



TALLER DE PROGRAMACIÓN I  
(75.42) CURSO VEIGA

# Informe técnico - Duck Game

2do Cuatrimestre 2024

Ascencio Felipe Santino	Zielonka Axel
110675	110310

## 1. Introducción

Este informe técnico describe el funcionamiento del programa a nivel de código. Su objetivo es proporcionar a futuros desarrolladores una comprensión profunda del sistema para que puedan mejorar el proyecto o adaptarlo según sus necesidades. Asimismo, está pensado para aquellos interesados en conocer el funcionamiento interno del juego.

## 2. Organización de los Directorios

A continuación se presenta la estructura de directorios y una breve descripción de cada uno.

### 2.1. Directorio `documentation`

Este directorio contiene todos los manuales e informes necesarios para la documentación y uso del juego.

### 2.2. Directorio `SDL`

Contiene las dependencias para las bibliotecas *SDL2*, necesarias para el manejo gráfico y de sonido en el juego.

### 2.3. Directorio `src`

En este directorio se almacena todo el código fuente del juego, dividido en varias subcarpetas, descritas a continuación:

#### 2.3.1. Directorio `client`

Incluye el código relacionado con el cliente, encargado de la interfaz gráfica y la comunicación con el servidor.

#### 2.3.2. Directorio `server`

Contiene el código del servidor, que gestiona la lógica del juego y la comunicación con los clientes.

#### 2.3.3. Directorio `common`

Guarda el código compartido entre los directorios `client` y `server`. Aquí se encuentran archivos con métodos y estructuras utilizadas en ambos programas.

#### 2.3.4. Directorio `tests`

Contiene el código de los 'tests' que verifican el correcto funcionamiento del 'Protocolo' y la 'Comunicación' Cliente-Servidor.

#### 2.3.5. Directorio `editor`

Actualmente solo se encuentra un esqueleto de la estructura para un futuro 'editor', la idea sería en próximas actualizaciones añadirlo para agrandar las funcionalidades del juego.

### 2.3.6. Directorio data

Este directorio contiene todos los recursos multimedia del juego, organizados en las siguientes categorías:

- Sprites.
- Música.
- Sonidos.
- Pantallas de victoria y derrota.
- Fuentes de texto.
- Mapas.
- Archivos de inicialización de la matriz de mapas (utilizados por el servidor).

### 2.3.7. Directorio SDL2pp

Este directorio contiene todo lo necesario para utilizar la librería "SDL2pp" para el apartado gráfico y de sonido del juego.

### 2.3.8. Directorios cmake y cmake-build-debug

Contienen la configuración y ejecución de los archivos para la compilación del proyecto mediante CMake.

### 2.3.9. Archivo de configuración config-juego.yaml

Este archivo permite la 'configuración-modificación' de los aspectos personalizables del juego.

## 2.4. Archivos adicionales en el directorio raíz

En el directorio raíz se encuentran los archivos principales de instalación y ejecución del juego, incluyendo el archivo **README**, el cual provee instrucciones básicas para la instalación y uso del juego, facilitando el proceso a usuarios sin experiencia previa.

## 3. Documentación del Código

### 3.1. Servidor

El servidor es responsable de:

- Controlar la lógica del juego.
- Comunicar el estado del juego a todos los jugadores conectados.
- Administrar la sincronización y la coexistencia de los jugadores, eliminándolos si están inactivos o cuando termina la partida.

A continuación se adjuntarán diagramas de clases y de secuencia para las partes más relevantes (UML).

### 3.1.1. Diagrama de clase: Arma

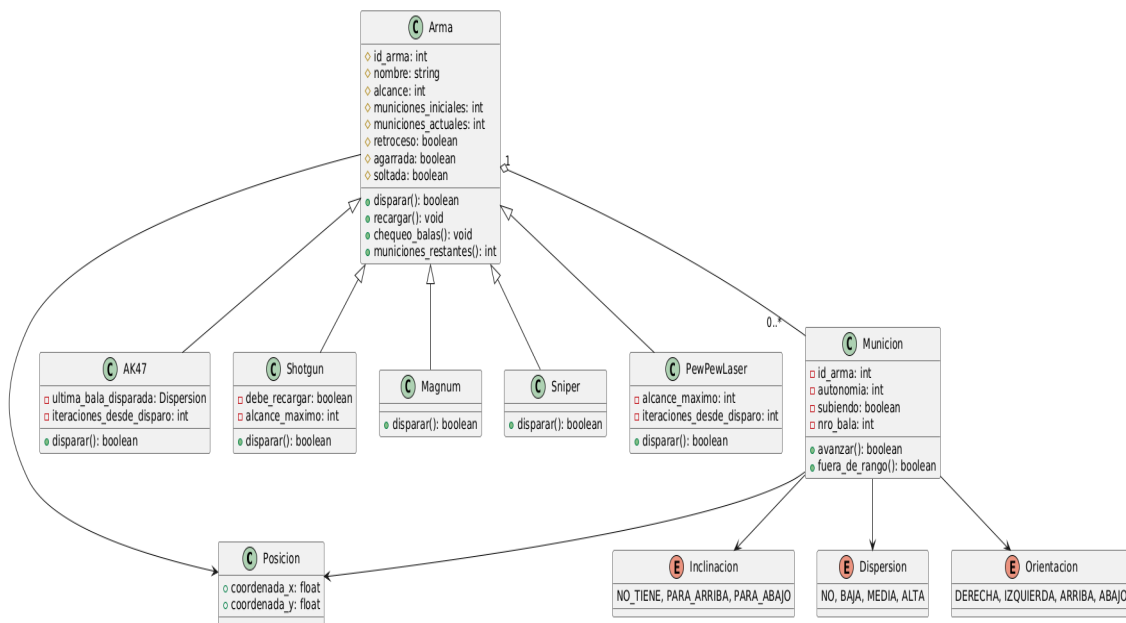


Figura 1: Diagrama de Clases UML : 1.

### 3.1.2. Diagrama de clase: Estado del Juego

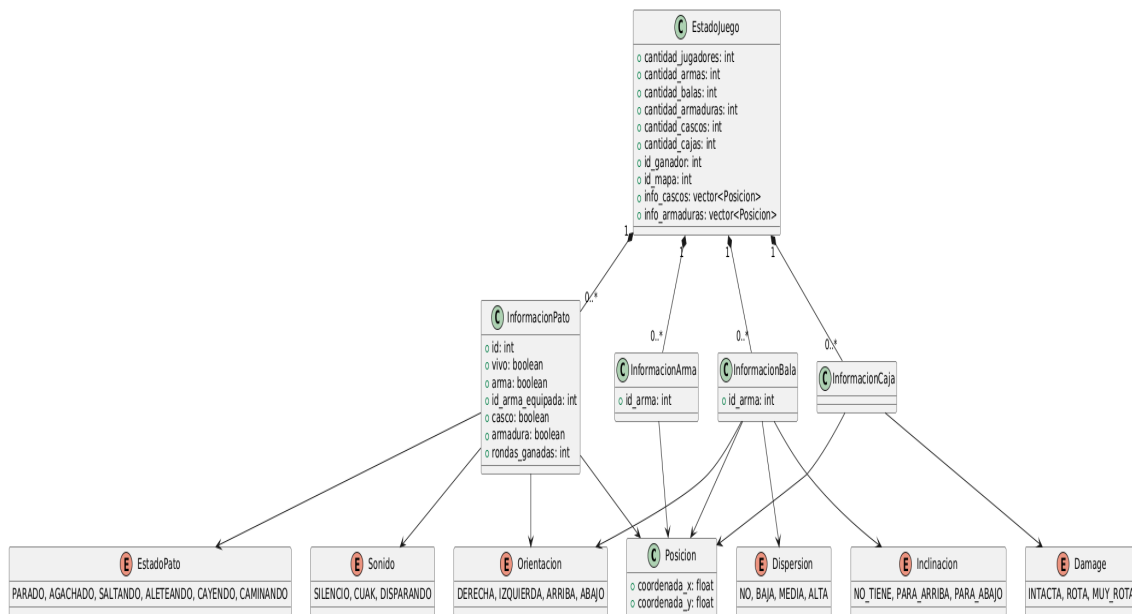


Figura 2: Diagrama de Clases UML : 2.

