Universitat de Barcelona

Client-Servidor

Enfonsar la flota

Pràctica 1

Author:
Christian José Soler
Nicolás Martín Forteza Ocaña

6 d'abril de 2016

Introducció

En esta práctica se nos pide implementar el juego célebre *Hundir la flota* para que juegan diferents clientes contra un mismo servidor.

El objetivo del juego es adivinar la situación de los barcos del enemigo y hundirlos indicando las coordenadas donde creemos que están. Para que cliente y servidor se entiendan entre ellos se debe implementar el protocolo espeficado en el enunciado.

Nuestra finalidad pues será implementar este programa, aprendiendo así a utilizar los mecanismos de programación cliente-servidor de Java, como la API de Sockets y de Selectors.

El cliente deberá tener un modo manual, un modo con IA sencilla y un modo con IA más compleja para jugar contra el servidor. El servidor deberá tener los mismos modos, exceptuando el manual.

Tendremos que implementar dos versiones de servidor que para cliente serán ambos transparentes: Un servidor donde cada cliente será gestionado por un thread aparte y otro, donde el servidor recibirá los mensajes de los clientes mediante un selector y gestionará todas las partidas a medida que van llegando los mensajes de los clientes.

Proceso de desarrollo

Inicialmente, tuvimos que plantear bien el proyecto y hacer un buen diagrama de clases, ya que sabíamos que eso permitirá que nuestro código sea robusto y a la larga, nos ahorrará mucho tiempo y prevendrá posibles errores.

Nuestro primer objetivo fue pues crear una versión local del juego, donde el cliente juega contra su propio tablero, pudiendo así probar la mecánica del juego básico, sin comunicación.

Una vez arreglados los errores de flujo básico, comenzamos a preparar la comunicación y simultáneamente empezamos a poner la base del servidor con Threads. En este punto nos dimos cuenta que estos proyectos compartían tanto código que merecía la pena crear una librería común, llamada swd-core, donde vinieran todas aquellas clases que ambos usan. Eso se debe a que nos parecía poco óptimo tener que modificar el mismo código varias veces.

Como hemos ido aplicando bastantes patrones de diseño y creamos swd-core, hemos podido reducir considerablemente el tiempo que necesitamos para hacer funcionar la comunicación entre cliente-servidor.

Cuando conseguimos tener estable la versión cliente-servidor con threads, pasamos a la última tarea de esta entrega: crear el servidor con selectores. Nos inspiramos en el ejemplo que se nos dio en clase. La verdad, no nos costó demasiado porque swd-core ya tenía todas las herramientas necesarias para hacer funcionar el juego y el ejemplo de clase era bastante completo.

Hechas todas las tareas, hicimos nuestras últimas pruebas en local, y posteriormente, en la sesión de test.

Con eso acabamos el trabajo.

Detalles del desarrollo

En este apartado iremos detallando las particularidades de la implementación de cada proyecto.

Empezaramos hablando de la librería swd-core que contiene la mayoría de las clases de la implementación (unas 20). Vamos por paquetes:

1. **Communication:** Se encarga de todo aquello referente a comunicar cliente con servidor y viceversa. La clase principal es Communication. Esta es la clase que se relaciona directamente con ComUtils y su tarea es preparar los mensajes a mandar y acabar de enviar-los con la ayuda de ComUtils, al igual que recibir mensaje con la ayuda de ComUtils.

2. **Controller:** Contiene la clase Controller que sirve para controlar el flujo correcto del juego. Similar al del patrón MVC, pero en nuestro caso acabó volviéndose una Fachada prácticamente, ya que sirvió como capa de abstracción tanto para cliente como servidor.

   (a) **subpaquete gameModes:**

   - **GameMode:** Clase abstracta que representa un modo de juego. Tiene dos métodos, uno abstracto que genera la siguiente posición a disparar y un otro que guarda el último movimiento en la cola de movimientos ejecutados por el algoritmo. El último movimiento si se guarda o no va en función de la respuesta del adversario, porque si es un ERROR consideramos que ha habido problemas al enviar la casilla a disparar. Esto es para evitar juegos que no terminan, porque podría pasar que justamente no se mandara bien alguna casilla que hubiera acabado en HIT.
   - **ManualGame:** Representa el modo de juego manual
   - **RandomAI:** Representa el modo de juego que dispara aleatoriamente sin repetir
   - **BetterAI:** Representa el modo de juego con una pequeña IA. Dispara aleatoriamente hasta que le dé a algún barco, en cuyo caso explora las 4 direcciones hasta hundirla.
   - **GameModeFactory:** Aplicación del patrón Factory donde se crea el modo de juego en función del parámetro que nos viene por consola.

