# Informe Proyecto Grupal – Programación I

Por: Santiago Panozzo y Mateo Rodriguez

# Análisis del problema:

## Tarea por realizar:

El trabajo que llevar a cabo trata de la realización de una versión simplificada del popular juego *“Batalla Naval” (Battleship)* desde cero en el lenguaje de programación Python. El juego consta de un tablero cuadrado con una cantidad de casillas especificada por el jugador (de 3x3 a 10x10), asimismo el número de enemigos y la dificultad serán elegidas por este. La principal diferencia con el juego original es el hecho de que esta versión es para un solo jugador, constando de barcos enemigos que se posicionan de manera aleatoria en casillas del tablero; el trabajo del jugador es localizar estos barcos indicando coordenadas para atacar. El jugador deberá entonces atacar a las casillas donde crea que los barcos enemigos se encuentran y, dependiendo de si acertó o no en su predicción, derribará el barco que en esa casilla se encontrase.

El problema recae en la implementación de las varias mecánicas del juego. Si bien la simplificación de las mecánicas del juego facilita el desarrollo del proyecto, sigue siendo una tarea compleja que involucra el uso de diversas funciones trabajando en conjunto para generar cada elemento del juego utilizando la aleatoriedad como herramienta.

En nuestro caso, nos decantamos por crear varias funciones separadas que generaban cada parte del juego, desde el propio tablero hasta la generación aleatoria de enemigos y el disparo del jugador. También tuvimos que programar los diversos menús que quisimos implementar para la facilidad de uso del programa.

## Posibles maneras de llevar a cabo la tarea:

Para la creación del tablero en sí se nos ocurrió utilizar un objeto de tipo lista de Python. Dentro de esta lista, habría otra lista para representar las filas horizontales del tablero, dentro de la cual habría otra lista para cada columna y un diccionario para cada casilla individual. Como cabría de esperarse, esto resultó innecesariamente complejo y, tras una mejor organización, dimos con la solución más simple de usar una lista para las filas, otra para las columnas y diccionarios para las casillas. Otras posibles soluciones podrían haber sido utilizar directamente diccionarios para cada casilla y prescindir de las listas, lo cual habría sido más simple; sin embargo, cabe recordar que el usuario es quien especifica el número de casillas, por lo cual habría que hacer de manera manual como mínimo 100 diccionarios (la cantidad de casillas para un tablero de 10x10 que se considera el más grande posible), lo cual sería bastante tedioso y no muy organizado. Algo similar ocurre con usar solo una lista, o dos, al final sería menos eficiente. La manera que elegimos es considerablemente más optimizada y ahondaremos sobre ella más adelante.

En cuanto a la generación de enemigos, se podría hacer de varias formas. Por un lado, podríamos aprovechar los diccionarios que representan cada casilla del tablero y almacenar en ellos los enemigos, dependiendo de su posición, con todos sus datos pertinentes. Por otro lado, podríamos llevar una o varias listas de enemigos, o un diccionario, separados del tablero en sí, que tuvieran almacenados la posición de cada enemigo y su estado. En nuestro caso decidimos ir por la primera opción, ya que aprovechaba de mejor manera el tablero y nos podía ser más cómodo para realizar ciertas operaciones.

En cuanto a partes más visuales, el tablero en sí podría ser representado de varias maneras. Se podría utilizar una forma sencilla como simples paréntesis rectos (“[ ]”) ya que tienen una forma relativamente cuadrada y representan bien las casillas. También está la opción de utilizar caracteres especiales como emoticonos o símbolos ASCII. Si bien partimos utilizando la primera opción ya que era más simple, pudiendo representar los enemigos y disparos con facilidad (agregando por ejemplo una “[ x ]” para representar los disparos), terminamos utilizando emoticonos que se veían mejor gráficamente con un estilo más consistente y cómodo a la vista.

