30-11-2016

Práctica 2

Ampliación de Programación

Oscar García de Lara

Jorge Cifuentes Fernández

APARTADO A:

Hemos implementado diferentes ficheros:

- **API_Hundir_Flota**: Incluye las definiciones de tipos y las funciones que llaman la interfaz del juego y las funciones de uso interno.
- **GUI_Hundir_Flota**: Son las funciones que se comunican con el usuario, permitiendo que introduzca los datos por teclado, y se encargan de usarlos conjuntamente con la API. Para jugar llamamos a *mainPlay*.
- **Tester**: Algunas pruebas que hemos realizado para comprobar la API.

DATOS

- **Coordenada**: La usamos para indicar la posición de la casilla o la inicial del barco, tiene dos campos X::Integer, Y::Integer.
- **Estado**: Contiene la situación que puede tener la casilla.
- **Casilla**: Contiene una coordenada y su estado.
- **Barco**: Tiene dos constructores, uno para los barcos verticales y otro para los horizontales. Ambos tienen una coordenada inicial y su tamaño.
- **Tablero**: Es una lista de casillas.
- **Flota**: Es una lista de barcos.
- **Mosaico**: Es una lista de string, que nos sirve de ayuda auxiliar para poder imprimir la Flota o el Tablero.

INSTANCIAS

Todos instancian read y show, destacamos estas dos:

Estado implementa Ord, para cada uno tenga un peso y poder hacer comparaciones más rápidas.

Y usando el pragma {-# OVERLAPPING #-} en instance show [a], nos permite tener un show independiente del show de a, ya que es más legible y usable poner print flota, por ejemplo, que tener que llamar print $ mostrarFlota flota.

FUNCIONES

API_HUNDIR_FLOTA

- Hemos implementado las funciones *incluirFlota*, *incluirBarco*, *incluirTablero* e *incluirCasilla* que son muy similares a las equivalentes en los ejercicios del miércoles incluirCuadrado.
- Para poder añadir un barco usamos *agregarBarco*, donde comprobamos que sus coordenadas estén dentro del límite y que tenga espacio suficiente.
    - Para comprobar espacio nos ayudamos de listas por comprensión para generar las coordenadas alrededor del barco, y comprobamos para cada barco de la flota que sus coordenadas no estén en la lista generada por las listas por comprensión.
- Para obtener las coordenadas que ocupa un barco está *posicionesBarco,* que gracias a una lista por comprensión recorre la coordenada X o Y, según sea un barco horizontal o vertical.