3. **Model:** Representa lo que es el modelo del juego, es decir el tablero con sus celdas y barcos. Es el conjunto de clases que se encarga de recibir disparos y controlar que el adversario no repita casillas (en cuyo caso se manda un error al adversario y se termina la comunicación).

4. **Utils:** Como su nombre indica, contiene una serie de utilidades. Entre estas se encuentran:

    - **Command:** Enum que representa todos los posibles comandos a mandar con su código correspondiente.
    - **Actor:** Enum que representa cliente o servidor con su timeout y texto a escribir en el log correspondiente.
    - **Orientation:** Enum que representa la orientación de un barco. No tiene utilidad real, es simplemente para organizarse mejor el código.
    - **ShipType:** Enum que representa los tipos de barco con su nombre, longitud y cardinalidad inicial.
    - **LogCreator:** Clase que se encarga de la escritura de logs en función de qué Actor qué Comando hizo con qué Params.
    - **Message:** Representa los mensajes que se mandan y se reciben del adversario. Tiene dos atributos: su Command y sus Params (params puede ser null). Es una pequeña aplicación del patrón Builder, aunque no del todo: Message es un objeto que tiene sentido de por sí sin llamar a build, pero el método *buildPackage()* crea el String que se deberá mandar al adversario.
    - **ComUtils:** La clase que se nos dió, aunque algo modificado. Es Communication el único que accede a él. En caso del Selector ni se usa.

5. **Exception:** En este paquete hemos creado dos excepciones nuevas:

    - **ReadGridException:** Salta en caso que haya problemas al crear el tablero.
    - **EndGameException:** Salta en caso que el juego acaba sea por error y porque alguien ganó. Se usa solo con el selector, porque el acabar las conexiones debe efectuarse en otro sitio.

Procedemos a hablar del proyecto cliente, es decir de swd-client. Tiene todas las clases de swd-core y además:

- **Client:** Se encarga de parsear los argumentos de línea de comandos y llamar a la clase que prepara, ejecuta y finaliza el juego. Si hay problemas al parsear los argumentos, acaba.
- **Game:** Se encarga de preparar, ejecutar y luego terminar el juego.
- **ClientCtrl:** Hereda de Controller y se le añaden las funciones específicas de la comunicación mediante ComUtils.

Procedemos a hablar del proyecto server con threads, es decir de swd-server-threads. Tiene todas las clases de swd-core y además:

- **Server:** Se encarga de parsear los argumentos de línea de comandos y llamar a la clase que prepara, ejecuta y finaliza el servidor. Si hay problemas al parsear los argumentos, acaba.
- **ServerThread:** Clase principal del servidor y se encarga de aceptar clientes, asignarles un thread y cuando termine de servirles cerrar la comunicación con ellos.
- **Game:** Clase que representa el thread que se le asigna a cada cliente al conectarse con el servidor.

Finalmente, queda el servidor con selector, es decir de swd-server-selector. Tiene todas las clases de swd-core y además:

- **Server:** Se encarga de parsear los argumentos de línea de comandos y llamar a la clase que prepara, ejecuta y finaliza el servidor. Si hay problemas al parsear los argumentos, acaba.
- **ServerSelector:** Clase principal del servidor y se encarga de aceptar clientes, asignarles un juego e ir recibiendo sus mensajes, pasándoles a los threads.
- **Game:** Clase que representa el juego del cliente al que se le asigna este juego. Es una máquina de estados que va recibiendo los cachos de mensajes que le van llegando por el selector, los junta, actualiza el juego y devuelve el mensaje que se le debe mandar de vuelta al adversario mediante la clase ServerSelector.