### 3.1.3. Diagrama de clase: Partida y Gameloop

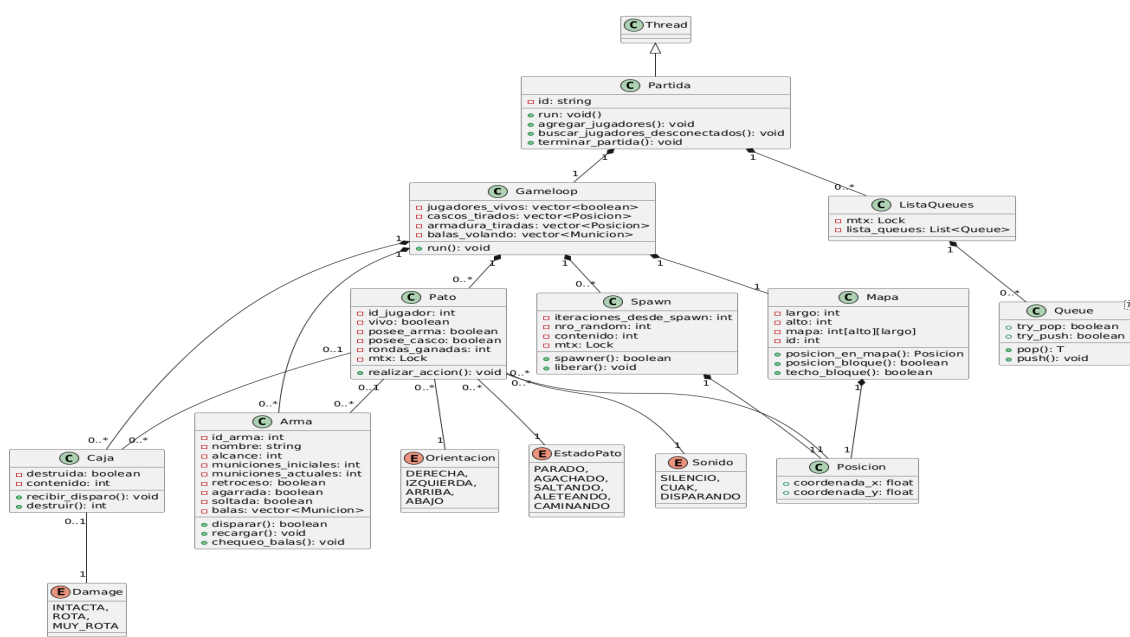


Figura 3: Diagrama de Clases UML : 3.

### 3.1.4. Diagrama de clase: Server

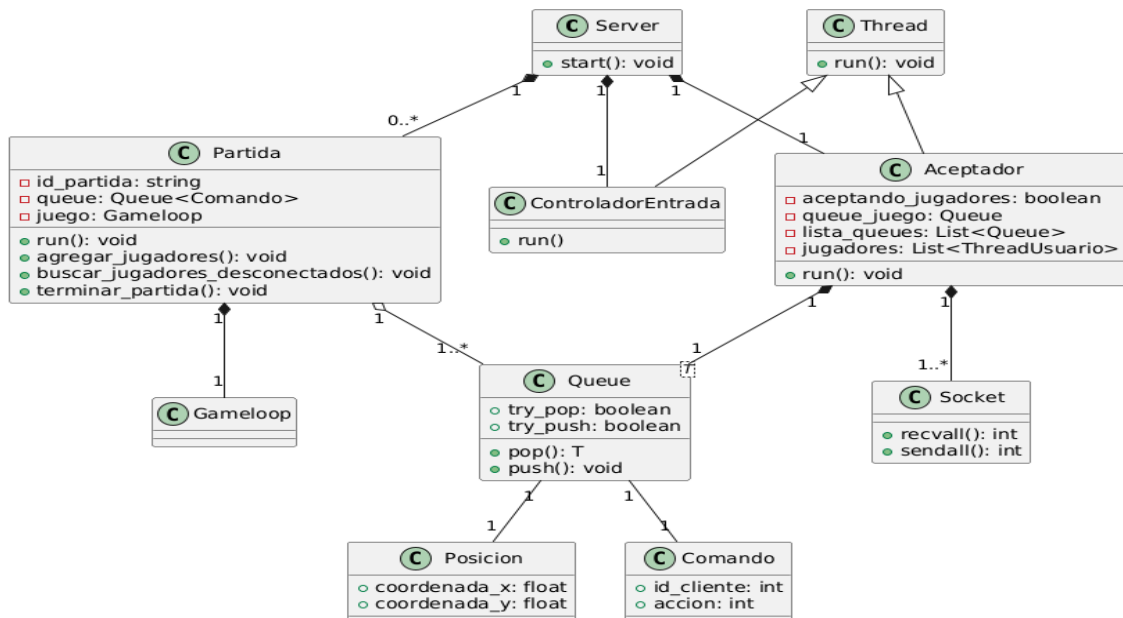


Figura 4: Diagrama de Clases UML : 4.

### 3.1.5. Diagrama de clase: Threads del Cliente

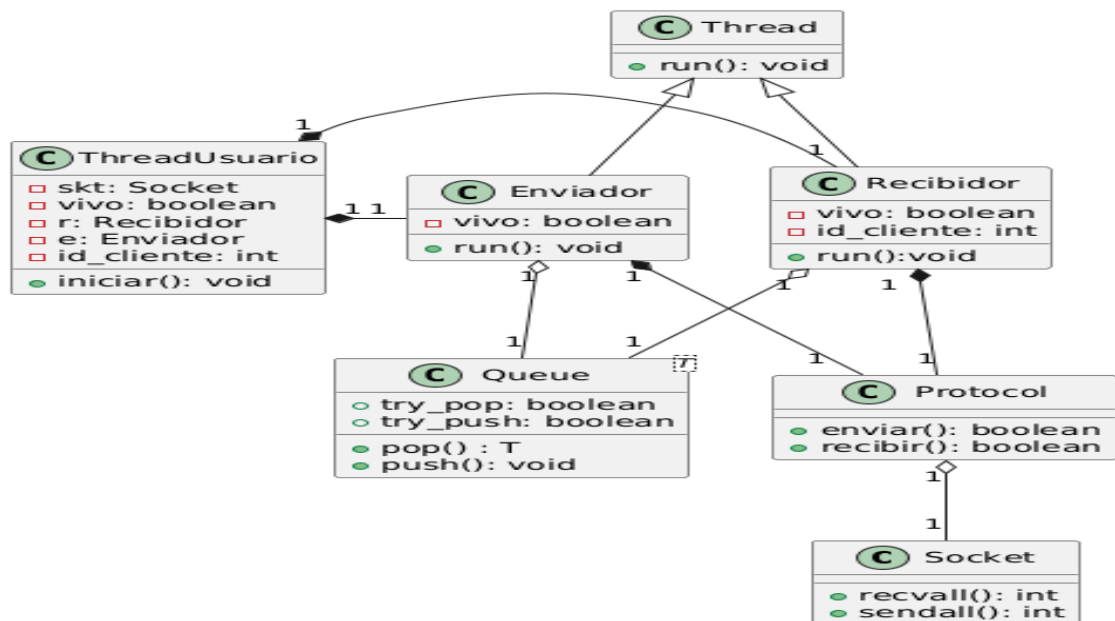


Figura 5: Diagrama de Clases UML : 5.