Otra manera de realizar todo esto que planteamos hasta ahora hubiese sido emplear clases y objetos. Podríamos crear una clase “casilla” con métodos para establecer si tiene un enemigo o no o si el usuario interactuó con ella, así como funciones para colocar enemigos, disparar a la casilla, mostrarla en pantalla, etc. Podríamos hacer casi prácticamente todo el juego en una clase; el problema recae en que, de nuevo, el tablero tiene un número de casillas entre 9 y 100, y no es muy realista generar 100 objetos “casilla” de manera procedural, tendríamos que hacerlo manualmente y podrían incluso estar ocupando memoria del sistema sin ser utilizados (por ejemplo, en un tablero de 3x3, las 91 posibles casillas restantes no serían utilizadas, pero estarían gastando recursos de todos modos). Por todo esto decidimos no seguir esta idea.

# Explicación del código

## Generación del tablero:

Como mencionamos anteriormente, la primera tarea en la que decidimos embarcarnos fue la de generar el propio tablero para el juego, teniendo en cuenta que el jugador sería quien ingresara las dimensiones del mismo.

Para esto creamos una función que obtiene como parámetros las dimensiones del tablero (la cantidad de casillas en uno de los lados del cuadrado). Donde “N” es la cantidad de casillas de cada lado y N2 es por tanto el área del cuadrado, la función genera una lista con N elementos que representa cada fila horizontal del tablero, las cuales denominaremos con números del 3 al 10 (debido a que el tablero debe tener unas dimensiones de mínimo 3x3 y máximo 10x10 casillas). Para ejemplificar, supongamos un tablero de 7 de lado, es decir N = 7, la función creará una lista llamada “Filas” con 7 elementos *(Figura 1)*.

A continuación, la función convierte estos elementos en sub-listas (a las que llama “columnas”) que contienen cada columna vertical *(Figura 2)*, cada elemento de estas sub-listas está inicialmente vacío. Posteriormente, la función cambia los elementos vacíos por diccionarios, que son los que realmente representan cada casilla individual en sí, con parámetros que representan la presencia de un enemigo en cada una y el estado del enemigo (*Figura 3*). Por defecto esta función simplemente colocará todas las casillas como vacías de enemigos y además establecerá que el jugador aún no ha interactuado con las mismas.

Table

Description automatically generated Table

Description automatically generated

Figura 1 : Lista "Filas" elementos 1 a 7 Figura 2: Convertidos los elementos de “Filas” en listas llamadas “columnas”

Table

Description automatically generated

*Figura 3: Todos los elementos de “columnas” son cambiados por diccionarios individuales que representan cada casilla individual.*

## Muestra del tablero:

Una vez tenemos los parámetros del tablero generados, debemos mostrárselo al usuario por medio de la función de Python “print()”, la cual implementamos en nuestra propia función “*mostrar\_tablero()*”.

A esta función le pasamos letras como parámetro para que las imprima en la pantalla del lado horizontal, y números para que haga lo mismo del lado vertical, esto con el propósito de que el usuario pueda orientarse dentro del tablero. En otras palabras, añadimos una manera visual de representar las coordenadas de manera que, por ejemplo, a la casilla en tercera posición vertical y cuarta horizontal se le pueda referir como “3D”.

La función también se dedica de mostrar el tablero en sí, representando todas las casillas del mismo con un “⬛” *(Figura 4)* y, en base a la interacción que el usuario le dé a esta, cambiará de la siguiente manera: si atacó dicha casilla y no había un enemigo, ahora se la representara con una “o” *(Figura 5)*, en caso de ataque y que sí haya un enemigo, se la representara con una “x” *(Figura 6)*. A su vez, esta misma función le devolverá al usuario la cantidad de enemigos y tiros restantes.

## A picture containing table Description automatically generated Graphical user interface Description automatically generated Graphical user interface Description automatically generated

*Figura 6: Representación de casillas interactuadas con enemigos*

*Figura 5: Representación de casillas interactuadas sin enemigos*

*Figura 4: Representación de las casillas del tablero*

## Generación de enemigos:

La función *“gen\_enemigos()”*, por medio de la librería “random” de Python, toma posiciones aleatorias del tablero y coloca un enemigo en ella. Como mencionamos anteriormente, cada casilla es un diccionario con varias “propiedades”, por defecto no hay enemigos, los enemigos no están vivos, y el jugador no ha interactuado con las casillas; pero cuando ejecutamos están función, cambia en esas casillas aleatorias estas propiedades para reflejar que ahora sí hay un enemigo y se encuentra con vida. A medida que coloca enemigos, la variable que representa la cantidad de enemigos ingresada por el usuario se restará hasta quedar en 0, esto lo hacemos para que siga colocando enemigos hasta que estén todos en el tablero, sin contar las veces que la función elige la misma casilla aleatoriamente dos veces.

## Cálculo de la cantidad de disparos:

La función *“get\_disparos()”* realiza una operación matemática para calcular el número de disparos de los cuales dispondrá el usuario. Para que esto suceda, se le pide al usuario que ingrese la dificultada deseada (fácil, intermedio o difícil) y en base a su respuesta se toma un porcentaje u otro de las dimensiones del tablero.

Para la dificultad fácil se le da al usuario una cantidad de disparos equivalente al 70% de las casillas del tablero; para la intermedia se le da la equivalente al 50%; y para la difícil se le da el equivalente al 30% de las casillas. El resultado de esta operación será redondeado hacia arriba si la fracción supera el 0,1%, y hacia debajo de lo contrario.

## Cálculo de la coordenada que ingresa el usuario:

Para facilitar nuestro trabajo diseñamos una útil función llamada *“getCoordenada()”*, la cual ubica la coordenada indicada por el usuario en el tablero. La misma ordena las coordenadas ingresadas por el usuario y realiza una equivalencia de las letras del tablero en números para saber la posición de la que el usuario se refiere. También plantea una serie de condiciones para evitar errores. Es decir, si el usuario ingresa “2C” o “C2”, la función en ambos casos separará esto en “2” vertical y “3” horizontal, para luego realizar el resto de las operaciones necesarias en el juego.

## Disparar

La función *“disparar()”* recibe como parámetros el tablero, la cantidad de disparos restantes para el usuario, y coordenadas verticales y horizontales a las cuales disparar. Con esto, evalúa si el usuario cuenta con disparos disponibles, así como si la coordenada ingresada existe o no, si ya fue atacada y si hay un enemigo en ella. Habiendo evaluado esta información, devuelve True o False dependiendo de si el disparo fue exitoso. Posteriormente, en el código del juego, dependiendo de si el disparo fue exitoso o no, se cambia el estado de los diccionarios de cada casilla para reflejar la nueva información, si es necesario.

Para ejemplificar, supongamos que el usuario quiere atacar las coordenadas “4” horizontal y “2” vertical, esto lo pasaríamos a la función, que leería el diccionario que se encuentra en esas coordenadas del tablero. Si la coordenada ya fue atacada con anterioridad *(“testeada”: True)*, se lo comunica al usuario y ningún disparo es efectuado. Si la coordenada no fue atacada *(“testeada”: False)* y hay un enemigo *(“enemigo”: True)* la función cambia el estado del enemigo a muerto *(“vivo”: False)*, establece ahora que ya fue atacada *(“testeada”: True”)*, y descuenta uno de los disparos del usuario, notificándole que el disparo fue exitoso. En caso de que no hubiera un enemigo, simplemente se establece que ya fue atacada y se resta un disparo, pero devuelve que no el disparo falló.

## Probar:

Esta función es una función de prueba que simula el funcionamiento de las anteriores, pero con un par de cambios. Recibe como parámetros dos listas que contienen las posiciones de los enemigos y de los disparos que deben ser efectuados. Igual que las anteriores, establece los estados por defecto de los diccionarios (enemigos vivos donde deben estar, ninguna casilla fue atacada, etc.), con la diferencia de que la posición de los enemigos ya no es aleatoria. Estos estados van a cambiar según la interacción del usuario con las casillas, esencialmente simulando una partida en un instante, efectuando los disparos donde deben ir y cambiando los estados de los diccionarios apropiadamente. A su vez, según la cantidad de enemigos restantes, esta función definirá si el usuario ganó o perdió (si no hay enemigos restantes el usuario ganó, de lo contrario perdió), devolviéndole un mensaje con los resultados.

# Conclusiones:

## Santiago Panozzo:

Personalmente el trabajo me pareció bastante interesante, tuvimos que poner en práctica mucho de lo que aprendimos, especialmente al trabajar con listas y diccionarios. Cuando tuvimos la idea de utilizar clases y objetos me pareció una forma muy buena de intentar utilizar una programación un poco más compleja para optimizar el proceso, pero implicaba demasiado trabajo y que el proceso terminara siendo aún menos óptimo. Con todo, al menos fue una manera creativa de usar programación orientada a objetos que tendré en mente para un futuro trabajo que lo requiera ya que, aunque complicada y no muy viable, era una idea dentro de todo posible. Seguro hay una manera de hacerlo que no requiera tanto trabajo, si hay algo que cambiaría sería eso, que me encantaría haberlo hecho así, pero viendo que no se nos ocurrió como implementarlo estoy satisfecho con el resultado final de todas formas.

Hubo algunas partes que nos dieron bastante trabajo, en un principio estábamos haciendo el código de una manera muy desordenada ya que estábamos probando cosas que se nos ocurrían hasta que algo funcionara y lo íbamos dejando así. Eventualmente tuvimos que pararnos un momento y reescribir el código de una forma más ordenada y con comentarios que nos ayudaran a organizarnos mejor y entender de mejor manera nuestro propio código. No es decir que nuestro código fuera malo en un principio, hicimos lo que pudimos y el programa funcionaba para lo que le pedíamos, pero viendo que algunas cosas nos podían limitar a futuro decidimos reestructurarlo de manera preventiva. Al final nos ayudó bastante, ya que simplificamos bastantes cosas y refinamos nuestras ideas a algo más funcional y entendible, lo cual nos ayudó a programar de manera más rápida y precisa de ahí en adelante.

También aprendimos a usar herramientas colaborativas; para la mayoría de nuestro trabajo utilizamos una herramienta web llamada “Replit” que nos permitía trabajar en el código de manera simultánea en tiempo real, permitiéndonos así que nos dediquemos a diferentes cosas a la vez, ahorrando tiempo y dividiendo tareas. Tiene sus problemas, y en el proceso descubrimos otras herramientas similares que podrían ser más cómodas de usar, lo cual tendremos en cuenta para el futuro, pero nos fue de gran ayuda.

## Mateo Rodríguez:

Nosotros realizamos este proyecto usando listas y diccionarios, ya que fue la mejor manera que se nos ocurrió e implementamos lo aprendido en clase. En un momento pensamos en realizar el proyecto utilizando clases, si bien creemos que es una manera eficiente de hacerlo, no logramos adaptar nuestro código para que funcione utilizando este método.

El problema más grande que tuvimos en este proceso fue el orden y la optimización del código, al punto que tuvimos que borrar ciertas partes para reescribirlas de una forma más optima y ordenada. Nosotros programamos en simultaneo usando la página de “Replit”, lo que, en lo personal, me generó un pequeño percance, que fue el hecho de entender del todo el código de mi compañero y dar a entender el mío, aunque sé que es parte del proceso de programar en conjunto.

Esta experiencia pareció bastante interesante e innovadora, ya que nunca programe en conjunto con otra persona, lo que me parece muy útil para aprender e ir adentrándome en la creación de proyectos de forma grupal.