Título: Proyecto PII 2022 - Batalla Naval

Segunda entrega

Ingeniería en informática

Curso Programación II

Integrantes del grupo:

Kevin Alvarez

Carol Glass

Antony Pereira

Docente: María Parapar

Docente: Matias Olave

20.06.2022

Análisis del problema

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. 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.

El juego consistirá en que los usuarios podrán crear una partida, ubicarán en el tablero la cantidad de naves habilitadas según la dificultad del juego, así como también el tipo de barco, y luego el usuario deberá elegir una celda dentro de un tablero con coordenadas número-numero para disparar. Turnando turnos entre ambos usuarios. Si lo logra adivinar, ha tocado un barco enemigo y sobre el tablero aparecerá “x” donde el usuario decidió disparar, de lo contrario, si en la coordenada elegida no hay ninguna parte de la nave aparecerá “|” ya que ha disparado al agua. Una vez que se ha tocado todas las coordenadas de cierta nave, dicha nave quedará hundida.

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.

Esta entrega se enfoca en la lógica del bot de Telegram.

La propuesta

Aquí se enlista una explicación general e informal 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 en el método WantToPlay() dentro de la clase Usuario. En ese momento se agrega el usuario al contenedor de lobby instanciando a LobbyContainer, consiste en una lista de usuarios esperando para jugar. Sin embargo, la responsabilidad de iniciar una partida debe ser del Administrador, quien toma los dos primeros usuarios en el LobbyContainer y

comienza una partida entre ellos. El jugador que comienza es el “Active_Player”, quien elige el tamaño del tablero y posiciona las naves. El jugador “Inactive_Player” aguarda y luego mediante el método Swap de la clase Utils es que cambian los roles.

- Como jugador debe posicionar las naves antes de comenzar con el primer movimiento. Esto es responsabilidad de la clase Game mediante un método PlaceShip() que con creator crea la nave según la opción elegida y modifica el tablero de naves. Hay una excepción puesta que no permite que se agregue un barco si ya hay otro en esa posición.
- 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, Board hereda hereda 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 es también que se busca la coordenada en la lista de barcos del jugador y la convierte a true con el método SearchCoordInShipsList(). Se imprimen ambos 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.
- Como jugador debe poder indicar una posición de ataque: Esto se puede observar en el método MakeShot() dentro de Player.
- Como administrador del juego, debe poder registrar usuarios: La clase Administrator tiene un método que registra usuarios con un “id” que se crea automático propio del usuario y su “name” o nombre y “surename” o apellido. Se guarda en UserContainer que también es estático. Si el usuario ya existe se lanza una ContainerException.

- Como administrador del juego, debe poder conectar dos jugadores que se encuentren esperando para jugar: Esto se ocurre con el método JoinUsersAndPlay() que crea una partida entre los usuarios que llegan al lobby.
- 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 .
- Como administrador del juego, luego de cada ataque debe indicar el resultado del mismo a ambos participantes (si el ataque coincide con la posición de un barco: "Tocado" en caso de haber más partes aún sin atacar, "Hundido" sino quedan partes de la nave por atacar, o bien si no ha tocado ninguna parte de una nave indicará "Agua". 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: Se cumple en el método EndGame() de la clase Administrator.

En cuanto a los test unitarios para las historias de usuario anteriormente mencionadas, no se pudo llegar a terminar para esta segunda entrega y esperamos que para la última se pueda llegar.

Solución V

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. Al seguir investigando acerca de los distintos patrones GRASP y principios SOLID que existen es que se llegó a la solución número 5 que será la solución propuesta para llevar a cabo el proyecto. Las clases son: Administrator, Game, User, Player, AbstractBoard, Board, ShotBoard, ShipBoard, Ship, Submarine, LightCruiser, Frigate, Spot, Utils, IPrinter, BoardPrinter, GamesContainer, LobbyContainer, UserContainer, ContainerException, ReceiveShotException, OptionException, SizeException, GameNotFinishedException, CoordException.

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 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 Administrator es quien toma decisiones y dirige de cerca las acciones de los demás. El rol de estructurador es de 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. La clase Board es proveedor de servicios ya que realiza el trabajo de crear el tablero como una matriz bidimensional.

Se tiene una clase que cumple el rol de interfaz que es IPrinter la cual será implementada por BoardPrinter.

Una gran variación de la primera entrega a esta, además de los Containers y las excepciones 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".

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 o 4 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. 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 depende del tamaño del tablero, siendo esta el 50% del tamaño del lado del tablero. Se obtiene como un atributo del tablero a crear, se llama MaxShipQuantity u es de tipo int.

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.

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. 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.

Conclusiones del trabajo y problemas a los que nos enfrentamos

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

Esta entrega permite tener aún más acercamiento al proyecto y estructura del mismo. Se comenzó a comprender el funcionamiento del bot y sirvió para verificar algunos métodos que se pensaban correctos pero no fueron así.

El equipo ahora entiende mejor las características del proyecto y así es que se debe seguir trabajando de manera más ordenada, dividiéndose y delegando tareas. También permitió observar aún mas cómo aplicar de manera práctica los patrones GRASP y principios SOLID que se aprendieron en el curso aunque el equipo reconoce que faltan algunos por cumplir.

Se reconoce que faltan tests para las historias de usuario, así como también un funcionamiento completo del bot.

En el diagrama UML se observan todas las clases del juego realizadas y sus relaciones. Falta agregar, terminar y testear Handlers.

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

Todo lo que falta estará para la entrega final.

En nuestra opinión, faltan algunas cosas para poder decir que se cerró la entrega con éxito. Esta última semana 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.

Bibliografía

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

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