Anexos

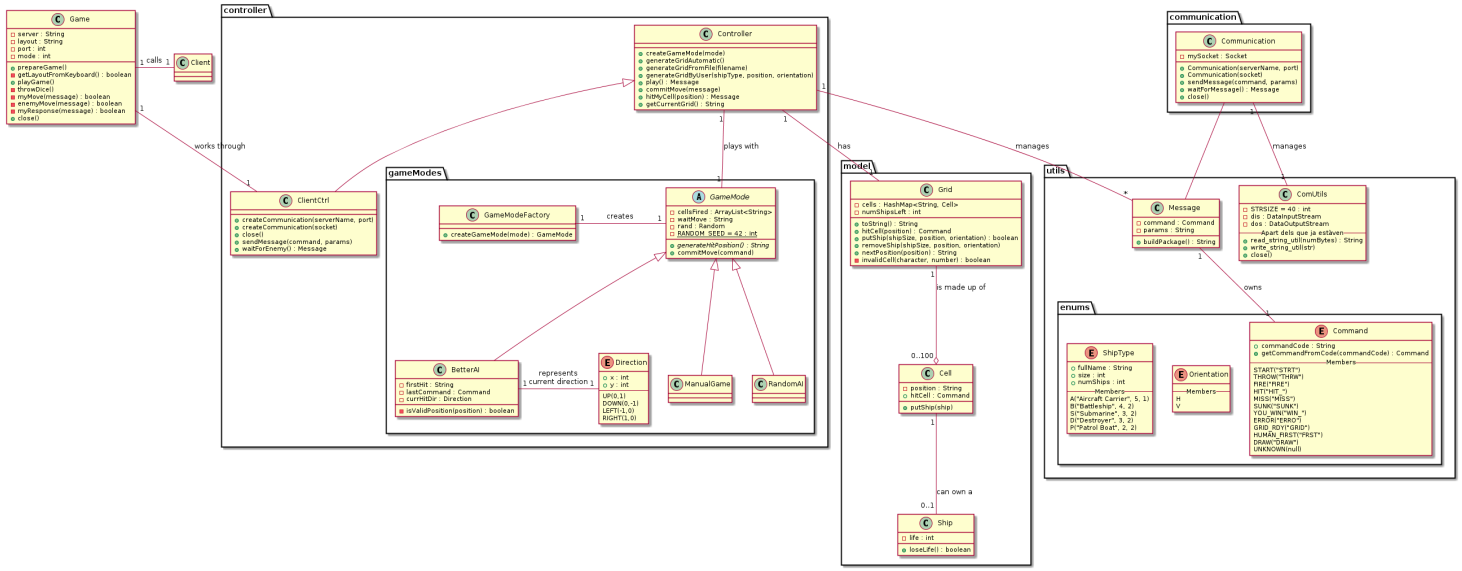

Figura 1: Client Diagram

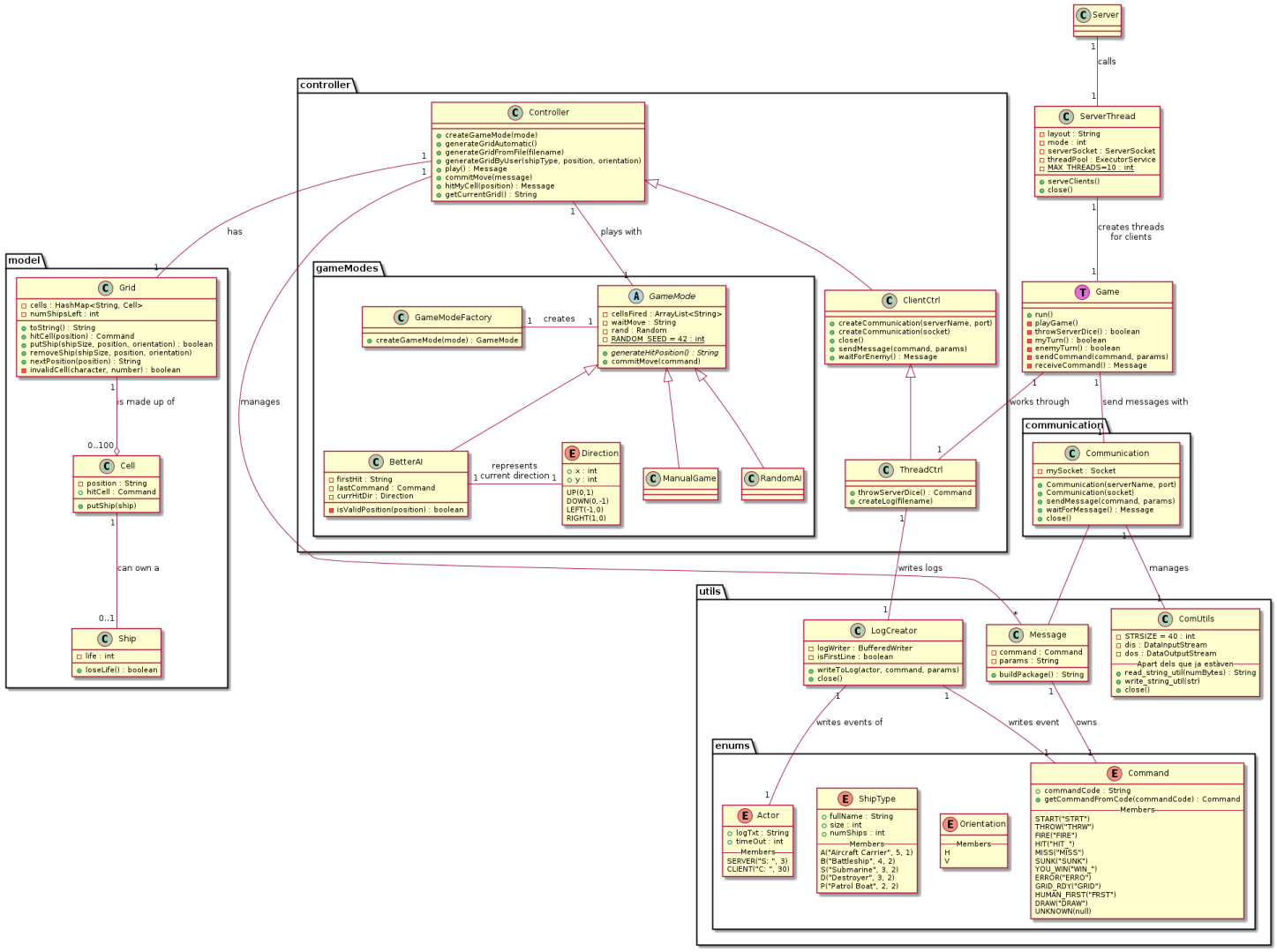

