TALLER 5 – STATE DESIGN PATTERN

1. ACERCA DEL PATRÓN

El patrón de diseño de Estado (State Design Pattern) le permite a un objeto alterar su comportamiento cuando su estado interno cambia. Por ello, es clasificado como un patrón de comportamiento.

1.1 Aplicabilidad

Puede ser aplicado cuando la conducta de un objeto depende de su estado, y éste debe cambiar en tiempo de ejecución. De igual forma, es útil cuando las operaciones tienen extensas sentencias condicionales que dependen de la naturaleza actual del objeto. Esto se debe a que el patrón trata el estado como un objeto en sí mismo, lo que suele convertir a cada rama del condicional en una clase separada.

1.2. Estructura

Se conforma de un contexto (Context), es decir el objeto principal que mantiene una instancia de cualquiera de los estados. También tiene una interfaz de estado (State), que permite encapsular los comportamientos e implementar los estados concretos (ConcreteStateA, ConcreteStateB)

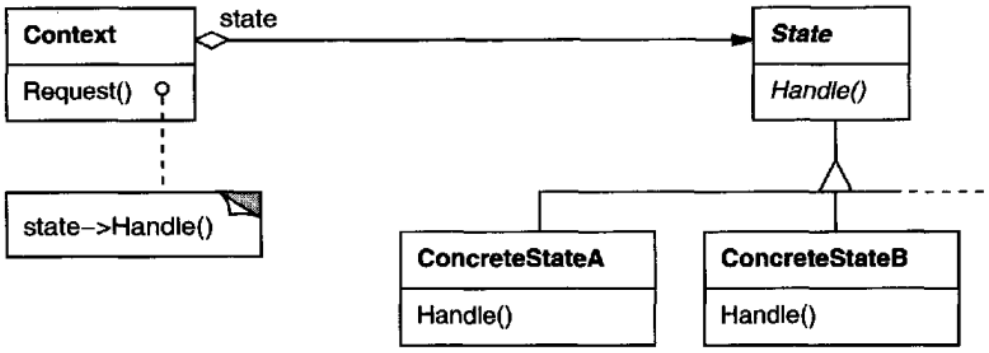


Diagrama UML del State Design Pattern

2. ACERCA DEL PROYECTO

El proyecto Pac-Man es un juego desarrollado en Java bajo patrones de diseño reconocidos, como lo son Abstract Factory, State y Strategy. Se basa en el videojuego clásico, donde el protagonista come ciertos premios dentro de un laberinto mientras evita ser capturado por los fantasmas. El proyecto hace uso de la librería Swing para implementar la interfaz gráfica.

URL del proyecto: https://github.com/lucasvigier/pacman.git

2.1. Retos de diseño

Los autores encontraron tres retos principales. El primero fue representar a cuatro familias de fantasmas con apariencia y conducta distinta, pero que guardan cierta dependencia. Otro es modelar la estrategia de cada familia para perseguir y huir de Pac-Man de forma óptima y escalable.

Estos dos desafíos se relacionan directamente con cambiar completamente el estado de un fantasma bajo ciertas condiciones del juego. Estas incluyen temporizadores propios, si una SuperPacGum ha sido comida, si el fantasma ha sido comido, si está acechando al protagonista y su ubicación respecto a su refugio

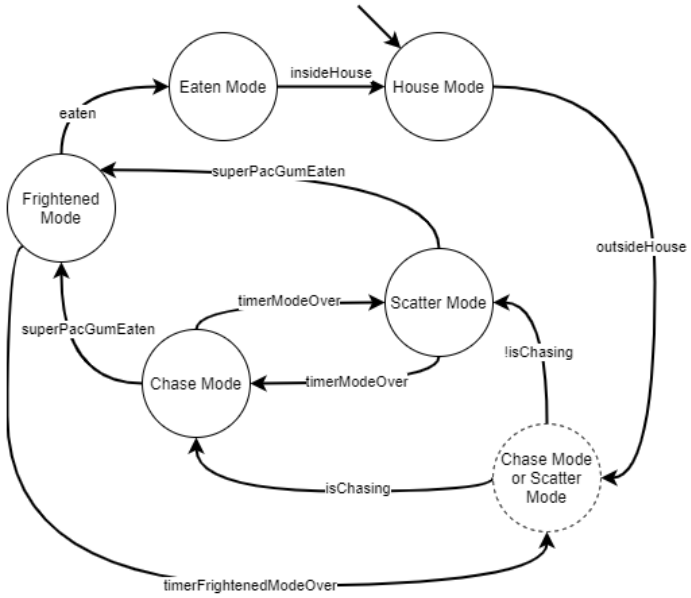
HouseMode: El fantasma intenta salir de su refugio cuando inicia el juego