### 3.1.6. Diagrama de secuencia: Loop del Aceptador y creación de partidas

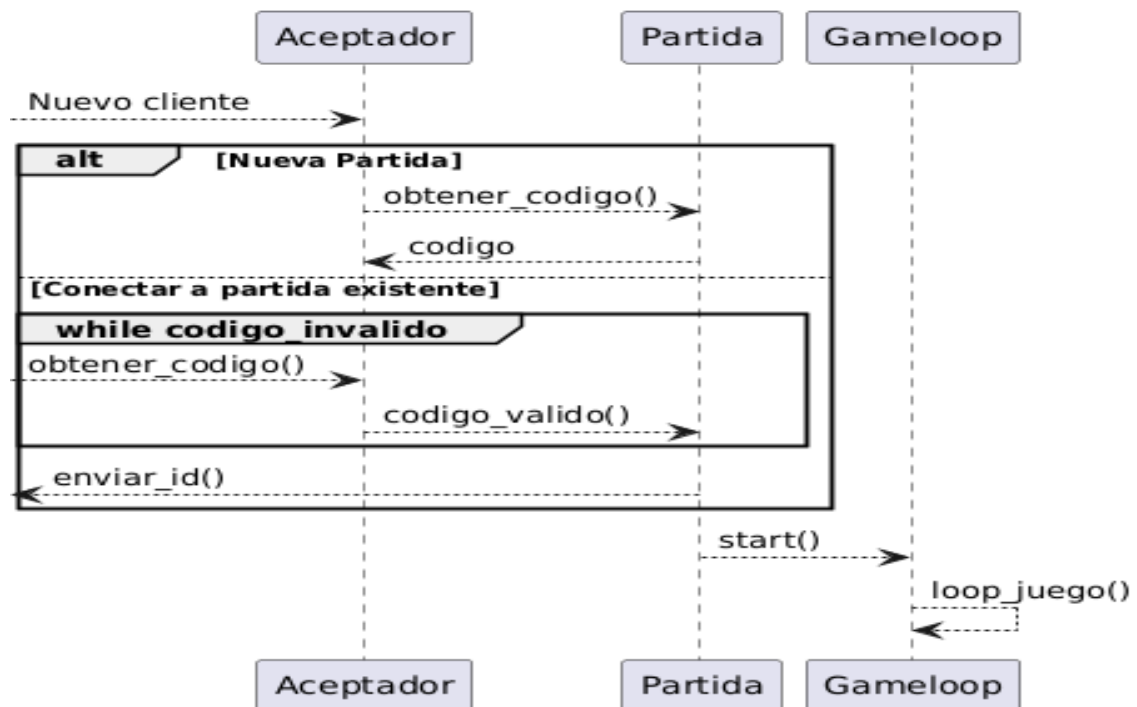


Figura 6: Diagrama de Secuencia: 1.

### 3.1.7. Diagrama de secuencia: Hilo Enviador

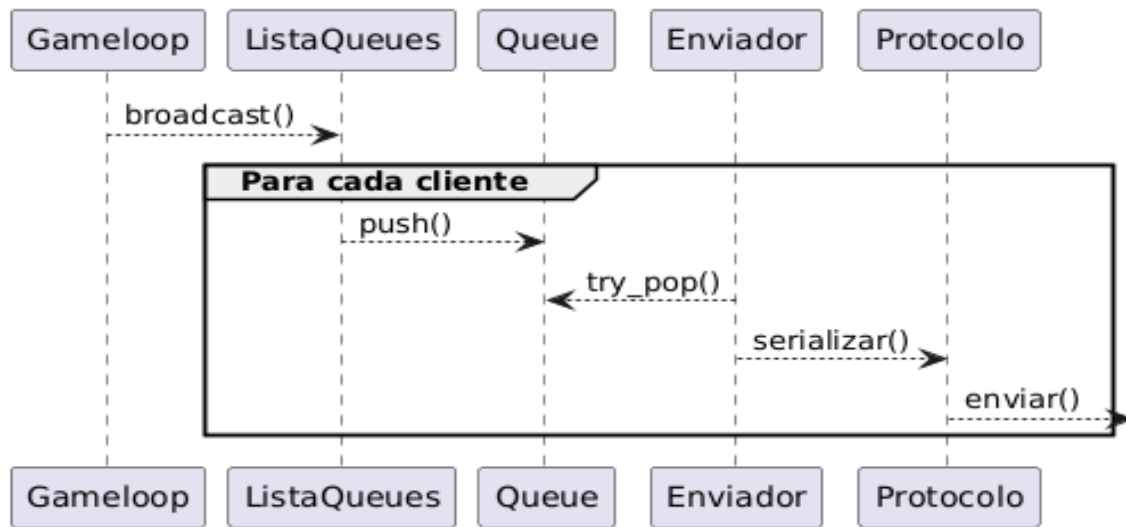


Figura 7: Diagrama de Secuencia: 2.

### 3.1.8. Diagrama de secuencia: Hilo Recibidor

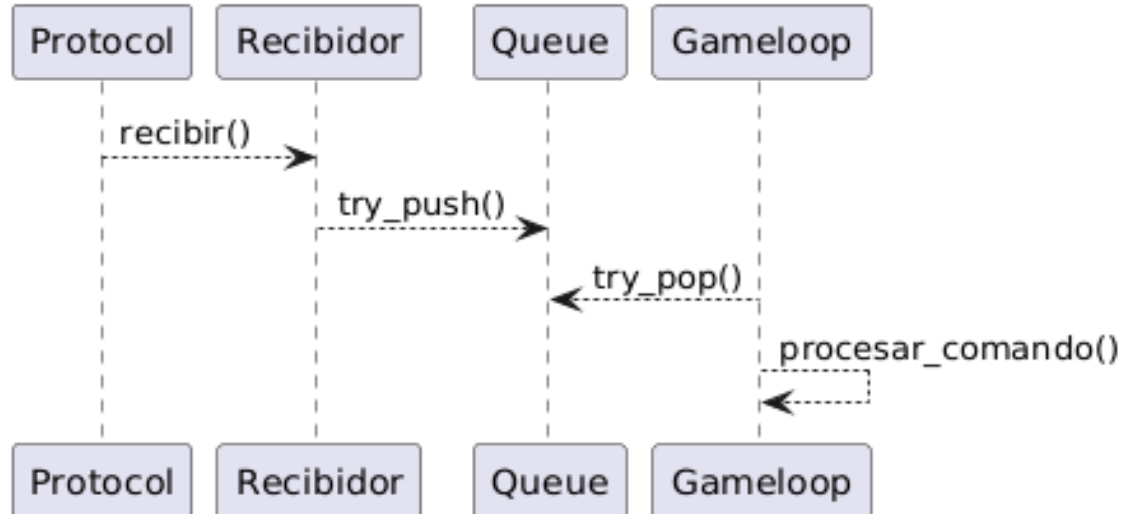


Figura 8: Diagrama de Secuencia: 3.

