18 de diciembre, 2023

Informe Arquitectura de software

Integrantes: Renato Atencio, Miguel Ormeño, Orlando Contreras

Índice

Introducción

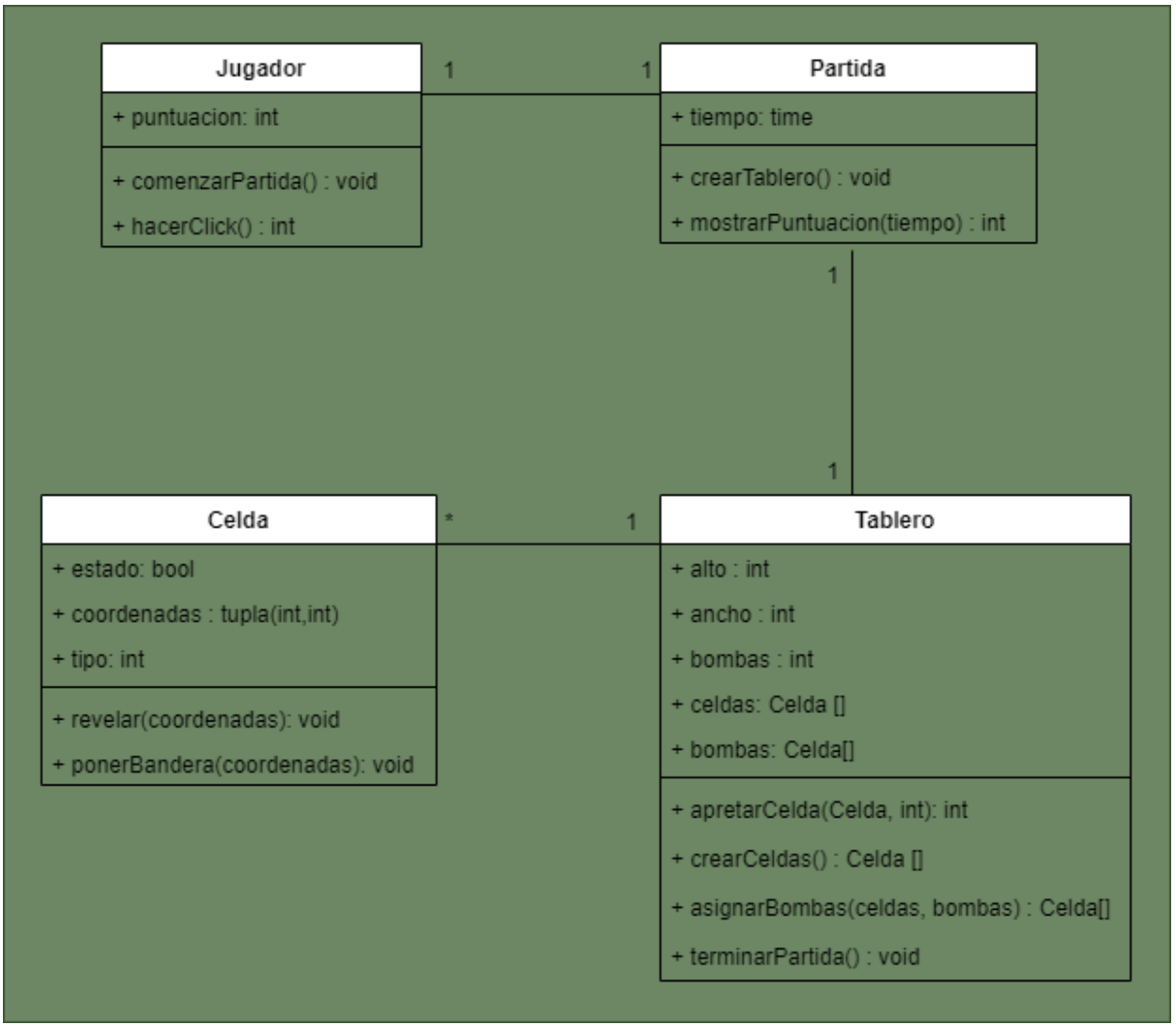
Nuestro proyecto es un software diseñado para poder jugar al Buscaminas. Basados en los puntos indicados que se dieron en la rúbrica, el informe de este proyecto abarca las metodologías 4+1 (Diagrama de caso de uso, diagrama de clases, diagrama de componentes y diagrama de secuencia) , así como también las buenas prácticas y el código funcional del juego en Python. Los diagramas mostrados son prototipos de la versión beta del software, ya que inicialmente se iban a agregar más clases, pero basándonos en el patrón singleton usamos una sola clase buscaminas la cual se instancia una vez para dar inicio a la partida. Aun así, los diagramas guardan coherencia con el empleado en el software por lo que se respeta la arquitectura en el uso de métodos y atributos que se encuentran en los siguientes diagramas:

Metodología 4+1

La metodología 4+1 es un modelo basado en el uso de 4 ‘vistas’ concurrentes y un escenario. Estas ‘vistas’ buscan describir el sistema desde el punto de vista de diferentes interesados. En este proyecto el objetivo es implementar una arquitectura de software para el juego buscaminas siguiendo esta metodología de diseño. En los siguientes puntos se explicarán las ‘vistas’ de la metodología y se muestran los diagramas relacionados al sistema.

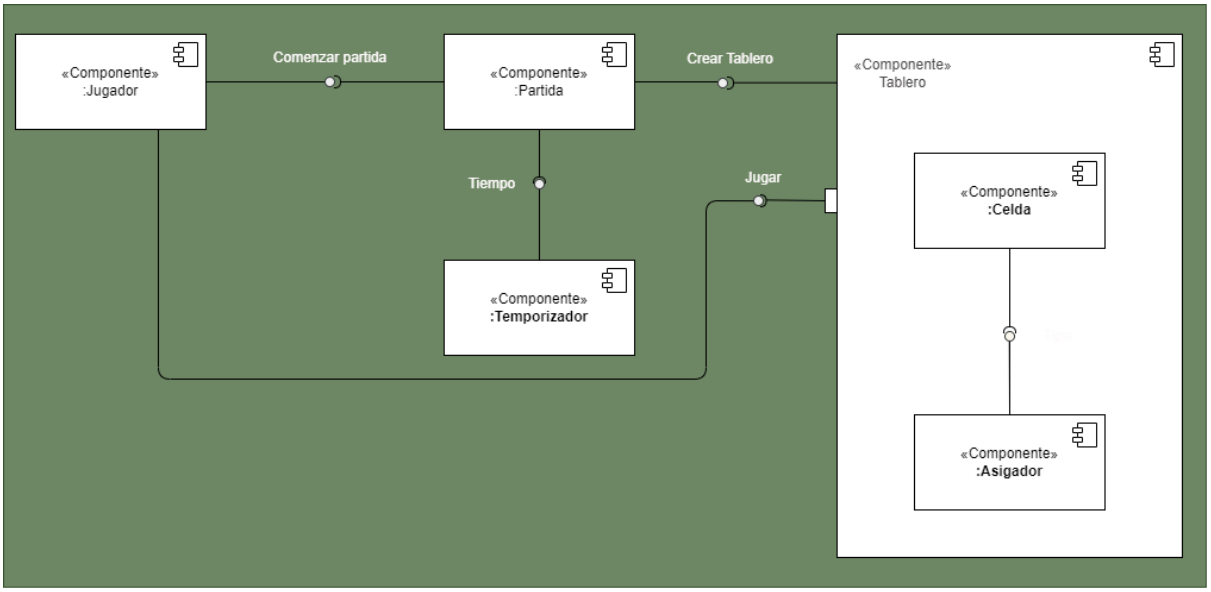
- ➔ **Vista lógica**: Enfocada en describir la estructura y funcionalidad del sistema. Usando diagramas como **diagrama de clase** y **diagrama de comunicación**.

