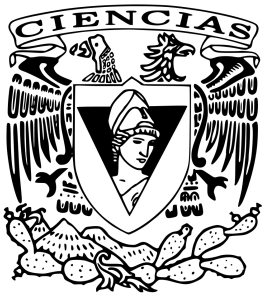
Universidad Nacional Autónoma de México
Facultad de Ciencias
Modelado y Programación
Proyecto Final.
04.09.2020

Amador González, Noe [redacted]
Hernández Sánchez, Oscar [redacted]



Descripción del proyecto

El proyecto consiste en una simulación de pelea entre un ejército y un enemigo, esta pelea se describe por medio de la terminal. Se cumple con todos los requerimientos solicitados en las indicaciones del *Proyecto Final*. Algunos puntos a considerar:

- Uso de la versión 8 de Java.
- El diagrama de clases no contiene la representación de la clase *Main.*
- En el diagrama de clases no se especifican los métodos constructores.

Ejecución del proyecto

El programa cuenta con una clase nombrada *Main* la cual se encarga de construir y correr todo el proyecto. Una vez compilados todos los archivos *.java y ejecutada la clase *Main*, se imprimirá un menú en la terminal, con este menú se podrá interactuar únicamente respondiendo/escribiendo números naturales incluido el cero; el programa también responderá a las peticiones. Es de señalar que el juego cumple con los requerimientos de impresión descritos en las indicaciones del *Proyecto Final* con un detalle adicional: cuando el Enemigo ha sido derrotado, se vuelve a imprimir el menú de acciones, esto permite que el jugador pueda volver a ver el *reporte* de cada soldado y *mover* para saber la localización de sus soldados, sin embargo, si el jugador quiere *atacar* otra vez entonces el juego imprimirá un mensaje diciendo que el Enemigo ha sido derrotado, una felicitación y la ejecución se detendrá. Para volver a jugar hay que correr nuevamente el programa.

Justificación

Para la implementación de nuestro proyecto hemos utilizado 3 patrones de diseño que estudiamos y vimos en clase: Strategy, Composite y Builder.
Elegimos estos patrones con base en los *hints* proporcionados en la descripción del proyecto:

- *HINT 1:* Piensen cómo es que cada soldado individual debe hacer cada acción respetando esas reglas de un soldado en general.
- *HINT 2:* Piensen cómo deben organizarse los soldados en el sistema.
- *HINT 3:* Piensen cómo se pueden comunicar los soldados y hasta qué nivel deben comunicarse dado el hint anterior.

- *HINT 4:* Piensen cómo se puede organizar la creación de estos ejércitos de forma unificada.
- *HINT 5:* Piensen cómo se puede unificar las distintas tareas.

Nuestra justificación para utilizar ***Strategy*** está impulsada por el *HINT 1*. Un soldado en general tiene 3 acciones fundamentales: moverse, atacar y reportarse. En Strategy tenemos como principio de diseño: "programar a una interfaz, no a una implementación"; esto nos permite agregar comportamientos al Soldado sin modificar los existentes; en este caso los comportamientos/interfaces serían Movimiento, Ataque y Reporte. La funcionalidad de Strategy nos permite asignar a cada una de las especialidades de los soldados un comportamiento sin desviarnos del objetivo principal que es asignarles las tres acciones fundamentales a cualquier soldado en general. También es importante conocer que la implementación de Strategy nos suma flexibilidad a la hora de crear soldados ya que a futuro podríamos agregar nuevos comportamientos y funcionalidades a los soldados.

Nuestra justificación para utilizar ***Composite*** está impulsada por el *HINT 2* y *HINT 3*. La descripción del problema nos indica que un ejército está conformado por pelotones, y a la vez estos están conformados por soldados que reciben órdenes de un comandante. Esto nos da una idea de jerarquización, lo que nos redirige a una definición vista en clase: "El patrón de diseño *Composite* te permite componer objetos en estructura de árbol para representar jerarquías, *Composite* deja al cliente tratar objetos individuales y composiciones de objetos uniformemente.". Esta definición se adhiere a nuestras necesidades. Cabe resaltar que hemos limitado la utilidad de este patrón a Comandante-Soldado y hemos dejado a un lado los ejércitos y los pelotones. Una vez implementada la jerarquía donde sabemos que el comandante le da instrucciones a los soldados, entonces podemos aplicar el patrón de diseño. Hay que notar que la clase Comandante, al ser el objeto compuesto y el que da indicaciones, debe recibir una lista de soldados a los que les va a mandar instrucciones, pero a su vez el Comandante también es un "tipo" soldado. Esto coincide con el diseño de *Composite* y sirve para completar los *HINTS 3* y *4*.

Nuestra justificación para utilizar ***Builder*** está impulsada por el *HINT 4* y *HINT 5*. Hay varios patrones que nos permiten construir objetos de manera personalizada, pero en nuestro proyecto nos hemos dado cuenta que necesitamos crear ejércitos ya construidos sin tanta necesidad de que el usuario sepa paso por paso la construcción. Esto coincide con algunas razones que vimos en clase para utilizar Builder: encapsulamos la creación de ejércitos en un objeto (llamémoslo un "builder"(constructor)) y le solicitamos al builder que construya la estructura de los ejércitos. Con esto logramos construir por piezas a nuestro ejército y que el usuario simplemente seleccione qué ejército es el que quiere. Es por esto que Builder parece ser una buena opción. Por otro lado, la implementación de Builder no requiere de una gran cantidad de clases especiales ya que Builder también nos permite unificar las tareas de los ejércitos sin importar qué ejércitos se hayan construido; esto facilita la interacción con el proyecto completo. Finalmente, Builder al igual que Strategy, nos brindan flexibilidad y facilitan la construcción de nuevos ejércitos a futuro.