Documento 4+1 Vistas + CODIGO (Proyecto Rummy)

Alumnos—
Pablo Jesús Galán Valenzuela - 00000244752
Héctor Francisco Báez Luque - 00000247283
Francisco Valdez Gastelum- 00000246904
Kathya Margarita Cordova Soto - 00000246801
Manuel Octavio Pérez Dominguez - 00000247439

Asignación—
Proyecto Final - Diagramas y Repositorio de Codigo

Fecha—
5 de Diciembre del 2024

Materia—
Arquitectura de Software

Profesor—
Gilberto Borrego Soto

ÍNDICE

1. Descripción del Juego

El Rummy es un juego de cartas que se juega con 104 fichas, incluyendo dos comodines y números del 1 al 13 en cuatro colores diferentes. Cada jugador recibe 14 fichas al azar y debe formar al menos 30 puntos con ellas para comenzar a jugar, utilizando combinaciones de grupos (3 o 4 fichas del mismo número pero de diferentes colores) o secuencias (3 o más números consecutivos del mismo color). Durante su turno, un jugador puede añadir o mover fichas en el tablero, realizar movimientos como cambiar o utilizar fichas, separar grupos, sustituir fichas o tomar una ficha del montón si no puede jugar. El juego termina cuando un jugador coloca todas sus fichas en el tablero.

Además, se permite personalizar el rango de números de las fichas y la cantidad de comodines, y puede ser jugado por 2 a 4 jugadores en un entorno de red local. El desarrollo del juego será realizado en Java, utilizando el patrón MVC para la interfaz gráfica.

2. Repositorio GitHub y cómo iniciarlo

Para poder ejecutar el código es muy sencillo, simplemente se tienen que abrir los 3 proyectos que vienen en el repositorio y abrirlos en su IDE de preferencia. De ahí tiene que construirlos con dependencias (Build with Dependencies) Empezando por dominio, siguiendo con red y terminando con GUI. Puede que llegue a dar error por versión, en ese caso solo tiene que cambiar dentro del apartado de propiedades del proyecto la plataforma de java en la opción de Compilar y en Sources en caso de ser necesario.

Al realizar eso, se tiene que ejecutar la clase MainServer del proyecto Red_Rummy en el paquete servidor. Después se pueden ejecutar hilos del GUI en la clase Prueba del paquete mvc.pruebas del proyecto GUI_Rummy

https://github.com/xSpaceKat/RummyEquipo6

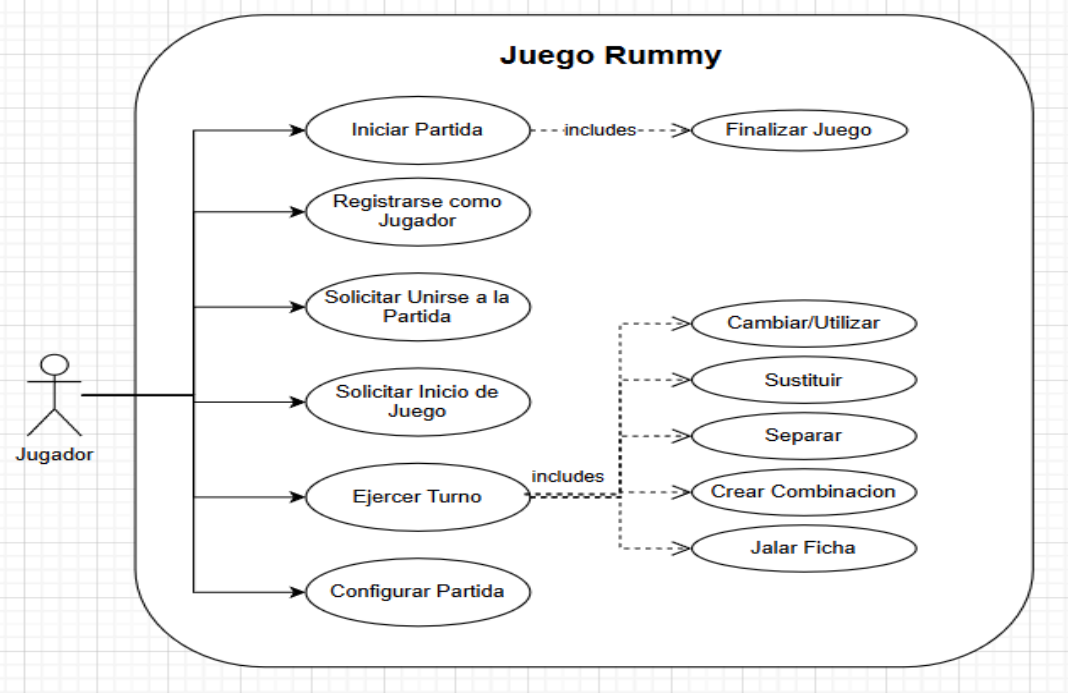
3. Patrón arquitectónico: Pipes & Filters

En el desarrollo de nuestro proyecto de Rummy, hemos adoptado el estilo arquitectónico Pipes and Filters para estructurar el flujo de datos y facilitar la validación de las reglas del juego. Este enfoque nos permite diseñar un sistema modular y extensible, alineado con los principios de separación de responsabilidades y escalabilidad.

El patrón Pipes and Filters organiza un sistema como una secuencia de componentes independientes denominados filtros, que procesan datos y los transforman paso a paso. Los pipes actúan como conectores que transfieren datos entre los filtros. En el contexto de nuestro proyecto, esto permite un flujo claro y predecible de validaciones y transformaciones, garantizando que las reglas del juego se apliquen en cada etapa.

4. Vista de escenarios

Diagrama de casos de uso:



Los siguientes casos de uso, fueron planteados por el docente con el propósito de separar las funcionalidades del sistema.

- **Configurar partida:** En este caso de uso el jugador establece el rango de fichas y el número de comodines, así como pone como disponible la partida.

- **Registrarse como jugador**: Aquí es cuando se registra nombre, avatar y colores de las fichas del jugador.
- **Solicitar unirse a la partida**: Ya con una partida configurada, otros jugadores pueden pedir unirse a dicha partida, pero no cuando esta ya esté iniciada.
- **Solicitar inicio de juego**: Es cuando uno de los jugadores pide que se inicie el juego. Se incluye la confirmación o negación de inicio por parte del resto de los jugadores.
- **Iniciar partida**: Esto sucede una vez que todos los jugadores aceptaron la solicitud de inicio, o bien, se conectaron 4 jugadores. Luego de ello se hace la repartición de fichas a cada jugador y se determina el orden de los turnos.
- **Ejercer turno**: Implica cualquier serie de movimientos que el jugador pueda hacer en cualquier turno, desde la tirada inicial, pasando bajar grupos, cortar grupos para ingresar fichas en ellos, tomar fichas del mazo, hasta terminar el juego porque se ganó.

Sin embargo, bajo un consenso del equipo, se decidió agregar un caso de uso externo a los propuestos por el empleador, el cual consiste en lo siguiente:

- **Finalizar juego:** Implica que, cuando el primer jugador quede sin fichas, el juego llegue a su fin.

Asimismo, dado que el caso de uso planteado "Ejercer turno", implicaba sus propias funcionalidades complementarias más complejas, se separó este mismo caso de uso en partes, denominadas y descritas de la siguiente manera:

- **Cambiar/Utilizar**: En el caso de que haya tres fichas juntas, se puede añadir una en un extremo y sacar la que hay en el otro extremo, para utilizarla en otra parte de los grupos de fichas puestas en el tablero. En el caso de que haya más de 3 fichas, podrá tomar cualquiera de los extremos sin añadir ninguna más.
- **Sustituir**: En el caso de los grupos de tres fichas de números iguales, se podrá tomar una ficha con un número de un color y en su lugar poner otra con el mismo número, pero de diferente color.
- **Separar**: Si hay más de 6 fichas en secuencia, se podrá separarlas y crear dos grupos de 3 o más y añadir las piezas del jugador en turno, siguiendo la misma secuencia.
- **Crear combinación**: Permite al jugador bajar una lista de fichas a su tablero, permitiendo ir reduciendo su número de fichas. Tiene dos opciones, crear una combinación de 3 o 4 fichas de números iguales o de caso contrario, a partir de una combinación de 3 fichas en adelante crear una secuencia de un mismo color
- **Jalar ficha**: Supone que, si un jugador no tiene más movimientos a realizar, el susodicho se dispone a jalar una ficha del mazo, teniendo como consecuencia el cambio de turno.

5. Explicación de patrones aplicados

Durante el desarrollo de nuestro análisis y diseño, como equipo, hemos podido implementar de buena manera ciertos patrones que han sido de gran utilidad para el desarrollo del proyecto. Estos patrones fueron:

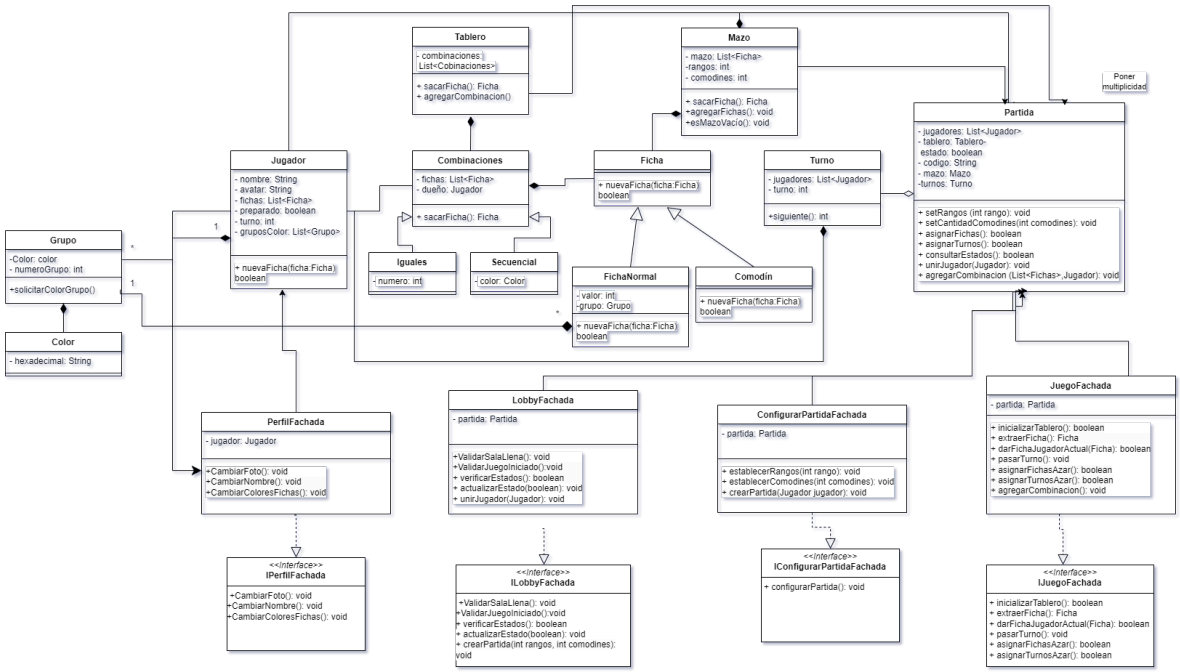
- **Singleton**: Asegura que una clase tenga solo una única instancia en todo el sistema y proporciona un punto de acceso global a esa instancia. Hemos implementado el patrón Singleton para evitar la necesidad de múltiples return en una pipeline. De esta forma, cuando se llega a la última etapa de la pipeline, simplemente se establece el valor del resultado usando getInstance del Singleton. Esto garantiza que solo exista una única instancia que almacena el resultado final de la pipeline, facilitando el acceso a ese valor en cualquier parte del proceso sin tener que propagar el resultado en cada etapa.
- **Strategy**: Si las decisiones sobre qué algoritmo o comportamiento ejecutar están delegadas a objetos específicos que se pueden intercambiar, puede estar presente el patrón Strategy. Facilita cambiar el comportamiento de un objeto en tiempo de ejecución. Y justo como su descripción, nos vimos obligados a implementar este método dentro de las pipelines de P&F. Este se nota en los diagramas de clase de diseño, en donde mediante este patrón podemos variar el comportamiento del método “enviar()” de nuestra clase “Pipe” con el objetivo de cambiar a qué filtro deberá apuntar la pipa y saber cual es su siguiente filtro a llegar o si es que directamente llegará al sink (que varía dependiendo el CU).
- **Fachadas**: Utilizado para agrupar un conjunto de operaciones relacionadas con la partida bajo una única interfaz, simplificando la interacción con el sistema interno. Esta idea fue dada debido a que consideramos que el tener que usar los métodos específicos de cada clase en nuestro estilo arquitectónico sería un tanto complicado de y tedioso de implementar. Por lo que decidimos implementar una sola fachada (que posteriormente sería segregada) en donde tuviéramos los métodos listos para usar en nuestra arquitectura. Pudiendo acceder a nuestro dominio mediante una interfaz para hacer operaciones de procesos.
- **Segregación de interfaces**: Aplicamos este principio para poder reducir el tamaño de las y crear interfaces específicas y más especializadas. Este patrón fue descubierto al proponer la fachada ya mencionada en donde propusimos una fachada para todas las entidades, en donde pudiera tener métodos que nos ayudará en la lógica del juego al momento de usar nuestro estilo arquitectónico. Al poner los métodos en una sola fachada descubrimos que había muchos métodos que podían ser agrupados, evitandonos tener una interfaz gigante. Así fue como lo implementamos y dividimos nuestras fachadas en una para cada parte fundamental del proceso, una para el perfil, otra para el lobby, otra para la partida en sí mismo, etc.
- **MVC y Observer:** Este patrón puede ser reconocido cuando hay múltiples componentes o clases observando y reaccionando a cambios en el estado de otros

objetos. Y esto en esencia se cumple en nuestros diagramas, puesto que la vista se actualizará conforme todo el proceso y cambio que haya generado el modelo. El patrón observer es necesario para poder implementar un MVC clásico. Sin este patrón sería imposible lograrlo.

- **Chain of Responsability:** Hemos implementado el patrón Chain of Responsibility de manera natural al utilizar el estilo arquitectónico Pipes and Filters. En nuestra implementación, cada filtro representa una etapa en la cadena de procesamiento, donde cada componente (o filtro) realiza una operación específica sobre los datos y luego los pasa al siguiente en la cadena donde Chain of Responsibility permite que cada filtro decida si procesa o simplemente pasa los datos al siguiente.