### 3.1.9. Diagrama de secuencia: Gameloop

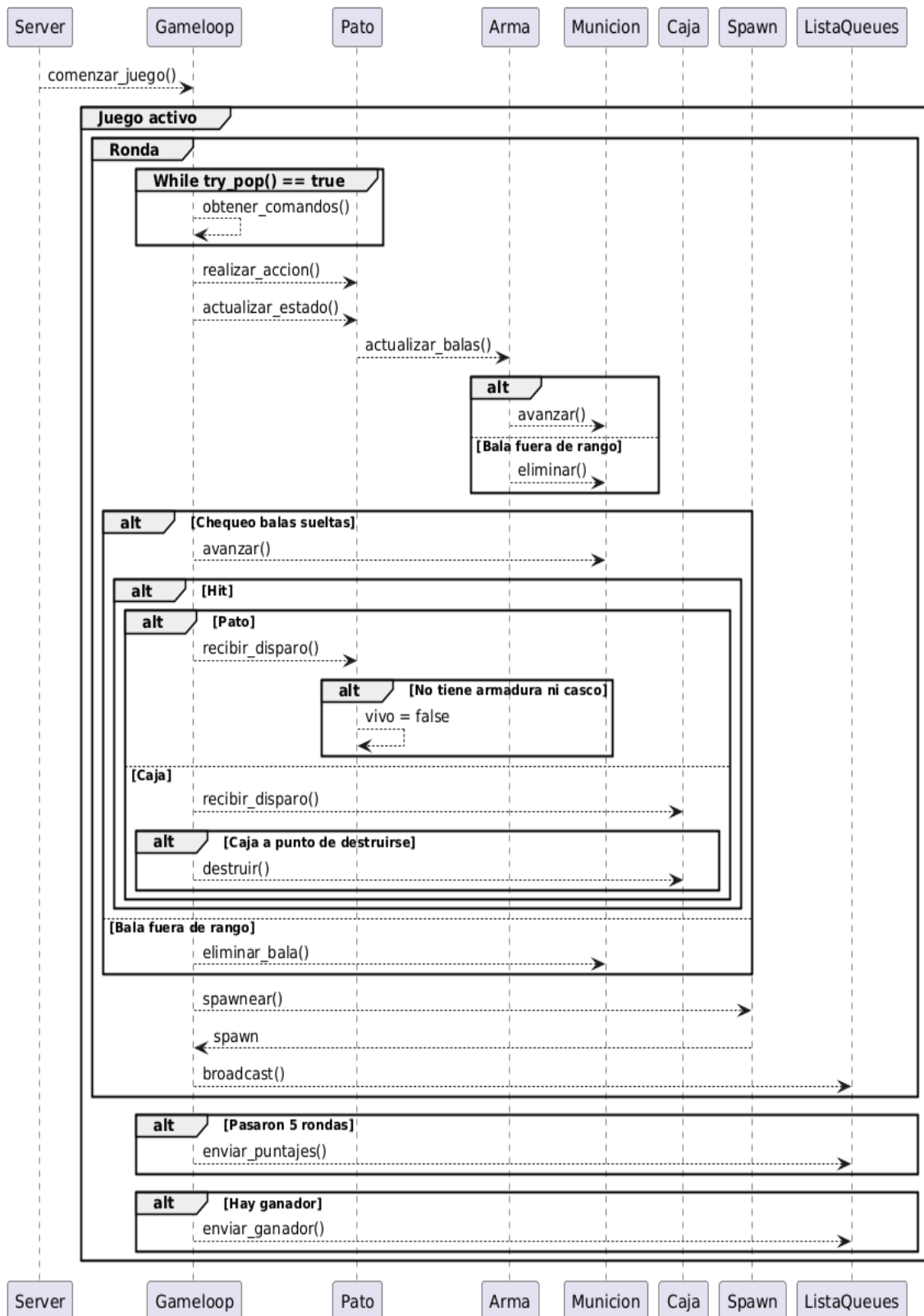


Figura 9: Diagrama de Secuencia: 4.



### 3.1.10. Diagrama de secuencia: Inicio, loop y finalizacion del Servidor

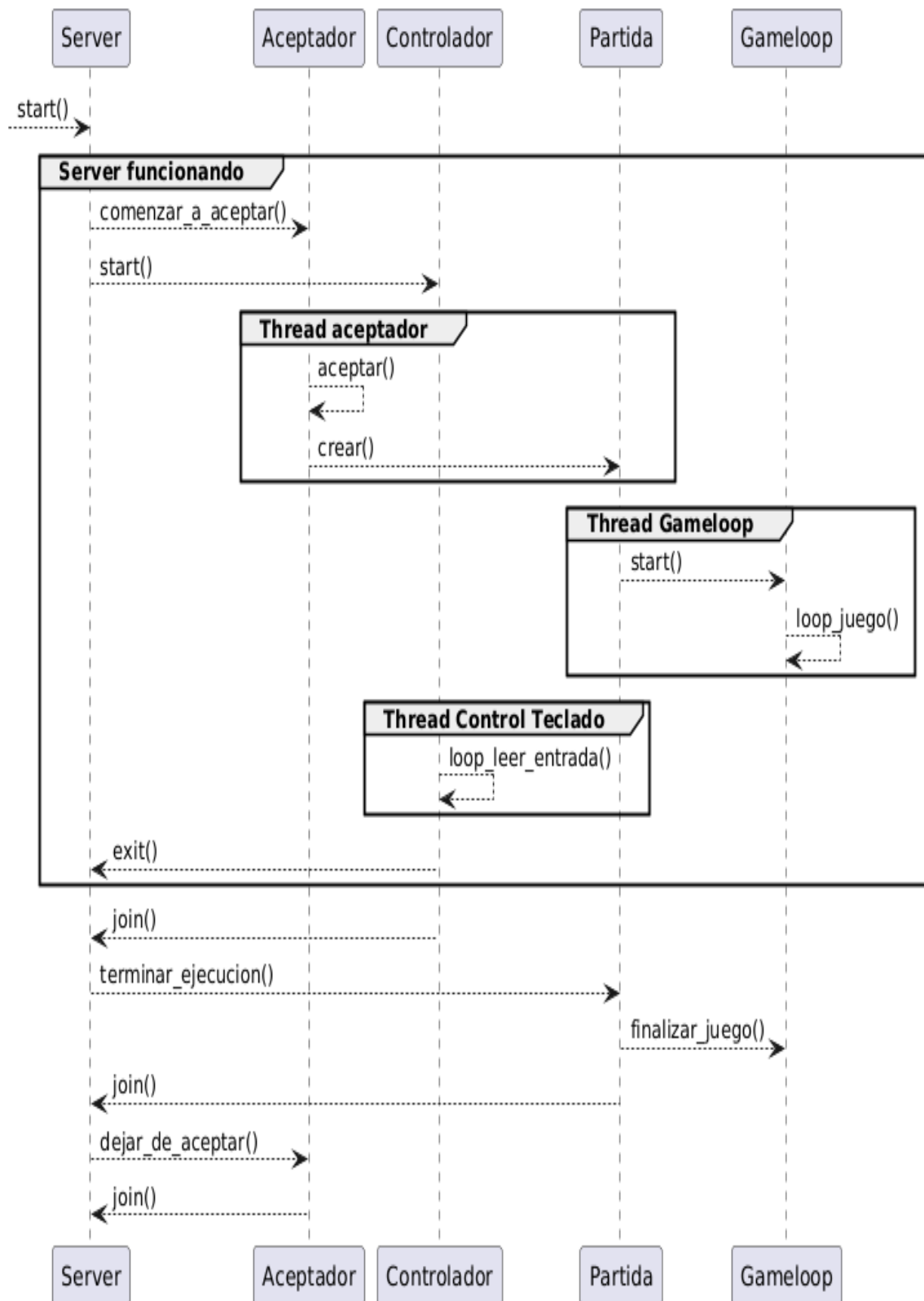


Figura 10: Diagrama de Secuencia: 5.

### 3.2. Cliente

El cliente es el encargado de:

- Representar visualmente el estado del juego para el jugador.
- Enviar las actualizaciones del jugador al servidor.
- Gestionar y utilizar las texturas y sonidos requeridos.

A continuación se anexarán diagramas de clases y de secuencia para las partes más relevantes (UML).

