Título: Proyecto PII 2022 - Batalla Naval

Entrega final

Ingeniería en informática

Curso Programación II

Integrantes del grupo:

Kevin Alvarez

Carol Glass

Antony Pereira

Docente: María Parapar

Docente: Matias Olave

05.07.2022

Índice

Introducción

En esta oportunidad el problema consiste en programar un juego conocido como “Batalla Naval” entre dos usuarios que se comunicarán a través de un Chatbot a través de Telegram. Según el material brindado, un chatbot (o bot conversacional) es un programa que simula mantener una conversación con una persona al proveer respuestas automáticas a entradas hechas por el usuario.

En el siguiente informe, por un lado, se encontrará los roles que cumplen las clases creadas junto a sus justificaciones. Por otro lado, se encontrará la ubicación para cada una de las historias de usuario solicitadas, las pruebas realizadas para cada método estará dentro de la carpeta /Test/LibraryTests.

La justificación para cada una de las clases creadas para llevar a cabo la tarea junto con patrones GRASP y principios SOLID aplicados se encuentra en la documentación generada con Doxygen. (“Refactoring Design Patterns”)

Además del respectivo informe, el proyecto final incluye:

- Código de solución (src/Library/): En GitHub
- Código para iniciar el chatBot (src/Program/): En GitHub
- Casos de prueba divididos por partes (test/LibraryTests): En GitHub
- Documentación generada con doxygen

Análisis del problema

El juego debe consistir en que los usuarios puedan jugar una partida contra otro jugador a través de Telegram. Cada jugador ubicará en el tablero la cantidad de naves habilitadas, así como también el tipo de barco habilitado. Luego, el usuario deberá elegir una celda dentro de un tablero con coordenadas letra-número para disparar turnándose entre ambos usuarios. Si lo logra adivinar, ha tocado un barco enemigo y sobre el tablero aparecerá “X” donde el usuario decidió disparar y aparecerá un mensaje de “tocado” , de lo contrario, si en la coordenada elegida no hay ninguna parte de la nave aparecerá “|” ya que ha disparado al agua y aparecerá un mensaje de “agua”. Una vez que se ha tocado todas las coordenadas de cierta nave, dicha nave quedará hundida y aparecerá un mensaje de “hundido”.

Dado que la letra del problema indica que se debe permitir tantas rondas como sean necesarias para cada una de las partidas, la partida continuará hasta que haya un ganador. El ganador se da una vez que hundió todas las naves del oponente.

Esta entrega se enfoca en poder jugar entre dos jugadores a través de Telegram, permite una sala de espera que unirá jugadores que deseen jugar, se llamará Lobby.

El chatbot guardará la información de las partidas, jugadores, usuarios mediante una interfaz.

La propuesta

Aquí se enlista una explicación general de las funciones del software, escrita desde la perspectiva del usuario final y donde fue realizada cada una.

- Como jugador debe poder iniciar una partida: Esto se realiza a través de @paco_pii_G8_bot en Telegram. Mediante *Handlers*, respuestas predefinidas a cualquier acción/mensaje específico en el bot (“Customize Zobot using various Handlers”). El primer Handler es el de *HelloHandler* y a través de éste comienza todo. El jugador debe escribir “Hola” y en ese instante seguir las instrucciones enviadas por el chatbot, hasta que le

encuentre un oponente con el mismo tamaño de tablero y allí comenzará la partida.

- Como jugador debe posicionar las naves antes de comenzar con el primer movimiento: Esto es responsabilidad de la clase Game mediante un método booleano PlaceShipOnBoard() que pone la nave según la opción elegida y modifica el tablero de naves. Si agrega un barco donde ya hay otro en esa posición o no entra en el tablero retornará false. Es un método booleano debido a que facilita a PlaceShipHandler con la ejecución.
- Como jugador debe poder acceder a dos tableros, uno para visualizar las propias naves y otro para que contenga los disparos realizados: Para poder cumplir con esta consigna es que se crean dos clases ShipBoard y ShotBoard que son subclases de Board. Al mismo tiempo, los disparos se muestran distintos para cada jugador. Por ejemplo, al realizar un disparo, el "Active_Player" hace el disparo (editando su tablero de tiros) mientras que el "Inactive_Player" recibe el disparo (editando su tablero de barcos). Este método invoca al método que busca la coordenada en la lista de barcos del jugador y la convierte a true con el método SearchCoordInShipsList(). Se imprimen los tableros por pantalla con la clase BoardPrinter que implementan la interfaz IPrinter, la idea es que al implementar la interfaz no tenga que existir una clase para imprimir cada tablero, la clase TelegramBoardPrinter que también implementará a la interfaz IPrinter pero su responsabilidad será imprimir el tablero en el chat de telegram.
- Como jugador debe poder indicar una posición de ataque: Esto se puede observar en el método MakeShot() dentro de Player. Para el Bot, esto se realiza en *MakeShotHandler.*