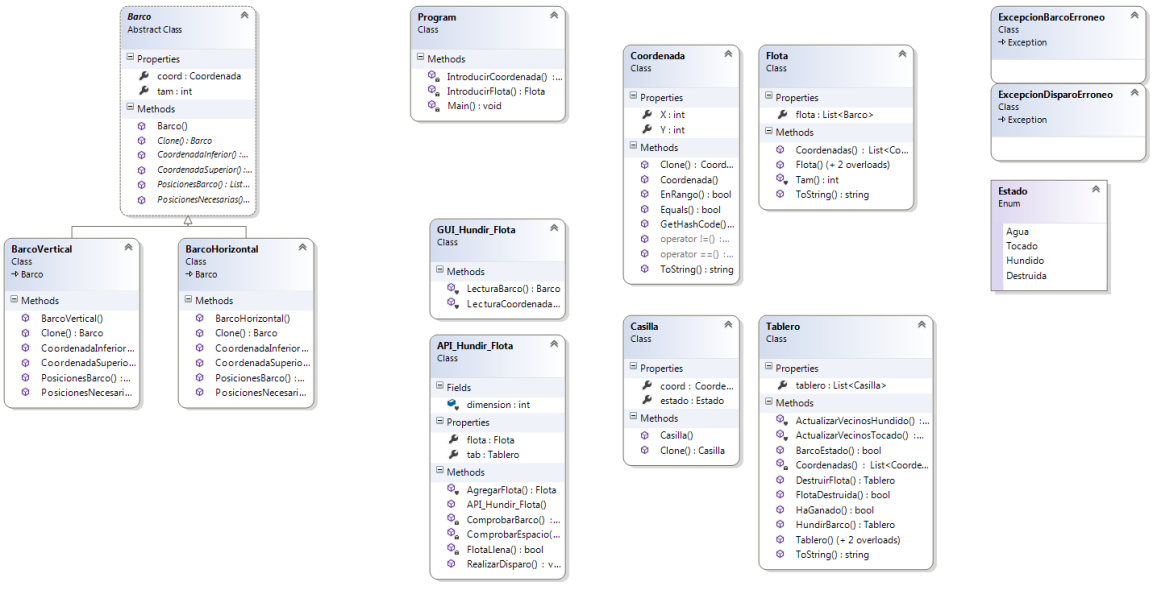
- Con las funciones *actualizarVecinos* obtenemos una lista de casillas con agua que tiene cada posición tocada de un barco. Para evitar añadir casillas fuera de rango del tablero usamos un filter (ver código).
- El método *barcoEstado* nos permite saber si todas las coordenadas de un barco están en el tablero y tienen el mismo estado, si esto es así devuelve True, si no False. Esto nos permite tener un código más limpio ya que tenemos otras funciones como *barcoHundido* que la usan.
- También tenemos *hundirBarco,* que cuando se confirma que el barco está hundido genera un nuevo tablero a partir del anterior cambiando los estados del barco de tocado a hundido. Además *destruirFlota* que hace lo mismo que *hundirBarco* pero cambiando todos los barcos a destruido.
- Con *realizarDisparo* comprobamos que la coordenada proporcionada esté dentro de rango, y si ha sido contra agua o un barco. En caso de tocar un barco, hacemos todas las comprobaciones para confirmar si el disparo ha tocado o hundido el barco, y en última instancia si ha destruido toda la flota.

GUI_HUNDIR_FLOTA

- La función *introducirFlota* usa una auxiliar *introducirFlota'*, que recibe una *flotaInicial* (lista vacía), lo cual le permite introducir al jugador tantos barcos como quiera (siempre teniendo en cuenta el límite de tamaño de la flota). Para pasar de String a Barco no usamos read, si no la variante *readMaybe* que nos devuelve Nothing si la cadena introducida es incorrecta o un Just barco. Esto permite controlar errores con strings inválidos.
- Con la función *realizarDisparos* también usamos la auxiliar *realizarDisparos',* donde le pasamos el *tableroInicial*, también usamos *readMaybe* para controlar una coordenada invalida. Al jugador se le permite realizar tantos disparos como quiera teniendo la única condición de parada es que se destruya la flota y gane el juego.

APARTADO B:

Diagrama de clases:



La lógica del juego la hemos dividido en más archivos respecto a la versión de Haskell, así que pasaremos a comentar las diferencias entre ambas versiones. No hemos usado ninguna estructura ya que, en principio, no nos aportaba nada nuevo, y tenía algunos inconvenientes como no ser nullable.

En general, el estilo de código en Haskell es más conciso y elegante, aunque tiene algunos puntos "molestos", como la indentación o la falta de un IDE estandarizado como Visual Studio, con sus múltiples opciones.

PROGRAM

En este archivo tenemos el propio main que nos permite jugar y dos funciones para introducir coordenada y flota. Ambas muestran información por consola, intentan leer la coordenada o flota y cogen la excepción en caso de error, reintentando la introducción por teclado. Las funciones que procesan el string introducido las hemos movido a la GUI, en aras de la simplicidad y reusabilidad.

GUI_HUNDIR_FLOTA

Esta clase tiene dos funciones estáticas e internas: LecturaBarco y LecturaCoordenada. Ambas reciben un string (el que introduce el usuario por teclado), lo dividen (función Split), e intentan parsearlo (función tryParse que nos evita tener que hacer un manejo explícito de las excepciones). SI todo esto funciona, se crea el Barco o la Coordenada y se devuelve. En este archivo se usan las excepciones creados por nosotros mismos, usando un bloque try-catch.

En general, hemos encontrado más facilidades para la introducción por teclado en C#, al no existir esa separación de código puro/impuro.

