Premisa: Analizar la aplicación de principios y patrones en el diseño del programa.

Principios Solid.

SRP – Principio de responsabilidad única.

El principio de responsabilidad única nos dice que cada clase debe encargarse de una única responsabilidad, este patrón lo cumplimos haciendo uso fuertemente de herencia y de cadena de responsabilidades para separar razones de cambio.

Entre los usos de herencia para separar responsabilidades podemos destacar el tablero que se encuentra repartido en cuatro clases $AbstractVesselSaver$ (cuya responsabilidad es conocer y definir el comportamiento necesario para almacenar barcos), $AbstractField$ (conocer y definir el comportamiento de los campos de los que se compone el tablero), $AbstractAttackable$(definir la lógica de recepción de diferentes ataques) y $AbstractTable$ (resumir toda la jerarquía para definir el tablero).

Entre los usos de la cadena de responsabilidades para separar razones de cambio, encontramos $CommandsHandlers$ donde se separan todos los comandos disponibles para el usuario en distintas clases, lo que mejora el desarrollo del código por su orden y limpieza.

Un caso de no cumplimiento de patrón en nuestro código podría ser $AbstractVesselAttacker$, donde se nuclean todas las formas de ataque de todos los barcos, éstas podrían separarse usando herencia. Otro ejemplo podría ser, $TelegramPlayers$ donde se unen el Singleton de la clase y toda la lógica sobre el manejo del diccionario.

OCP – Abierto a extensión cerrado a modificación.

En el patrón abierto a la extensión, cerrado a las modificaciones, aparece nuevamente la cadena de responsabilidades como herramienta para permitir el agregado de nuevas funcionalidades agregando eslabones a distintas cadenas.

Un ejemplo ya citado es el de $CommandsHandlers$, donde el agregado de un comando solo implica la herencia de la clase $AbstractHandler$, su implementación y simplemente agregarlo a la cadena correspondiente. Otro ejemplo, con el uso de cadenas de responsabilidades,

podría ser ante la necesidad de agregar un nuevo evento, donde basta con implementar la interfaz $IEvent$ y agregarlo a la cadena[1] $AbstractNextEvent$.

LSP – Principio de sustitución de Liskov.

Toda clase que extiende o hereda (clase sucesora) de otra es un subtipo de esta (clase base), y en cualquier instancia donde se haga uso de la clase base, se puede utilizar la clase sucesora.

Un ejemplo, el método que se define en la interfaz $IItemValidator$ hace uso de $AbstractItemSaver$ una clase de la que se desprende por ejemplo $Battleship$ por lo tanto $Battleship$ puede ser la instancia con la que se haga la invocación al método.

ISP – Principio de segregación de interfaces.

Las interfaces no abundan en nuestro programa, sin embargo, aquellas que utilizamos tienen a lo sumo una sola operación, para evitar que los clientes dependan de tipos que no usan.

Un ejemplo de no cumplimiento de este patrón se observa en nuestra primera entrega donde $IItem$ definía tanto como evitaba un ataque, además de cuando era agregable a un barco.

DIP – Principio de inversión de dependencias.

El programa cuida sumamente las dependencias, cada clase abstracta y cada interfaz depende solamente de otras clases abstractas e interfaces; dejando solamente la herencia final de una cadena de extensiones de clases abstractas o la implementación de una interfaz a clases de bajo nivel de abstracción.

Los casos de no cumplimiento de este patrón se pueden dividir en dos; los justificados, por ejemplo, por el uso del patrón Singleton, siendo ejemplos las clases: $TelegramBot$, $TelegramPlayer$ e $ItemContainer$; la justificación de la dependencia sobre estas clases es la necesidad de una funcionalidad por ejemplo: para mantener todos los jugadores juntos en un mismo lugar para su consulta, o mantener una relación jugador-siguiente ítem.

Entre el incumplimiento injustificado se encuentra el uso de la palabra $static$ para la creación de métodos de clase, el único caso ejemplo se encuentra dentro de la clase $StringToInt$.

[1] Basta con eso solamente para agregar el evento, para la ejecución total del evento dentro del juego hace falta, por ejemplo, la implementación de su $AbstrackAttacker$ o su $ToString$.

Patrones Grasp.

Experto.

Encontramos una íntima relación entre el concepto de encapsulación y el patrón experto, sin embargo, haciendo uso del principio de sustitución de Liskov esto permite seguir el cumplimiento del patrón de responsabilidad única.

Los ejemplos que citaremos se encuentran dentro de la clase $AbstractRoomSaver$, más específicamente se trata los métodos de nombre $GetRoomBy$ ..., todos ellos retornan un tipo $Room$ si no se usase la palabra clave $protected$ se perdería la encapsulación; por el patrón de sustitución de Liskov se puede hacer uso de este método desde clases sucesoras, así lo hace $AbstractRoomsCommunication$ en los métodos $SendAllBy$ ... y mantener una separación de responsabilidades.

Polimorfismo.

El polimorfismo permite enviando el mismo mensaje se responda, desde distintos objetos, de diferente manera. Existen dos ejemplos en nuestro código ambos referidos al manejo de ítems. El primero de ellos recae sobre las implementaciones de la interfaz $IItemValidator$ donde cada ítem puede definir si es agregable o no de distintas maneras. El segundo refiere a las implementaciones de la interfaz $IAttackValidator$ donde cada ítem define su respuesta ante los distintos ataques que puede recibir el barco.

Creador.

El patrón creador hace presencia en las cadenas $AbstractItemToAddValidator$ y $AbstractItemToAttackValidator$, cuyo propósito es dado un ítem retornar su validador (el agregable o el de ataque), además hace presencia en los métodos de recepción de ataque de la clase $AbstractAttackable$ donde se crea la instancia que será guardada en el tablero tras el ataque; en la misma jerarquía donde se guardan las instancias de los campos de los tableros.

En otro punto del programa se crean las instancias de los jugadores que se agregan a un signleton, clase $TelegramPlayers$.Finalmente, también se aplica en $AbstractRoomSaver$ donde se coleccionan instancias de $Room$.

Otros patrones.

Cadena de responsabilidades.

Las cadenas de responsabilidades son sin lugar a duda uno de los patrones mejor aprovechados a lo largo y a lo ancho del desarrollo, además de sus comúnmente conocidas bondades, como lo son la separación de responsabilidades y la apertura a la extensión notamos la fácil corrección de bugs por el fácil recorrido sobre el código y la fácil implementación de nuevas funcionalidades mediante el envió de una excepción al final de una cadena que obligamos a tener un eslabón. Un ejemplo de esto último puede ser cualquier cadena de conversión de la que necesariamente esperamos respuesta.

Singleton.

Singleton es uno de los patrones más concretos en su implementación, pero nos ha parecido más irreconocible en cuanto al diseño, entre sus virtudes destacamos la unificación de la información que necesitamos nucleada para su consulta, como ha de ser el $TelegramPlayers$ así como su perseverancia, como lo es $ItemContainer$ donde la cantidad de ítems agregados y el siguiente ítem a agregar no deben ser descuidados.

Finalmente.

Si no nos falla la memoria, existen dos conceptos no mencionados del curso, delegación y tipos genéricos; delegación existe, por ejemplo, en toda la jerarquía de la clase $Player$ donde cada método se encarga de delegar su comportamiento a otros, mientras que tipos genéricos no aparece en el desarrollo del código, podría hacer una aparición en un singleton genérico desde donde se implementaran los otros o de una manera difícil en las cadenas de responsabilidades.