Santiago Andrés Gélvez Galvis – 202212387

Diseño y Programación Orientado a Objetos.

Taller #5 – Patrones

Link del repositorio: https://github.com/AlmasB/FXGLGames.git

1. Información general del proyecto:
   a. ¿Para qué sirve?
      Se trata de un repositorio con varios juegos de muestra creados en Java con ayuda de FXGL (FX Game Library). FXGL es una librería de desarrollo de juegos basada en JavaFX8. En otras palabras, FXGL usa los *frameworks* de gráficos y gestión de aplicaciones de JavaFX. Entre los 18 juegos del proyecto se puede encontrar una versión de Flappy Bird, de Bomberman, de TicTacToe, de Pac-Man, de Space Invaders, y otros juegos sencillos.
   b. ¿Cuál es la estructura general del diseño?
      Cada carpeta del proyecto hace referencia a un juego diferente. Dentro del *source* (src) un juego en específico, por ejemplo, Pong, existen dos paquetes: uno que contiene la implementación en java y otro que contiene los recursos (música, sonidos, texturas, imágenes, etc.). Con el ejemplo en concreto (Pong), hay pocas clases, sencillas y fáciles de distinguir: bola, bate, bate_enemigo, tipo_entidad, controlador_UI, aplicación y Factory (del que se hablará más adelante).
      Así como en Pong, los demás juegos cuentan con un diseño similar: clases referentes a los componentes del juego, una clase que distingue las entidades, un controlador, la aplicación y un Factory.
   c. ¿Qué grandes retos enfrenta?
      Como se utiliza la librería FXGL, se debe tener conocimiento y manejo de esta. Sin embargo, como se basa en el *framework* de JavaFX, su API es popular y por tanto fácil de aprender. Además, tiene una documentación extendida y se utiliza en casos reales de desarrollos de juegos. Finalmente, esto también implica que los bugs que pueda tener JavaFX se trasladan a la librería y luego al proyecto. En la sección de *Issues* se evidencian varios problemas del proyecto relacionados con el *framework*.

2. Información y estructura del fragmento del proyecto donde aparece el patrón:

   El patrón *Abstract Factory* se utiliza en todos los juegos (src) del proyecto. Se trata de una clase que tiene el nombre del juego y Factory al final (eg: *PongFactory, TicTacToeFactory, etc.*). Todas las clases *Factory* del proyecto implementan una interfaz llamada EntityFactory.

3. Información general sobre el patrón:
   a. ¿Qué patrón es?
      El patrón es *Abstract Factory.*
   b. ¿Para qué se usa usualmente?
      Se utiliza por medio de una interfaz para poder crear objetos sin necesariamente especificar sus clases en concreto. Además, permite que objetos sean agregados o modificados durante el tiempo de ejecución. Las subclases de la interfaz son las que se encargan de decidir cuál