6. Vista Lógica

6.1. Diagrama de dominio



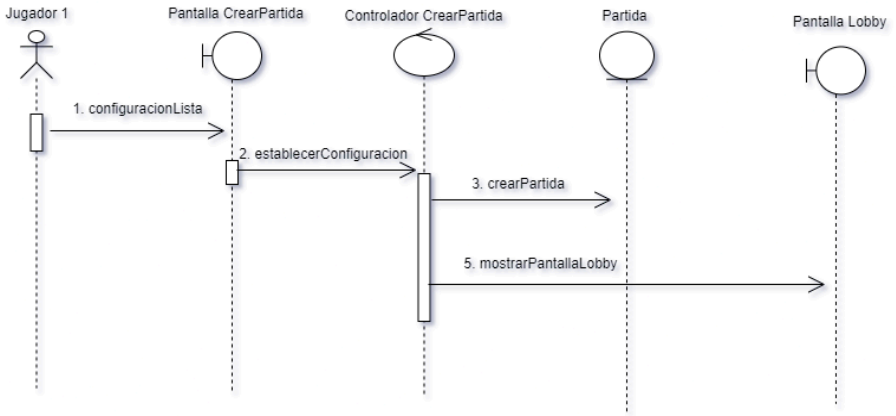
Explicación de clases de dominio:

- **Partida:** Representa una partida del juego Rummy. Naturalmente, contiene las características esenciales que conecta todo el juego, entre ellas el Mazo de fichas, los Jugadores, el Tablero y el orden de Turnos.

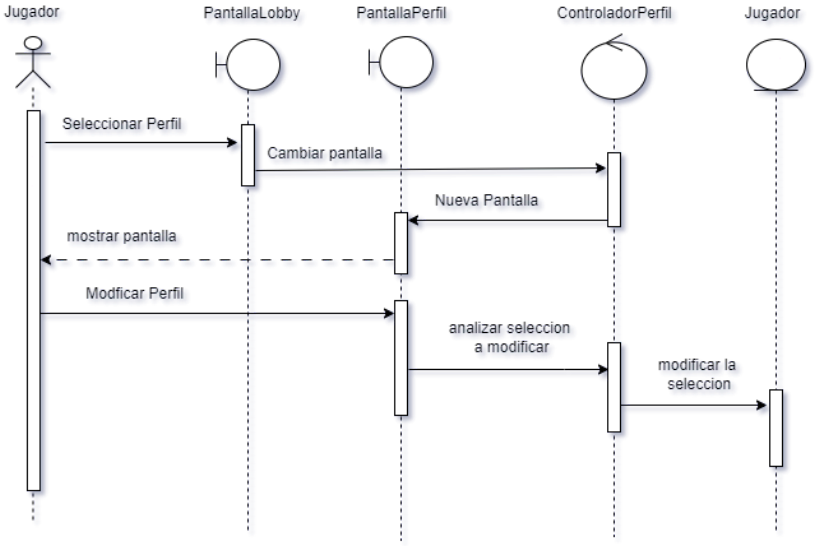
- **Tablero:** Simula un tablero del juego, por lo mismo, se compone del grupo de combinaciones que serán colocados dentro de él.
- **Jugador:** Contiene las características primordiales que puede tener un jugador en un juego en línea, ya sea su nombre, su avatar, el conjunto de fichas que posee en su mano, y el turno que tiene indicado.
- **Mazo:** Representa la totalidad de fichas que los jugadores podrán utilizar durante la partida, se compone tanto de fichas normales como de comodines. Aquel que cree la partida podrá escoger el rango máximo de fichas de cada color así como también la cantidad de comodines que contendrá la misma.
- **Combinaciones:** Son el conjunto de maneras en las cuales se puede hacer una combinación de ficha como movimiento en el tablero.
    - **Secuencial:** Este tipo de combinación de fichas se basa en que sea una serie numérica secuencial o consecutiva, con la única diferencia que las fichas deben de ser del mismo color, si alguna no cumple con esta regla, puede afectar en no poder descartar muchas fichas o incluso no poner ninguna en el turno.
    - **Iguales:** Para este tipo de combinación debe de ser lo opuesto a la combinación secuencial, aquí las fichas deben de ser de un mismo valor numérico pero no necesariamente del mismo color.
- **Ficha:**
    - **Normal:** Este tipo de fichas contienen un color y valor, dichos atributos son los necesarios para usarse en alguna combinación o movimiento durante la partida.
    - **Comodín:** Las fichas comodín, como su nombre lo dice, es una ficha especial para usarse de la manera que gustes durante la partida, dichos usos son su agregación a cualquier tipo de combinación, ya que toma la responsabilidad de tomar el valor o color a su conveniencia para la combinación.
- **Turno:** Permite tener un control y registro del orden en que los jugadores podrán realizar sus movimientos.
- **Grupo:** Representa los grupos de fichas normales en las que será dividido el mazo, teniendo así, un par de cada color, con una cantidad de 4 colores, y, por consiguiente, 8 grupos de fichas normales.
- **Color:** Cada grupo de fichas normales es representado por un color, es decir, es la característica que necesita una ficha normal para representar su color.

6.2. Modelos de análisis

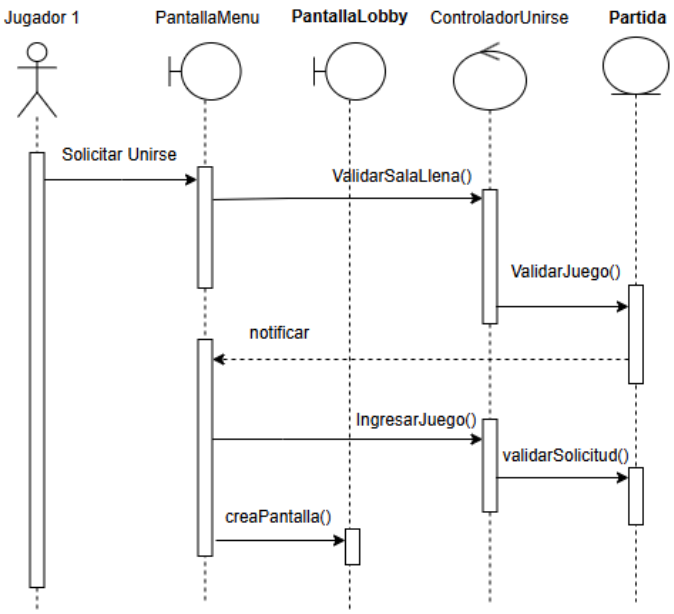
6.2.1. CU1: Configurar partida



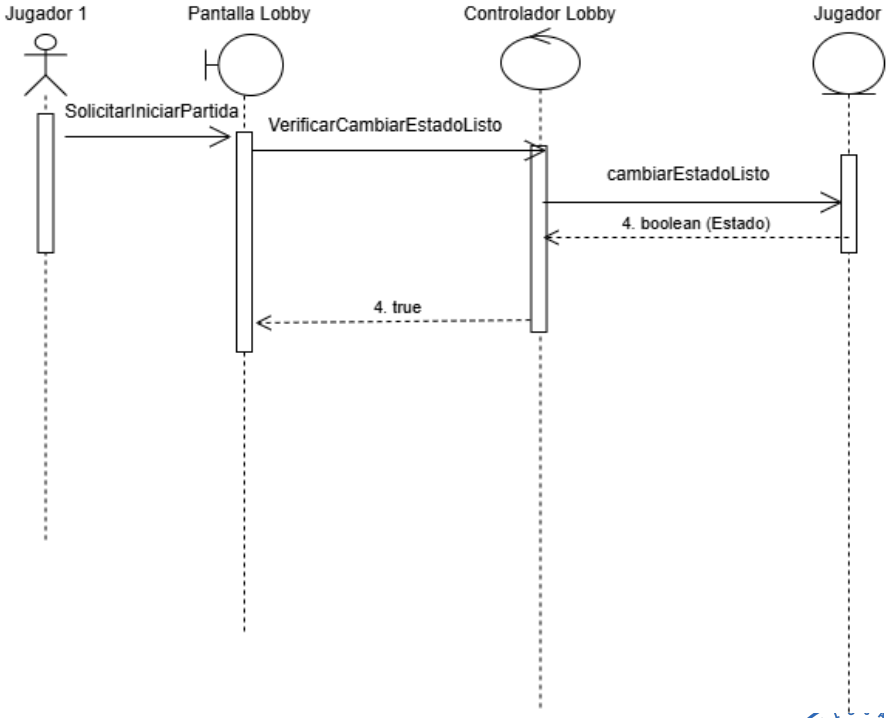
6.2.2. CU2: Registrarse como jugador



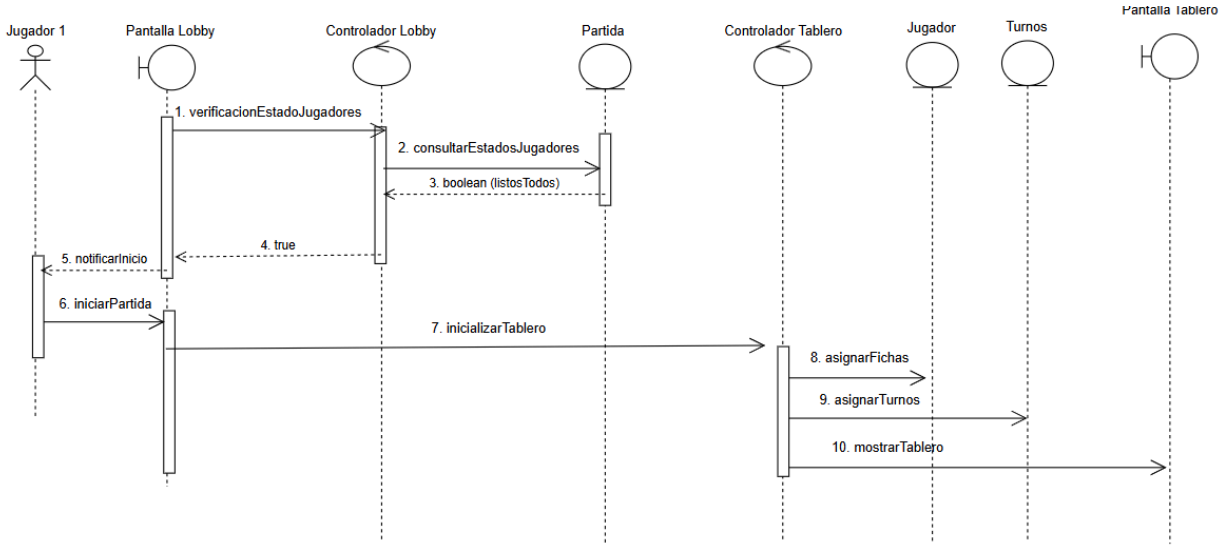
6.2.3. CU3: Solicitar unirse a la partida