  - ◆ Diagrama de clase:



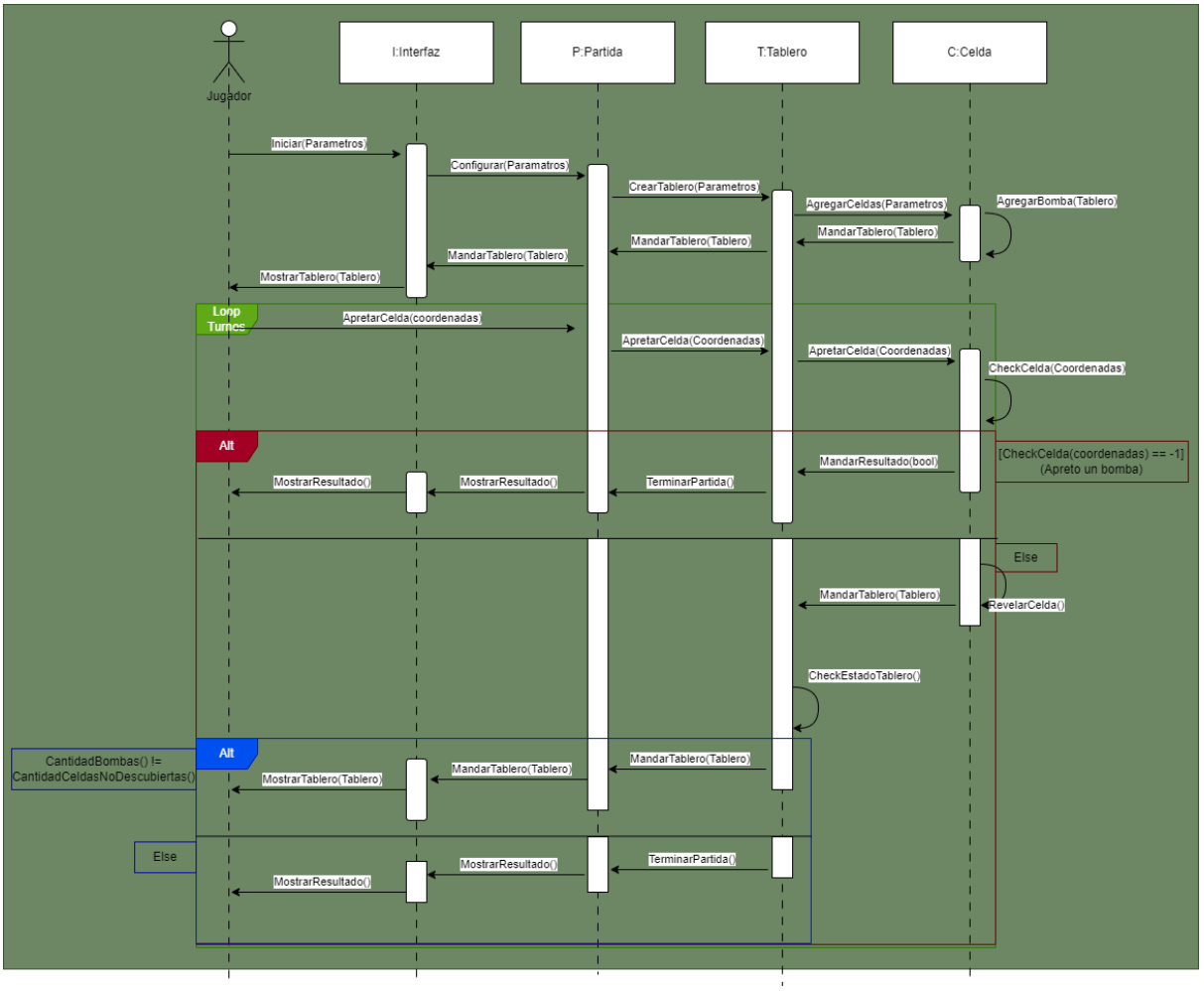
➔ **Vista de desarrollo:** Ilustra el sistema desde la perspectiva del programador y está enfocado en la administración de los artefactos de software. Se hace uso de **diagrama de componentes** y **diagrama de paquetes**.

