Susana Vilarnovo Allal
Laura Palmeiro Castro
GRUPO 1.3

UNIVERSIDADE DA CORUÑA

PRÁCTICA DE DISEÑO

E1: Gestión de hotel

Nos han encargado un software para la gestión de las reservas de un hotel y de su servicio de limpieza de habitaciones. La idea es que el personal del hotel pueda gestionar las habitaciones mediante los métodos del sistema que, dependiendo del estado de la habitación, serán aplicables o no.

Respecto a los principios de diseño SOLID:

- Principio de responsabilidad única: este principio plantea que cada objeto debe tener una responsabilidad única que esté enteramente encapsulada en la clase. Todos los servicios que provee el objeto están estrechamente alineados con dicha responsabilidad.

  En nuestro programa, este principio se cumple de la siguiente manera para cada estado: a pesar de que todos implementan las funciones de la interfaz EstadoHabitacion, cada estado se encarga únicamente de funciones específicas:

  - "Aprobada" cuenta con el método *reservarHabitacion*, encargado de cambiar al estado Ocupada y establecer el cliente de la habitación. También tiene el método *aprobarHabitacion*, que revoca la limpieza y reinicia el proceso de limpieza de la habitación (transiciona al estado PendLimpieza).
  - "Ocupada" implementa los métodos *terminarReserva* que cambia al estado Aprobada y desasigna el cliente de la habitación; y *liberarHabitacion*, que transiciona al estado PendLimpieza y libera el campo cliente de la habitación.
  - "PendLimpieza" tiene el método *limpiarHabitacion*, que cambia el estado de la habitación a PendAprob y le asigna un limpiador pasado por parámetro.
  - "PendAprob" implementa el método *aprobarLimpieza*, que verifica que el supervisor que realiza la acción es el adecuado y, en ese caso, cambia al estado Aprobada y se desasigna el limpiador.

  Adicionalmente, todas las llamadas a los métodos de los estados se realizan desde la clase Hotel, pero esta delega sus responsabilidades en estos métodos de los estados, lo que mantiene el código mucho más claro y coherente con el principio de responsabilidad única.

- Principio abierto-cerrado: este principio plantea que las entidades software deberían ser abiertas para permitir su extensión, pero cerradas frente a la modificación. Es

decir, que se pueda extender el módulo, clase, etc., creado con nuevas implementaciones pero sin afectar a lo que ya existe.

En nuestra implementación, este principio se refleja en los estados de la habitación. Si el hotel desea crear un nuevo estado aplicable a las habitaciones, simplemente se crea una clase que implemente la interfaz EstadoHabitacion y se implementan sus métodos. De esta manera, el programa se vuelve flexible a cambios en el hotel, cumpliendo con el principio abierto-cerrado.

Además, cabe mencionar la presencia de los principios DRY (Don't Repeat Yourself), KISS (Keep It Simple, Stupid) Y YAGNI (You Aren't Gonna Need It), en el desarrollo del programa. Estos principios han sido aplicados para evitar la adición de código redundante, duplicado e innecesario.

El uso de la encapsulación de objetos con múltiples variaciones de comportamiento, a través del polimorfismo y la ligadura dinámica, ha permitido desarrollar el patrón estado utilizado en el programa. Este enfoque ha contribuido a mantener el código limpio y evitar repeticiones innecesarias.

También, se ha evitado la introducción de complejidades innecesarias y la implementación de código que no se necesita actualmente, siguiendo los principios KISS y YAGNI.