6.2.4. CU4: Solicitar inicio de juego

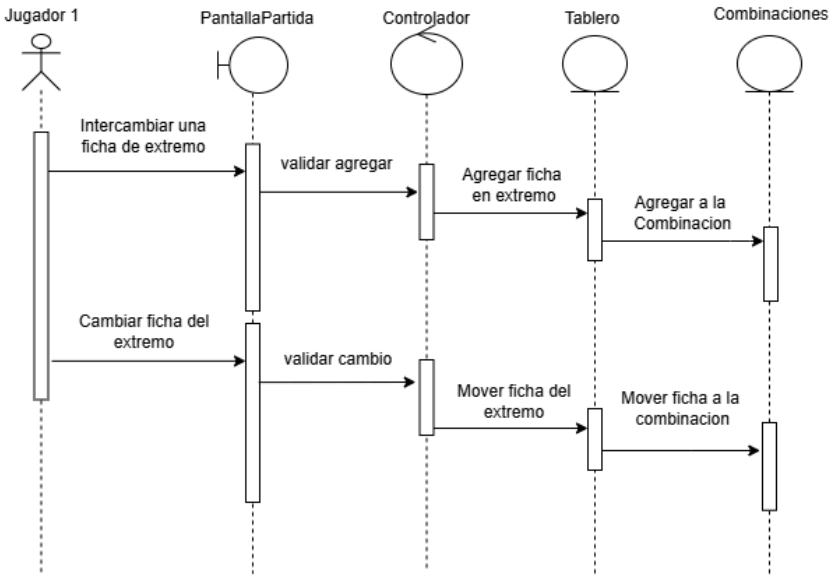


6.2.5. CU5: Iniciar partida

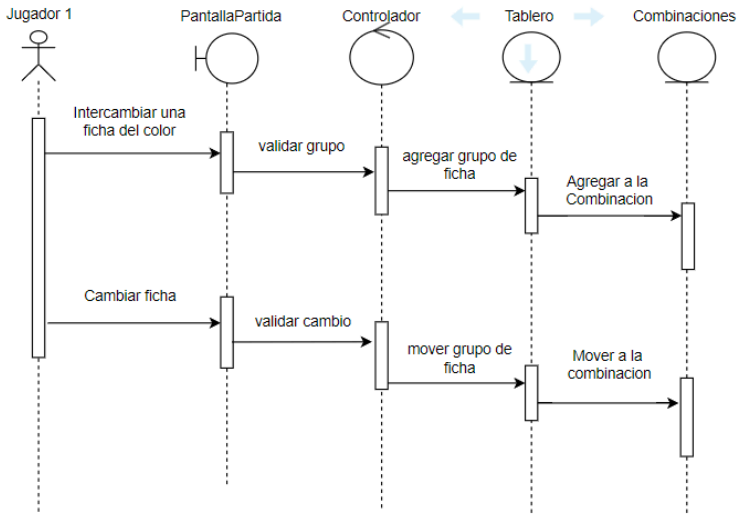


6.2.6. CU6: Ejercer turno

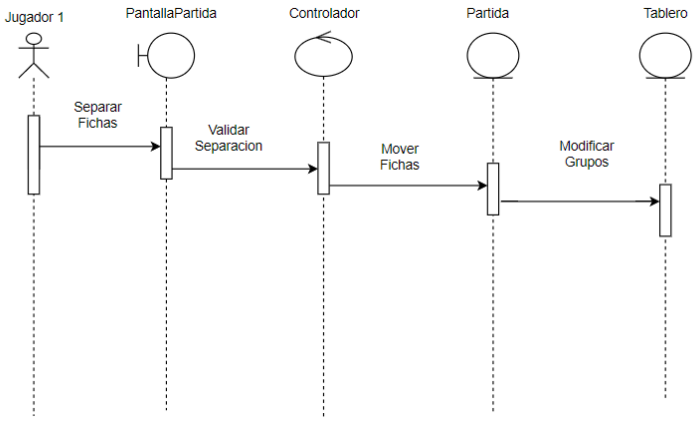
6.2.6.1. Parte 1: Cambiar/Utilizar



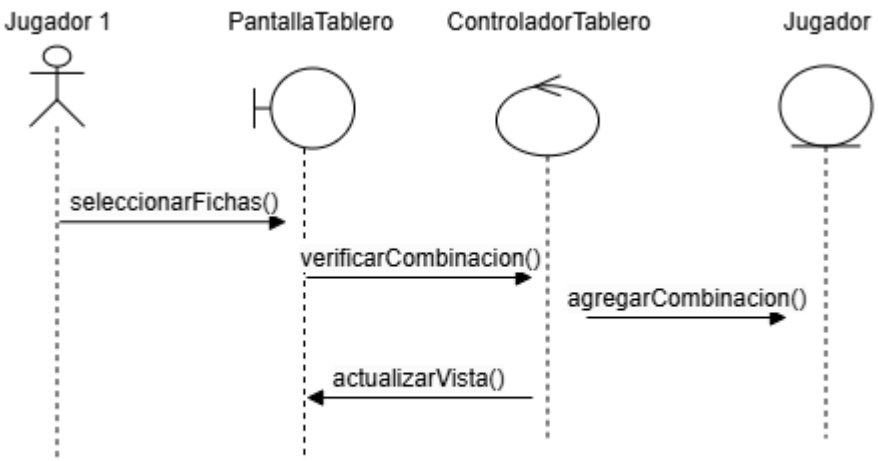
6.2.6.2. Parte 2: Sustituir



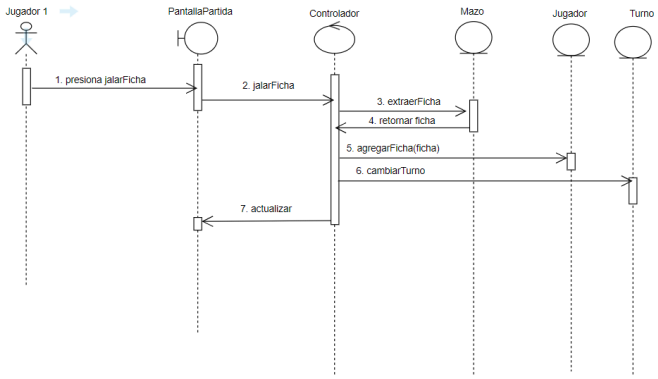
6.2.6.3. Parte 3: Separar



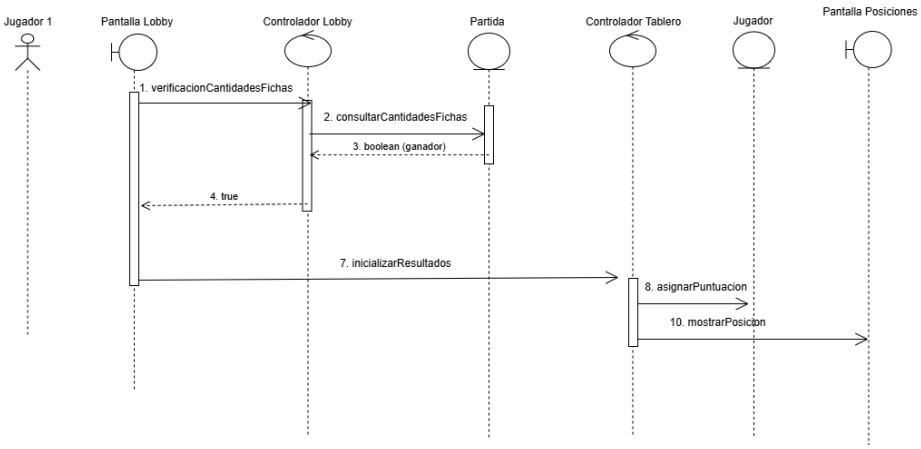
6.2.6.4. Parte 4: Crear combinación



6.2.6.5. Parte 5: Jalar ficha

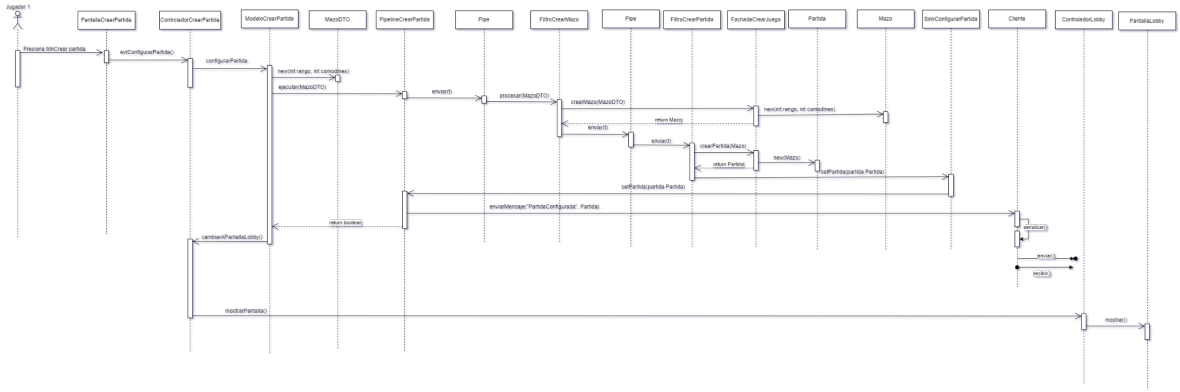


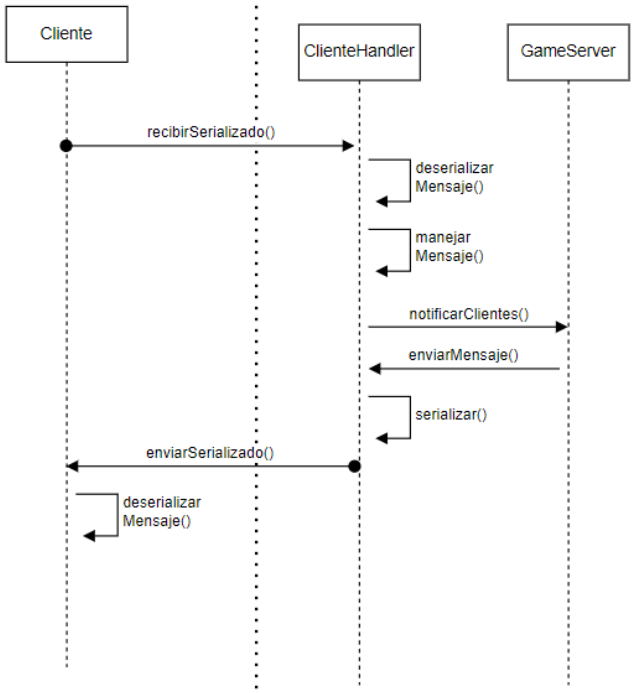
6.2.7. CU7: Terminar partida

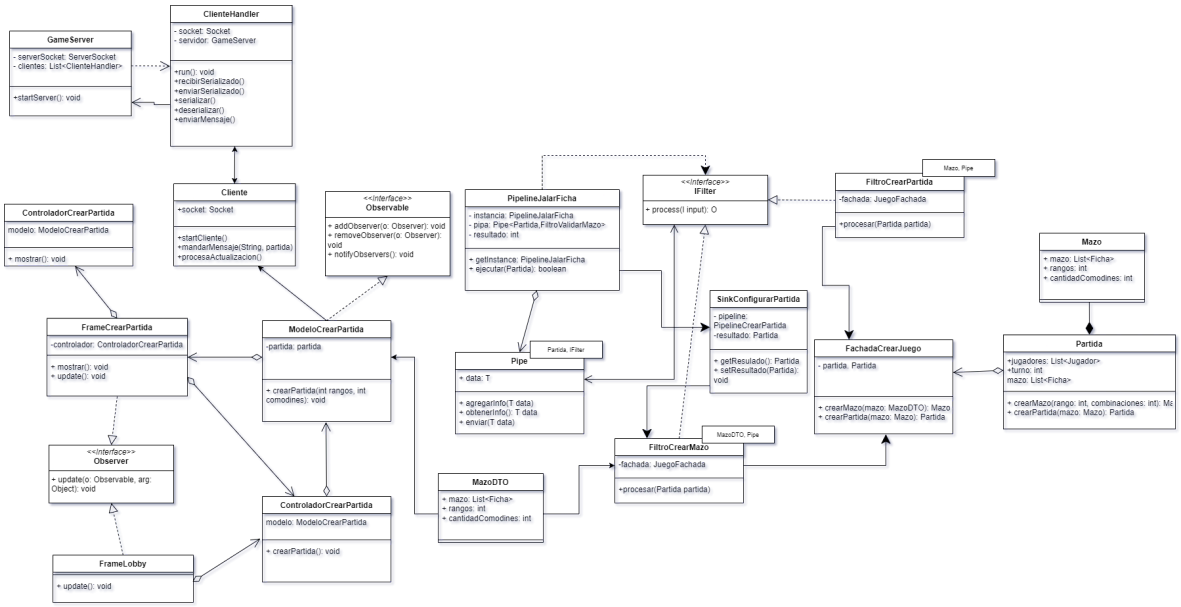


6.3. Modelos de diseño

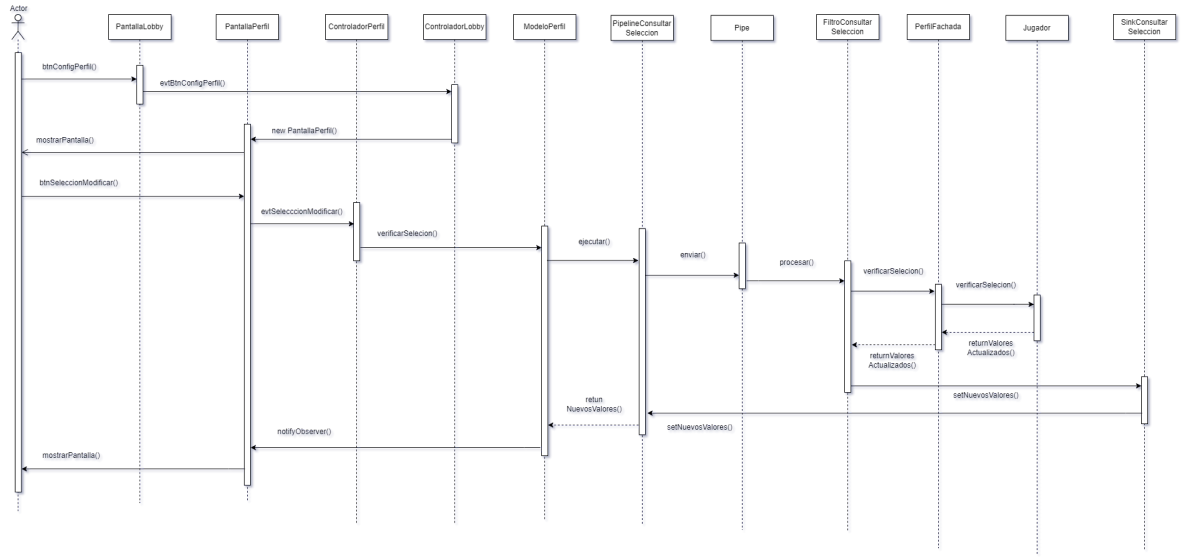
6.3.1. CU1: Configurar partida

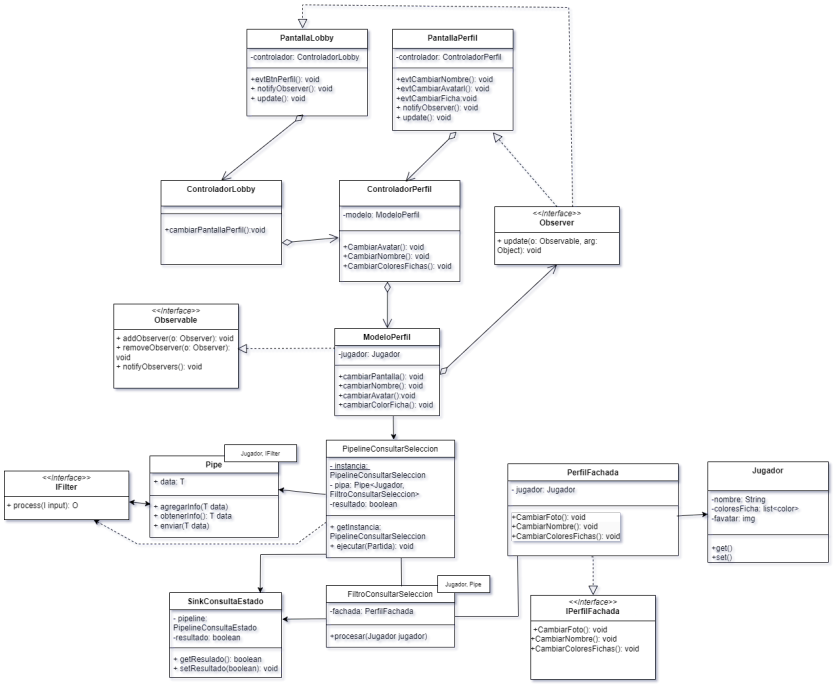




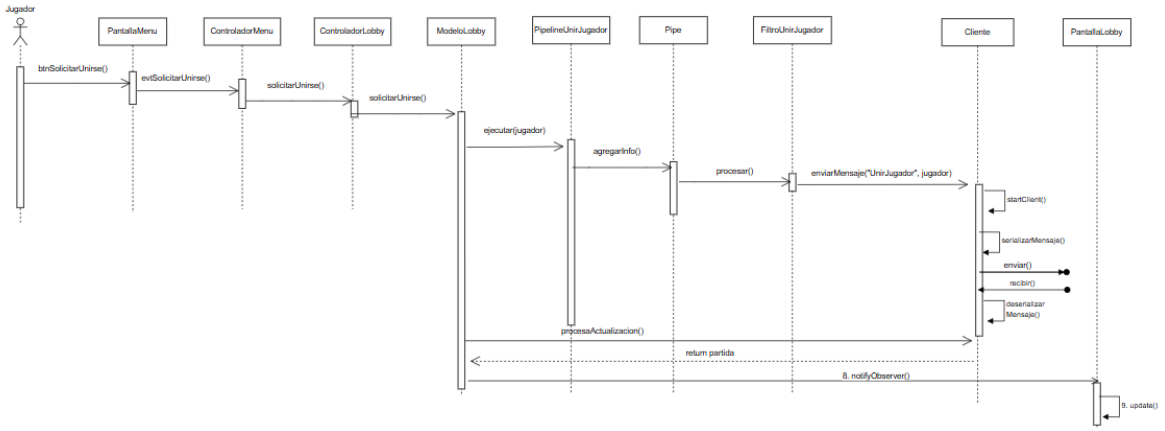


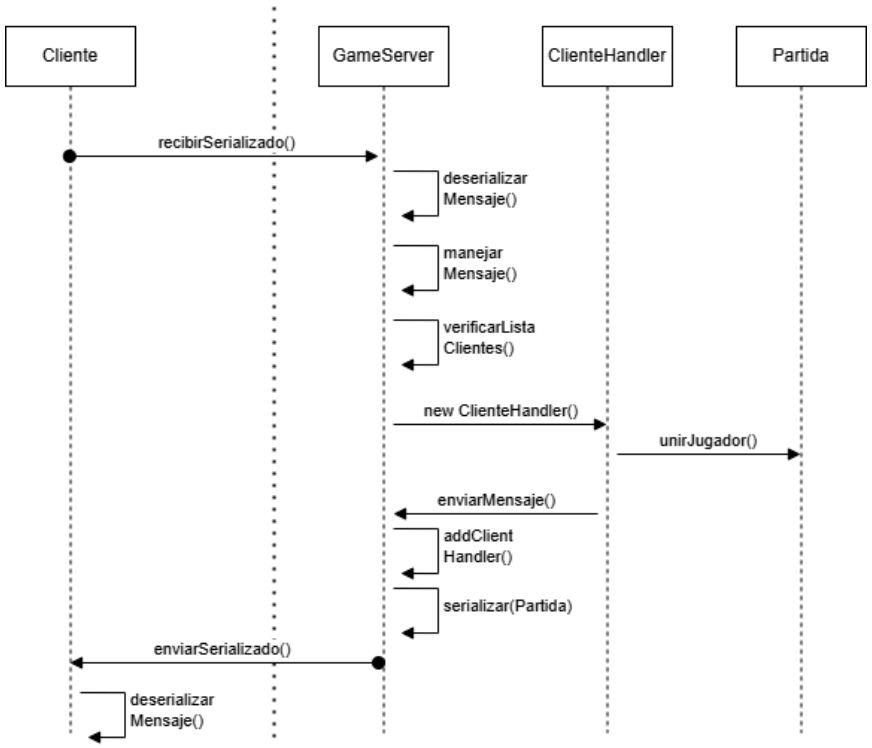
6.3.2. CU2: Registrarse como jugador

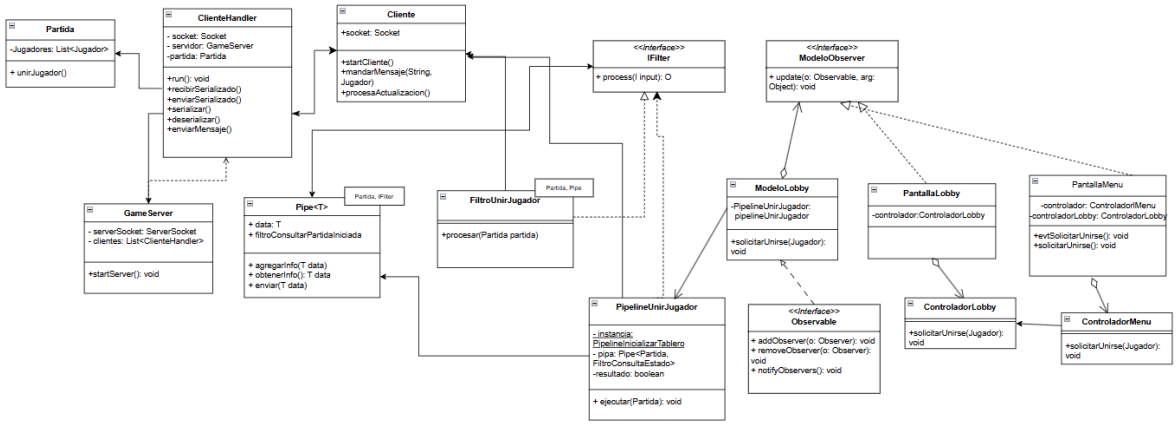




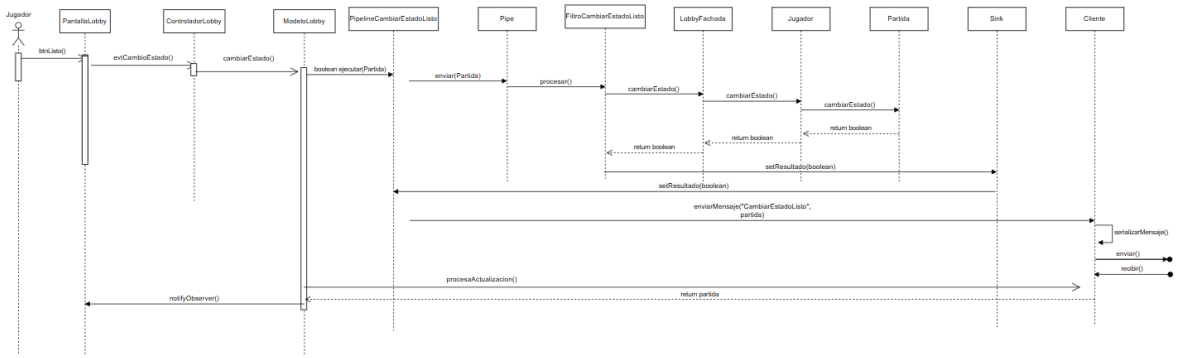
6.3.3. CU3: Solicitar unirse a la partida

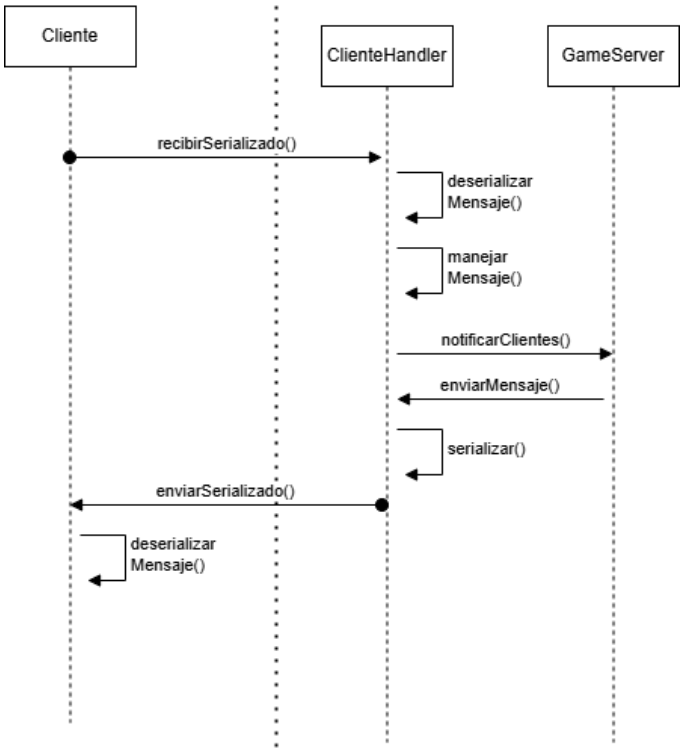


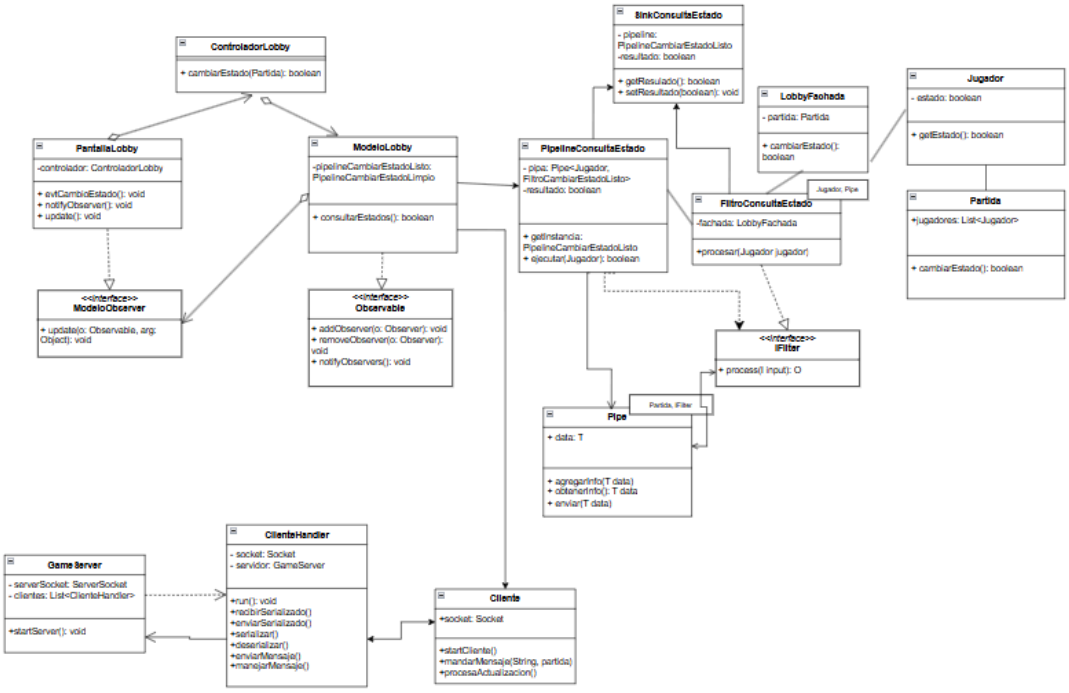




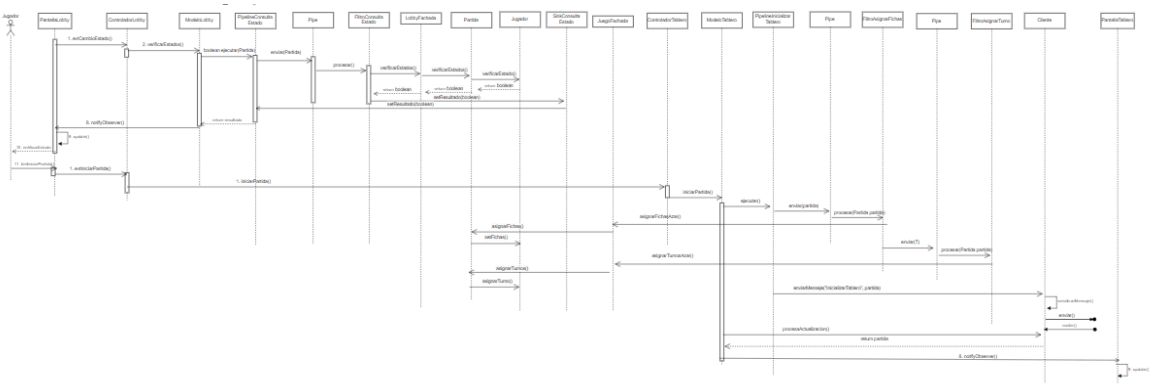
6.3.4. CU4: Solicitar inicio de juego

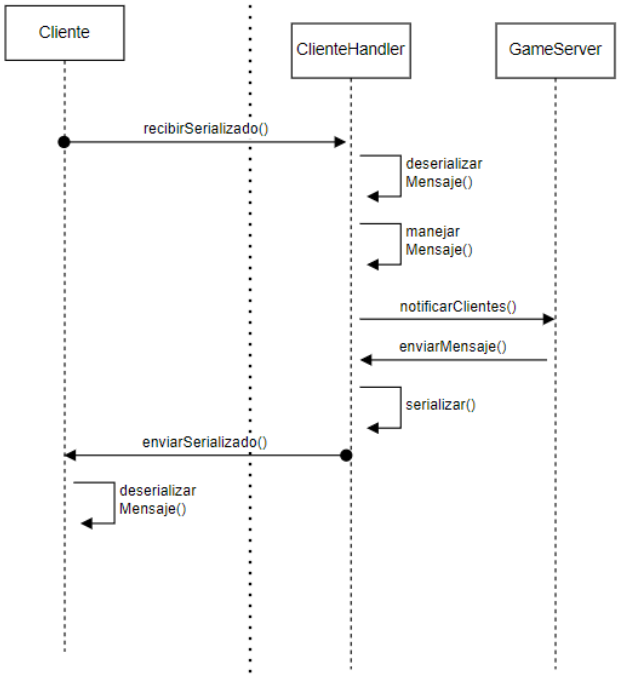

Jugador
PantallaLobby
ControladorLobby
ModeloLobby
PipelineCambiarEstadoListo
Pipe
FiltroCambiarEstadoListo
LobbyFachada
Jugador
Partida
Sink
Cliente
btnListo()
evtCambioEstado()
cambiarEstado()
boolean ejecutar(Partida)
enviar(Partida)
procesar()
cambiarEstado()
cambiarEstado()
cambiarEstado()
return boolean
return boolean
return boolean
setResultado(boolean)
setResultado(boolean)
enviarMensaje("CambiarEstadoListo", partida)
serializarMensaje()
enviar()
recibir()
procesaActualizacion()
return partida
notifyObserver()

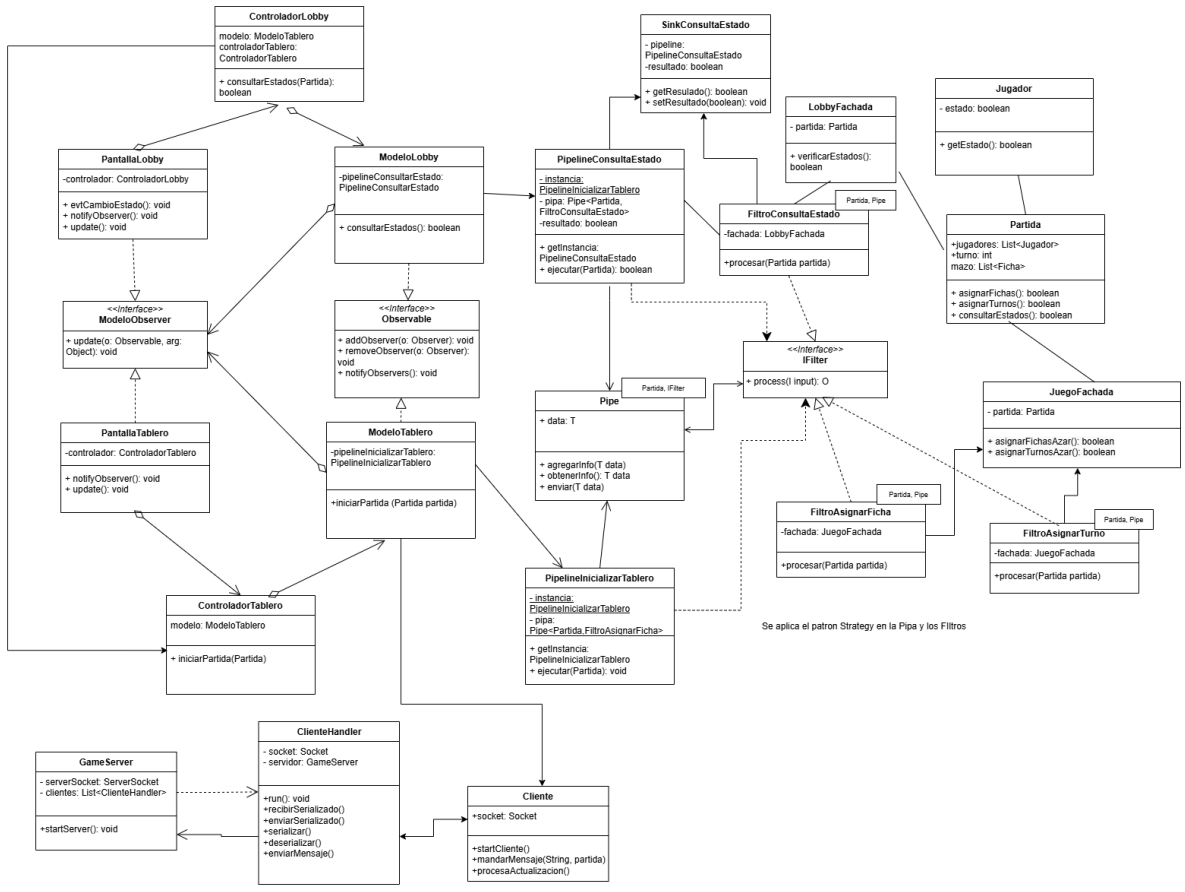

Cliente
ClienteHandler
GameServer
recibirSerializado()
deserializar Mensaje()
manejar Mensaje()
notificarClientes()
enviarMensaje()
serializar()
enviarSerializado()
deserializar Mensaje()



6.3.5. CU5: Iniciar partida

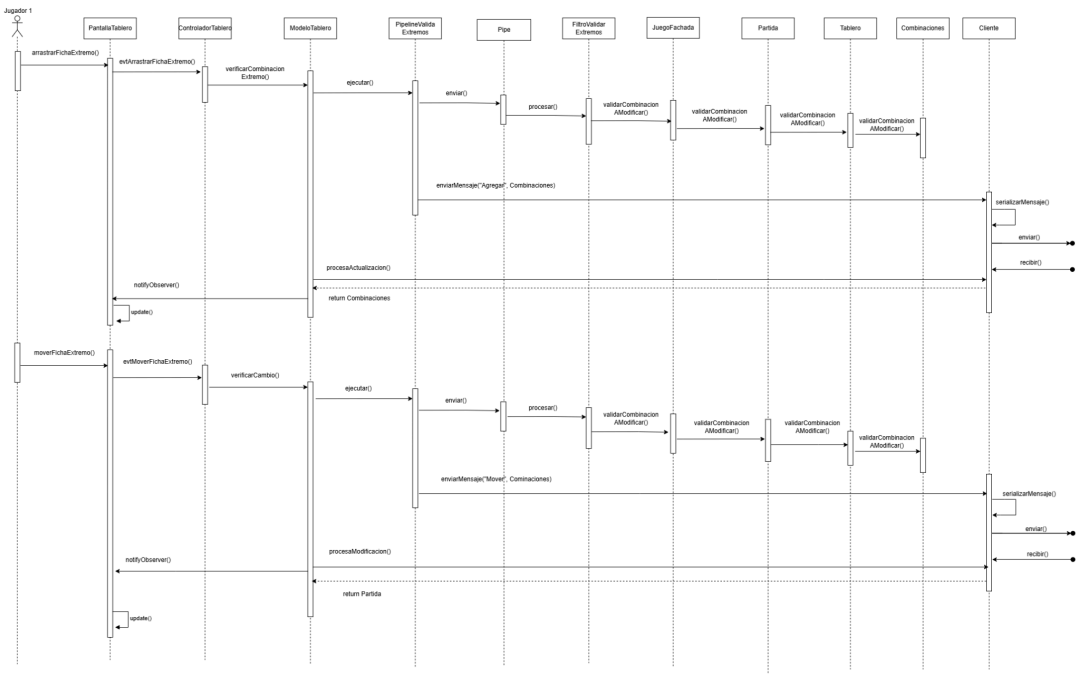


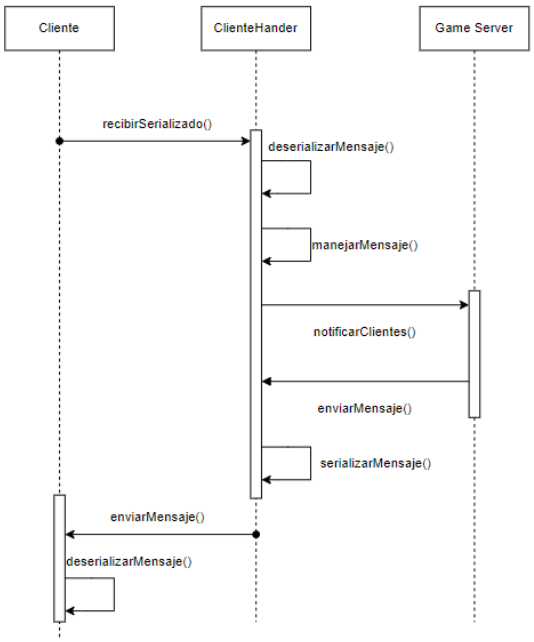

Cliente
ClienteHandler
GameServer
recibirSerializado()
deserializar Mensaje()
manejar Mensaje()
notificarClientes()
enviarMensaje()
serializar()
enviarSerializado()
deserializar Mensaje()

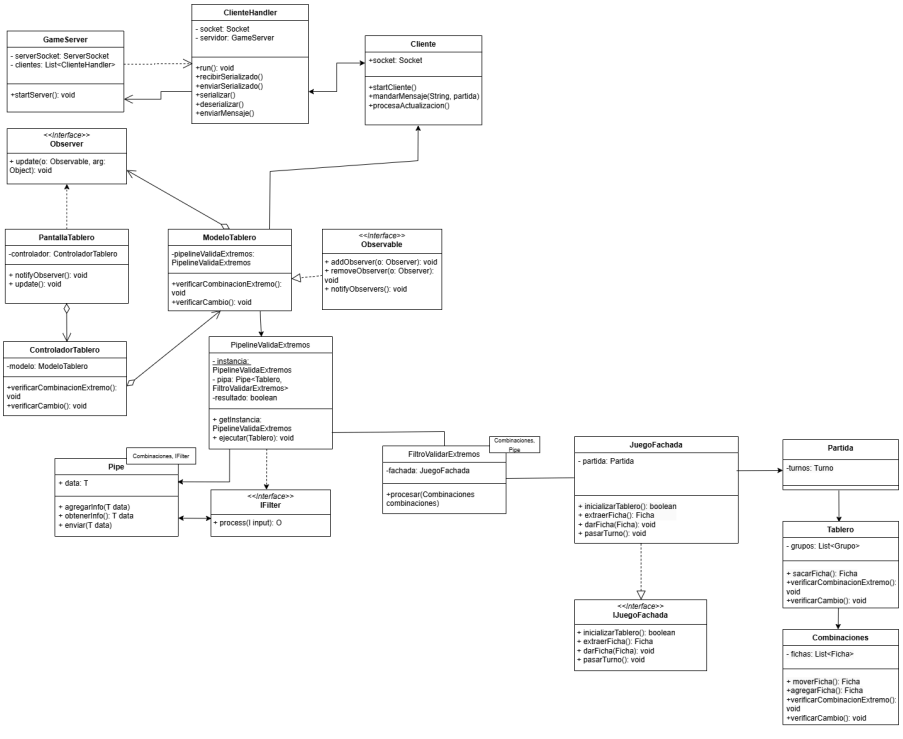

ControladorLobby
modelo: ModeloTablero
controladorTablero: ControladorTablero
+ consultarEstados(Partida): boolean
SinkConsultaEstado
- pipeline: PipelineConsultaEstado
-resultado: boolean
+ getResulado(): boolean
+ setResultado(boolean): void
LobbyFachada
- partida: Partida
+ verificarEstados(): boolean
Jugador
- estado: boolean
+ getEstado(): boolean
PantallaLobby
-controlador: ControladorLobby
+ evtCambioEstado(): void
+ notifyObserver(): void
+ update(): void
ModeloLobby
-pipelineConsultarEstado: PipelineConsultarEstado
+ consultarEstados(): boolean
PipelineConsultaEstado
- instancia: PipelineInicializarTablero
- pipa: Pipe<Partida, FiltroConsultaEstado>
-resultado: boolean
+ getInstancia: PipelineConsultaEstado
+ ejecutar(Partida): boolean
Partida, Pipe
FiltroConsultaEstado
-fachada: LobbyFachada
+procesar(Partida partida)
Partida
+jugadores: List<Jugador>
+turno: int
mazo: List<Ficha>
+ asignarFichas(): boolean
+ asignarTurnos(): boolean
+ consultarEstados(): boolean
<<Interface>>
ModeloObserver
+ update(o: Observable, arg: Object): void
<<Interface>>
Observable
+ addObserver(o: Observer): void
+ removeObserver(o: Observer): void
+ notifyObservers(): void
<<Interface>>
IFilter
+ process(I input): O
Partida, IFilter
Pipe
+ data: T
+ agregarInfo(T data)
+ obtenerInfo(): T data
+ enviar(T data)
JuegoFachada
- partida: Partida
+ asignarFichasAzar(): boolean
+ asignarTurnosAzar(): boolean
PantallaTablero
-controlador: ControladorTablero
+ notifyObserver(): void
+ update(): void
ModeloTablero
-pipelineInicializarTablero: PipelineInicializarTablero
+iniciarPartida (Partida partida)
Partida, Pipe
FiltroAsignarFicha
-fachada: JuegoFachada
+procesar(Partida partida)
Partida, Pipe
FiltroAsignarTurno
-fachada: JuegoFachada
+procesar(Partida partida)
ControladorTablero
modelo: ModeloTablero
+ iniciarPartida(Partida)
PipelineInicializarTablero
- instancia: PipelineInicializarTablero
- pipa: Pipe<Partida,FiltroAsignarFicha>
+ getInstancia: PipelineInicializarTablero
+ ejecutar(Partida): void
Se aplica el patron Strategy en la Pipa y los FIltros
GameServer
- serverSocket: ServerSocket
- clientes: List<ClienteHandler>
+startServer(): void
ClienteHandler
- socket: Socket
- servidor: GameServer
+run(): void
+recibirSerializado()
+enviarSerializado()
+serializar()
+deserializar()
+enviarMensaje()
Cliente
+socket: Socket
+startCliente()
+mandarMensaje(String, partida)
+procesaActualizacion()

6.3.6. CU6: Ejercer turno

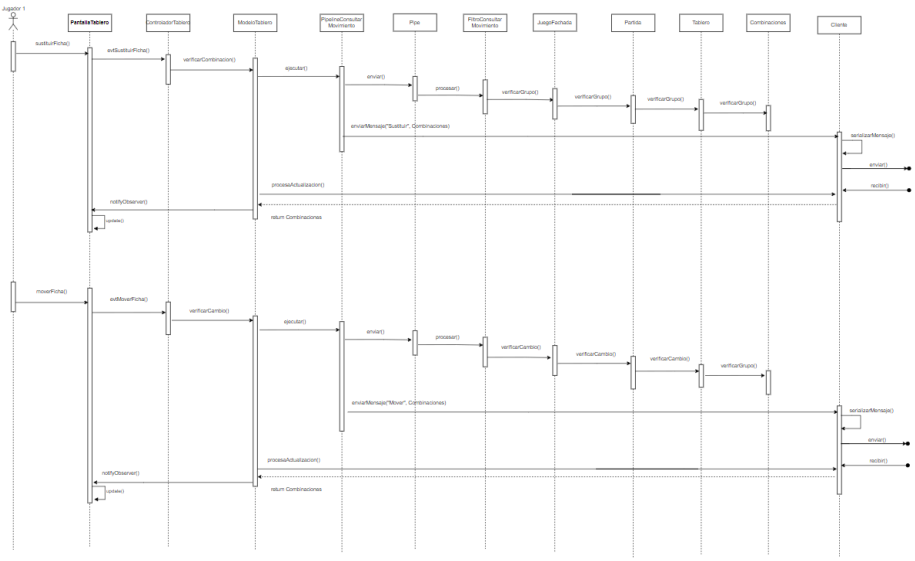
6.3.6.1. Parte 1: Cambiar/Utilizar

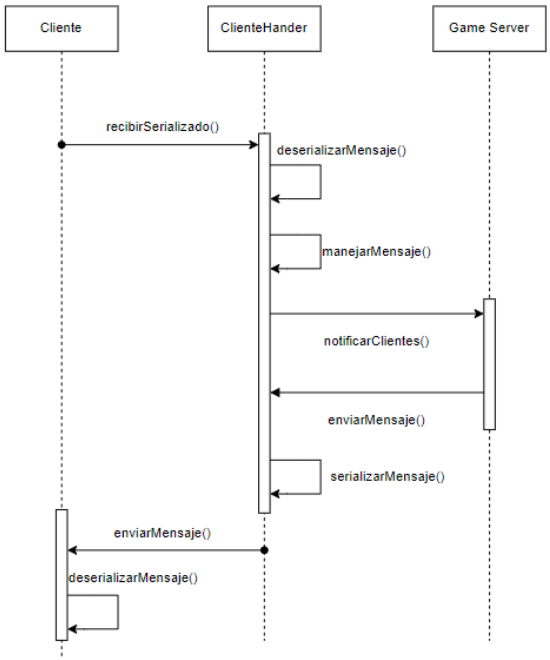


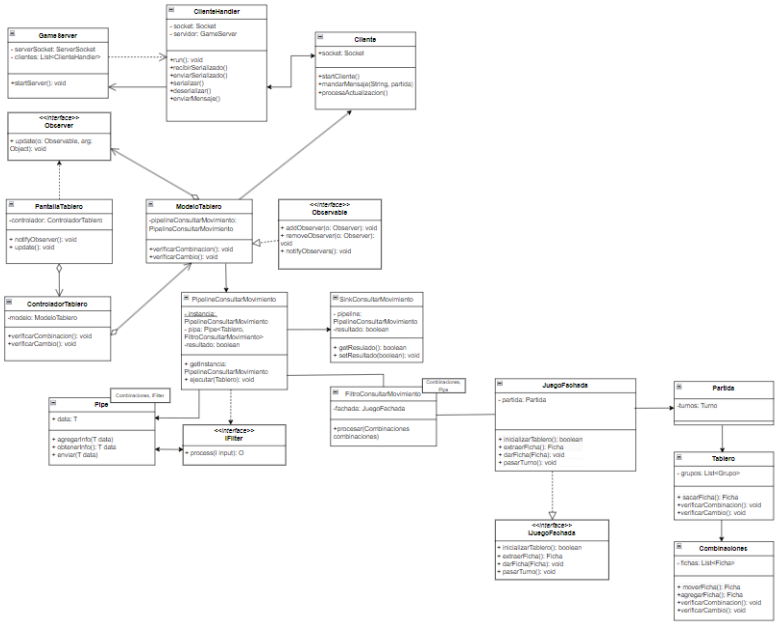




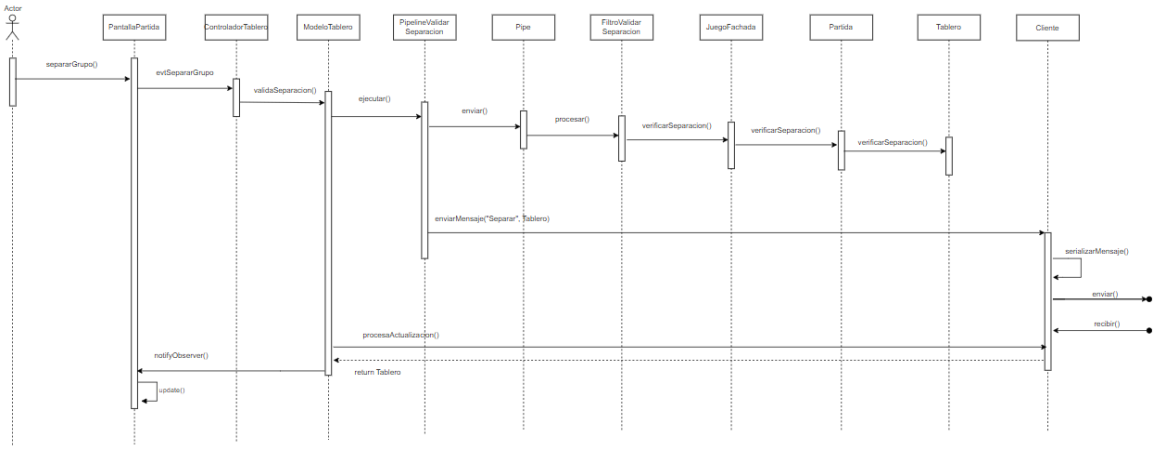
6.3.6.2. Parte 2: Sustituir

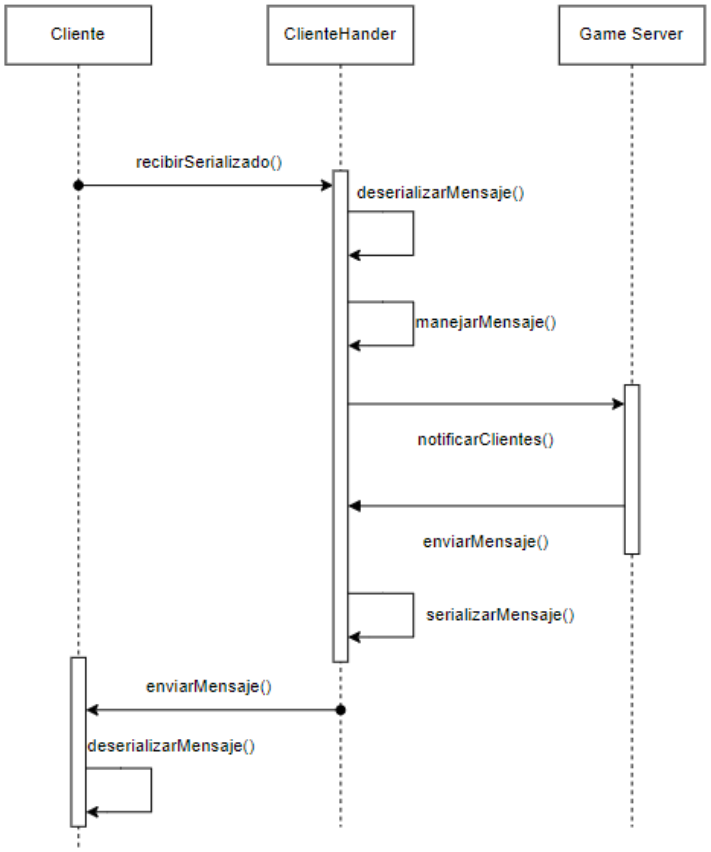

Jugador 1
PantallaTablero
ControladorTablero
ModeloTablero
PoseInvConsultorMovimiento
Pipe
FiltroConsultorMovimiento
JuegoFichada
Partida
Tablero
Combinaciones
Cliente
sustituirFicha()
evtSustituirFicha()
verificarCombinacion()
ejecutar()
enviar()
procesar()
verificarGrupo()
enviarMensaje("Sustituir", Combinaciones)
serializarMensaje()
procesaActualizacion()
notifyObserver()
update()
return Combinaciones
moverFicha()
evtMoverFicha()
verificarCambio()
enviarMensaje("Mover", Combinaciones)

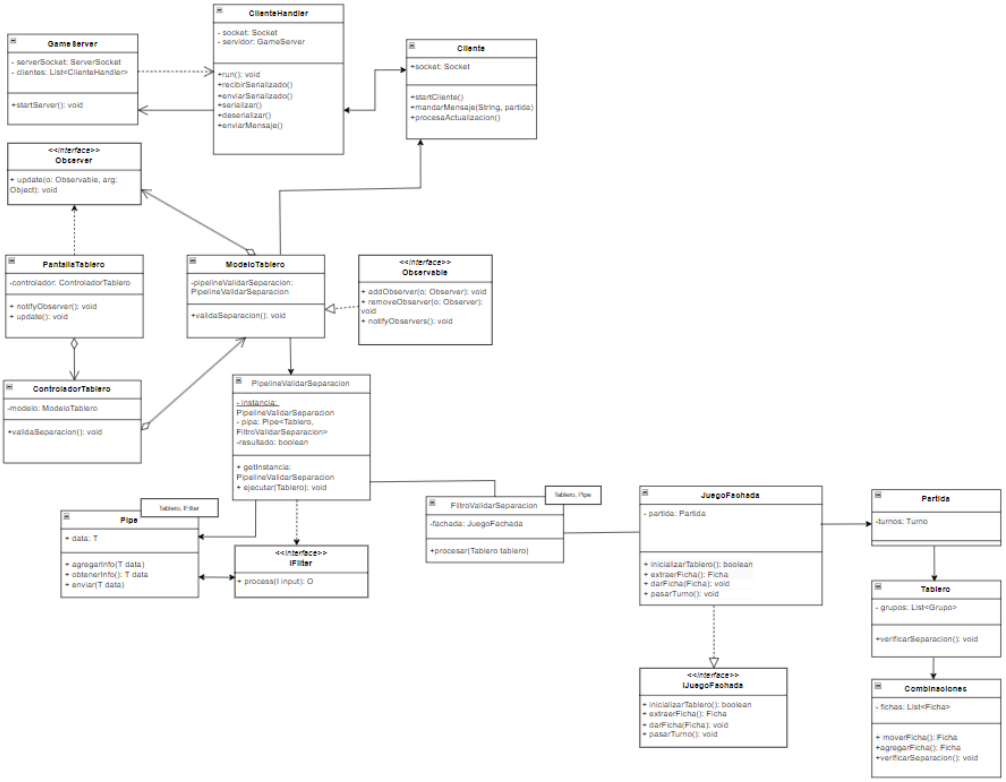

Cliente
ClienteHander
Game Server
recibirSerializado()
deserializarMensaje()
manejarMensaje()
notificarClientes()
enviarMensaje()
serializarMensaje()
enviarMensaje()
deserializarMensaje()



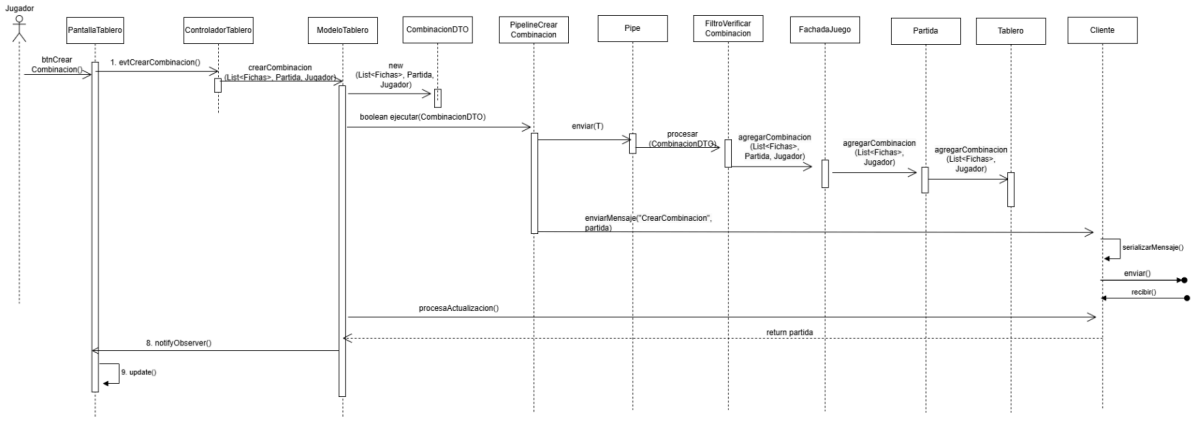
6.3.6.3. Parte 3: Separar

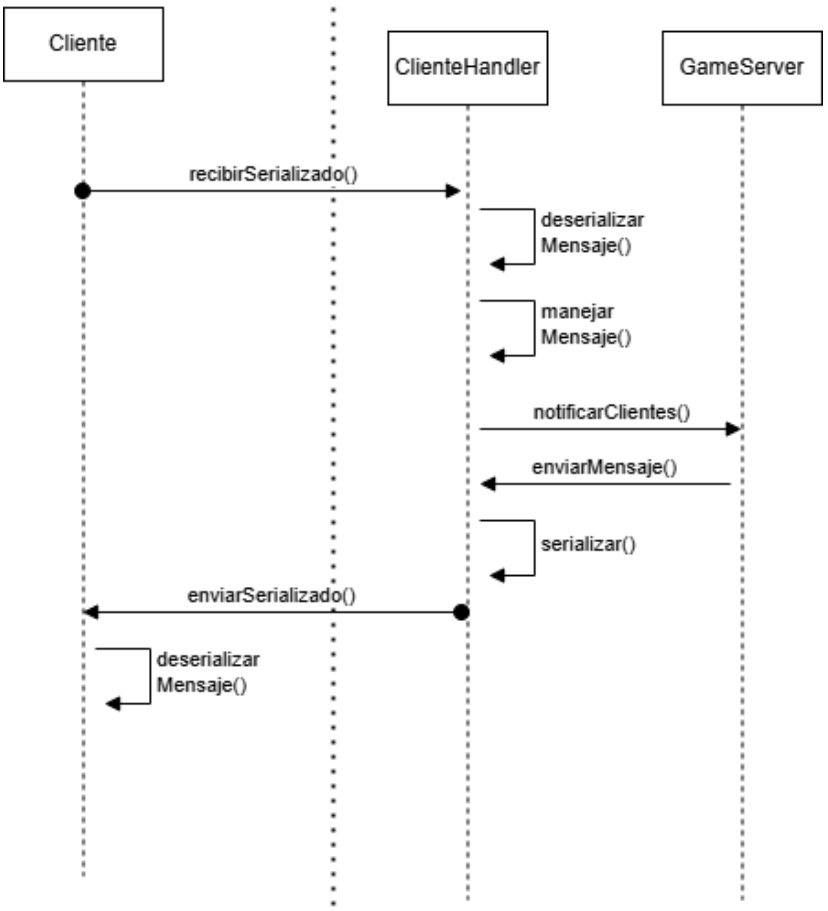


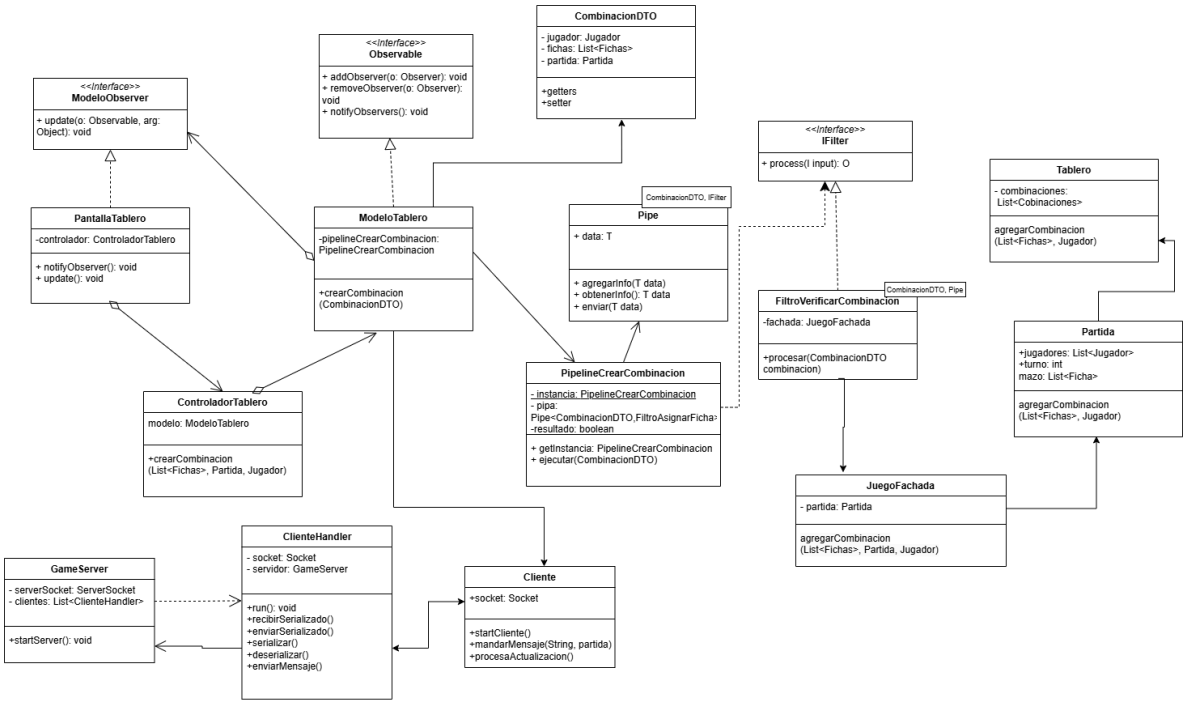

Cliente
ClienteHander
Game Server
recibirSerializado()
deserializarMensaje()
manejarMensaje()
notificarClientes()
enviarMensaje()
serializarMensaje()
enviarMensaje()
deserializarMensaje()



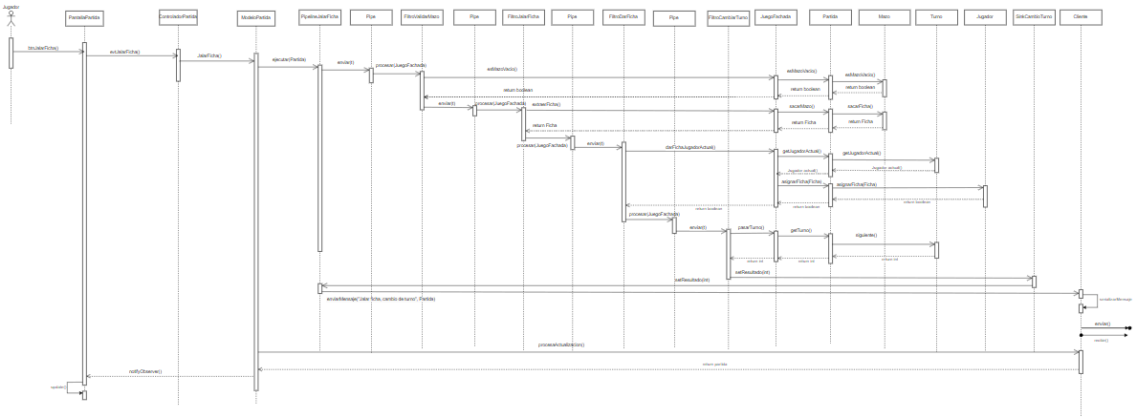
6.3.6.4. Parte 4: Crear combinación

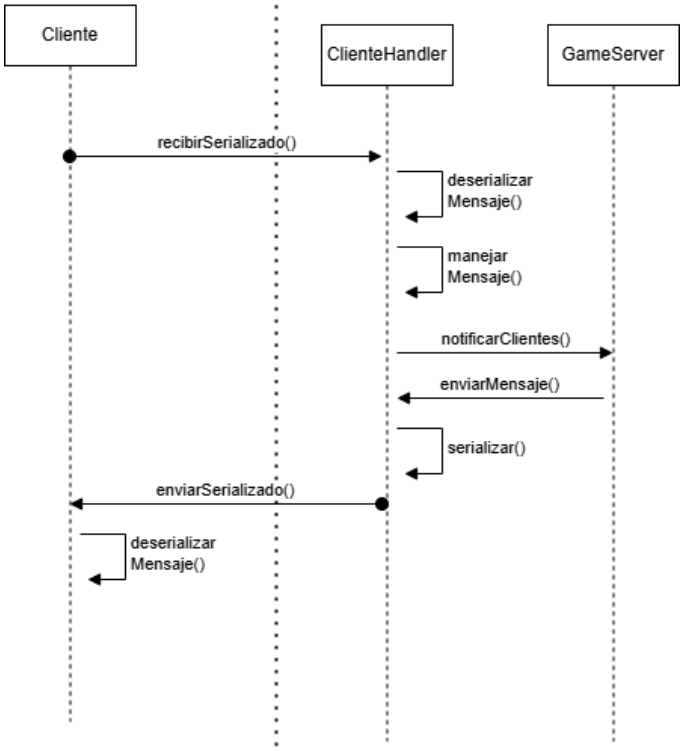


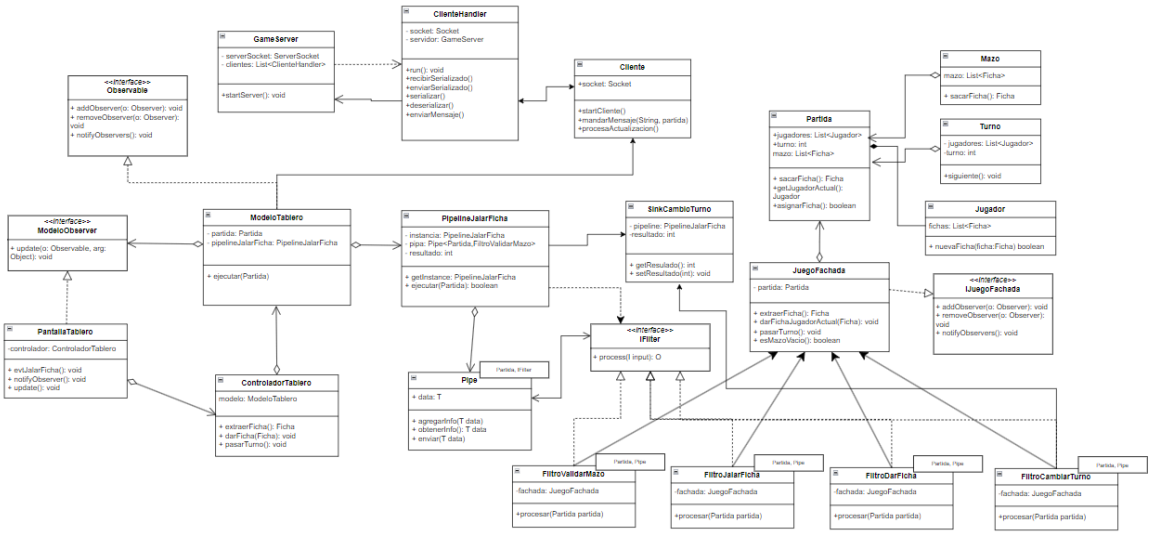

Cliente
ClienteHandler
GameServer
recibirSerializado()
deserializar Mensaje()
manejar Mensaje()
notificarClientes()
enviarMensaje()
serializar()
enviarSerializado()
deserializar Mensaje()


<<Interface>> ModeloObserver
+ update(o: Observable, arg: Object): void
<<Interface>> Observable
+ addObserver(o: Observer): void
+ removeObserver(o: Observer): void
+ notifyObservers(): void
CombinacionDTO
- jugador: Jugador
- fichas: List<Fichas>
- partida: Partida
+getters
+setter
<<Interface>> IFilter
+ process(I input): O
Tablero
- combinaciones: List<Cobinaciones>
agregarCombinacion (List<Fichas>, Jugador)
PantallaTablero
-controlador: ControladorTablero
+ notifyObserver(): void
+ update(): void
ModeloTablero
-pipelineCrearCombinacion: PipelineCrearCombinacion
+crearCombinacion (CombinacionDTO)
CombinacionDTO, IFilter
Pipe
+ data: T
+ agregarInfo(T data)
+ obtenerInfo(): T data
+ enviar(T data)
CombinacionDTO, Pipe
FiltroVerificarCombinacion
-fachada: JuegoFachada
+procesar(CombinacionDTO combinacion)
Partida
+jugadores: List<Jugador>
+turno: int
mazo: List<Ficha>
agregarCombinacion (List<Fichas>, Jugador)
ControladorTablero
modelo: ModeloTablero
+crearCombinacion (List<Fichas>, Partida, Jugador)
PipelineCrearCombinacion
- instancia: PipelineCrearCombinacion
- pipa: Pipe<CombinacionDTO,FiltroAsignarFicha>
-resultado: boolean
+ getInstancia: PipelineCrearCombinacion
+ ejecutar(CombinacionDTO)
JuegoFachada
- partida: Partida
agregarCombinacion (List<Fichas>, Partida, Jugador)
GameServer
- serverSocket: ServerSocket
- clientes: List<ClienteHandler>
+startServer(): void
ClienteHandler
- socket: Socket
- servidor: GameServer
+run(): void
+recibirSerializado()
+enviarSerializado()
+serializar()
+deserializar()
+enviarMensaje()
Cliente
+socket: Socket
+startCliente()
+mandarMensaje(String, partida)
+procesaActualizacion()

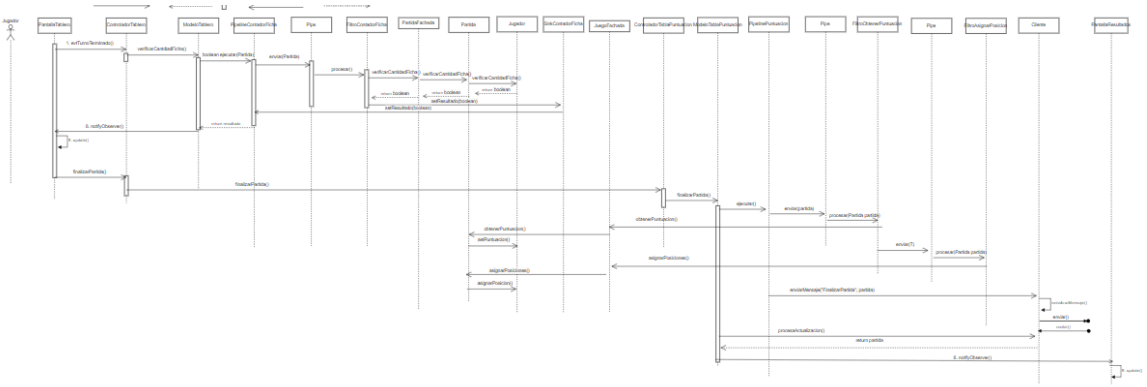
6.3.6.5. Parte 5: Jalar ficha

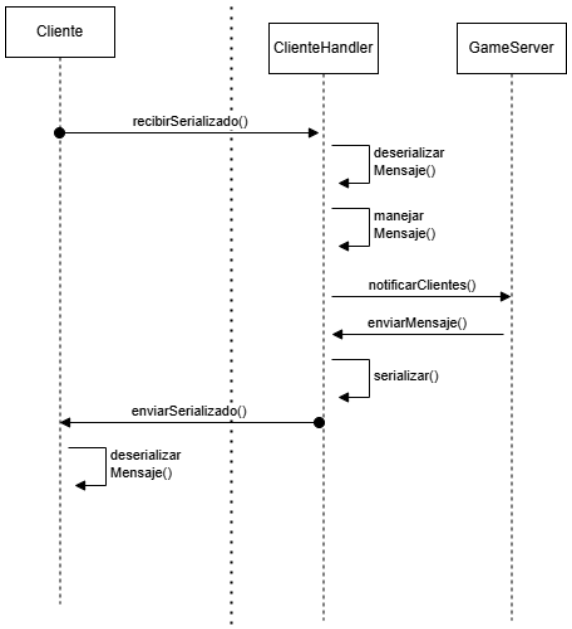


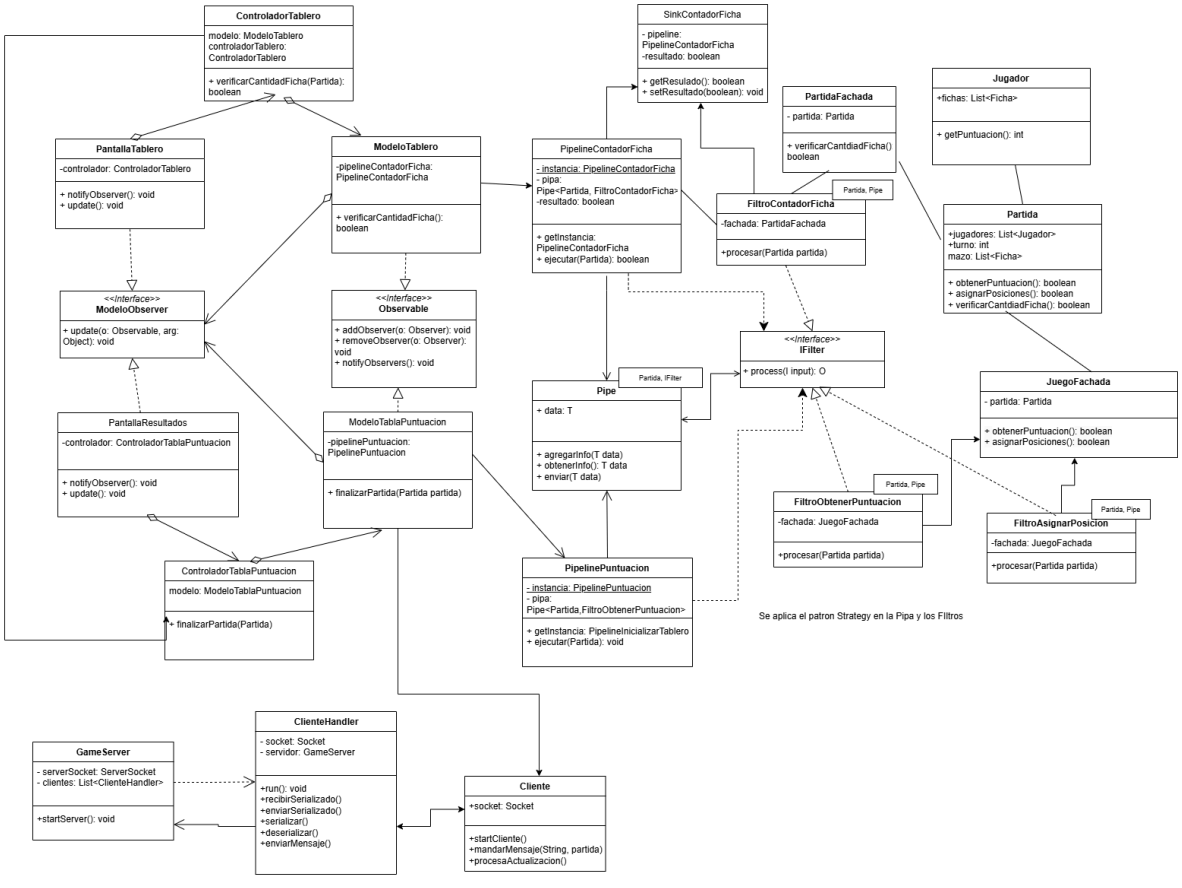




6.3.7. CU7: Terminar partida




Cliente
ClienteHandler
GameServer
recibirSerializado()
deserializar Mensaje()
manejar Mensaje()
notificarClientes()
enviarMensaje()
serializar()
enviarSerializado()
deserializar Mensaje()

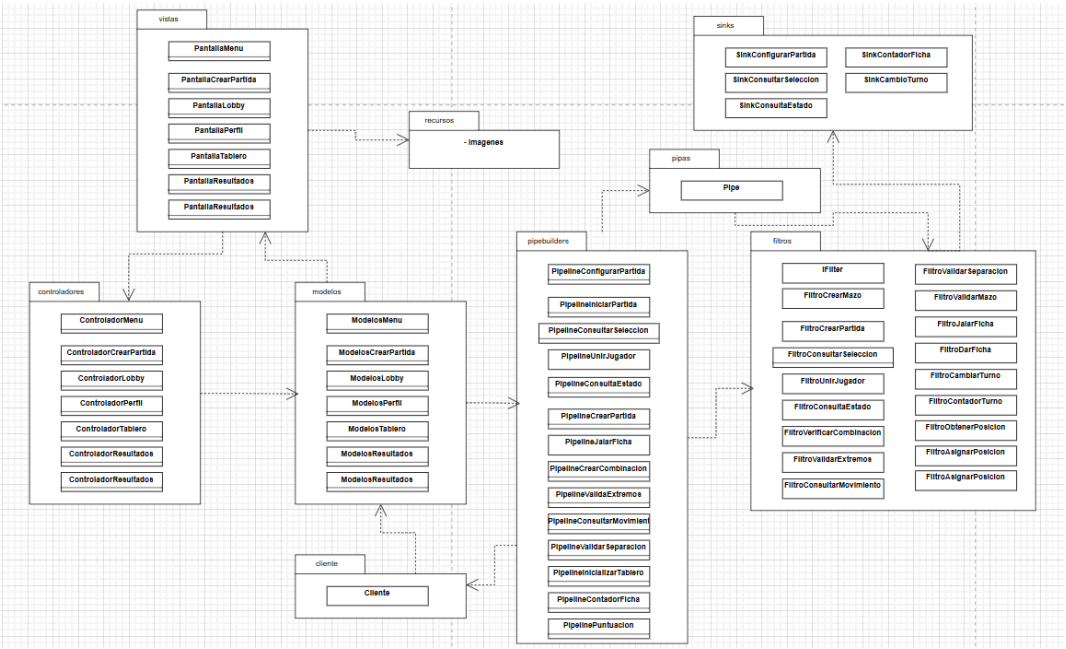

ControladorTablero
modelo: ModeloTablero
controladorTablero: ControladorTablero
+ verificarCantidadFicha(Partida): boolean
SinkContadorFicha
- pipeline: PipelineContadorFicha
-resultado: boolean
+ getResulado(): boolean
+ setResultado(boolean): void
PartidaFachada
- partida: Partida
+ verificarCantdiadFicha() boolean
Jugador
+fichas: List<Ficha>
+ getPuntuacion(): int
PantallaTablero
-controlador: ControladorTablero
+ notifyObserver(): void
+ update(): void
ModeloTablero
-pipelineContadorFicha: PipelineContadorFicha
+ verificarCantidadFicha(): boolean
PipelineContadorFicha
- instancia: PipelineContadorFicha
- pipa: Pipe<Partida, FiltroContadorFicha>
-resultado: boolean
+ getInstancia: PipelineContadorFicha
+ ejecutar(Partida): boolean
Partida, Pipe
FiltroContadorFicha
-fachada: PartidaFachada
+procesar(Partida partida)
Partida
+jugadores: List<Jugador>
+turno: int
mazo: List<Ficha>
+ obtenerPuntuacion(): boolean
+ asignarPosiciones(): boolean
+ verificarCantdiadFicha(): boolean
<<Interface>>
ModeloObserver
+ update(o: Observable, arg: Object): void
<<Interface>>
Observable
+ addObserver(o: Observer): void
+ removeObserver(o: Observer): void
+ notifyObservers(): void
<<Interface>>
IFilter
+ process(I input): O
Partida, IFilter
Pipe
+ data: T
+ agregarInfo(T data)
+ obtenerInfo(): T data
+ enviar(T data)
JuegoFachada
- partida: Partida
+ obtenerPuntuacion(): boolean
+ asignarPosiciones(): boolean
PantallaResultados
-controlador: ControladorTablaPuntuacion
+ notifyObserver(): void
+ update(): void
ModeloTablaPuntuacion
-pipelinePuntuacion: PipelinePuntuacion
+ finalizarPartida(Partida partida)
Partida, Pipe
FiltroObtenerPuntuacion
-fachada: JuegoFachada
+procesar(Partida partida)
Partida, Pipe
FiltroAsignarPosicion
-fachada: JuegoFachada
+procesar(Partida partida)
ControladorTablaPuntuacion
modelo: ModeloTablaPuntuacion
+ finalizarPartida(Partida)
PipelinePuntuacion
- instancia: PipelinePuntuacion
- pipa: Pipe<Partida,FiltroObtenerPuntuacion>
+ getInstancia: PipelineInicializarTablero
+ ejecutar(Partida): void
Se aplica el patron Strategy en la Pipa y los Filtros
GameServer
- serverSocket: ServerSocket
- clientes: List<ClienteHandler>
+startServer(): void
ClienteHandler
- socket: Socket
- servidor: GameServer
+run(): void
+recibirSerializado()
+enviarSerializado()
+serializar()
+deserializar()
+enviarMensaje()
Cliente
+socket: Socket
+startCliente()
+mandarMensaje(String, partida)
+procesaActualizacion()

7. Vista de despliegue

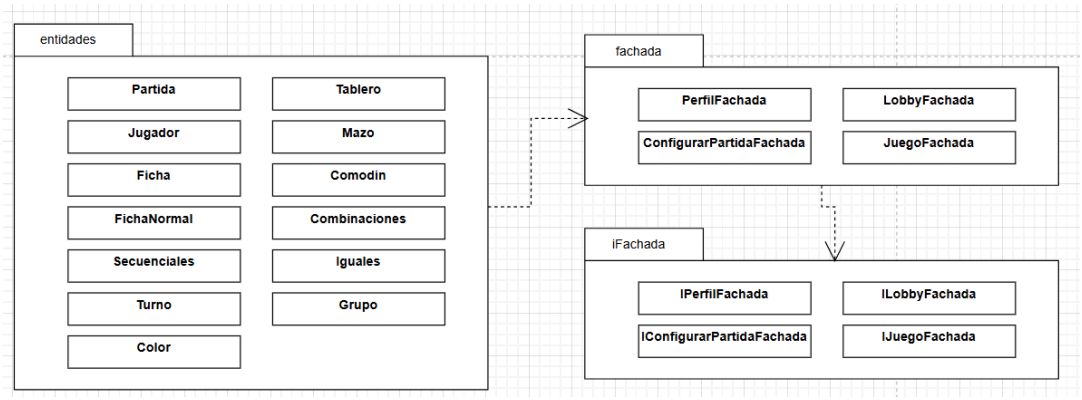
7.1. Diagrama de paquetes

Esta organización sigue principios de diseño como la separación de responsabilidades y modularidad. La agrupación de elementos en secciones específicas (vistas, controladores, modelos, pipeline builders, filtros, pipes y sinks) permite que cada componente cumpla una función específica y bien delimitada dentro de la arquitectura. El paquete de entidades concentra la lógica del negocio, asegurando cohesión y reutilización. El paquete de fachada simplifica el acceso al sistema, actuando como intermediario entre las entidades y los clientes externos. Por último, el paquete de interfaces de fachada define contratos para garantizar flexibilidad y desacoplamiento. Esta organización facilita el mantenimiento, la escalabilidad y la claridad en el diseño del sistema.

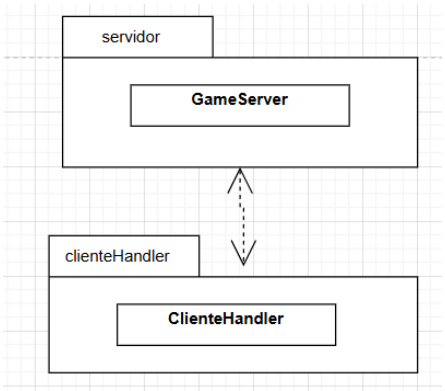
7.1.1. Componente GUI



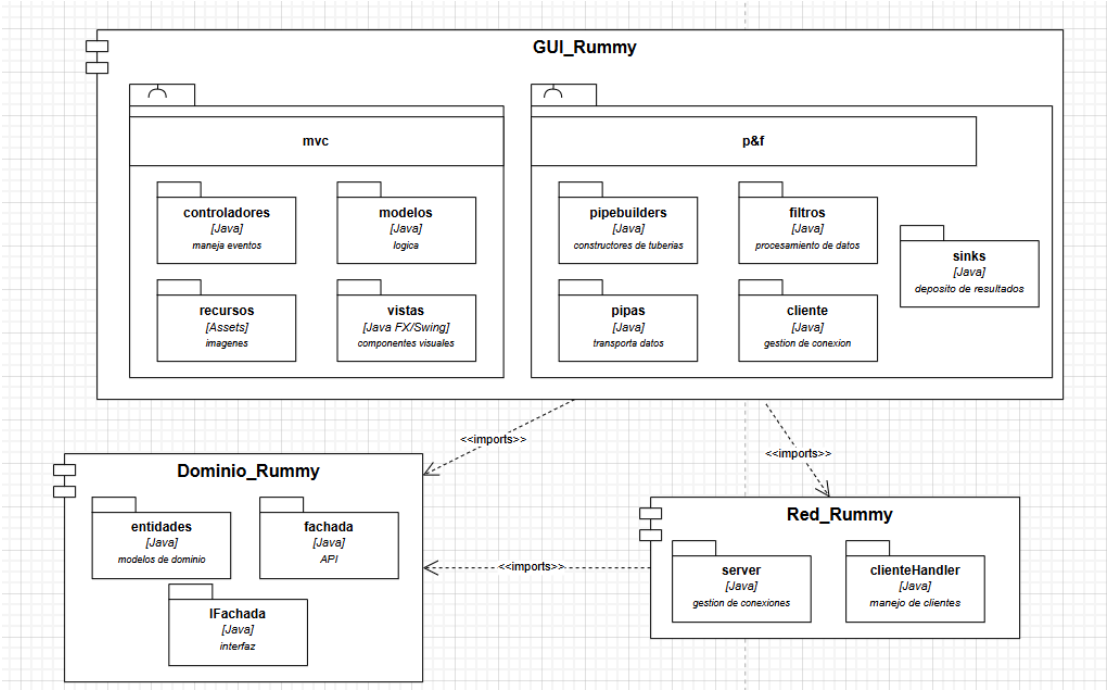
7.1.2. Componente Dominio



7.1.3. Componente Red

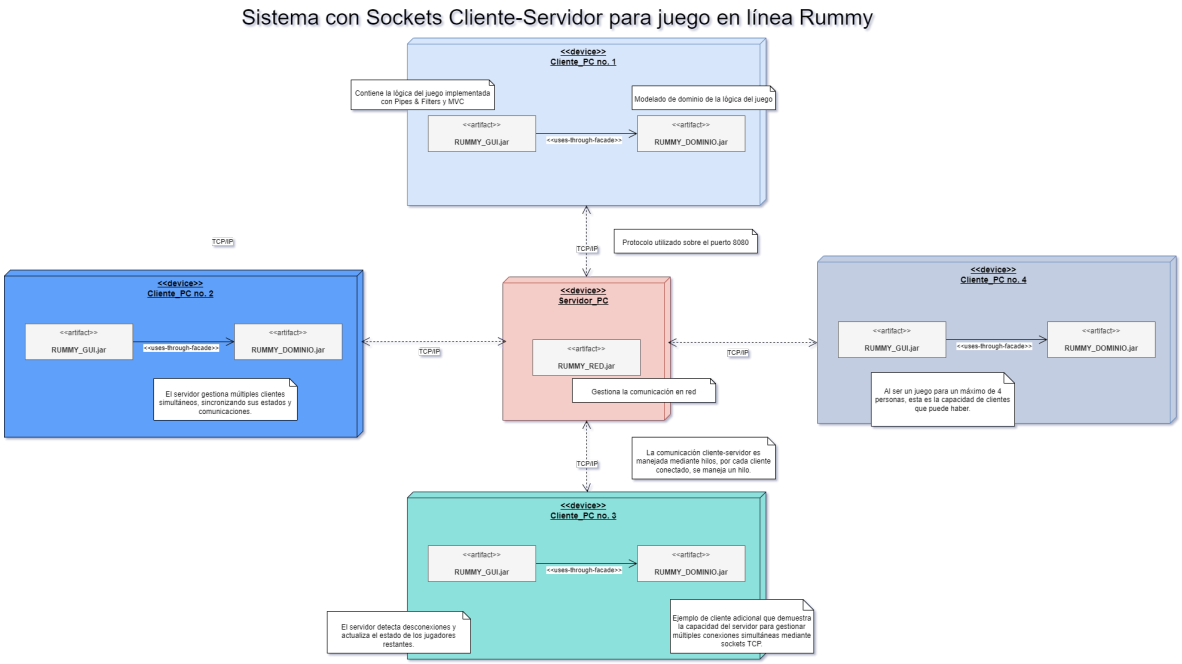


7.2. Diagrama de componentes



8. Vista física

8.1. Diagrama de despliegue



Descripción

- **Nodo 1:** Servidor_PC: Dispositivo el cual lleva consigo el proyecto del servidor, permitiendo el intercambio de información entre los nodos por medio del protocolo TCP utilizando sockets. Utiliza el proceso de serialización de objetos para enviar y recibir información con los nodos cliente. El servidor es capaz de gestionar múltiples clientes simultáneos, sincronizando sus estados y comunicaciones. Permite una multiplicidad de 0 a 4 nodos, sin embargo, para llevar a cabo el juego, se necesitan preferentemente un mínimo de 2 dispositivos cliente. Utiliza el puerto 4444 para sincronizar los nodos cliente.
- **Nodos 2-5:** Cliente_PC: Dispositivos encargados de llevar la comunicación del cliente, cargar con la lógica del juego y el apartado visual del mismo por medio del patrón de diseño Facade. Llevan consigo el cliente encargado de establecer una conexión bidireccional con el servidor.
- **Conexión**: Por medio del protocolo TCP e hilos, es aquel apartado encargado del intercambio de información al mandar objetos serializados.

Requerimientos de instalación:

- Puerto 4444 disponible para uso en todos los dispositivos.
- Un dispositivo debe utilizarse para mantener el server en funcionamiento.
- Los demás dispositivos llevarán consigo los archivos .jar responsables de la lógica y la presentación del juego.