◆ Diagrama de componentes:



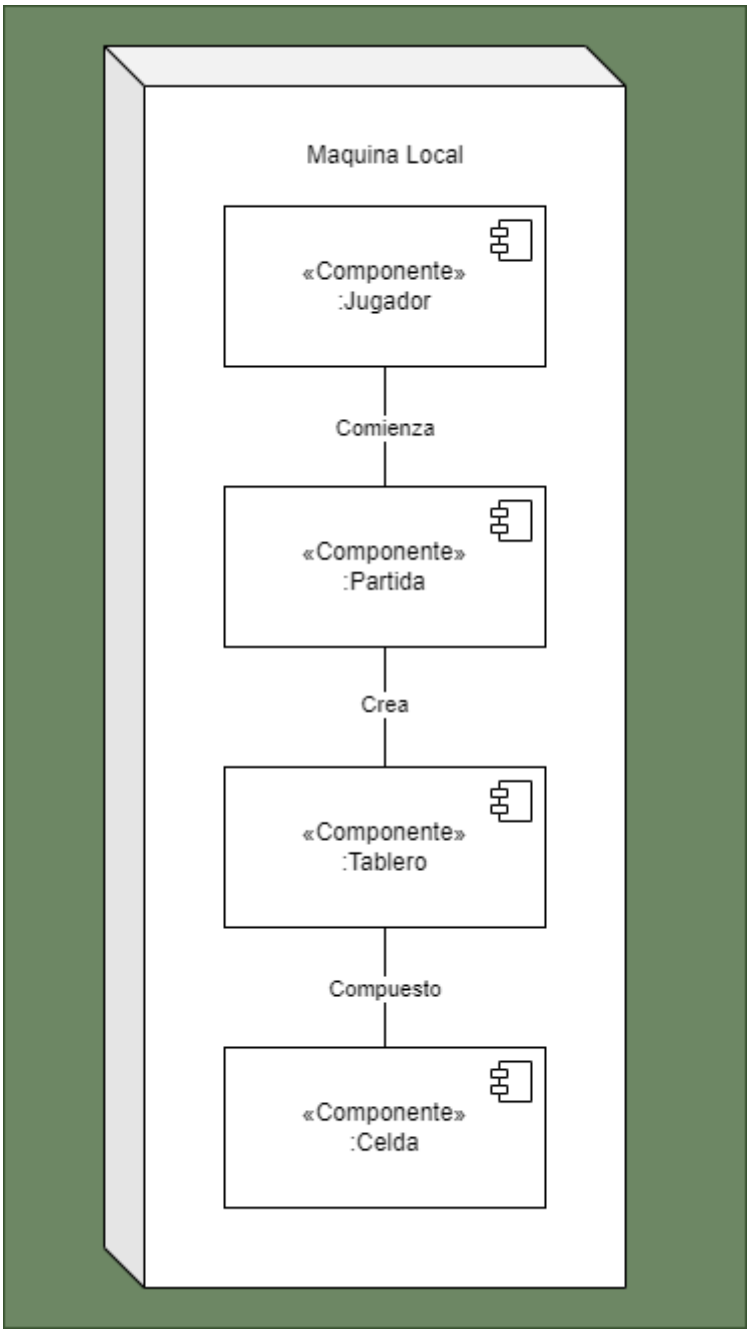
➔ **Vista de proceso**: Trata los aspectos dinámicos del sistema, explica los procesos de sistema y cómo se comunican a través de los **diagramas de actividad** y **diagrama de secuencia**.

◆ Diagrama de secuencia:

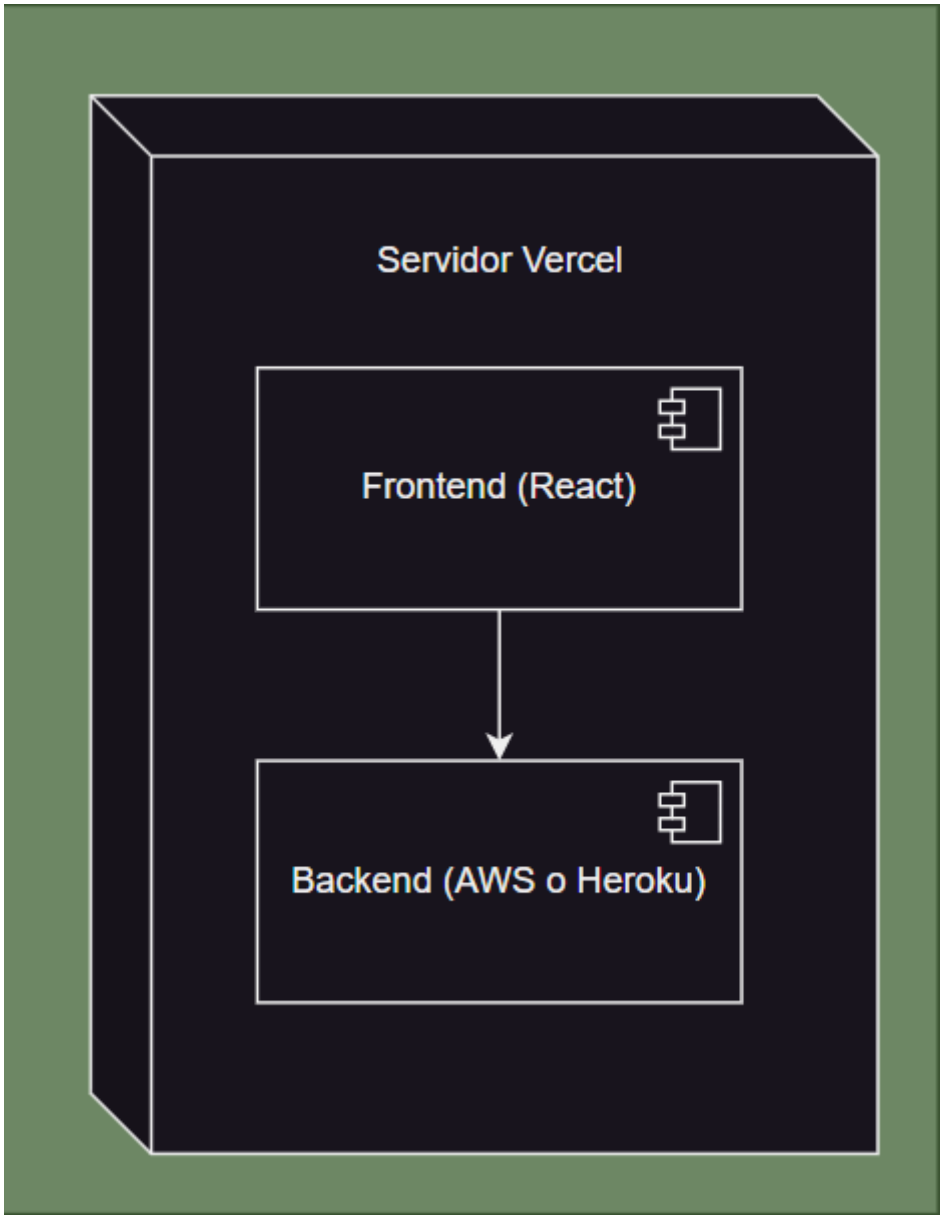


➔ **Vista física**: Describe el sistema desde el punto de vista de un ingeniero de sistemas usando un **diagrama de despliegue**.

- ◆ Diagrama de despliegue (Correr código localmente):

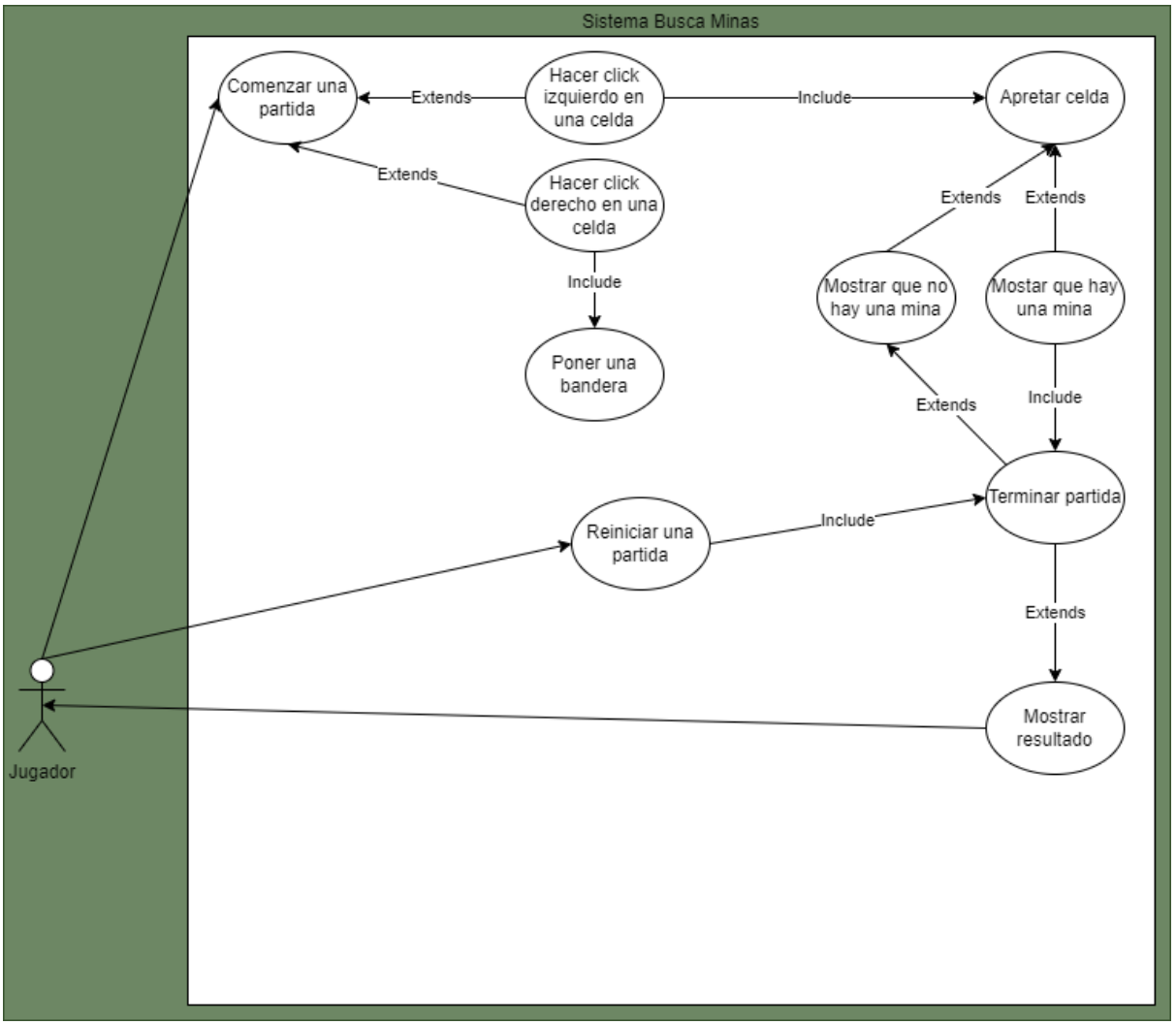


- Diagrama de despliegue (Página):



➔ **Escenarios**: La descripción de la arquitectura se ilustra utilizando un conjunto de casos de uso, es decir a través de un **diagrama de casos de uso**

◆ Diagrama de casos de uso:



Patrón de diseño

Explicación
El patrón de diseño que usamos en este código es el patrón de diseño Singleton. Este patrón asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a esa instancia.
El patrón Singleton de tal forma de garantizar que solo haya una instancia de la clase buscaminas en toda la aplicación. Esto ayuda en momentos donde se requiere exactamente una instancia de una clase para gestionar recursos compartidos o para garantizar la coherencia en el comportamiento del programa.

Como se implementó:

Clase singleton:

- Definimos una variable de clase **_instance_** que almacenará la instancia única de la clase.
- Sobreescribimos el método **_new_** para tener control sobre la creación de nuevas instancias. Esto es, el método verifica si ya existe una instancia de la clase y en caso de ser así, devuelve esa instancia en lugar de crear una nueva.