- Como administrador del juego, debe poder registrar usuarios: Nuestro Bot está programado para que registre a los usuarios según su Id y Nombre en Telegram, esto evita que el usuario tenga que registrarse a sí mismo y reciba un mejor servicio. De todas formas, la clase Player permite cambiar el UserName Se guarda en UserContainer que también es una clase estática.
- Como administrador del juego, debe poder conectar dos jugadores que se encuentren esperando para jugar: Esto lo realiza el *JoinPlayersHandlers* y ocurre con el método JoinPlayersWithSameBoardSize() en LobbyContainer. Une a dos jugadores que hayan seleccionado el mismo tamaño de tablero
- Como administrador debe poder actualizar el tablero de registro de ataques, cuando un jugador ataca a otro: Se logra en el método ShotMade() que invoca a MakeShot() y ReceiveShot() de cada jugador y allí se van actualizando las matrices. A su vez, aquí es cuando wasHit del Spot que representa una coordenada de la nave pasa a ser true si es que le embocó a una parte de la nave.
- Como administrador del juego, debe poder almacenar todas las partidas en juego, junto con sus datos (tableros, jugadores, etc.): Se realizó una clase GamesContainer que tiene una lista de tipo Game.
- Como administrador del juego, se debe permitir tantas rondas como sean necesarias para cada una de las partidas: Esto se cumple ya que el juego solo finaliza cuando se hunden todos los barcos de algún jugador y se confirma en el método AreAllShipsSinked().
- Como administrador del juego, luego de cada ataque debe indicar el resultado del mismo a ambos participantes. Los mensajes a aparecer son:

1) "Nuestros satélites 🛰️ nos indican que tu misil ha dado en el blanco, el enemigo está en apuros. Es el turno de tu enemigo 😨." Significa que hay más partes del barco aún sin atacar
2) "Capitán, se le informa que ha hundido el barco enemigo 😎. Felicitaciones 👌. Vamos por buen camino." Si no quedan partes de la nave por atacar
3) “Le has dado a una ola 🌊. Es el turno de tu enemigo 😨.” Significa que no ha tocado ninguna parte de una nave.

Estos mensajes le aparecerán al usuario que estaba atacando. ¿Cómo se entera el otro usuario? Al momento de que es su turno atacar, podrá visualizar en su tablero de barcos si tiene un barco color rojo (le disparó y tocó) o violeta (disparó pero no tocó) mientras que el otro usuario se entera mediante su tablero de barcos, luego de enviar la coordenada a atacar.

Se realiza en la clase Player() en el método ReceiveShot() ya que éste tiene la información del tablero de cada usuario. Se indica también que el barco se hundió completamente con un atributo de tipo booleano de Ship llamado IsSinked chequeando que todos los atributos wasHit de las coordenadas de posición de dicho barco estén en true.

- Como administrador debe indicar que se finalizó la partida cuando todas las naves de un jugador queden hundidas: El bot puede indicarlo mediante el método ReceiveShot().

En cuanto a los test unitarios para las historias de usuario se encuentran en la carpeta Test/LibraryTest divididas por clases.

Solución a la propuesta

Las 9 clases del juego que se habían pensado en la primera entrega, para la segunda se transformaron en 26 clases más los Handlers y las clases del bot. Para la entrega final, se hizo 28 clases. Al seguir investigando acerca de los distintos patrones GRASP y principios SOLID y tomando en cuenta las correcciones de las entregas anteriores, es que se llegó a la solución final. Las clases son: AbstractBoard, Board, ShipBoard, ShotBoard, Spot, GamesContainer, LobbyContainer, UsersContainer, ContainerException, CoordException, GameNotFinishedException, OptionException, ReceiveShotException, SizeException, BoardPrinter, IPrinter, TelegramBoardPrinter, AircraftCarrier, Frigate, LightCruiser, Ship, Submarine, Game, Player, Singleton, User, Utils.

