Instituto Tecnológico de Costa Rica
Sede Interuniversitaria de Alajuela

BV2019-DOC:

Documentación

BattleVoid

Implementación del juego de mesa Battleship.

Autores
Alejandro Rojas Jara
Jose Antonio Güell

18/10/2019

Tabla de Contenidos

1 INTRODUCCIÓN

1.1 Propósito

El principal objetivo de este proyecto es el de reforzar conocimientos en modelado y programación orientada objetos, mediante la implementación de un juego en Java. Se pretende que los estudiantes diseñen e implementen un Juego de Batalla Naval , en la realización del mismo los estudiantes harán uso de los siguientes temas estudiados en clase:

1. Modelado de Clases en UML.
2. Herencia y Polimorfismo.
3. Entorno de Desarrollo (IDE) para programar en Java.
4. Programación en Java.
    a. Objetos.
    b. Clases.
    c. Atributos.
    d. Métodos.
    e. Modificadores de visibilidad.
    f. Instanciación.

2 DESCRIPCIÓN DEL PROBLEMA

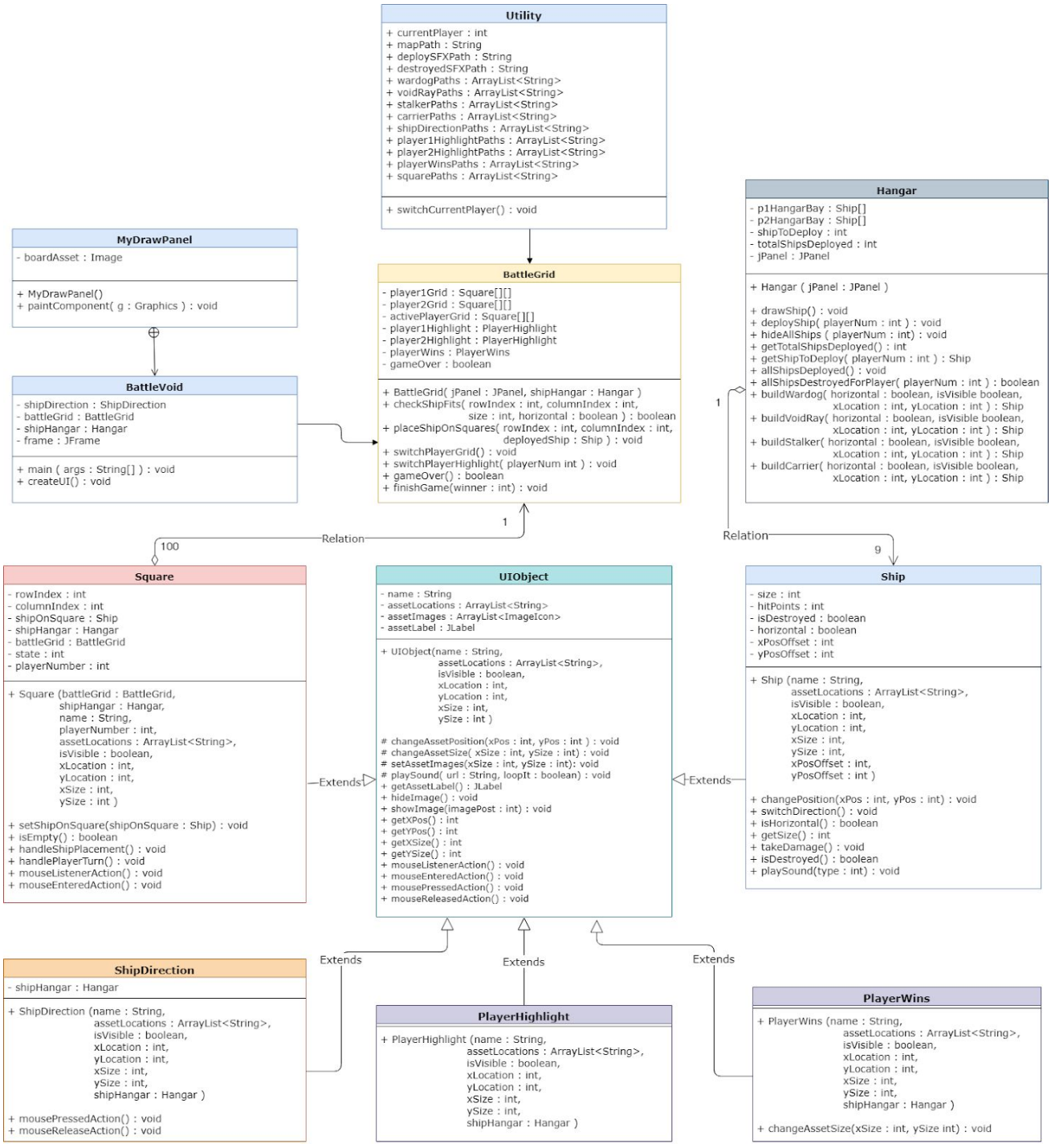
2.1 Alcance y Objetivos

En este proyecto se pretende implementar el juego de mesa Battleship (Batalla Naval) utilizando los conocimientos de modelado y programación orientada a objetos aprendidos en clase. Dentro del alcance del proyecto se incluye las siguientes funcionalidades y cualidades.

1. Interfaz gráfica para mejor visualización del juego.
2. Cada jugador podrá acomodar sus naves a su gusto. Sin embargo, el juego tendrá la lógica necesaria para evitar que dos naves se coloquen en un mismo espacio, ni que excedan de los límites del tablero.
3. Flujo de juego completo, es decir, se podrá jugar una partida de principio a fin.
4. Se utilizarán las reglas estándar del juego de mesa.
    a. Cada jugador tendrá un tablero de 10x10 casillas.