#### 3.2.1. Diagrama de clase: Comunicación con el Server

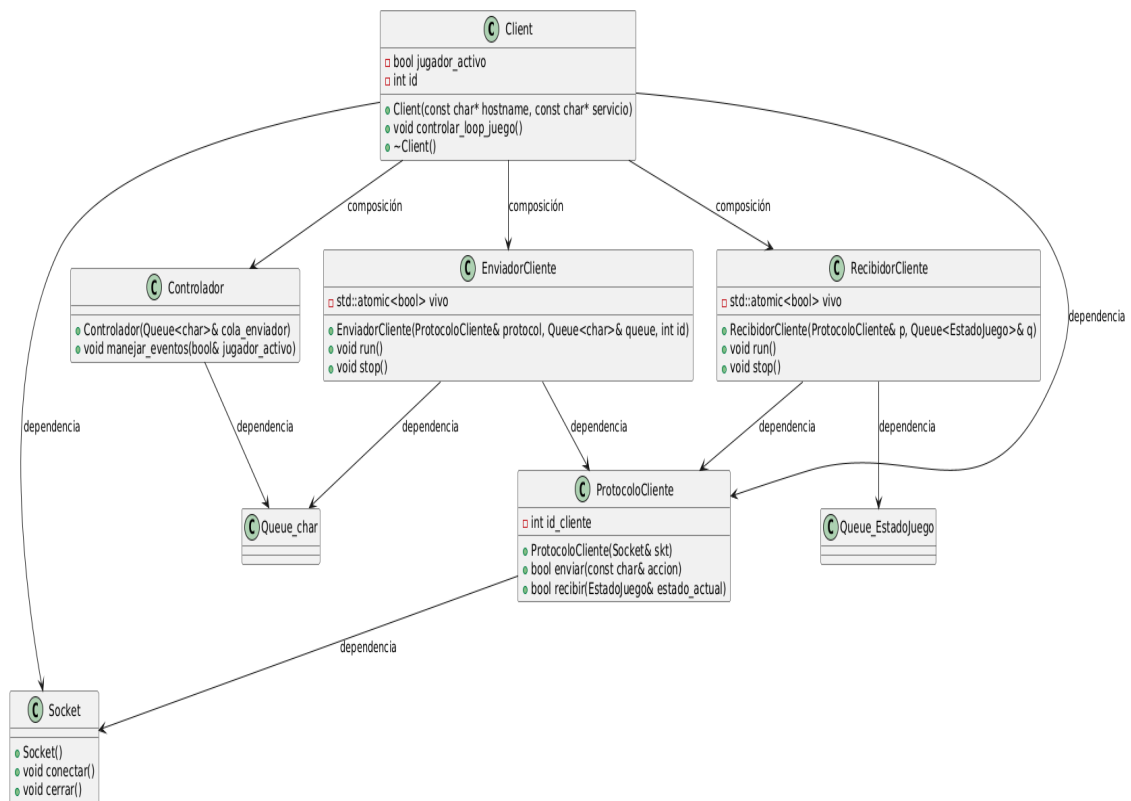


Figura 11: Diagrama de Clases UML : 6.

### 3.2.2. Diagrama de clase: Dibujo de la partida

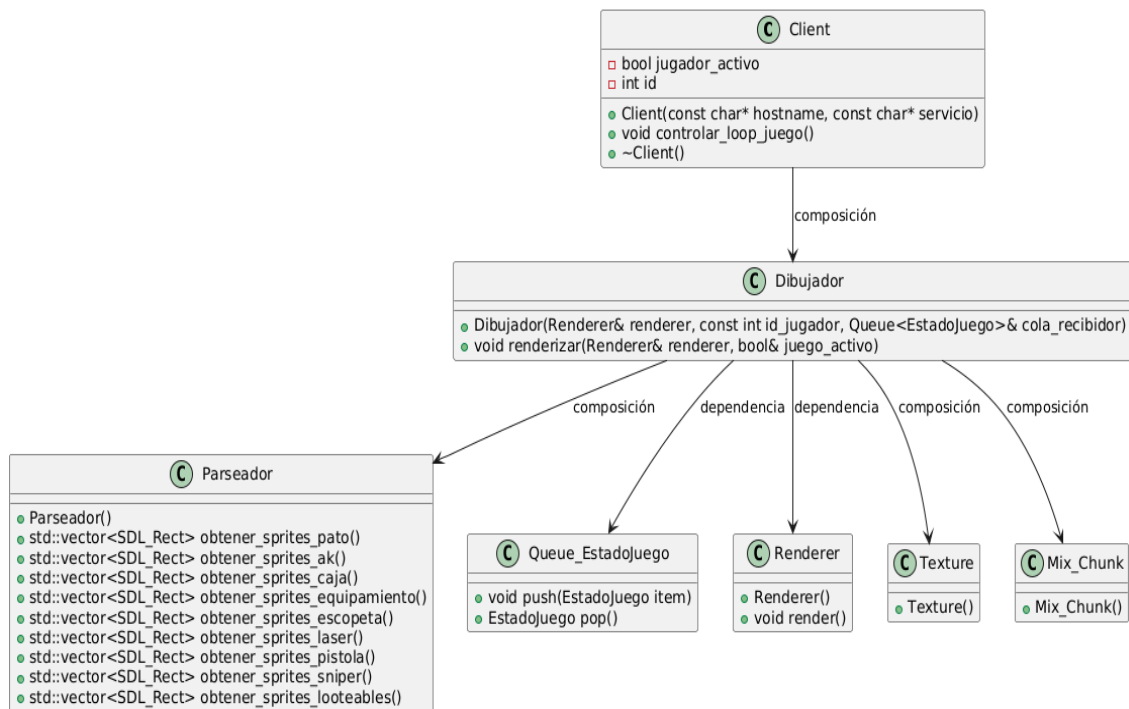


Figura 12: Diagrama de Clases UML : 7.

### 3.2.3. Diagrama de secuencia: Conexión con el Servidor

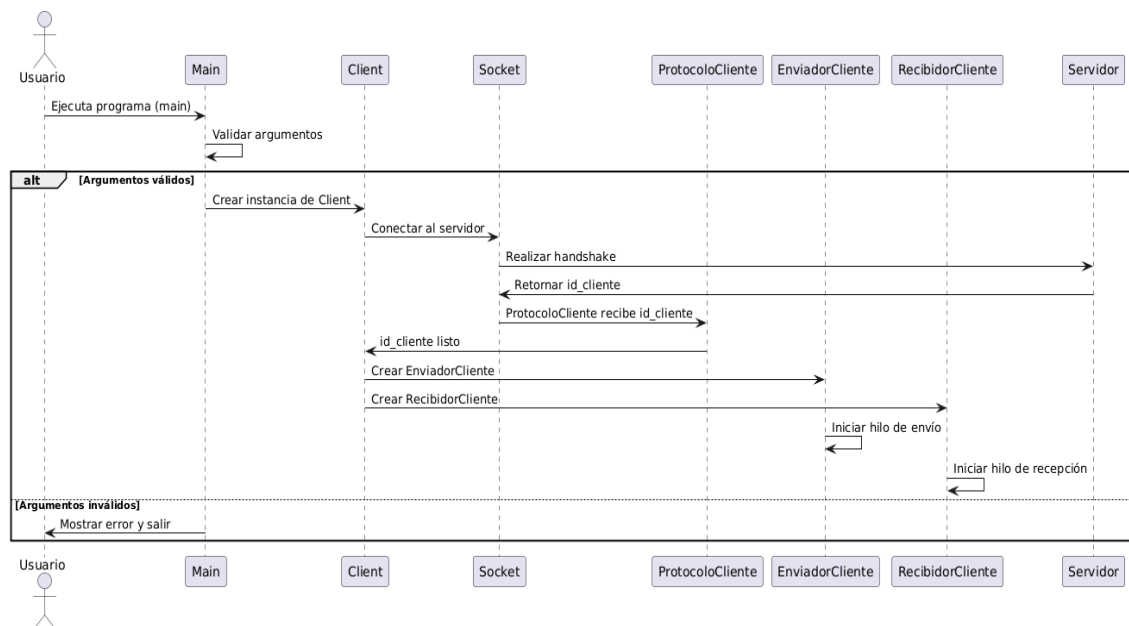


Figura 13: Diagrama de Secuencia: 6.