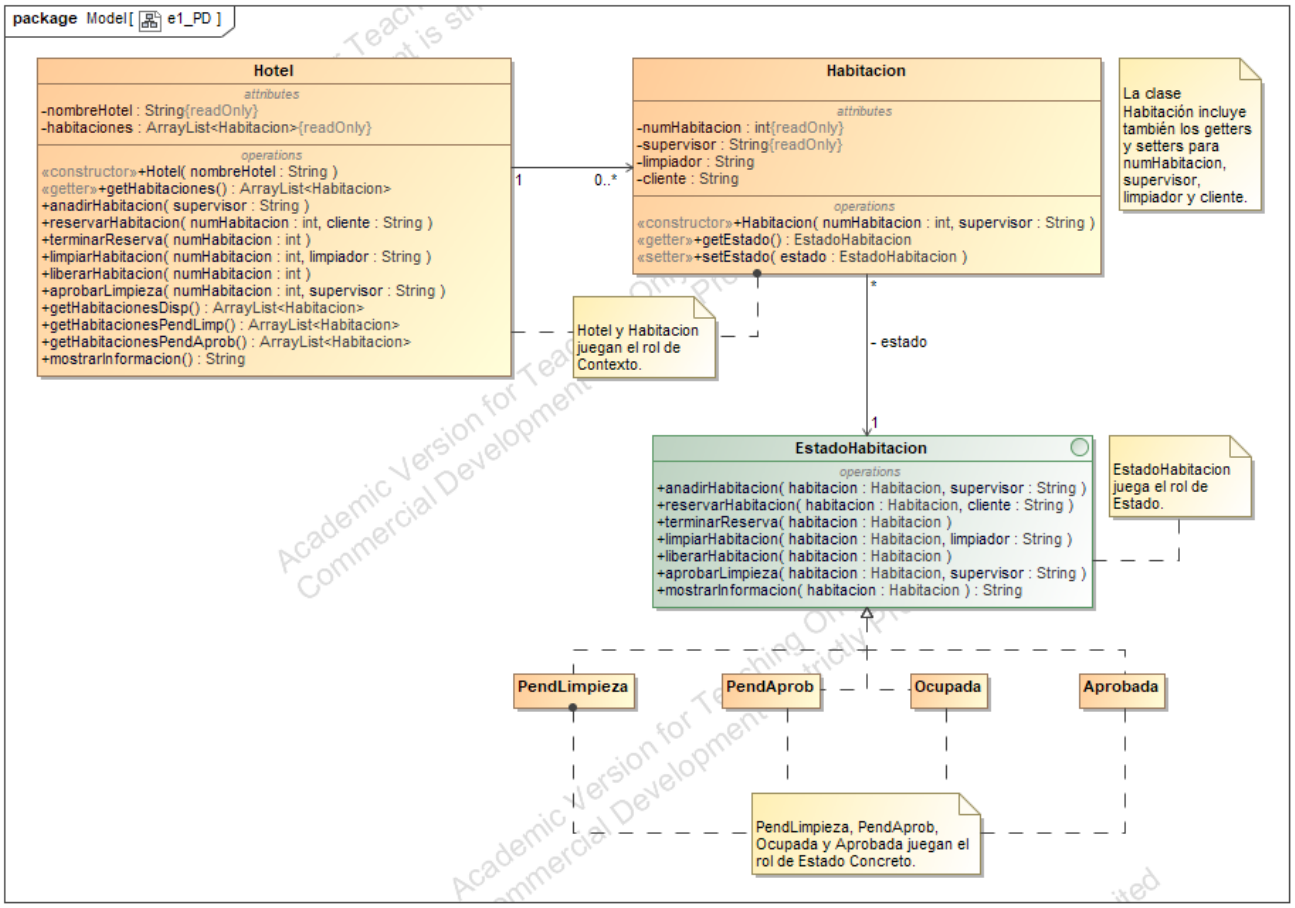
Patrón de diseño utilizado en este problema:

Para resolver este problema hemos utilizado el **patrón de estados**.

