Proyecto 02

Equipo: **Los Eso Brad**

- Gómez López Erik Eduardo 320258211
- Sánchez Victoria Leslie Paola 320170513
- Vázquez Reyes Jesús Elías 320010549

Para ver el programa, colóquese dentro de la carpeta src y ejecute 'javac EsoBradPets.java'. Siga las indicaciones.
*Se utilizó java 11.

Problemática:

La empresa Zakeh no se encuentra en su mejor momento. A pesar de contar con un juego mundialmente conocido como lo es Pou, el cual tiene más de 10 millones de descargas en total, se sabe que la popularidad de éste ha disminuido drásticamente en los últimos años. Por ello, se te encomendó la tarea de desarrollar un sucesor espiritual de Pou, sin embargo, los altos mandos de Zakeh tienen ciertas condiciones.

Los ejecutivos quieren que el juego ofrezca una experiencia diferente a su anterior éxito. Sugieren (por no decir que te exigen) inspirarte en la mascota virtual Tamagotchi para realizar este trabajo. En pocas palabras, consistiría en cuidar a una mascota virtual hasta que fallezca por descuido del jugador, o hasta que el propio jugador decida terminar la partida.

Se debe poder elegir entre 3 mascotas. Puedes decidir si serán animales comunes o personajes conocidos; ellos se preocuparán por las licencias después. Además, cuando se elige la mascota, se debe desplegar un menú donde estén detalladas las características principales de cada animal, las cuales son nombre, una breve descripción, puntos de hambre, puntos de energía, puntos de felicidad, y el saldo con el que cuentas al inicio dependiendo de la mascota que elegiste. Así mismo, debe existir un modo fácil y uno difícil.

Las acciones que se podrán hacer con la mascota son jugar, dormir, despertar y alimentar. Mientras juega, la mascota no puede dormir ni alimentarse, mientras duerme no puede hacer nada más que despertar, y mientras come no puede jugar ni dormir. Si muere, lógicamente no puede hacer nada de lo anterior.

Al momento de alimentar, *si el refrigerador está vacío*, el usuario tendrá la opción de ir al minisuper a abastecerse de comida y debe poder visualizar la lista de productos a su disposición; estos productos alimenticios varían de precio según su calidad. Naturalmente, el jugador no puede comprar alguno si su saldo no es suficiente.

En la opción de jugar, debe haber algunos minijuegos que permitan ganar monedas y puntos de felicidad para la mascota.

La mascota puede morir por aburrimiento o depresión si no se juega con ella. Fallece por cansancio si se juega demasiado con ella. Puede morir si no se le alimenta, naturalmente. También puede morir si se le alimenta con comida de mala calidad por mucho tiempo.

El jugador debe poder ver cuántos días logró cuidar de la mascota una vez que el juego finalice.
Finalmente, los ejecutivos pusieron mucha énfasis en que la experiencia al jugar el título fuera ***desafiante***.

Dejando de lado todo lo anterior, se te dio "libertad total" para realizar el juego.

Patrones utilizados:

1. Prototype: La interfaz 'Cloneable' y las clases 'MascotaVirtual', 'BaseDeDatosMascotas' y 'CrearMascota' se encargan de esta implementación.
   Debemos tener 3 mascotas disponibles para ser adoptadas, pero según el nivel del juego que el usuario elija, sus comportamientos cambian (el valor de sus atributos y algunos diálogos).
   Este patrón nos permite crear el molde para una mascota y al clonarlo, podemos brindarle al usuario distintas versiones de una misma mascota. además de ahorrarnos código y clases que pudieran ser repetitivas.

2. Composite: La interfaz 'Producto' y las clases 'Alimento', 'Inventario' se encargan de esto.
   Un usuario tendrá un refrigerador inicialmente vacío (un objeto Inventario). Cuando requiera comprar, se creará un carrito virtual (otro objeto Inventario) donde el usuario añade los alimentos que quiere comprar. Como el usuario tiene dinero, antes de realizar su compra se debe verificar que el dinero le alcance. Si tiene suficiente saldo, el carrito virtual se agrega a su refrigerador, de lo contrario el carrito virtual será eliminado con el recolector de basura de java y el usuario podrá intentar comprar de nuevo después sin ningún problema.

3. Iterator: Las interfaces 'Iterator', 'Catalogo' y las clases 'CatalogoAlimento', 'ListaJuegos' e 'IteradorLista' se encargan de implementar el patrón.
   Tenemos una colección de 'Producto' disponibles en el minisuper para que el usuario pueda escoger cuales comprar para alimentar a su mascota, esta colección no tiene porqué ser modificada (a menos que nosotros, como dueños del minisuper queramos cambiar el catalogo).
   Este patrón también nos permite hacer del catálogo un sistema independiente y abierto a la extensión (pues en un futuro, podríamos venderle al usuario accesorios para su mascota y guardar la información en una lista (quizá no sea conveniente), como lo es la Lista de eventos.

4. State: La interfaz EstadoMascota, y las clases 'ModoComer', 'ModoJugar', 'ModoDormir', 'ModoDespertar', 'ModoSuspender', 'ModoMorir' y 'Hogar' se encargan de implementar el patrón.
   La mascota puede hacer 3 cosas básicas: comer, dormir y jugar. Pero tenemos un cuarto estado (secreto) que puede ocurrir: morir.
   Si la mascota está durmiendo, necesita despertar. Si ya se despertó puede hacer cualquier cosa. Si está jugando o comiendo no puede hacer otra cosa al mismo tiempo. Como las acciones del usuario afectan directamente a la mascota, cualquiera descuido puede hacer que la mascota muera dando fin al juego (ya que no puede hacer ninguna actividad de las mencionadas anteriormente). La clase auxiliar Hogar nos permite simular los cambios de estado de la mascota, así nos evitamos que la clase 'MascotaVirtual' crezca todavía más.

5. Facade: Para tener un código más organizado, dividimos el problema en:

- Darle una bienvenida al usuario (contexto del programa)
- Elegir el nivel del juego (fácil o difícil)
- Adoptar una mascota (en caso de aceptar la misión)
- Convivir con su mascota (en caso de aceptar la misión)
- Darle una felicitación o mensaje de despedida (en el primer caso, ocurre si el usuario acepta la misión y su mascota sigue viva y en el segundo caso si el usuario decide rechazar la misión desde el principio)

Este es el papel de la clase 'CentroAdopcion'.

6. MVC: Para tener un mejor control de las clases y respetar el UML, creamos paquetes para cada patrón que fueron introducidos dentro de la carpeta Modelo o Vista o Controlador. La jerarquía de carpetas quedó así:

src

package.Modelo--> package.Prototype, package.Composite, package.Iterator

package.Vista--> package.Facade

package.Controlador--> package.State

package.Minijuegos

EsoBradPets.java (main)

El paquete de minijuegos se considera un subsistema "independiente" de los patrones anteriores y las clases que los conforman.