Cada clase cumple un rol, los roles que pueden cumplir son *titular de información, estructurador, proveedor de servicios, coordinador, controlador, interfaz.* (Wirfs-Brock and McKean, 4).

Para el caso del diagrama el titular de la información es la clase Game ya que tiene acceso a casi todas las clases, por lo tanto tiene información sobre el juego en sí. La clase ShipBoard y ShotBoard también son titulares de información ya que tiene los datos acerca de donde se encuentran los barcos que el usuario agregó o los disparos que realizó. La clase Player puede cumplir rol de titular de información pues contiene la información de tableros, barcos, realiza y recibe tiros. Los proveedores de servicios son BoardPrinter, TelegramBoardPrinter y Utils ya que realiza el trabajo de imprimir los tableros, ofreciendo servicios informáticos, y brinda propiedades para el mejor funcionamiento e interacción con el usuario. La clase Board es proveedor de servicios también ya que realiza el trabajo de crear el tablero como una matriz bidimensional.

Las clases para todos los *Handlers* son quienes toman decisiones y dirigen de cerca las acciones del usuario. El rol de estructurador también es cumplido por la clase Player ya

que mantiene las relaciones entre los objetos y la información sobre esas relaciones, pues todas las clases están relacionadas de cierta forma con ella. A la clase Ship se le puede dar un rol de coordinador, pues reacciona a los eventos delegando tareas a otros como lo es cuando se agrega el barco, o cuando recibe el disparo para ver si el barco quedó hundido. Un barco queda hundido cuando se dispara a todas las coordenadas que lo componen cuando se le dispara a la coordenada de vulnerabilidad. Se tiene dos clases que cumplen el rol de interfaz que es, por un lado, IPrinter, la cual será implementada por BoardPrinter y TelegramBoardPrinter y por el otro AbstractBoard, pues si bien es abstracta se está utilizando como interfaz.

Una gran variación de la primera entrega a esta, además de los Containers y las excepciones y los Handlers es la clase Player. Esta clase hace que el constructor dependa de que el usuario esté registrado. Dicha clase permite dividir más responsabilidades, pues el Player es quien jugará, comenzará la partida, disparará, mientras que el User tendrá la información del usuario la cual incluye un Id generador "automático" por Telegram. Para la entrega final, se incluyen todos los *Handlers* y el funcionamiento del bot.

Por otro lado, otra variación es la de asignar al jugador al lobby según el tamaño de tablero que seleccione, esto es para asegurar que ambos jugadores tengan el mismo tamaño de tablero.

La explicación de cada clase y donde se aplicaron patrones GRASP y principios SOLID se encuentra en el DoxyGen del proyecto.

Negociando con el cliente

Dado que se debe agregar características extras en el juego, se decidió agregar:

1) El usuario podrá ubicar 3 tipos de naves distintas, cada una ocupará 2, 3, 4 o 5 coordenadas en el tablero y cada una será una clase. El usuario podrá seleccionar el tipo de barco a agregar. Los nombres para cada barco son: LightCruiser, Frigate, Submarine,

AircraftCarrier. Serán todas subclases de Ship, pues una subclase hereda los atributos y métodos de otra clase y habitualmente agrega nuevos atributos y nuevos métodos

2) La cantidad de barcos que podrá agregar el jugador estará limitada a 4, uno de cada tipo. para poder hacer que sea justo para ambos jugadores.

3) Cada barco tendrá una coordenada “aleatoria” del largo del barco que será considerada como su punto vulnerable, al acertar el tiro en su punto vulnerable el barco será completamente eliminado. Esto se realizó en barco un atributo de Ship, VulnerableCoord con un método que si se dispara allí cambia todos los atributos wasHit de las coordenadas a true. Será utilizado cada vez que se realiza un disparo.

4) Los jugadores tienen únicamente tres opciones de tableros “Malvinas” de tamaño 8, “Donbas” tamaño 10 y “Laos” tamaño 12.

