permite facilidad para hacerle pruebas unitarias, pues cada estrategia se puede probar de manera independiente a las otras, y ninguna estrategia va a afectar a las demás. Por esto, las pruebas se van a comportar de una manera similar a como vimos en clase con el ejemplo en la calculadora, en donde se puede probar cada función por aparte y sin necesitar relacionarlas entre ellas. En cuanto a desventajas, lo único que se puede ver a primera vista de este proyecto es que al utilizar este patrón se crea una mayor cantidad de clases y con más clases y estructura se le puede estar dando una complejidad adicional al código.

Basado en lo visto en el último tema de clase, un reto que enfrenta este código es el manejo de errores, ya que, en este caso, el código asume que todo se va a utilizar correctamente, por lo cual en dado caso que el usuario haga algo inesperado, el código no va a poder manejarlo y va a terminarse. De igual manera, otro reto al que veo que podría enfrentarse es a la implementación de múltiples estrategias, ya que en el momento el código asume que solo va a haber una estrategia implementada, por lo cual el usuario solo podría elegir una acción que quiere que haga el personaje, pero en dado caso que se necesite que primero camine y después corra no se puede con este código.

En el caso de este proyecto, otra forma de haberlo implementado hubiera sido con métodos de correr caminar y arrastrarse, de esta manera se hubiera podido evitar la creación de tantas clases, sin embargo, posiblemente hubiera sobrecargado la clase GameCharacter.

El diagrama UML de este código se vería de la siguiente manera:



El personaje es representado por la clase GameCharacter. Esta clase tiene un método Move() que recibe un parámetro de tipo Movement\_Style, el cual le indica el tipo de movimiento que va a hacer (Caminar, Correr o Arrastrarse) . Las tres clases que representan y definen los diferentes tipos de movimiento son Crawling, Walking y Running.

El diagrama muestra el flujo de control para el método Move(). El método comienza llamando al método call () de la clase IMove. Luego, el método call() de la clase IMove llama al método Move() de la clase correspondiente al tipo de movimiento especificado por el parámetro Movement\_Style. Esto significa que el método Move existe en las 3 clases de movimiento y en la clase IMove que acaba siendo una clase tipo interfaz para estas otras 3, forzándolas a utilizar este método Move. Si se fuera a crear otro tipo de movimiento, debe seguir este mismo diagrama que siguen los otros 3 movimientos.

Por ejemplo, si el parámetro Movement\_Style tiene el valor Crawling, el método call() de la clase IMove llamará al método Crawl() de la clase Crawling y el personaje va a adoptar un movimiento de arrastrarse.

**Clase GameCharacter**

La clase GameCharacter representa el personaje de juego. La clase tiene los siguientes atributos y operaciones:

Atributos:

movementStyle: El tipo de movimiento actual del personaje.

Métodos:

Move(): Mueve el personaje al estilo especificado por el parámetro Movement\_Style.

**Clase IMove**

Métodos:

La clase IMove es una interfaz que define el método Move(). El método Move() recibe un parámetro de tipo Movement\_Style y devuelve un valor de tipo void.

**Clase Crawling**

Métodos:

La clase Crawling implementa el método Move() de la clase IMove. El método Move() de la clase Crawling hace que el personaje se mueva gateando.

**Clase Walking**

Métodos:

La clase Walking implementa el método Move() de la clase IMove. El método Move() de la clase Walking hace que el personaje se mueva caminando.

**Clase Running**

Métodos:

La clase Running implementa el método Move() de la clase IMove. El método Move() de la clase Running hace que el personaje se mueva corriendo.