### 3.2.4. Diagrama de secuencia: Hilo Enviador

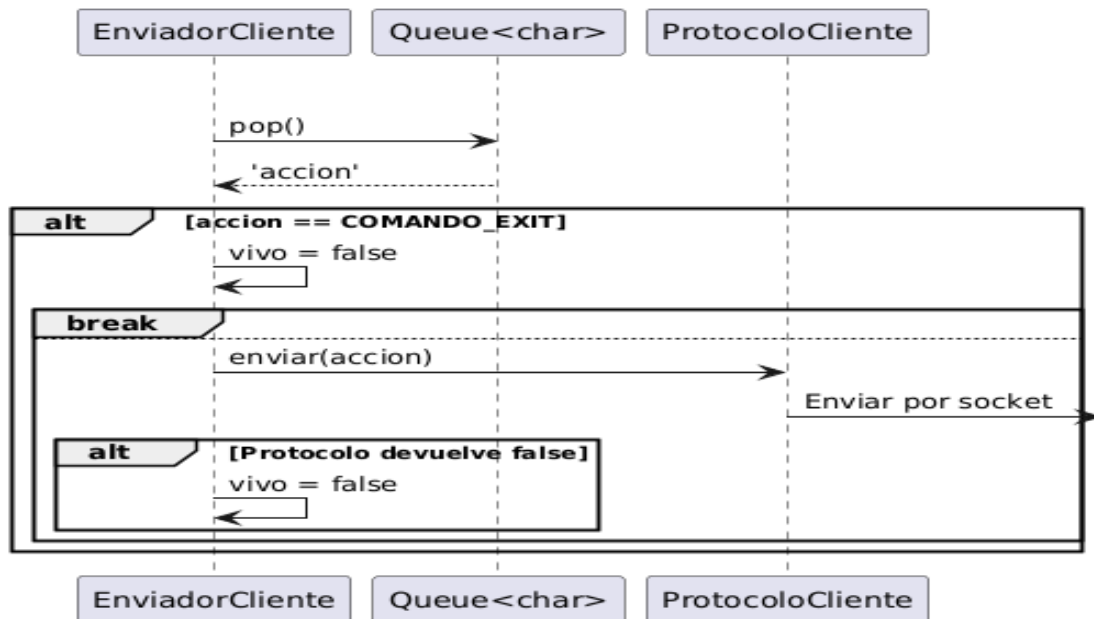


Figura 14: Diagrama de Secuencia: 7.

### 3.2.5. Diagrama de secuencia: Hilo Recibidor

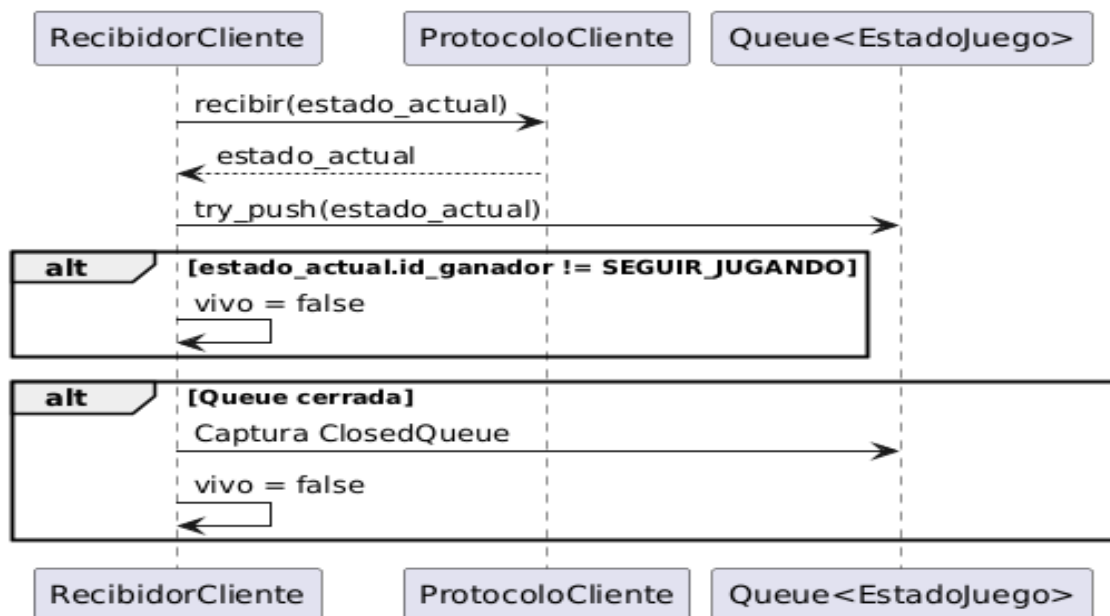


Figura 15: Diagrama de Secuencia: 8.

### 3.2.6. Diagrama de secuencia: Lógica del Cliente

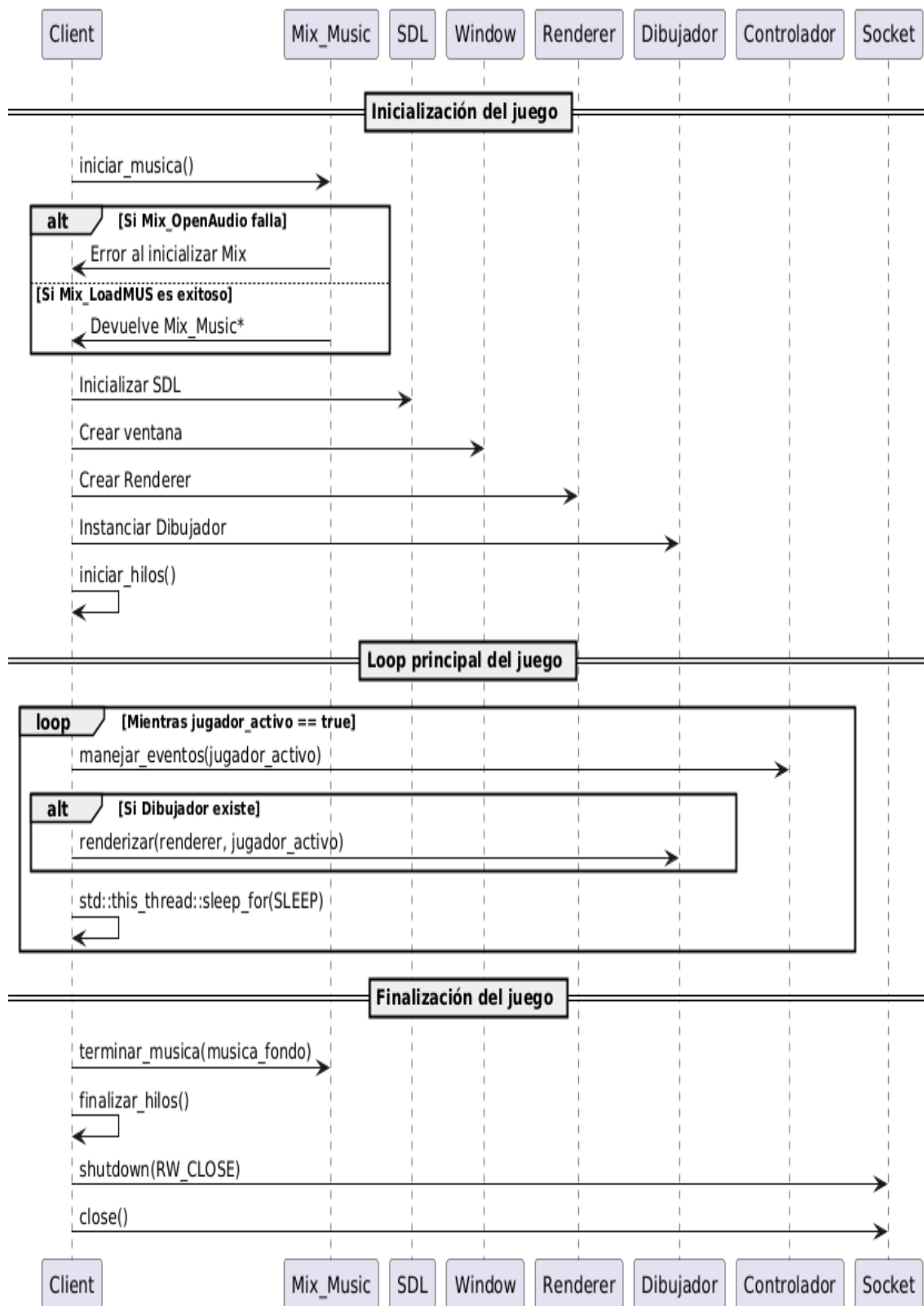


Figura 16: Diagrama de Secuencia: 9.