ChaseMode: Persecución del protagonista de acuerdo a la estrategia del fantasma

ScatterMode: El fantasma termina la persecución y se dispersa en el laberinto hacia su propia esquina

FrightenedMode: Cuando Pac-Man come un SuperPacGum, la dirección del fantasma es aleatoria y puede ser comido

EatenMode: El fantasma ha sido comido y busca regresar a su refugio. No representa un riesgo



Autómata finito para representar los estados del fantasma

2.2. Diseño

La app se inicializa en la clase GameLauncher y se relaciona con Game, el cual contiene las entidades móviles (Pacman y Ghost) junto a las estáticas (PacGum, SuperPacGum, Wall y GhostHouse)

Se puede instanciar un enemigo de tipo Blinky, Pinky, Clyde o Inky. Cada fantasma tiene un estado cambiante que extiende de GhostState, su estrategia, su temporizador de modo, su temporizador de temor y si está acechando al jugador o no.

GhostState es la clase más importante del patrón, ya que modela funciones compartidas por todos los estados y funciones abstractas que cambian el comportamiento de acuerdo al modo del fantasma.

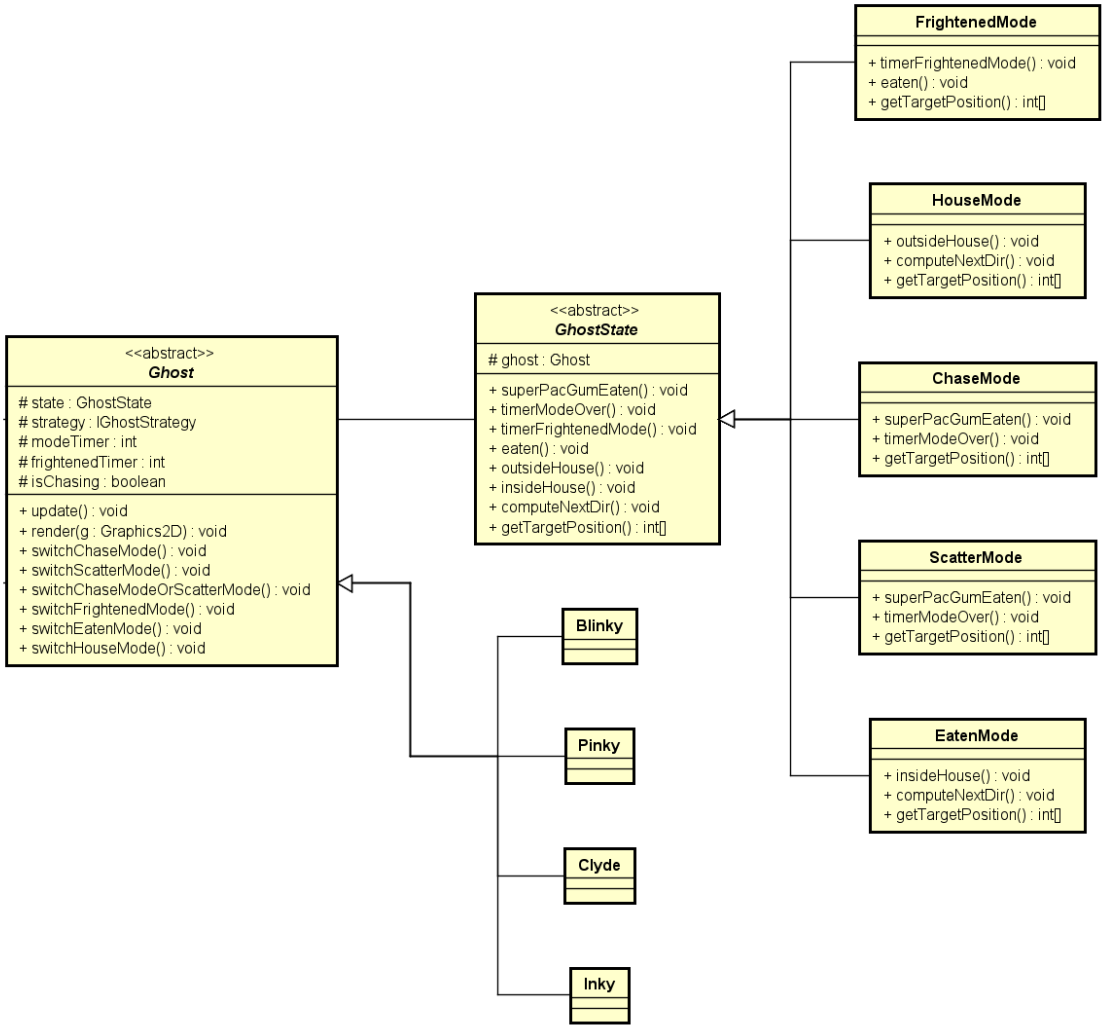


Diagrama UML del patrón State

Vale la pena rescatar que este patrón se complementa satisfactoriamente con Abstract Factory y Strategy.

Bajo el primero de ellos, se crea una interfaz para crear familias de objetos Ghost. De esta manera, el sistema es independiente de cómo sus objetos son creados y representados, mientras los usa conjuntamente de acuerdo a su estado.

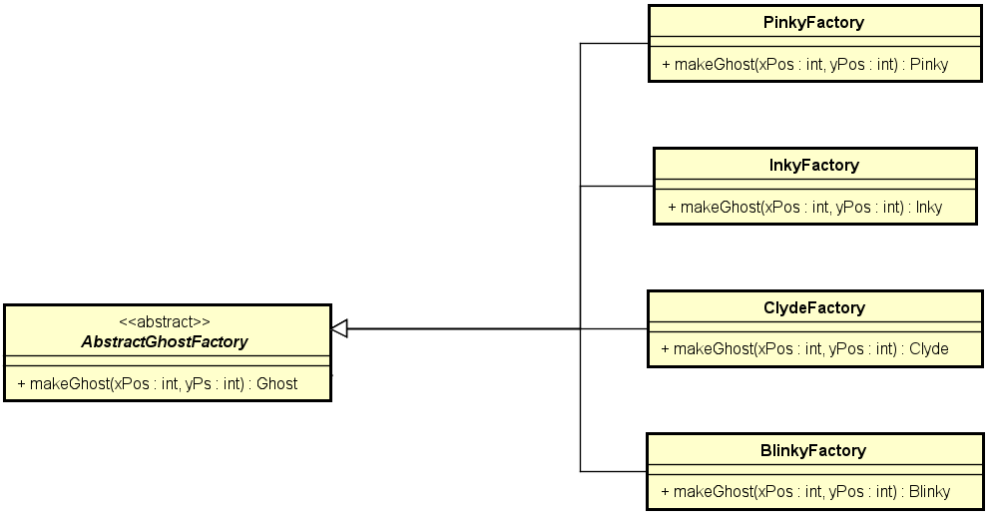


Diagrama UML del patrón Abstract Factory

Mediante el segundo, se toma un mismo procedimiento que puede ser realizado de distintas maneras. Posteriormente, se extraen todos los algoritmos de estrategia en diferentes clases, los cuales son objetos intercambiables.