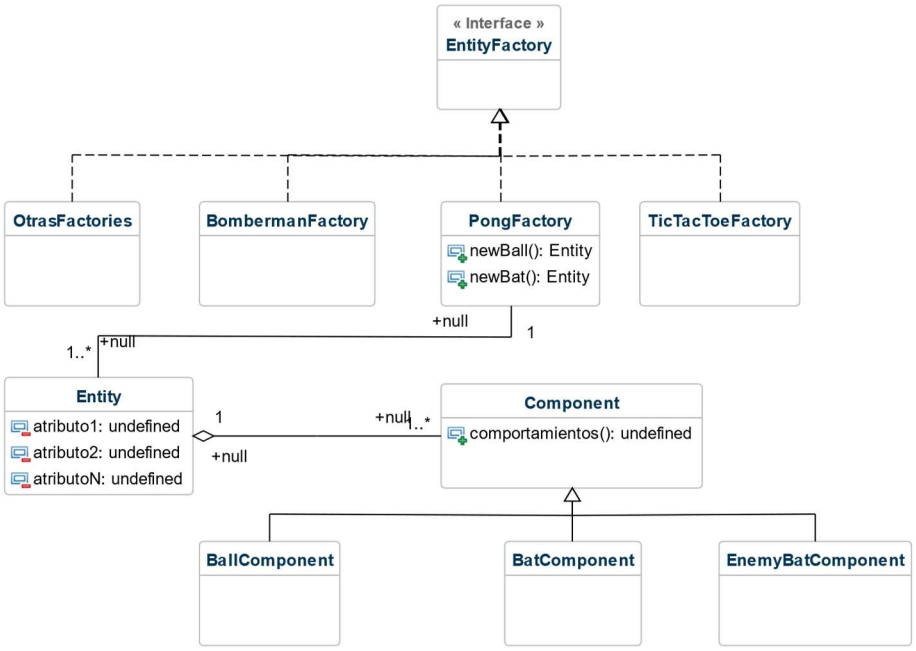
objeto instanciar. Es una versión más complicada del *Factory Method,* pero permite crear "familias" de productos (clases). Análogamente, se podría comparar con los *Generics*, o con polimorfismo, ya que facilita la manipulación de clases antes de que se sepa de qué tipo son.

4. Información del patrón aplicado al proyecto:

   En primer lugar, una entidad, según el proyecto, es un objeto de juego genérico. Es decir, tiene ciertas características mínimas que aplican para cualquier tipo de juego. Luego, la clase 'Componente' se utiliza para agregar datos y comportamientos a las entidades (composición).

   Específicamente hablando del juego de Pong, el patrón *Abstract Factory* se utiliza para crear entidades del juego. Puntualmente, la clase PongFactory implementa la interfaz EntityFactory (que también implementan otras *Factories* de otros juegos) para crear dos entidades en concreto: bola y bate. Esto se logra por medio de la anotación @Spawns("nombreEntidad") antes de su implementación. Además, es importante mencionar que no es necesario crear otro bate (el del 'bot' contrincante), ya que una entidad es suficiente para todas las instancias. Luego, cuando se instancian los componentes del juego, es cuando se agregan datos y comportamientos a las entidades iniciales.

   En el siguiente diagrama UML se pueden apreciar las relaciones (específicamente del juego Pong, aunque las demás *Factories* funcionan de igual manera):



5. ¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto?

   Porque el proyecto tiene diferentes géneros de videojuegos y desde el comienzo del proyecto, los desarrolladores no podían predecir exactamente cuáles entidades y componentes iban a tener todos y cada uno de los juegos.

   ¿Qué ventajas tiene?

   Por tanto, implementar una interfaz (EntityFactory) de la cual se implementan fábricas de entidades para cada juego (PongFactory, TicTacToeFactory, etc.) facilitaría el manejo de clases y métodos cuando tienen características similares.

6. ¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?

   En primer lugar, complejiza e incrementa la producción de código, ya que se deben hacer varias implementaciones de interfaces y en un juego pueden existir múltiples entidades (modifica tanto interfaces como clases concretas que implementan la *Factory*).

   En segundo lugar, atenta contra el principio SOLID de Open-Closed, ya que se necesita hacer modificaciones al código existente. Más concretamente, si se desea agregar un posible producto (en este caso, una nueva entidad del juego), la *Factory* debe considerar esta nueva posibilidad (con *cases*, por ejemplo) por medio de más líneas de código.

7. ¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?

   Como este proyecto es pequeño, es posible omitir la implementación de fábricas e individualizarlas a cada juego en particular. Es decir, que no todos los juegos implementen la interfaz EntityFactory, sino que dentro de cada implementación se instancien las entidades de cada juego (si se sabe puntualmente cuáles entidades debe tener cada juego). Esto permite menos complejidad en el código. Sin embargo, a medida que se creen más juegos en el proyecto, es necesario volver a utilizar este patrón porque no siempre se tiene conocimiento de cuáles entidades pueden crearse para un juego determinado.