### 3.2.7. Diagrama de secuencia: Control de eventos del teclado

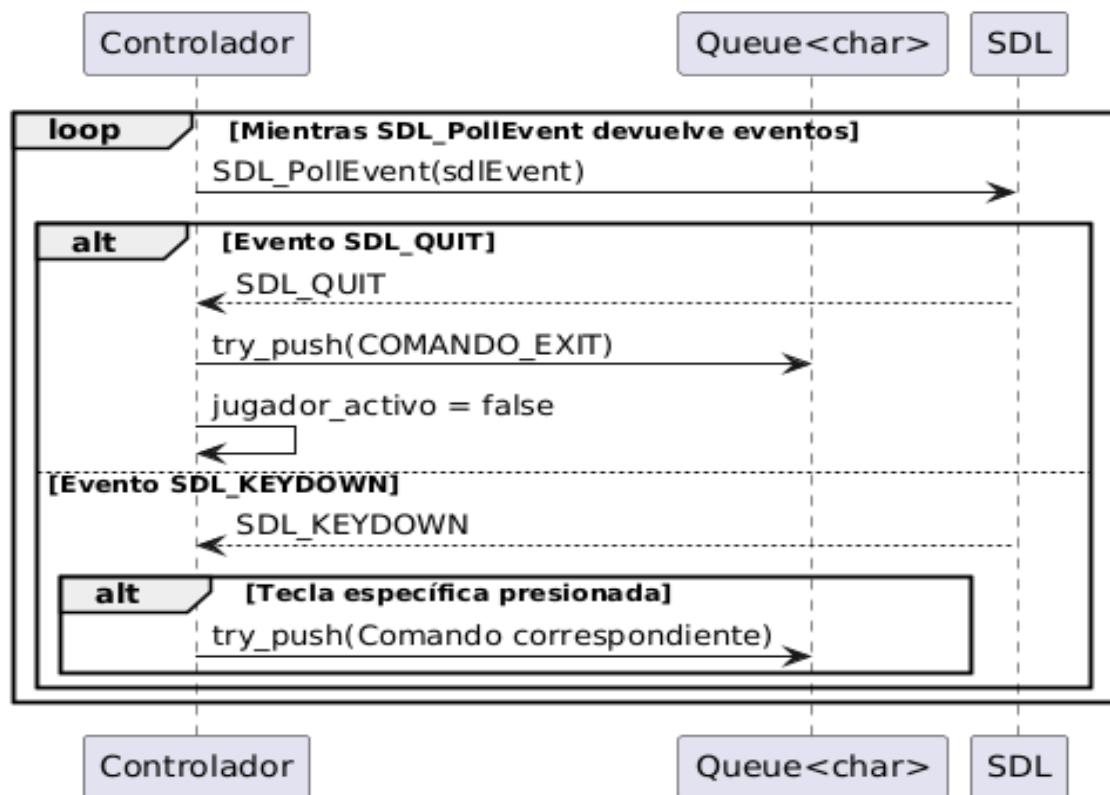


Figura 17: Diagrama de Secuencia: 10.

### 3.2.8. Diagrama de secuencia: Dibujo del estado actual del juego

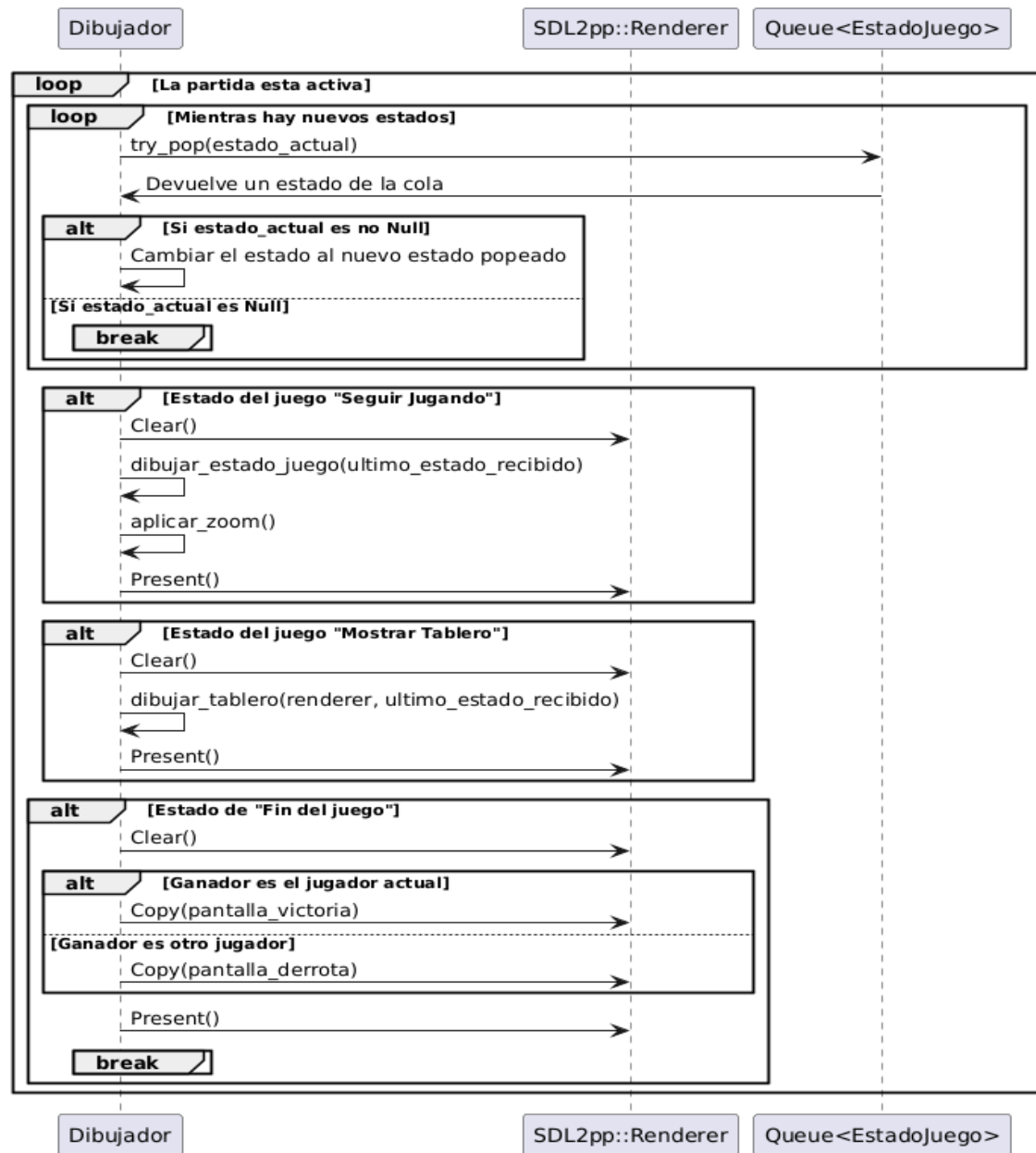


Figura 18: Diagrama de Secuencia: 11.

## 4. Análisis de Reporte de Valgrind

Los scripts que incluyen el indicativo 'valgrind' en su nombre permiten ejecutar tanto el servidor como los clientes utilizando esta herramienta, con el objetivo de verificar el comportamiento de la memoria y los accesos realizados por los programas.

Esta práctica resultó fundamental para garantizar que nuestro sistema optimice su funcionamiento y no incurra en errores relacionados con la gestión y el acceso a la memoria.

En el caso del servidor, la ejecución del script correspondiente muestra que el reporte de Valgrind

no detecta fugas de memoria ni errores en el funcionamiento del programa.