API_HUNDIR_FLOTA

Hemos creado una clase, que hará la función de juego, con dos propiedades internas: tablero y flota. El hecho de que C# sea orientado a objetos en este caso es una ventaja, ya que agrupamos un tablero y una flota en una sola clase, a diferencia de Haskell.

Dentro de esta clase hemos implementado varias funciones, en su mayoría auxiliares (comprobar si hay espacio para un barco, si la flota está llena en esta instancia del juego, realizar un disparo o agregar una flota al juego).

En cuanto a las diferencias con HS, este lenguaje nos ha supuesto una facilidad a la hora de codificar "realizarDisparo" (la función más compleja de este archivo), tanto por no tener que usar recursividad como por las facilidades para actualizar el tablero (incluso con la inmutabilidad).

Los restantes tipos de datos los hemos movido a diferentes archivos para hacerlos **clases** o **estructuras**:

COORDENADA

Hemos creado una clase Coordenada que, como en Haskell, tiene una posición cartesiana X e otra Y.

Lo que en Haskell conseguíamos con un instance de Eq, aquí lo hacemos con las funciones estáticas "==" y "!=", y un override de Equals, por lo que en este aspecto ambos lenguajes se parecen (aunque la instancia de Haskell sea más clara y concisa).

Esta clase además tiene un método para clonar instancias, uno para comprobar si una coordenada está en rango y un ToString. Las dos últimas se parecen bastante a sus equivalentes de HS, pero el hecho de tenerlas en una clase separada y de ser métodos de instancia, en nuestra opinión, añade claridad al código.

CASILLA Y ESTADO

Casilla nuevamente es una clase, que define en su interior Estado (ya que el estado es algo inherente a una casilla), que en esta versión hemos decidido que sea una enumeración, por simplicidad. Casilla tiene dos elementos, una coordenada y un estado, con propiedades automáticas. Aquí solo hemos creado una función Clone que clona una instancia.

TABLERO

Tenemos una clase tablero, pero en vez de definirla directamente como una lista de Casilla, hemos creído más conveniente añadir una propiedad que sea dicha lista de Casillas. Hemos añadido tres constructores (uno normal y dos para clonar una instancia añadiendo o no un nuevo elemento).

Tablero tiene un método para devolver las coordenadas presentes, un ToString y varias más:

- BarcoEstado: esta ha sido más tediosa de implementar aquí, al no disponer de un filter y demás funciones de orden superior como en HS.
- HundirBarco: de igual manera, está función es más larga que si correspondiente en HS al usar un if y un foreach en vez de map.
- DestruirFlota: sucede lo mismo que con la anterior, cambiamos una función de orden superior por una iteración con foreach.
- Las funciones de actualizar vecinos son más largas en esta versión, pero son más sencillas de entender (solo usamos constructores y un Add).

BARCO

Barco está implementado como una clase abstracta de la cuál heredan BarcoVertical y BarcoHorizontal. Pensamos que es la manera lógica de implementarlo, dado que no hay patrones en C#. Este cambio respecto a Haskell nos da la ventaja de indicar ciertas funciones abstractas que deben codificarse, lo que evita errores (por ejemplo, cometer el olvido en HS de no poner un patrón para un tipo de barco en una función).

Como métodos tenemos dos que devuelven las posiciones del barco, dos que devuelven la coordenada inferior y superior, y uno para clonar instancias. Las funciones que crean una lista de coordenadas han tenido que ser implementadas con un bucle principal que añada cada coordenada, comprobando ciertos requisitos (ver código). Esta implementación en Haskell era más directa usando simplemente listas por comprensión, especificándole ciertas restricciones.