Lógica del bot

Programar el bot, implementarlo y que puedan jugar 2 usuarios a la vez no fue tarea fácil. En nuestro proyecto, todo comienza y se lleva a cabo en la clase HandlerConcadenation. Esta clase es titular de información, facilita la información de los handlers y es en ésta donde se marca el órden en que se van a ir pasando las responsabilidades, para ser más específicos, la clase contiene un método InitializeHandlers() que es quien inicializa todos los handlers según el momento. El método es luego invocado por la clase ChatBot, la cual es un singleton y es la encargada de toda la gestión del Bot.

La clase IHandler es la clase donde se aplicará el patrón “Chain Of Responsibility”. Consiste en que cada Handler decide si procesa el mensaje, o si le pasa la responsabilidad al

siguiente. La clase que hereda de esta interfaz es la clase abstracta BaseHandler, es la superclase y los Handlers existentes son las subclases.

Todos los handlers implementan él método InternalHandler() de la superclase BaseHandler los cuales se acceden mediante keywords. Por ejemplo, en el primer caso, se usa la keyword “hola”, ”Hola”, “/Hola”, cuando el usuario ingresa la palabra clave, InternalHandler realizaría la tarea del mismo, comparando la palabra ingresada por el usuario con la lista de keywords. En este caso, al ingresar al InternalHandler lo que se hace es crear una nueva instancia de ChainData, donde si el usuario no estaba utilizando ya el Bot, se guardan los datos de éste en un diccionario, esta misma clase es singleton para que los datos de la comunicación entre el usuario y el bot tengan persistencia. ChainData tiene clave el Id del usuario y el valor es una colección con las posiciones dentro del Handler.

Si la Keyword es correcta o el Id del se añadió se comienza a procesar el Handler.

Se valida la posición en la que se encuentra el usuario mediante iteraciones para saber qué comportamiento debe seguir, esto se hace con userPositionHandler, en el caso de que esté en la posición 0, significa que el usuario ingresó /hola, en el segundo que aceptó ayudar en la batalla y así sucesivamente.

Al salir del handler, todo el contenido de la colección de esa clave es limpiado.

El resto de los Handlers aplican el mismo funcionamiento y se acceden mediante la clase HandlerConcadenation, en el método estático InitializeHandlers(). Los handlers que se concatenan son los siguientes:

1. HelloHandler: Se activa con la palabra clave /Hola, “Hola” u “hola”, agrega el id del user y el nickname de Telegram a un diccionario.
2. RegisterUserHandler: Agrega la información del usuario al usersContainer.
3. BattlePlacerHandler: Le da al usuario los tamaños de tableros que puede elegir y éste debe seleccionar uno.

4. SelectLobbyHandler: Al seleccionar el tipo de tablero (Maldivas, Donbas, Laos), SelectLobbyHandler se encarga de crear los tableros del usuario (ShotBoard y ShipBoard) para agregar al jugador al LobbyContainer que le corresponde.
5. JoinPlayersHandler: Este handler es el que une a dos usuarios que hayan elegido el mismo tamaño de tablero. Se crea la instancia de Game entre ambos jugadores y se remueven a estos dos jugadores de la lista de LobbyContainer.
6. PlaceShipHandler: Pide a los jugadores que comiencen a ubicar los barcos. las keywords se corresponden con los nombres de los barcos que puede agregar cada uno.
7. MakeShotHandler: Es el Handler encargado de los disparos

Indicaciones a tener en cuenta

- Al comenzar por primera vez a hablar con el bot luego de indicar “/start”, volver a escribir “Hola”, “hola” u “/Hola”.
- En caso de cometer un error al posicionar barcos, se debe volver a seleccionar el tipo de nave.
- En caso de no ser usuario de IPhone, lse recomienda ir a Ajustes/StickersAndEmojis y desactivar la opción “emojis grandes” del dispositivo para tener una mejor visualización a la hora de jugar.
- Si en el momento de estar jugando y de indicar “/atacarEnemigo” aparece el texto inesperado “¡Chau!, ¡Que andes bien!” antes de volver a ejecutar el Program, se recomienda intentar seleccionar nuevamente la opción. De lo contrario, se solicita volver a correr el programa.

Inicio y ejecución del proyecto

Para una ejecución exitosa se deberán de tener ciertos paquetes en su ordenador en el directorio C:\Users\<Su usuario>\.nuget\packages. Dichos paquetes con las versiones correspondientes están localizadas en la carpeta “docs” de nuestro repositorio en Github. La carpeta que contiene dichos paquetes es llamada “PaquetesUtilizados” y se deberá copiar su interior en el directorio mencionado anteriormente.

Por último pero no menos importante se deberá copiar y pegar el archivo llamado “appsettings.json” en la siguiente ubicación dentro de su ordenador “PII_EntregaFinal_G8\src\Program\bin\Debug\net6.0 ”. Esto se debe a que el archivo al ser subido al repositorio es ignorado por GitHub por lo cual debe de ser añadido manualmente en cada ordenador al ejecutar el proyecto. Cabe recalcar que el ignorar este archivo por parte de Github no es aleatorio y fue configurado previamente con el fin de evitar que el token de seguridad de nuestro chatbot pueda ser accedido y extraído por personas mal intencionadas.

Al iniciar la comunicación con el Bot debe ser con “Hola”, “hola”, “/Hola”.

Por cualquier otra consulta acerca de la ejecución de nuestro proyecto se agradece la comunicación con los desarrolladores que estarán disponible para usted las 24 hs.

Problemas que surgieron a lo largo del trabajo

Un problema que se presentó fue que saltaba un error con el tema del SecretToken. Para solucionar esto, utilizamos BotFather y le pedimos otro token para comenzar a crearlos.

Otro tema y desafío a resolver era generar que sea multiplayer, es decir que dos personas se puedan comunicar a través del bot.

El equipo sufrió la baja de un compañero, por lo que lo que se había planificado para dividir y hacer entre 4 terminamos siendo 3 lo cual también lo consideramos como un problema organizativo para cumplir con la entrega en su totalidad.

Otro tema que se tuvo que enfrentar fue hacer que los *Handlers* respondan a todo lo solicitado, pues en un momento el Bot dejó de funcionar y, una vez más, se tuvo que utilizar el “debugging”.

Reflexiones del trabajo

Se reconoce que a medida que pasó el tiempo, las clases con sus métodos y atributos cambiaron mucho, con el fin de cumplir con nuevos patrones procurando alcanzar buenas prácticas de programación y diseño.

Esta entrega permite visualizar todo lo trabajado en el semestre, y fue un proceso muy largo. Nos enfrentamos a la toma de decisiones en muchas ocasiones y aprendimos a trabajar en equipo.

Utilizar GitHub como herramienta fue clave, permitió tener más acercamiento y poder trabajar en conjunto sin tener que coincidir en todas las reuniones, lo cual concluyó en una entrega prolija, entendible y que funciona.

El equipo terminó entendiendo mejor el funcionamiento de un Bot y la implementación de la API, así como también la aplicación de patrones y principios.

En el diagrama UML se observan todas las clases del juego realizadas y sus relaciones, creemos que al compararlo con el programa concuerdan en su totalidad.

Para finalizar, se puede decir que esta entrega se cerró con éxito. A pesar de que un compañero abandonó luego de la primera entrega, se pudo cumplir con todas las historias de usuario y que el bot funcione que era lo más importante.

Bibliografía

"C# docs - get started, tutorials, reference." *Microsoft Docs*, https://docs.microsoft.com/en-us/dotnet/csharp/. Accessed 15 June 2022.

"Customize Zobot using various Handlers." *Zoho*, https://www.zoho.com/salesiq/help/developer-guides/bot-siq-scripts-handlers-2.0.html. Accessed 28 June 2022.

José García Peñalvo, Francisco, and Carlos fgarcia@.ubu.es Carlos Pardo Aguilar. "Diagramas de Clase en UML 1.1." Universidad de Burgos.

Martin, Robert C. *SOLID y GRASP - Buenas practicas hacia el exito en el desarrollo de software*, https://jbravomontero.files.wordpress.com/2012/12/solid-y-grasp-buenas-practicas-hacia-el-exito-en-el-desarrollo-de-software.pdf. Accessed 21 June 2022.

"Refactoring Design Patterns." *Refactoring.Guru*, https://refactoring.guru/design-patterns. Accessed 28 June 2022.

Wirfs-Brock, Rebecca, and Alan McKean. *Object Design (Roles, Responsibilities and Collaborators)*. Addison-Wesley, 2002.