Sin embargo, en el cliente se observó que las bibliotecas SDL2 y SDL2pp generan falsos positivos relacionados con pérdidas de memoria y errores en el acceso a la memoria durante la liberación, los cuales se reflejan en el reporte final.

A continuación, se presenta una breve explicación de estos errores para fines documentales, especialmente dirigida a futuros desarrolladores que trabajen con el código relacionado. Asimismo, se garantiza (como se evidencia en el reporte final) que en el código de autoría propia no existen problemas de gestión de memoria ni fugas (leaks).

#### 4.1. Reporte de Valgrind: Pérdidas de Memoria

Como se mencionó previamente, la biblioteca SDL2 genera falsos positivos en los reportes de Valgrind, clasificados como pérdidas de memoria del tipo 'still reachable'.

Adicionalmente, se reportaron dos pérdidas de memoria: Una del tipo definitely lost y otra indirectly lost.

Tras una investigación detallada, se concluyó que estos problemas son causados por bugs propios de la herramienta Valgrind.

##### 4.1.1. Captura del Reporte de Pérdidas de Memoria

```
==8566== HEAP SUMMARY:
==8566==    in use at exit: 241,533 bytes in 3,272 blocks
==8566== total heap usage: 117,465 allocs, 114,193 frees, 181,321,061 bytes allocated
==8566==
==8566== 56 bytes in 1 blocks are indirectly lost in loss record 2,204 of 2,439
==8566==    at 0x484D953: calloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==8566==    by 0x18370781: ???
==8566==    by 0x17DF1A63: ???
==8566==    by 0x17DD9B60: ???
==8566==    by 0x17DD9D47: ???
==8566==    by 0x17D8688E: ???
==8566==    by 0x17D89641: ???
==8566==    by 0x17D84144: ???
==8566==    by 0x17D2D8BF: ???
==8566==    by 0x17D6DDEB: ???
==8566==    by 0x17D69F7F: ???
==8566==    by 0x17CFD482: ???
==8566==
==8566== 112 (56 direct, 56 indirect) bytes in 1 blocks are definitely lost in loss record 2,374 of 2,439
==8566==    at 0x484D953: calloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==8566==    by 0x1837075D: ???
==8566==    by 0x17DF1A63: ???
==8566==    by 0x17DD9B60: ???
==8566==    by 0x17DD9D47: ???
==8566==    by 0x17D8688E: ???
==8566==    by 0x17D89641: ???
==8566==    by 0x17D84144: ???
==8566==    by 0x17D2D8BF: ???
==8566==    by 0x17D6DDEB: ???
==8566==    by 0x17D69F7F: ???
==8566==    by 0x17CFD482: ???
==8566==
==8566== LEAK SUMMARY:
==8566==    definitely lost: 56 bytes in 1 blocks
==8566==    indirectly lost: 56 bytes in 1 blocks
==8566==    possibly lost: 0 bytes in 0 blocks
==8566==    still reachable: 241,421 bytes in 3,270 blocks
==8566==    suppressed: 0 bytes in 0 blocks
==8566== Reachable blocks (those to which a pointer was found) are not shown.
==8566== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==8566==
```

Figura 19: Captura de pantalla: 1.



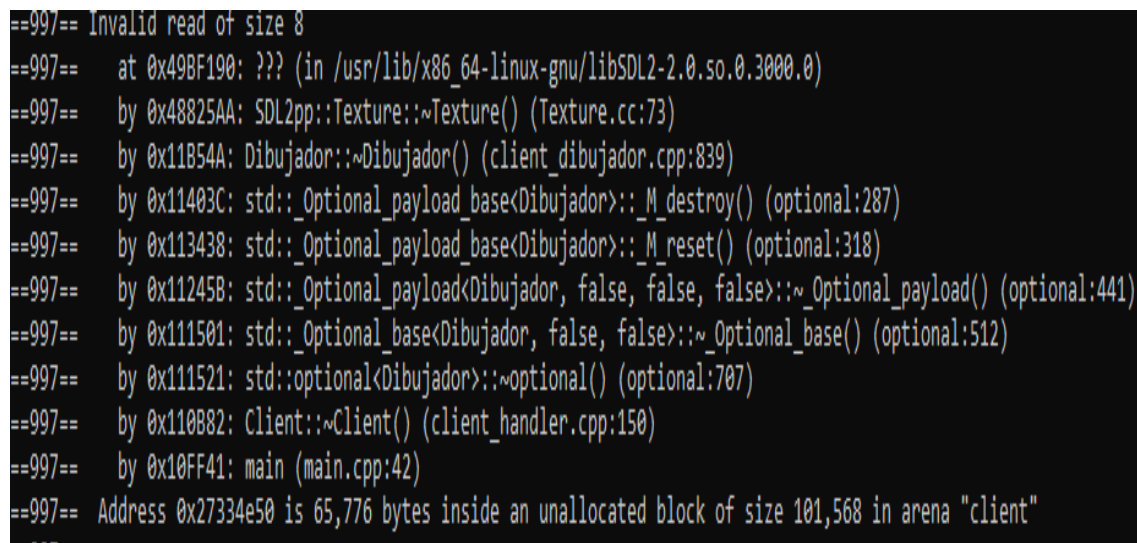
## 4.2. Reporte de Valgrind: Error en SDL2pp

El reporte de Valgrind también señala algunos errores relacionados con el acceso a memoria luego de ser liberada en el destructor de las texturas.

Nuestro modelo sigue el patrón RAII (Resource Acquisition Is Initialization), lo que significa que no manejamos memoria dinámica de manera manual, excepto en el caso de los sonidos.

Sin embargo, la biblioteca SDL2pp aparentemente no verifica si la textura que intenta liberar ya ha sido eliminada, lo que provoca un acceso indebido a la memoria y genera el error reportado. El detalle de este problema se muestra a continuación:

### 4.2.1. Captura del Reporte de Error en SDL2pp



```
==997== Invalid read of size 8
==997== at 0x498F190: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.3000.0)
==997== by 0x48825AA: SDL2pp::Texture::~~Texture() (Texture.cc:73)
==997== by 0x11854A: Dibujador::~~Dibujador() (client_dibujador.cpp:839)
==997== by 0x11403C: std::_Optional_payload_base<Dibujador>::~_M_destroy() (optional:287)
==997== by 0x113438: std::_Optional_payload_base<Dibujador>::~_M_reset() (optional:318)
==997== by 0x112458: std::_Optional_payload<Dibujador, false, false, false>::~~_Optional_payload() (optional:441)
==997== by 0x111501: std::_Optional_base<Dibujador, false, false>::~~_Optional_base() (optional:512)
==997== by 0x111521: std::optional<Dibujador>::~optional() (optional:707)
==997== by 0x110B82: Client::~~Client() (client_handler.cpp:150)
==997== by 0x10FF41: main (main.cpp:42)
==997== Address 0x27334e50 is 65,776 bytes inside an unallocated block of size 101,568 in arena "client"
```

Figura 20: Captura de pantalla: 2.

Se realizaron intentos para corregir este error en la biblioteca. Sin embargo, al no contar con acceso al archivo '.cpp' que implementa el destructor de las texturas, no fue posible solucionar el problema.

Esta situación queda documentada para referencia en futuras ocasiones.

## 4.3. Consideración final

Debido a la gran cantidad de falsos positivos relacionados con pérdidas de memoria que genera SDL2, el script que utiliza la herramienta Valgrind en el cliente incluye una configuración específica para suprimir los reportes de pérdidas del tipo still reachable. Esto se realizó con el objetivo de obtener reportes más claros y enfocados.

No obstante, se dejó comentada una variante del script que no suprime estos reportes, para que pueda ser utilizada si es requerida en el futuro.

## 5. Conclusión

Esperamos que esta documentación facilite la comprensión del funcionamiento del juego. Invitamos a los interesados a mejorar este proyecto, implementando sus propias ideas y funcionalidades. Si decides hacerlo, no dudes en contactarnos; nos encantaría probar tus contribuciones.