Uso del singleton:

- Se instancia la clase buscaminas una vez(**buscaminas = Buscaminas()**), y esta instancia es la que se utiliza en todo el programa.
- Cada vez que se llama a **Buscaminas()**, se devuelve la misma instancia existente en lugar de crear una nueva, gracias a la lógica implementada en el método **_new_.**
- Variable **_instance**: La usamos para almacenar la única instancia de la clase. Es una variable de clase que se inicia como None.
- **Método __new__**: Este método controla la creación de nuevas instancias de la clase. Verifica si la instancia **_instance** es None. Si es así, crea una nueva instancia de la clase llamando al constructor de la clase **super().__new__(cls**). Si **_instance** ya tiene una instancia, simplemente devuelve esa instancia existente.
- Método **__init__**: Este método inicializa los atributos y configuraciones necesarios para el juego. Se ejecuta solo si se crea una nueva instancia.

Buenas Prácticas

Se usaron prácticas como documentación (este informe), planificación de sesiones en las cuales mediante pequeñas reuniones comentamos la arquitectura e implementación del software. También se comentó el código para un mejor entendimiento por parte de quienes tengan la lectura de este para guiarse mejor , ya sea como usuario así como de desarrollador en caso de mejorar el código a futuro, dándole mantenibilidad (otra buena práctica y calidad de software). Se hicieron pruebas de tal modo identificar errores o pequeños bugs que si bien no detonaron en una caída del software , si arruinaban la calidad del juego en cuanto a jugabilidad, situaciones que fueron mejoradas o solucionadas gracias a la revisión. El código como bien se mencionó es diferente a la arquitectura propuesta en algunos aspectos se respetaron la mayoría de atributos y métodos de la clase buscaminas para una funcionalidad que abarque el patrón singleton que se implementó. Se adjunta una imagen a modo de ejemplo que verifica el buen uso de comentarios para la facilidad de lectura como se mencionó anteriormente.

```
# Crea el tablero
def generate_board(self):
    self.board = [[0 for _ in range(self.size)] for _ in range(self.size)]

# Colocar bombas en las celdas
def place_bombs(self):
    bombs_placed = 0
    while bombs_placed < self.bomb_count:
        x = random.randint(0, self.size - 1)
        y = random.randint(0, self.size - 1)
        if self.board[x][y] != -1:
            self.board[x][y] = -1
            bombs_placed += 1

# Indica el numero de bombas adyacentes a un celda
def calculate_numbers(self):
    for i in range(self.size):
        for j in range(self.size):
            if self.board[i][j] != -1:
                self.board[i][j] = self.count_adjacent_bombs(i, j)

# Cuenta las minas adyacentes a las celdas
def count_adjacent_bombs(self, x, y):
    count = 0
    for i in range(-1, 2):
        for j in range(-1, 2):
            if (
                0 <= x + i < self.size
                and 0 <= y + j < self.size
                and self.board[x + i][y + j] == -1
            ):
                count += 1
    return count
```

Protocolo de despliegue

El protocolo de despliegue del cual se podría hacer uso sería la utilización del servicio Vercel para desplegar el frontend de la página y usar otro servicio como Heroku o AWS para manejar la lógica del backend de la página.

En el caso de la implementación actual decidimos no hacer el despliegue ya que no tenemos experiencia con servicios para manejar el backend, aunque dejamos el backend y frontend en el repositorio así que su despliegue debería de ser posible si se hace uso de estos servicios.

Programa

Nuestro buscaminas es creado e iniciado como Singleton. Al construirse, parte con el tablero en tamaño fijo seguido por las bombas, que son colocadas aleatoriamente dentro del tablero con una cantidad a razón fija, asegurándose de que no se sobrepongan, después calcula la cantidad de bombas adyacentes a cada casilla vacía y las integra al tablero, y finalmente crea la interfaz del juego: le da roles a cada tipo de click (derecho para poner bandera e izquierdo para mostrar que hay detrás de la casilla), las revelaciones de casillas o el tablero, y el fin del juego ya sea victoria o derrota con las opciones de reintentar y salir.

Bibliografía

- Modelo de vistas 4+1
- Patrón de diseño Singleton
- Documento de buenas prácticas en diseño e implementación de software: https://www.velneo.com/blog/15-buenas-practicas-proyectos-desarrollo-software
- Vercel
- Heroku
- AWS
- Repositorio Git: ProyectoArquitecturaDeSoftware