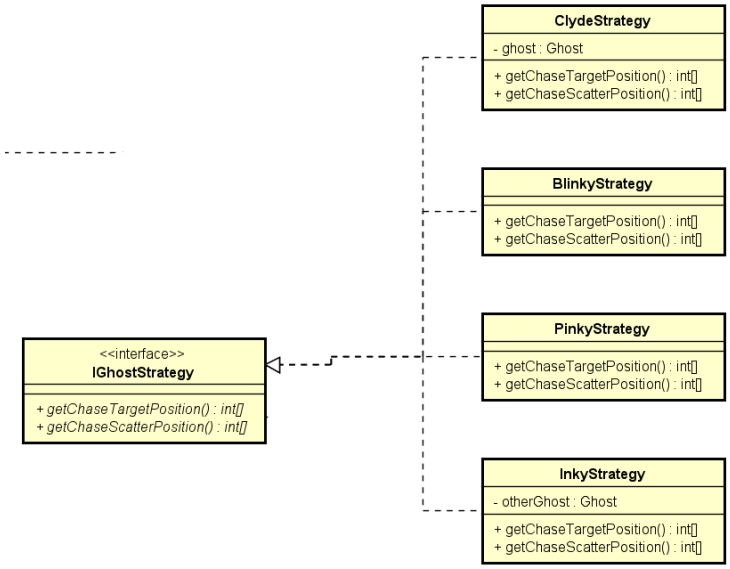


Diagrama UML del patrón Strategy

4. USO DEL PATRÓN EN EL PROYECTO

Respecto a la implementación, se evidencian las colaboraciones que propone el libro guía.

En primer lugar, el contexto (Ghost) delega las peticiones relacionadas con el estado del juego a su objeto de estado concreto (GhostState), y entre ellos existe una doble relación. De igual forma, el contexto es la interfaz principal para el cliente, que en este caso corresponde al juego en sí mismo (Game). Finalmente, el contexto es quien determina el flujo de los estados de acuerdo a sus atributos.

Teniendo en cuenta lo anterior, se analizará el uso del patrón en el proyecto con mayor detenimiento.

4.1. Uso habitual del patrón

Este patrón ha sido utilizado habitualmente en los protocolos de conexión TCP, en editores de imágenes con diferentes herramientas, en la secuencia de estados de un jugador en un videojuego, entre otros. Todos estos ejemplos comparten una estrategia que permite modelar una gran cantidad de estados y transiciones con facilidad. Por ello, su utilización es adecuada y tiene sentido en el proyecto.

4.2. Ventajas

- Respeta el principio de Responsabilidad Única al separar las clases
- Respeta el principio de Abierto/Cerrado al existir la posibilidad de añadir nuevos estados sin alterar clases del mismo tipo ni el contexto
- Simplifica el código al eliminar condicionales que dependen de muchos factores o atributos
- Genera comportamientos polimórficos cohesionados
- Cada estado tiene un nombre distintivo y un comportamiento más claro

4.3. Desventajas

- Cuando hay pocos estados o transiciones, puede complejizar innecesariamente el código
- En algunos casos, puede ser difícil saber quién debe definir las transiciones entre estados

Respecto a la creación y destrucción de objetos, puede impactar de dos formas:

- Se crean objetos cuando son necesarios y se eliminan después, lo que puede aumentar el tiempo de ejecución cuando hay variaciones continuas
- Se crean objetos antes de ser utilizados y nunca son eliminados, lo que puede aumentar la memoria utilizada durante toda la ejecución

Dentro del proyecto, cada fantasma tiene cinco atributos para los cinco estados, los cuales cambian constantemente. Por lo tanto, se escogió el mejor trade-off para el caso, ya que la cantidad de enemigos es moderada.

4.4. Alternativas de mejora

Una mejora que podría ser implementada en el diseño actual es el uso del patrón Flyweight. De esta manera, se puede guardar la gran cantidad de estados en un menor espacio, ya que se comparten las partes comunes de varios objetos, en vez de almacenar toda la información en cada objeto.

4.5. Alternativas de resolución

El problema de modelar una máquina de estados finitos podría tratarse con el patrón Template Method. Esto funcionaría si las implementaciones de los modos son lo suficientemente parecidas entre sí como para ser agrupadas. En este caso, podría definirse un esqueleto para el algoritmo, y las subclases podrían redefinir parte del algoritmo sin cambiar abruptamente su estructura

Otra alternativa es considerar al patrón State como una especialización de Strategy. Ambos se basan en la composición y cambian el comportamiento del contexto al delegar el trabajo a otros objetos. Como Strategy hace a estos objetos completamente independientes e inconscientes de los demás, podría utilizarse dado que no hay una dependencia fuerte entre los estados más allá de su extensión de una misma clase.

5. REFERENCIAS

Gamma, E., Helm, R., Johnson, R., Vlissides, J. M. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. ISBN: 0201633612

Refactoring Guru. (2023). State – Behavioral Patterns – Design Patterns. https://refactoring.guru/design-patterns/state