Figura 2: Threads-Server Diagram

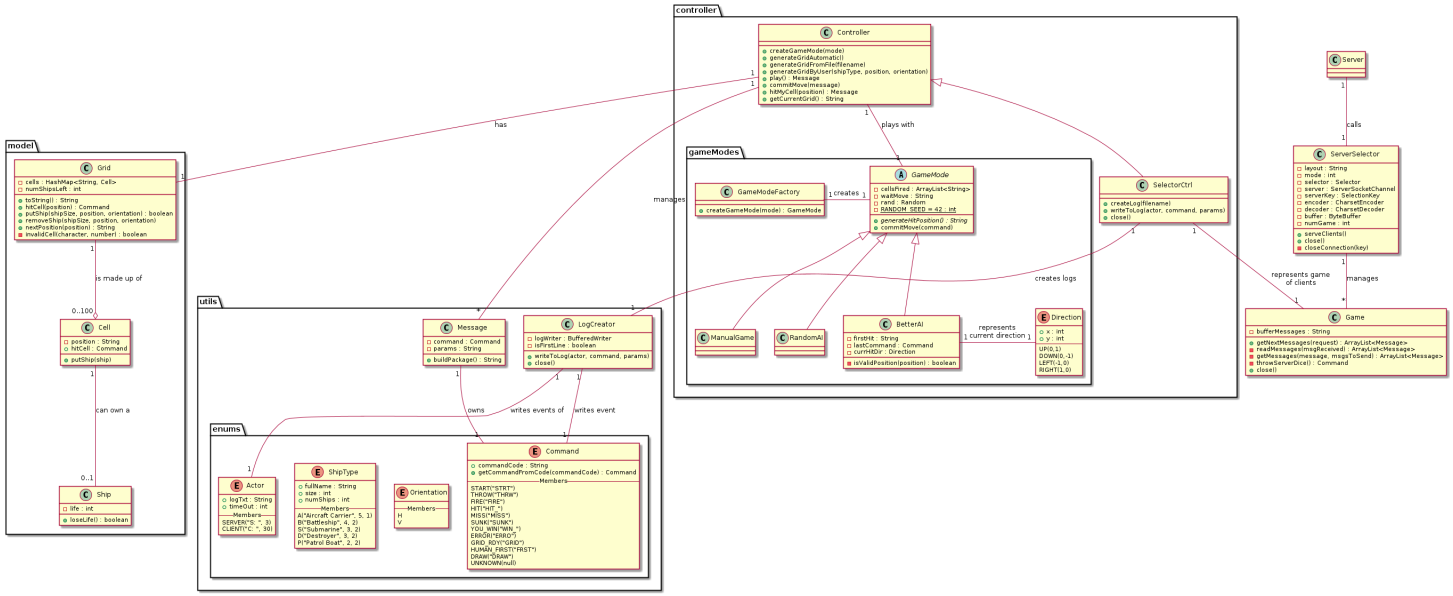

Figura 3: Selector-Server Diagram