FLOTA

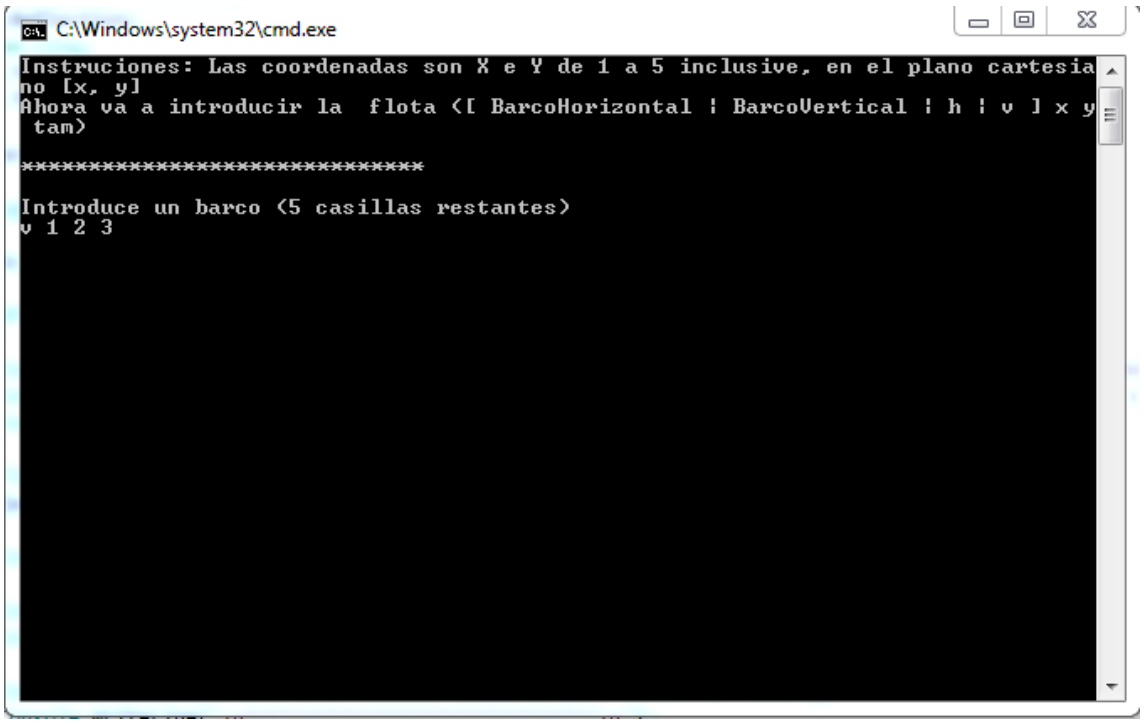
Flota está declarada como una clase, con una propiedad lista de barcos. La diferencia respecto a Haskell es que flota no es directamente una lista de barcos, lo que nos da mayor versatilidad. Tenemos constructores que clonan la flota (para evitar que se modifique una instancia, se crea una nueva y se añade lo que haga falta, simulando así la inmutabilidad de Haskell). Además, hay un ToString similar al Show de Haskell y dos funciones auxiliares (coordenadas de una flota y tamaño de esta, ambas con bucles foreach).