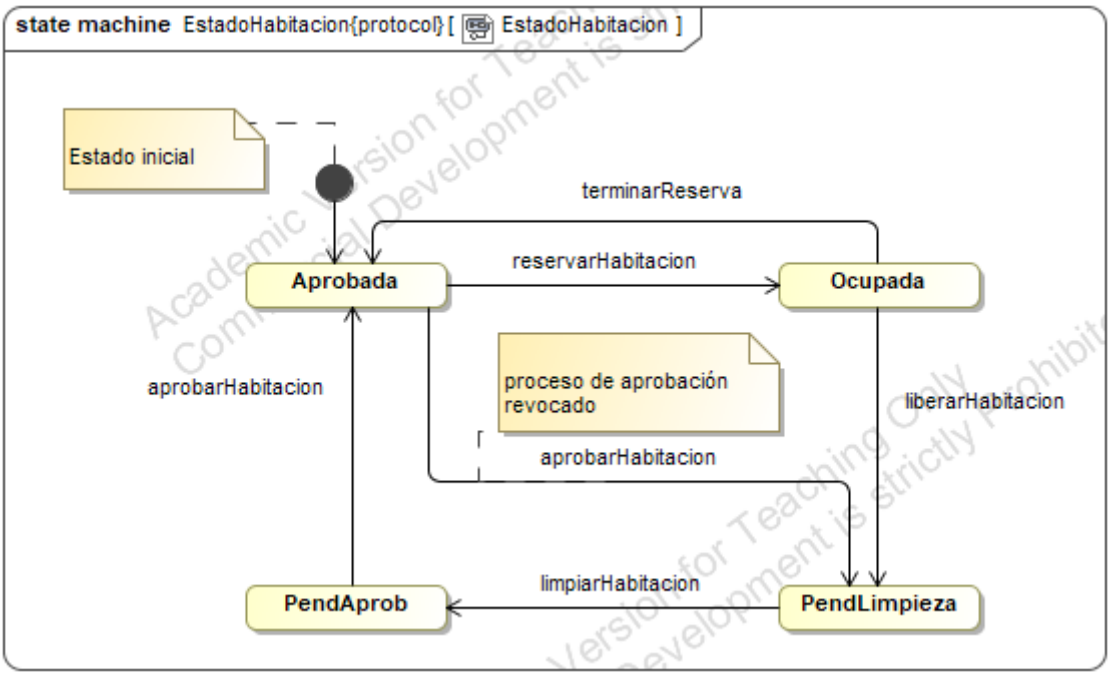
Este patrón permite a un objeto modificar su conducta al cambiar su estado interno. En este caso, resulta ideal para gestionar las habitaciones según el estado en el que se encuentran (Aprobada, Ocupada, PendLimpieza o PendAprob), ya que su comportamiento se vuelve dependiente de su estado interno.

Además, la implementación de nuevos estados se simplifica considerablemente, simplemente añadiendo nuevas subclases. Esto mantiene la gestión del hotel como un proceso abierto que puede aceptar modificaciones en el futuro de manera eficiente.

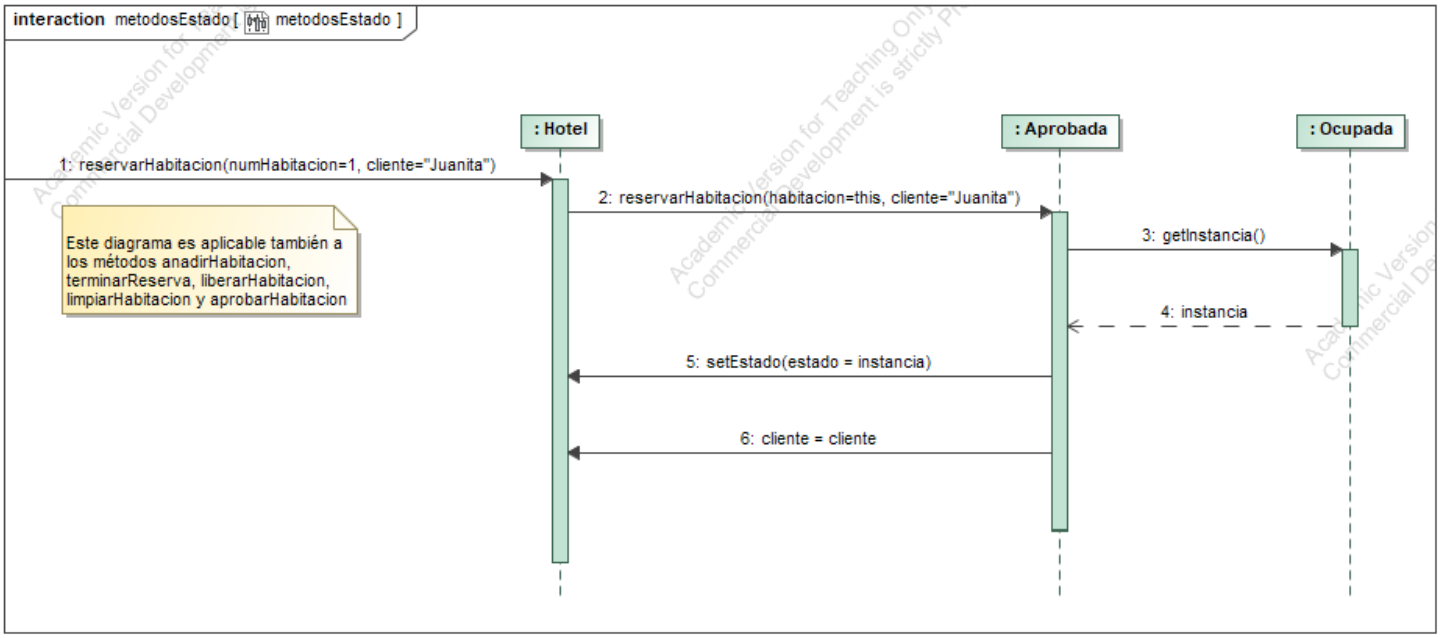
E1 - Diagrama de clases:



E1 - Diagrama de estados:



E1 - Diagrama de secuencia:



E2: Incursiones Navales

Nos han encargado un software para implementar las operaciones requeridas por un almirante en un mundo de guerra naval constituido por una flora que se embarca en incursiones presentes en un paisaje naval (mapa). La idea es que el almirante pueda realizar la incursión con una flota dada, determinar el número mínimo de nodos necesarios para alcanzar el final de un mapa y representar ese mapa en Formato Newick.

Respecto a los principios de diseño SOLID:

- Principio de responsabilidad única: este principio plantea que cada objeto debe tener una responsabilidad única que esté enteramente encapsulada en la clase. Todos los servicios que provee el objeto están estrechamente alineados con dicha responsabilidad.
  En nuestro programa, este principio se cumple, ya que la responsabilidad de determinar qué hacer en cada nodo (incursion), de calcular el número mínimo de nodos para alcanzar un NodoFin y de representar el mapa en texto recae en cada uno de los nodos (NodoAvistamiento, NodoBatalla, NodoAtaqueAereo, NodoTormentaMarina y NodoFin). De esta manera, el Almirante solo necesita realizar llamadas a los métodos de su mapa, manteniendo el código claro y coherente.

- Principio abierto-cerrado: este principio plantea que las entidades software deberían ser abiertas para permitir su extensión, pero cerradas frente a la modificación; es decir, que pueda extender el módulo, clase, etc., creado con nuevas implementaciones pero sin afectar a lo que ya existe.
  En nuestro programa, este principio se cumple en la implementación de los nodos. La creación de nuevos tipos de nodos se realizaría fácilmente mediante la herencia de las clases abstractas e implementando sus métodos abstractos que establecen el comportamiento de la incursión, el mínimo de nodos hasta un NodoFin y la impresión del mapa en ese nodo. De esta manera, el programa es flexible y puede adaptarse a la creación de nuevos tipos de nodos en el futuro sin necesidad de modificar la lógica existente.

- Principio de inversión de la dependencia: este principio plantea la estrategia de depender de interfaces o clases y funciones abstractas, en vez de depender de clases y funciones concretas.

En nuestro programa, este principio se refleja en las clases abstractas Nodo, NodoRutaFija y NodoBifurcacion, que implementan métodos abstractos para cada uno de los servicios del nodo. La implementación de cada tipo de nodo depende de estas abstracciones, invirtiendo así la dependencia y permitiendo una mayor flexibilidad en la extensión del sistema sin modificar las clases existentes.

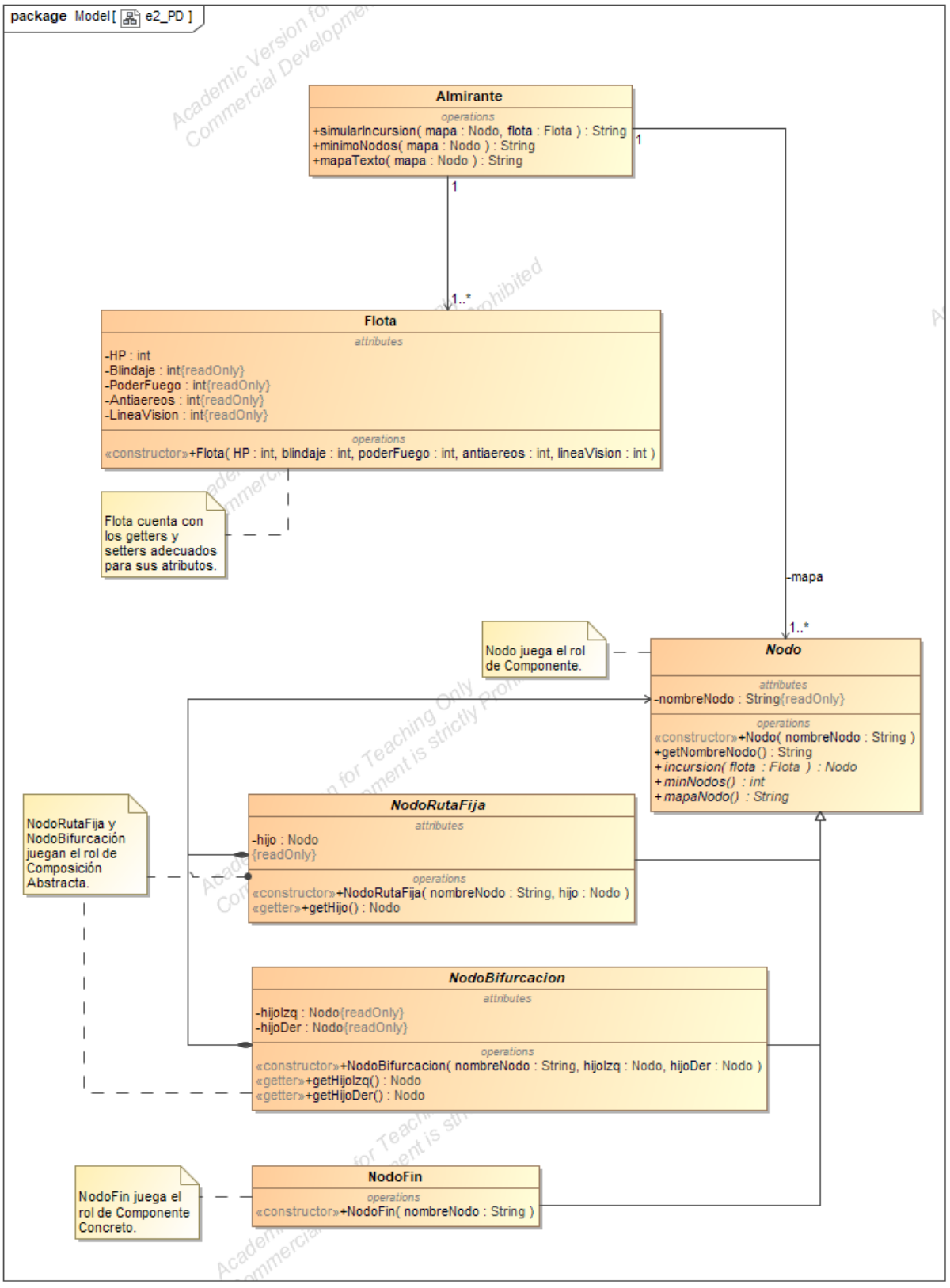
- Principio "Tell, Don't Ask": este principio se basa en que el código orientado a objetos debería centrarse en decirle a los objetos qué hacer en lugar de pedirles información y luego tomar decisiones.
  En nuestro programa, este principio se cumple debido a que en los métodos del Almirante, se llama a los métodos concretos de los nodos (incursion, minNodos y mapaNodo). De esta forma, el código se centra en el paso de mensajes entre los objetos (clase Nodo a clase Almirante), enviando las instrucciones al nodo en lugar de consultar su información para realizar las acciones. El Almirante le indica a los nodos qué acciones deben realizar, permitiendo que cada nodo maneje internamente cómo realizar esa acción en función de su estado y características.

Además, cabe mencionar la presencia de los principios DRY, KISS Y YAGNI. Se ha evitado añadir código adicional, duplicado e innecesario mediante la abstracción de métodos en la clase abstracta Nodo (incursion, minNodos y mapaNodo).
El principio DRY (Don't Repeat Yourself) se ha seguido al encapsular la lógica común en la clase base Nodo, lo que permite reutilizar funcionalidades en las clases derivadas sin repetir el código.
El principio KISS (Keep It Simple, Stupid) se ha aplicado al mantener la estructura simple y comprensible, evitando la complejidad innecesaria y facilitando la comprensión y mantenimiento del código.
El principio YAGNI (You Aren't Gonna Need It) se ha observado al abstenerse de implementar características o funcionalidades que no se necesitan actualmente, evitando así complicaciones innecesarias en el código.

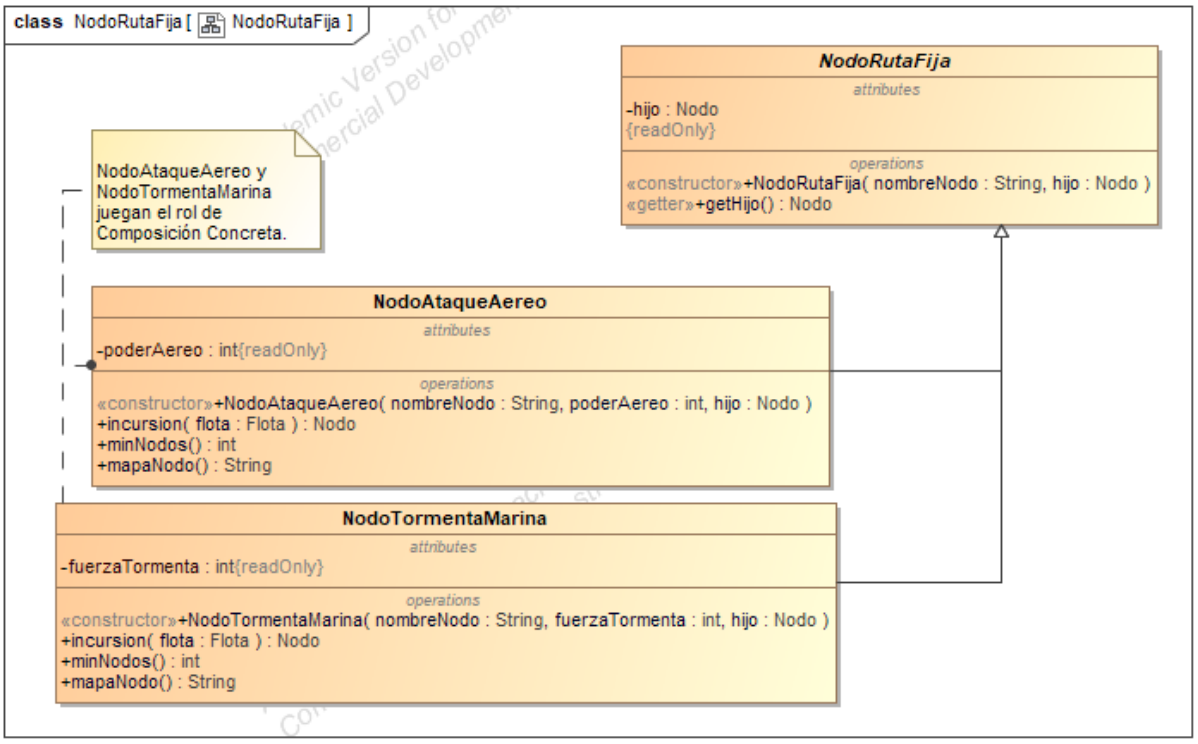
Patrón de diseño utilizado en este problema:
En este problema, hemos aplicado el **patrón composición** (más concretamente Composición Compleja), una forma de composición de objetos que se utiliza para construir estructuras de árbol que representan jerarquías todo-parte. Este patrón estructural es ideal para implementar la estructura de nodos solicitada en el ejercicio, haciendo uso de clases abstractas como Nodo, NodoRutaFija y NodoBifurcacion.

La implementación de este patrón nos permite tratar a los diferentes tipos de nodos de manera uniforme, proporcionando una interfaz común y facilitando la introducción de nuevos tipos de nodos en el futuro. La composición compleja también favorece la gestión y manipulación de la estructura jerárquica del mapa de manera eficiente y modular.

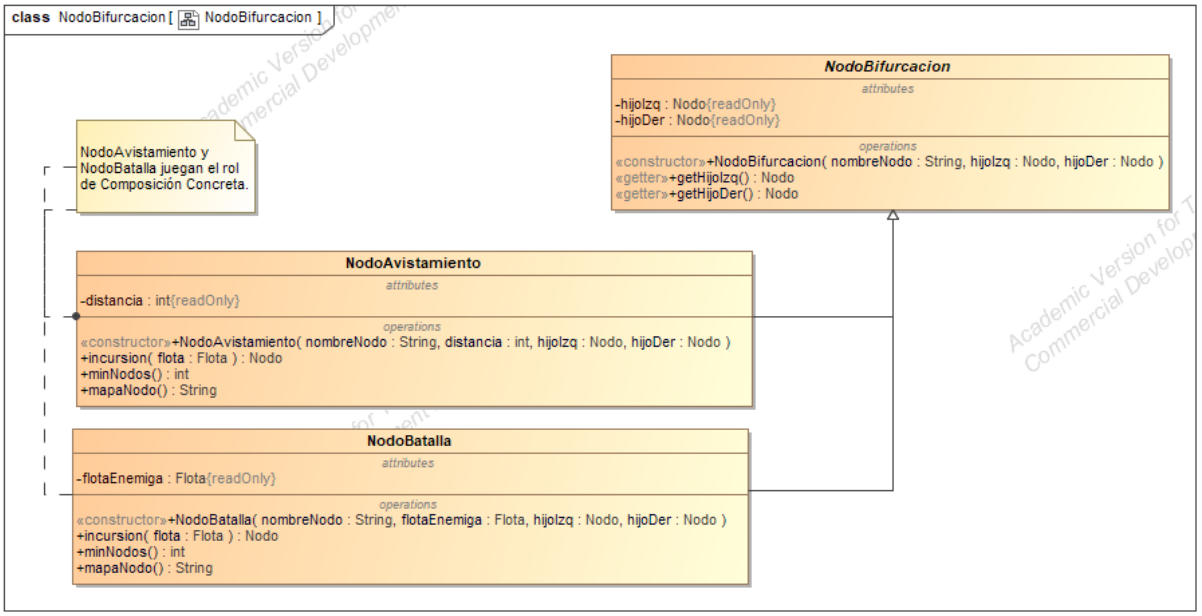
E2 - Diagrama de clases:



Subdiagrama para NodoRutaFija:

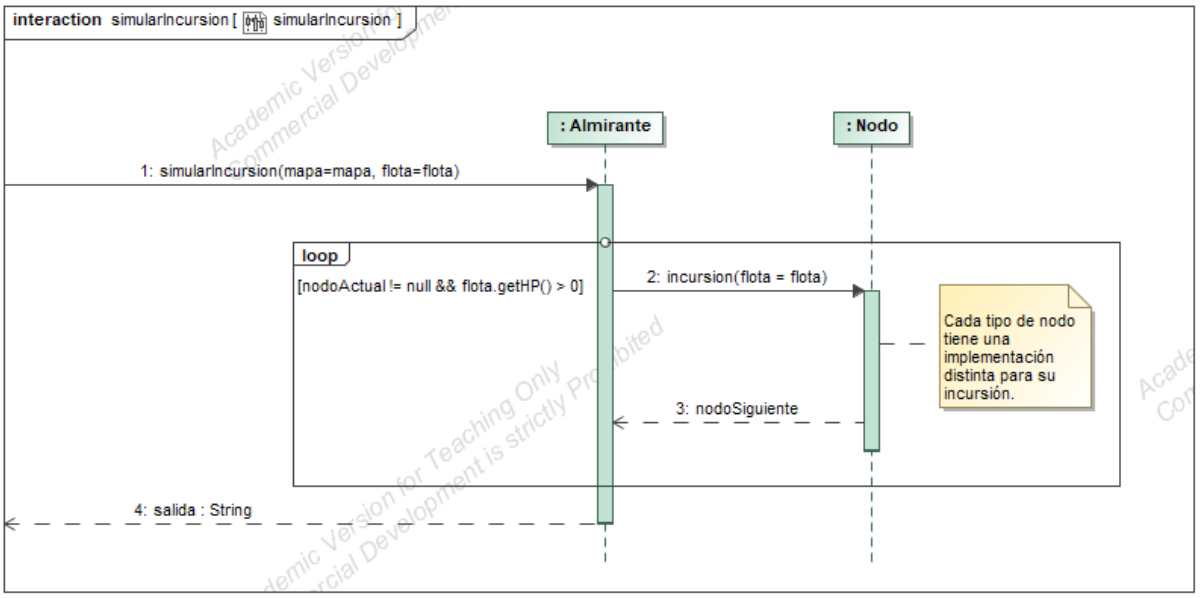


Subdiagrama para NodoBifurcación:

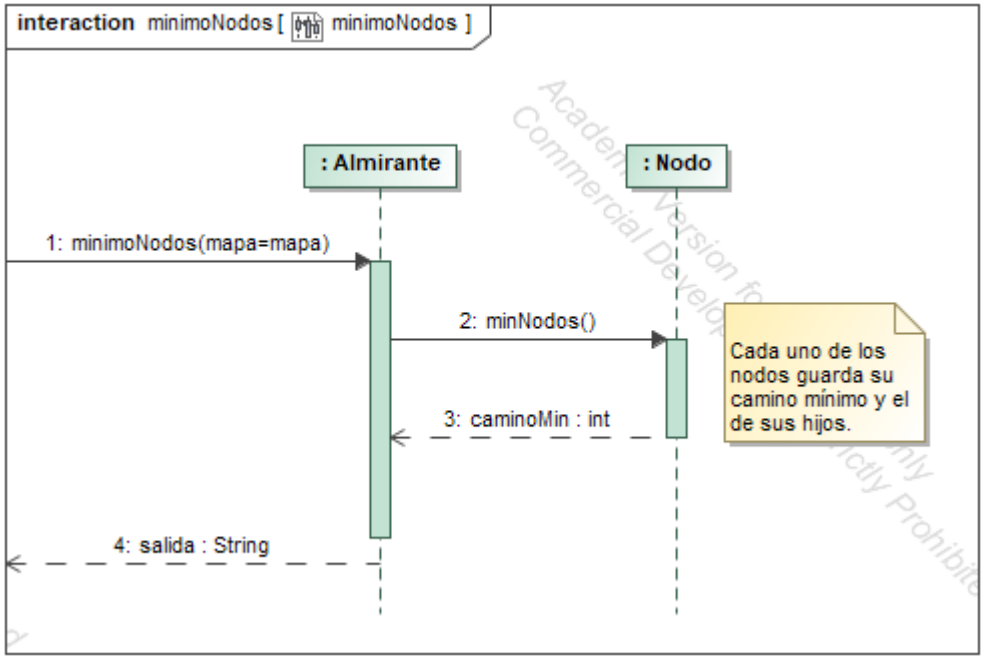


E2 - Diagramas de secuencia:

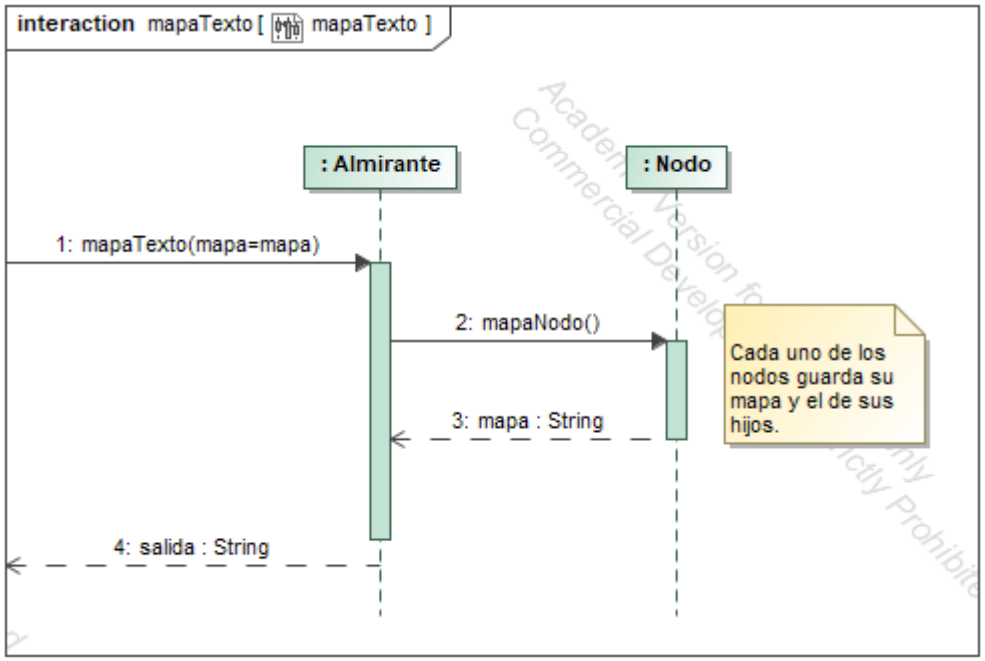
simularIncursion:



minimoNodos:



mapaTexto:


interaction mapaTexto [mapaTexto]
: Almirante
: Nodo
1: mapaTexto(mapa=mapa)
2: mapaNodo()
3: mapa : String
4: salida : String
Cada uno de los nodos guarda su mapa y el de sus hijos.