    b. Cada jugador tendrá 9 naves (2 de 1 casilla, 3 de 2 casillas, 3 de 3 y 1 de 4).
    c. Los jugadores toman turnos disparando hasta que algún jugador destruya todas las naves del enemigo, en cuyo momento gana la partida.

Queda fuera del alcance la capacidad de comunicación por red, por lo que las partidas serán locales en una única computadora.

3. DIAGRAMA DE CLASES



4. DISEÑO DEL PROGRAMA

4.1 Decisiones de diseño

Para este proyecto se decidió implementar interfaz gráfica, por lo que la mayoría de decisiones de diseño están basadas en este requerimiento.

Estructura del programa

Los archivos del proyecto se estructuraron crearon cuatro principales paquetes (*packages*) para organizar el código.

Entities: Contiene las 3 principales entidades del juego:

- **Hangar**: Construye y gestiona los objetos de las naves (Ship) de cada jugador.
- **Ship**: Gestiona los atributos lógicos específicos de una nave. Extiende de la clase ***UIObject*** para poder colocar la imagen de cada nave en la interfaz.
- **Square**: Gestiona los atributos tanto lógicos como de interfaz, de una casilla específica en el tablero. Extiende de la clase ***UIObject*** para poder colocar la imagen de cada casilla en la interfaz.

Game: Contiene las dos clases que generan e inicializan el juego y la interfaz.

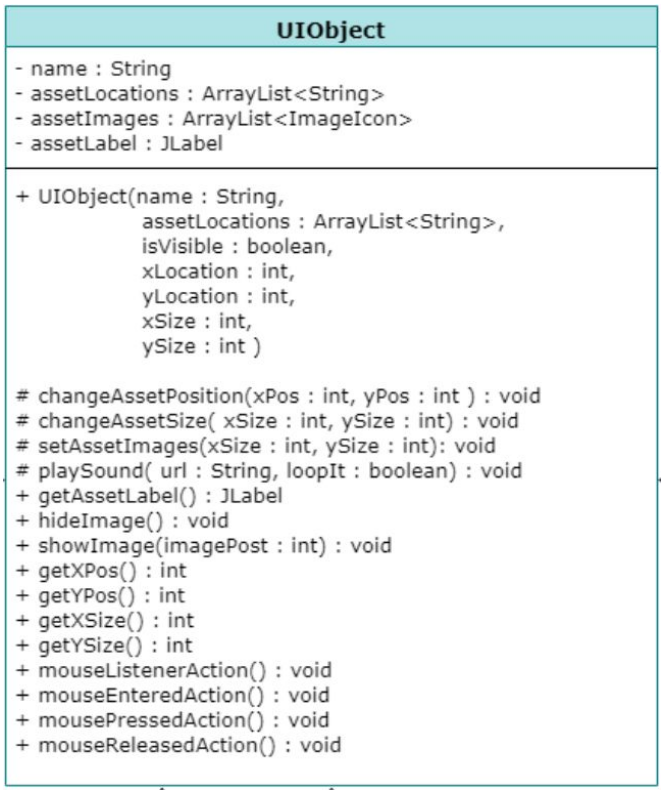
- **BattleGrid**: Construye los dos tableros de casillas (Square) 10x10 para cada jugador, e implementa los métodos necesarios para colocar las naves e iterar sobre estos.
- **BattleVoid**: Clase principal del juego. Contiene el *main* y se encarga de inicializar todos los componentes del juego, tanto lógicos como de interfaz.

UI: Contiene cuatro clases de objetos de interfaz, todos estos extienden de ***UIObject***.

- **PlayerHighlights** : *JLabel* que muestra de quién es el turno.
- **PlayerWins**: *JLabel* que muestra una imagen con el ganador de la partida.
- **ShipDirection**: *JLabel* que funciona como botón para cambiar la dirección de una nave (horizontal o vertical)
- **UIObject**: Clase principal que contiene todos los atributos necesarios para dibujar un objeto en la interfaz, tales como posición X Y, tamaño, imagen, etc.

- **Utility**: Contiene datos estáticos, principalmente las ubicaciones (paths) de los archivos de imagenes utilizados en la interfaz. Estos se encuentran en la carpeta de ***assets***.

Utilización exhaustiva de herencia (extends)



En base a esto, la principal decisión de diseño que se puede apreciar, es la creación de la clase **UIObject**, la cual es reutilizada (mediante *extends*) por todos los componentes del juego que necesitan una representación visual. Podemos notar como el constructor recibe los valores de posición X Y, tamaño y ubicación de la imagen, y si es inicialmente visible o no.

El método ***changeAssetPosition***, por ejemplo, permite mover las naves a través de las casillas mientras el jugador decide dónde colocarlas. Además, los métodos de **mouseAction**() son los que permiten la interacción con el usuario. Cada objeto que extiende UIObject, sobreescribe estos métodos e implementa un comportamiento específico para cada uno.

4.2 Algoritmos usados

Si bien el proyecto no requirió de la implementación de algoritmos complejos, se pueden mencionar algunas funcionalidades que requirieron de la programación de chequeos lógicos para el correcto comportamiento del programa.

Colocación de naves

Al inicio del juego, cada jugador debe colocar sus naves en el tablero, estas pueden estar en posición horizontal o vertical. El principal problema que presenta esto, es el hecho de que sin ningún chequeo, sería posible colocar una nave fuera de los límites del tablero. Es decir, sería posible colocar la nave de 4 casillas, en la columna 8, por ejemplo, lo que evidentemente es incorrecto.

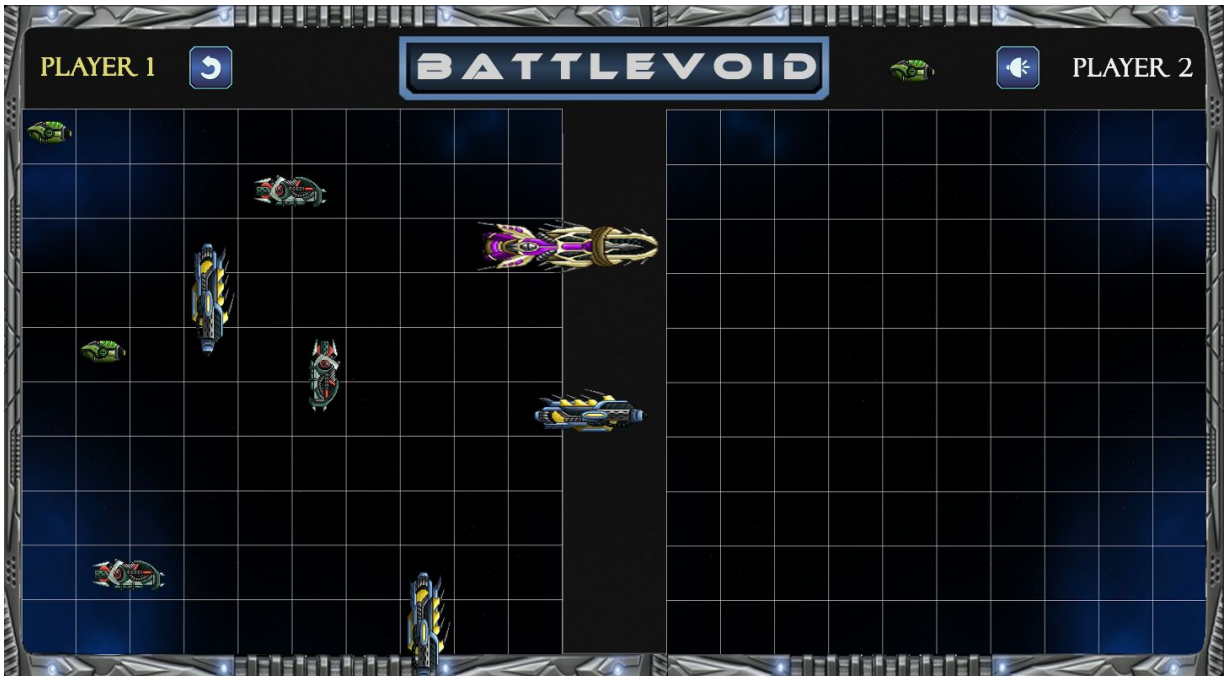


Imagen 1. Colocación **inválida** de naves.

Para evitar esto, antes de mover la nave a una casilla, se realizar un chequeo ***checkShipFits*.** Este recibe la **fila** y **columna** donde se pretende colocar la nave, así como el **tamaño** de esta, el primer chequeo hace la comparación ***10 - compareIndex >= size,*** donde *compareIndex* varía dependiendo de si la posición es horizontal o vertical. Es decir, si estamos intentando colocar la nave de 4, horizontal, en la casilla 8, entonces 10 - 8 >= 4, retornará falso.

```
public boolean checkShipFits(int rowIndex, int columnIndex, int size, boolean horizontal){
    int compareIndex = rowIndex;
    if(horizontal) compareIndex = columnIndex;
    //Check that ship doesn't exceed grid boundaries
    if(10 - compareIndex >= size){
        //Check that there aren't any other ship on the spaces
        for (int i = 0; i < size; i++) {
            //Found a ship on one of the nearby spaces, cant place
            if(horizontal){
                if(!this.activePlayerGrid[columnIndex+i][rowIndex].isEmpty()){
                    return false;
                }
            }
            else{
                if(!this.activePlayerGrid[columnIndex][rowIndex+i].isEmpty()){
                    return false;
                }
            }
        }
        return true;
    }
    else return false;
}
```

El segundo chequeo, consiste en verificar que desde la posición inicial donde se pretende colocar la nave, hasta **n** casillas adyacentes, donde **n** es el tamaño de la nave, no podrá haber ninguna otra nave. Si la nave se coloca horizontal, se verifica en las columnas, mientras que si es vertical se verifica en las filas. Para esto, se utiliza un método **isEmpty()** de la clase **Square,** es decir, cada casilla sabe si contiene una nave.

A parte de esto, las funcionalidades del programa son considerablemente sencillas, y no fue necesaria la implementación de algoritmos complejos.

4.3 Bibliotecas usadas

- La principal biblioteca utilizada es **Swing**, la cual permite la creación de JFrames, JLabels, etc para la construcción de la interfaz gráfica.
- Se utiliza **java.awt.event** para la interacción con el usuario mediante detección de entradas del mouse (clicks).
- También se hace uso extensivo de **java.util** para el manejo de **ArrayLists**.
- Finalmente **javax.sound** permite la reproducción de efectos de sonidos en el juego.

5. ANÁLISIS DE RESULTADOS

5.1 Objetivos alcanzados

Se logró completar la implementación del juego con el alcance y requerimientos propuestos. Es posible jugar una partida de principio a fin, sin errores de ejecución (*bugs*) detectados.

5.1 Objetivos no alcanzados

Se pretendía implementar un botón de “Jugar de Nuevo” al finalizar una partida, de manera que todos objetos y estado del juego volviera a su estado inicial, sin embargo no fue posible completarlo por falta de tiempo. Por lo tanto, es necesario ejecutar el programa nuevamente para iniciar una nueva partida.

6. MANUAL DE USUARIO

6.1 Instrucciones de Uso

Fase 1: Colocación de naves

Cada jugador deberá colocar sus naves en su respectivo tablero, iniciando el jugador 1.

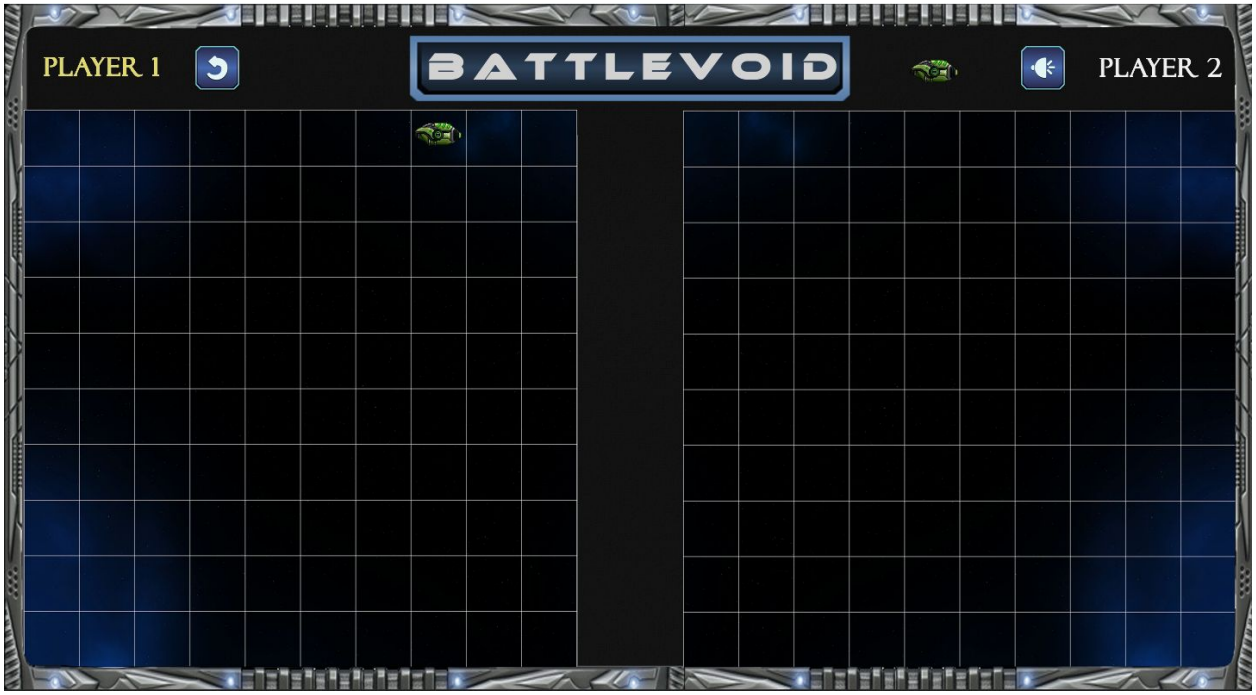


Imagen 2. Pantalla inicial del juego

Una vez que el jugador 1 haya colocado sus naves, le seguirá el jugador 2, haciendo lo mismo. Sin embargo, cabe destacar que una vez que el jugador 1 coloque todas sus naves, estas se desaparecerán del tablero, de manera que el jugador 2 no pueda verlas y hacer trampa.

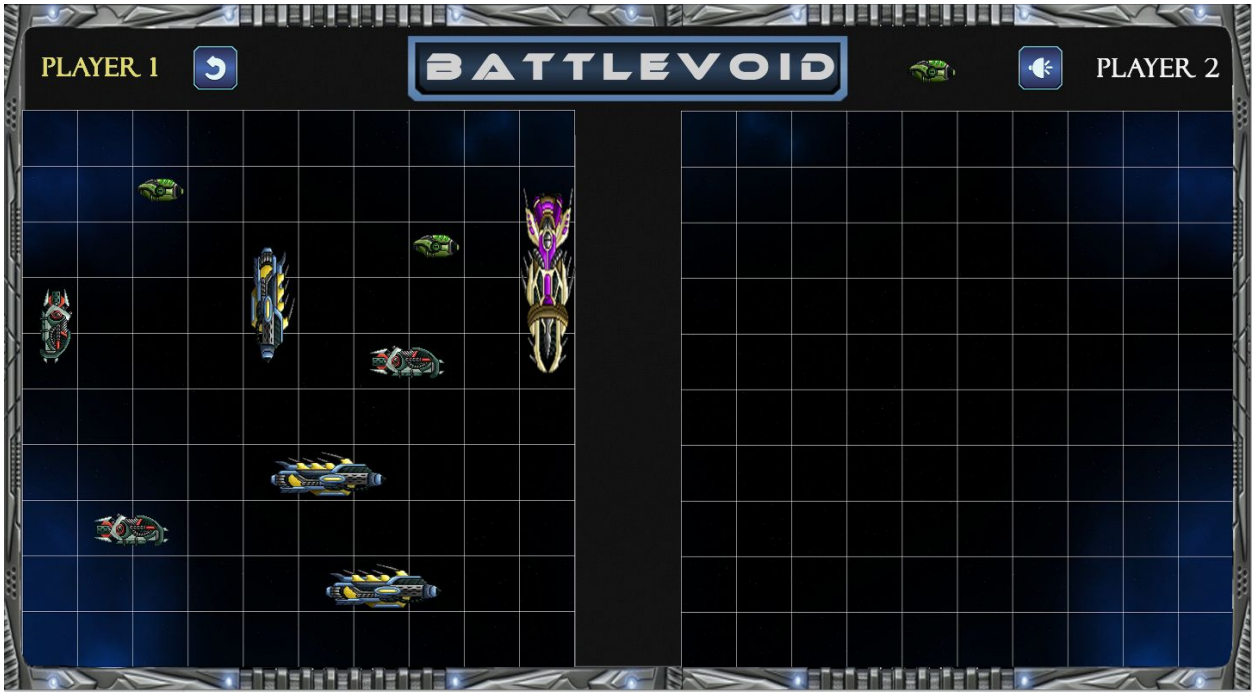


Imagen 3. Jugador 1 colocando sus naves.

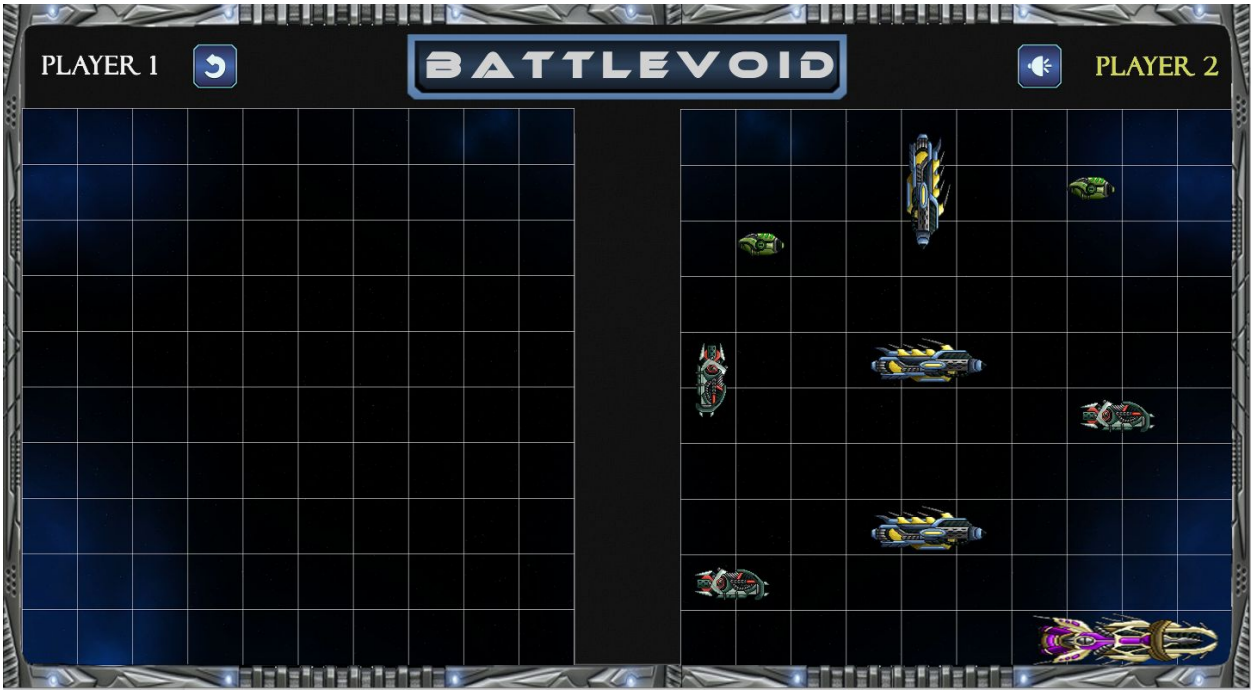


Imagen 4. Jugador 2 colocando sus naves. Las del jugador 1 están escondidas.

Fase 2: Inicio de la partida

Una vez que ambos jugadores hayan colocado sus naves, estos tomaran turnos “disparando” en el tablero del enemigo, dando *click* en la casilla que desea atacar, cuando se acierta a una nave, la casilla cambiará a verde, si es un fallo, se pondrá en rojo. Si una nave es destruida, esta volverá a aparecer.

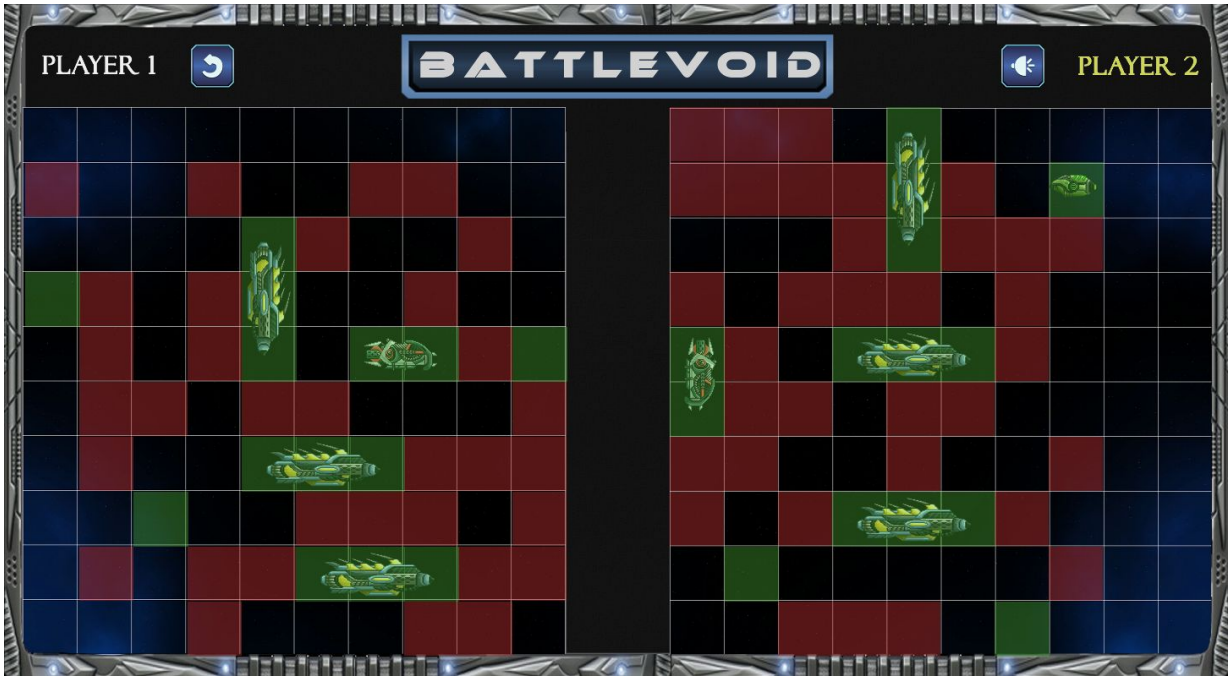
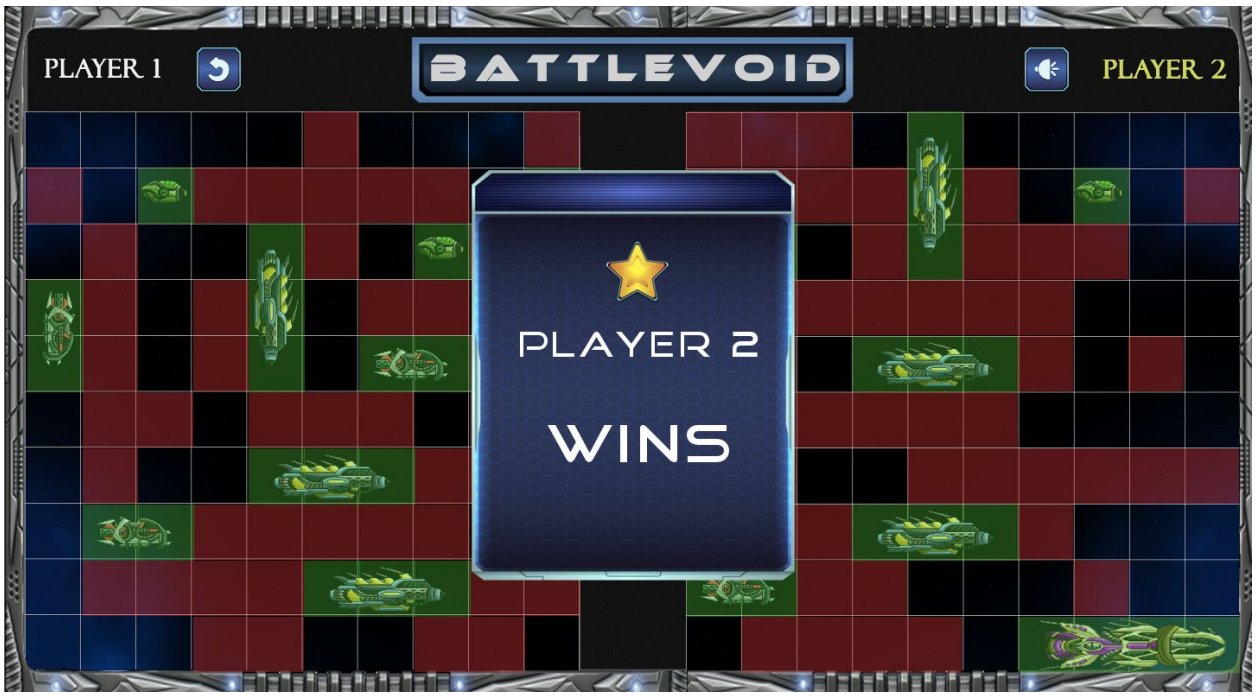


Imagen 5. Ambos jugadores toman turnos disparando al enemigo.

Fase 3: Fin de la partida

Finalmente, cuando un jugador haya destruido todas las naves del enemigo, se mostrará un mensaje felicitando al ganador.



6.2 Instrucciones de Ejecución

Se puede descargar el código del programa en el repositorio de GitHub. Y utilizar el IDE IntelliJ IDEA Community o Eclipse para ejecutar el programa.

7 CONCLUSIONES PERSONALES

7.1 Alejandro

La capacidad de abstraer objetos de la vida real en representaciones lógicas y programables es una herramienta sumamente útil para cualquier desarrollador. Por lo que considero que la realización de este proyecto ha sido de suma importancia para ya sea introducir o refrescar estos conocimientos en los estudiantes. Además fue posible aplicar conocimientos aprendidos en clase, como la utilización de *ArrayLists*, clases abstractas, *extends* y *override*.

7.2 Jose Antonio

8 AUTOEVALUACIÓN

Nombre	Evaluación
Jose Antonio	100
Alejandro	