EXCEPCIONES

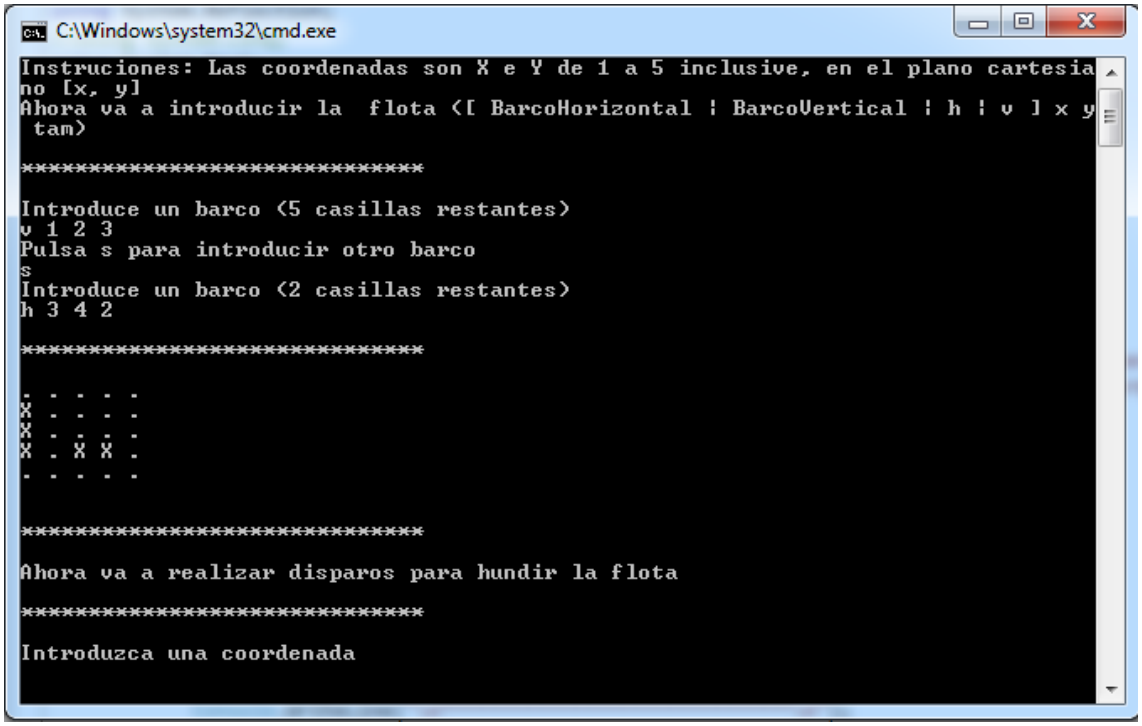
Hemos creado dos excepciones para poder usarlas en la GUI, ExcepcionBarcoErroneo y ExcepcionDisparoErroneo. En C# el manejo de excepciones nos parece más conciso que en Haskell.

IMÁGENES DEL FUNCIONAMIENTO

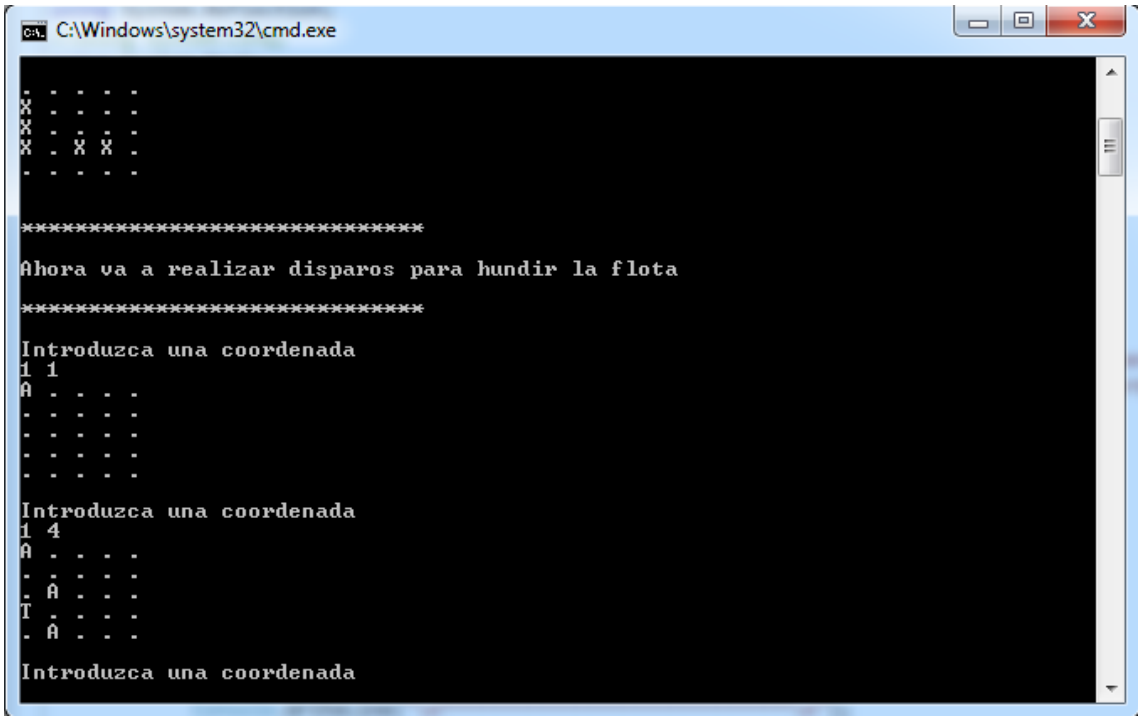
Introducción de barcos por teclado:



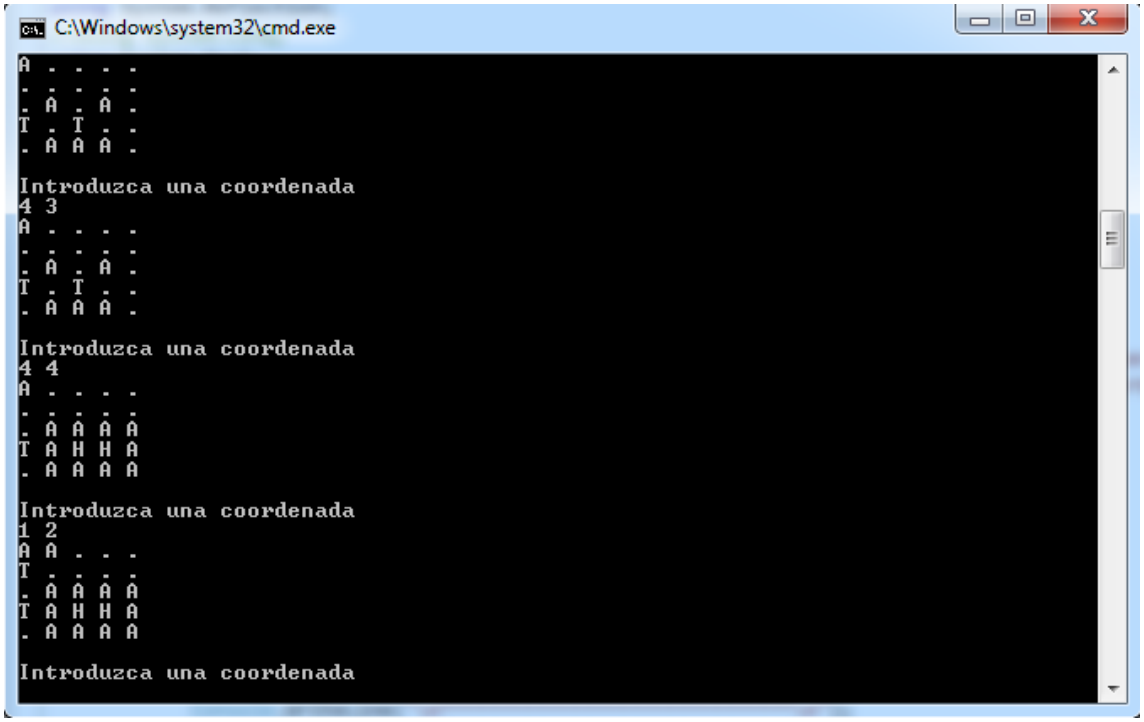
Aquí se muestra ya la flota creada:



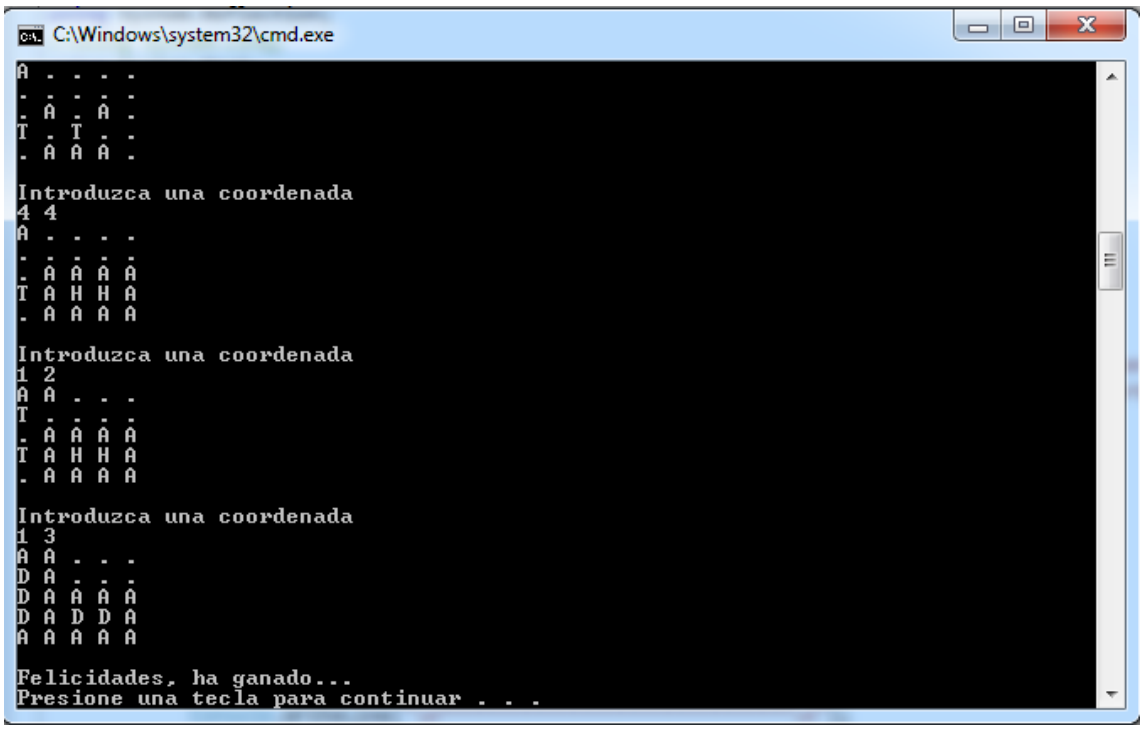
Introducimos coordenadas y se nos va mostrando el tablero actualizado:



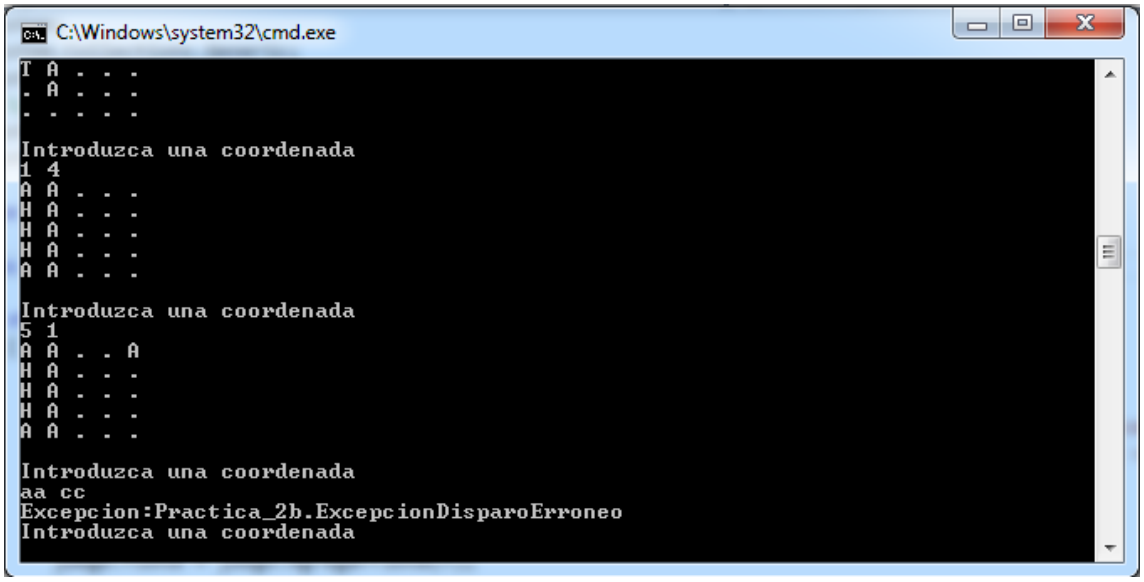
Hundimos un barco y vemos como efectivamente se actualizan sus vecinos:



Destruimos toda la flota y se nos avisa del fin del juego:



Probamos a introducir una coordenada mal:



Probamos a introducir un barco mal: