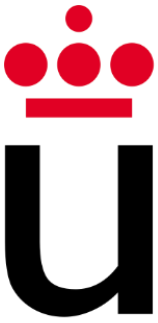


Universidad Rey Juan Carlos

PRÁCTICA ESTRATEGO

MEMORIA

EVOLUCIÓN Y MANTENIMIENTO DEL SOFTWARE 2024 / 2025

19 de mayo de 2025

Equipo 11:

Jorge Andrés Echevarría

Víctor Bartolomé Letosa

Iván Gutiérrez González

Arturo Enrique Gutiérrez Mirandona

Índice

1. Introducción

Esta práctica está dentro de la materia de Evolución y Adaptación de Software, asignada al periodo 2024/2025 del Grado en Ingeniería del Software. Su objetivo principal es implementar conceptos esenciales de mantenimiento y evolución de sistemas de software, a través de la reingeniería de un proyecto ya existente. Para ello, se ha suministrado como fundamento una implementación en Java del juego de estrategia Stratego, originalmente desarrollada en 2014.

La meta principal es convertir este software legado en una versión más fiable, extensible y en concordancia con los estándares de desarrollo actuales. Esto abarca tanto labores de actualización del código como la incorporación de nuevas características, conforme a una propuesta de progreso elaborada por el propio equipo. Esta práctica tiene como objetivo no solo incrementar la calidad técnica del software original, sino también aplicar técnicas de diseño, análisis de arquitectura y gestión de versiones colaborativas, componentes cruciales en el ciclo de vida del software profesional.

Antes de empezar a explicar los cambios fue necesario realizar un diagrama de clases UML, para entender la estructura del código original. Este diagrama, dividido en imágenes según las carpetas principales, fue extraído gracias a la reingeniería con StarUML a partir del código. Véase el Anexo 1 o visite el archivo *MainOld.svj* dentro de la carpeta "*documentation"* en la raíz del proyecto.

2. Mantenimiento

Esta primera parte de la práctica la hemos dividido en cambios generales, relacionados con la estructura del proyecto, y la comprobación de la correcta aplicación de los principios SOLID, explicados en clase, necesarios para aumentar la mantenibilidad del sistema, ayudando tanto a la claridad del código como a futuras mejoras que se hagan sobre este.

2.1. Cambios generales

El primer cambio no es técnicamente uno si no, ya que primero necesitábamos comprobar la funcionalidad del sistema, por lo que añadimos la librería de JavaFX correspondiente con la versión de Java más moderna (23), asegurando su compatibilidad con nuevas y antiguas versiones de este lenguaje.

Posteriormente, como lo ejecutamos todos los miembros del equipo en el IDE VSCode, creamos la carpeta .vscode con sus correspondientes *launch.json* y *settings.json*, encargados de ejecutar el servidor y el cliente, y usar la librería añadida, mencionada anteriormente.

Una vez teniendo el código funcional decidimos realizar una primera reestructuración del código dividida en dos pasos.

2.1.1. Creación del proyecto con la estructura en Maven

Para hacer este proyecto compatible con Maven tuvimos que crear un pom.xml en la rama padre del repositorio y tuvimos que adecuar los ficheros y recursos en la estructura propia de Maven, dejándolos de la siguiente manera:

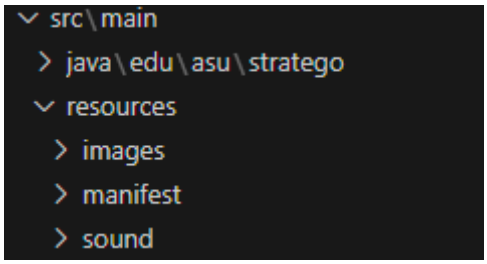


Ilustración 1: Estructura proyecto Maven

2.1.2. División de los archivos

Con el objetivo en mente de facilitar la manipulación en general del código se propuso la idea de dividir el código en dos directorios (Cliente y Servidor). Esto no solo facilitaría nuestro entendimiento de los flujos de datos en el sistema, sino que nos permitiría discernir posteriormente con mayor facilidad las relaciones entre las clases y el comportamiento de cada directorio como un individuo separado del otro. Esta propuesta fue de las primeras a realizar debido a la naturaleza de sus cambios. En retrospectiva podemos asegurar que la idea fue una gran ayuda para facilitar en gran medida la refactorización posterior.

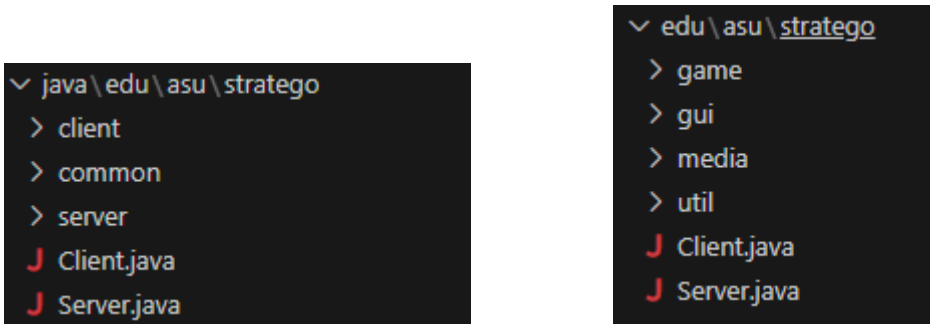


Ilustración 2: Comparación entre el antes (derecha) y el después (izquierda).

2.2. Principios SOLID

A continuación, explicaremos como realizamos la aplicación de los diferentes principios. En caso de llevar a cabo un patrón y/o modificación que aplique a varios principios, se explica en el principio que hemos considerado más beneficiado por esa mejora.

2.2.1. Principio de Responsabilidad Única (SRP)

El principio de responsabilidad única fue el primer principio abordado por el grupo. Esto se debe a que pretendíamos asentar una base estable sobre la que luego aplicar los demás principios sin ir solapándonos en el trabajo.

En específico el procedimiento comenzó con realizar un reconocimiento inicial por las clases en búsqueda de clases que contuviesen mucho código en ellas, porque puede ser un indicador de que presenta demasiadas responsabilidades. Tras ese reconocimiento inicial se realizó un segundo barrido en búsqueda de las clases que pudiesen haberse escapado en la iteración anterior.

Tras anotar todos los puntos a atacar, se comenzó con el trabajo. Las clases que han supuesto la mayor inversión de trabajo han sido: *ClientGameManager* y *ServerGameManager*, debido a que ambas clases eran no solo de un tamaño importante sino también presentaban más de 4 responsabilidades distintas en ellas. Entre las responsabilidades principales del *ServerGameManager* encontramos la gestión de las comunicaciones, de los jugadores, del tablero, del juego además de otras como el cálculo de movimientos válidos y etc. Con relación a *ClientGameManager* las responsabilidades que acumulaba tenían correspondencia con la lógica del juego, el manejo de cierta GUI, comunicaciones con servidor, etc. entre otras.

En resumen, era un punto débil del sistema que debía ser reestructurado. Por lo que en caso del *ServerGameManager* se separó en cuatro clases que harían cumplir este principio. *BoardManager*, encargado de la gestión del tablero y piezas; *ClientCommunicationManager*, encargado de las comunicaciones con los clientes; *GameFlowManager*, encargado del flujo del juego y por último *MoveValidator*, encargado de calcular los movimientos válidos.

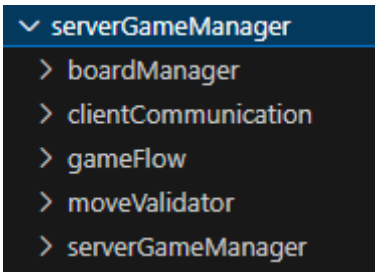


Ilustración 3: Resultado de la separación de responsabilidades en ServerGameManager

En el caso del *ClientGameManager* se separó en *ServerConnection* (manejo de conexiones con el servidor), *GameLogic* (lógica del juego), *GameUIManager* (gestión de la interacción con la GUI) y *ClientGameManager* (coordinación de operaciones).

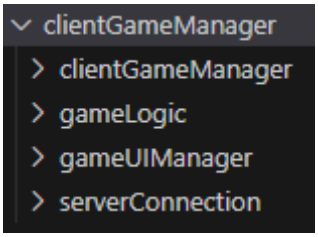


Ilustración 4: Resultado de la separación de responsabilidades en ClientGameManager

Adicionalmente hubo otros cambios de menor escala entre los que se encuentran la separación de la clase *BoardSquareEventPane* en lógica y eventos, debido a que

presentaba innecesariamente una gran cantidad de validaciones lógicas, por lo que se optó por separarlas a otra clase culla única responsabilidad es las validaciones lógicas.

En última instancia, es cierto que persisten algunas clases cuya responsabilidad puede interpretarse de forma ambigua, como es el ejemplo de *BoardManager* (esta clase controla tanto el tablero como las piezas en el servidor). Esto ocurre porque, en ciertos casos, las funcionalidades que gestionan están estrechamente relacionadas o no se delimitan con claridad. En estas situaciones, definir con precisión qué constituye una única responsabilidad puede ser subjetivo y depende, en parte, de decisiones de diseño y del contexto de las circunstancias.

2.2.2. Principio Abierto-Cerrado (OCP)

El cumplimiento del principio de abierto/cerrado se llevó a cabo con dos cambios principales en el diseño del código. El primero fue reorganizar cómo se trataban las piezas del juego. Originalmente, se hacía uso de enumerado que hacía demasiadas cosas: creaba instancias de piezas, tenía lógica de combate y gestionaba un constructor para los tipos de pieza. Esto hacía muy complicado añadir nuevos tipos de pieza. Para solucionarlo, se creó una clase por cada tipo de pieza que implementa la interfaz *PieceTypeInterface*, donde cada una define sus atributos y su propio método de ataque. Además, se modificó como se seleccionaban las imágenes correspondientes a cada tipo de pieza, originalmente era un bloque if-else inmenso para todas las piezas y ahora cada pieza tiene su propio método de selección de Sprite. Como muchos bucles dependían del antiguo enumerado puesto que este inicializaba las piezas, se creó la clase *PieceManager*, que centraliza la creación y acceso a las piezas mediante una lista estática.

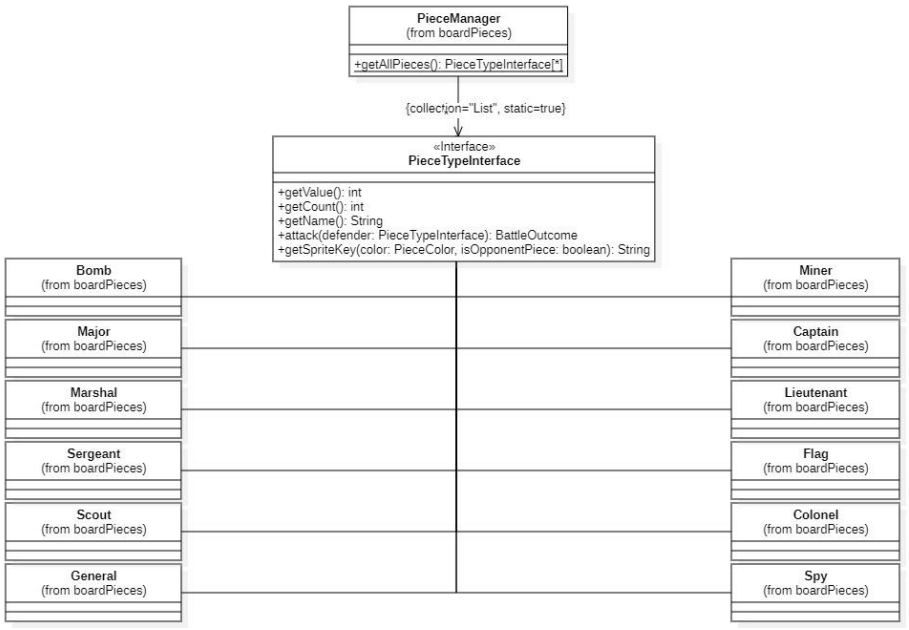


Ilustración 5: Resultado después de separar el enumerado PieceType en tipos de pieza individuales

El segundo cambio fue en el método *playGame* de la clase *GameFlowManager*, que tenía bloques if bastantes densos para decidir qué hacer según el resultado del combate (victoria, derrota o empate). Para organizar mejor esa lógica, se aplicó el patrón **Strategy**, usando polimorfismo. Se creó la interfaz *BattleResolutionStrategy* con un método resolve, e implementaciones específicas para cada caso: *WinStrategy*, *LoseStrategy* y *DrawStrategy*. Luego, se creó el *BattleStrategyManager*, que guarda un mapa con las estrategias y se encarga de ejecutar la que corresponda según el resultado del combate. Gracias a esto, se puede añadir una nueva resolución sin tocar el código existente, cumpliendo así con el principio de abierto/cerrado.

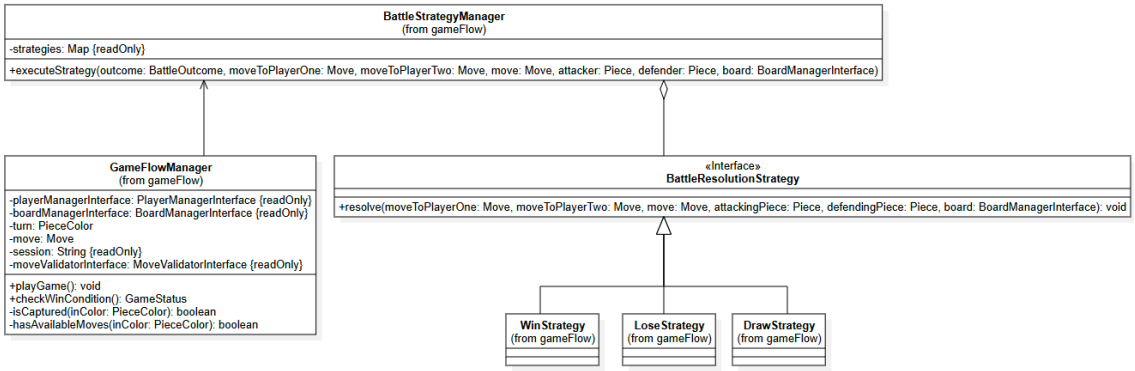


Ilustración 6: Resultado de aplicar el patrón Strategy en el resultado de las batallas

2.2.3. Principio de Sustitución de Liskov (LSP)

Este principio dice que la herencia ha de garantizar que cualquier propiedad que sea cierta para los objetos supertipo también lo sea para los objetos subtipo es decir que las clases hijas deben poder sustituir a las clases padres sin alterar el funcionamiento del programa.

Para ello comprobamos las clases abstractas y las interfaces de nuestro programa.

- Interfaces:

En nuestro programa hay un total de 22 interfaces y por cada una de ellas comprobamos que las clases que las implementan usan todos sus métodos y de manera coherente, que no se lanzan excepciones donde no se esperan y que se respetan los contratos entre las clases y las interfaces.

Las comprobaciones son todas correctas por lo que no realizamos ninguna modificación en el código actual, es decir podríamos sustituir la clase base por la hija y realizar las mismas funcionalidades y eso no rompería el funcionamiento del programa.

- Clases abstractas:

Contamos con la clase abstracta BaseScene.java implementada en clases como BoardScene.java, ConnectionScene.java, FinalScene.java y WaitingScene.java.

Estas clases cumplen el contrato con BaseScene.java ya que implementan su método *buildScene*, inicializan el atributo *scene*, invoca *buildScene* en el constructor y la función *getScene* puede usarse sin problema, lo que hace a *BaseScene.java* totalmente sustituible

por cualquiera de las clases que la implementan por lo que cumple el principio de sustitución den Liskov.

Por último, aunque esta acción puede estar más relacionada con otros principios SOLID, hemos utilizado el patrón de diseño **Factory Method** con el fin de desacoplar la creación de objetos de su uso, creando una clase fabrica tanto para el Server como para el Client.

Ilustración 7: Imagen de las clases fabrica del Client y el Server

Logrando con este patrón varios beneficios como desacoplamiento de la lógica de creación de objetos del código que los utiliza y en caso de modificación solo serán necesario cambiar el código de la clase fabrica y no del Server o Client.

Este patrón también nos permite reutilizar dichas clases en otros contextos para crear sesiones y clientes del juego diferentes, y probar la lógica de creación de sesiones y clientes de forma aislada.

2.2.4. Principio de Separación de la Interfaz (ISP)

El principio de segregación de interfaces establece que *"los clientes no deberían estar forzados a depender de interfaces que no utilizan"*. Es decir, se busca evitar interfaces demasiado grandes o genéricas que obliguen a las clases que las implementan a depender de métodos innecesarios.

Como se vio en el diagrama de clases inicial, el proyecto no tenía ninguna interfaz implementada, solo se han creado posteriormente para cumplir el principio de Inversión de Dependencias especificado a continuación.

Como bien se ha mencionado en el principio anterior, Sustitución de Liskov, todas las interfaces implementadas por este equipo cumplen perfectamente su función, es decir, no se detectaron casos en los que una clase implementara métodos no relevantes para su funcionalidad, lo cual evidencia que el diseño actual ya respeta el principio de segregación de interfaces.

2.2.5. Principio de Inversión de Dependencias (DIP)

Para el cumplimiento de este principio se han creado interfaces para todas las clases que tuvieran lógica dentro de ellas, es decir, a las clases formadas por atributos y cuyos únicos métodos sean simplemente getters y setters no se les han creado una interfaz.

Para todas las demás si se les ha creado una interfaz con los métodos públicos en ellas dejando los atributos, métodos estáticos y las implementaciones de los métodos públicos para las implementaciones concretas. Facilitando el intercambio de la clase implementada por otra cualquiera, que cumpla con la interfaz, en caso de ser necesario para una futura evolución del código. Por ejemplo, se creó una interfaz *ClientGameInterface*, implementada por la clase *ClientGameManager*, usando la interfaz dentro de la clase

Client, así como todas las subclases creadas a partir de dividirla en el principio SRP (*GameLogic, GameUIManager* y *ServerConnetion*). Este enfoque se ha seguido en clases como *ServerGameManager, SetupPieces, SetupPanel, etc.*

Además, se han separado las clases incrustadas en otras separándolas en diferentes ficheros y creando las interfaces pertinentes. Por ejemplo, las clases *SelectPiece* y *UpdateReadyButton* dentro de la clase *SetupPieces.*

Después de estos cambios básicos dentro de este principio se han aplicado diferentes patrones para solucionar conexiones circulares, desacoplamiento de código, etc.

En primer lugar, se aplicó el patrón **Mediator** para unificar comunicación entre *ServerGameManager*, *BoardManager, GameFlow,* y *PlayerManager*, con la clase *ClientCommunicationManager*. Dejando una estructura bastante similar a la siguiente:

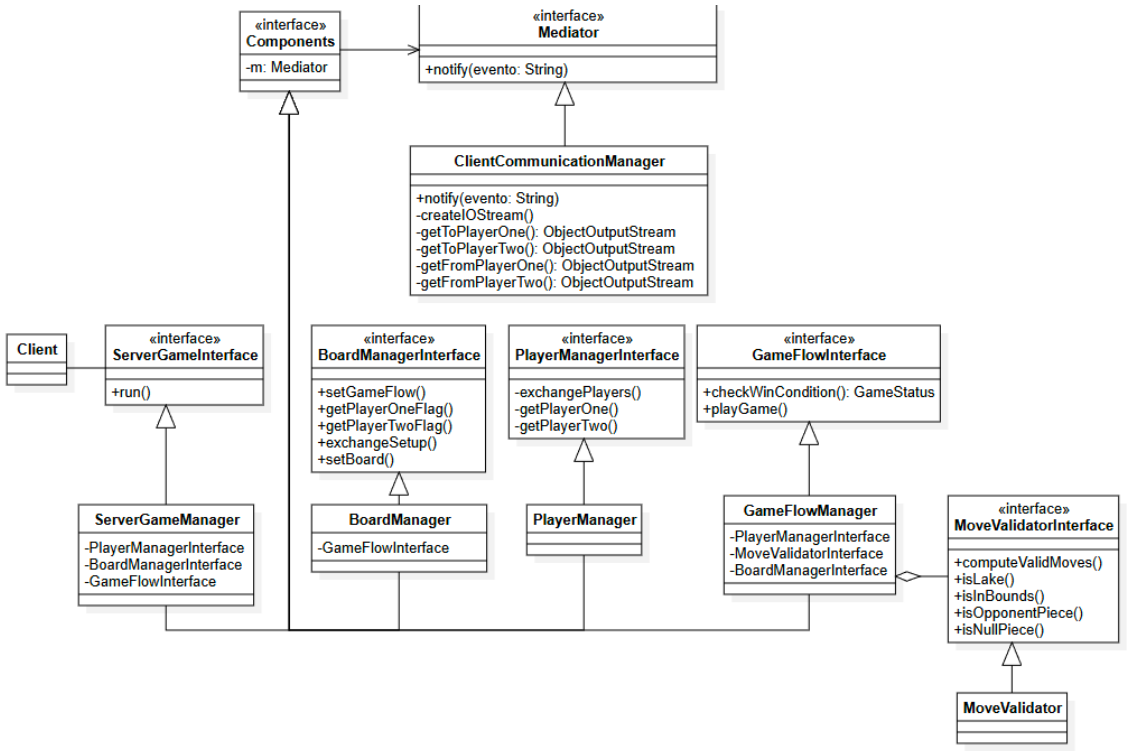


Ilustración 8: Resultado de la aplicación del patrón Mediador en la carpeta ServerGameManager

Por último, para asegurar un desacoplamiento de las diferentes escenas usadas a lo largo de la ejecución de la aplicación, se aplicaron los patrones **Template Method** y **Factory Method** para asegurar la fácil creación y/o modificación de una escena y la utilización de estas clases, respectivamente. Resultando en una nueva estructura reflejada por este diagrama UML:

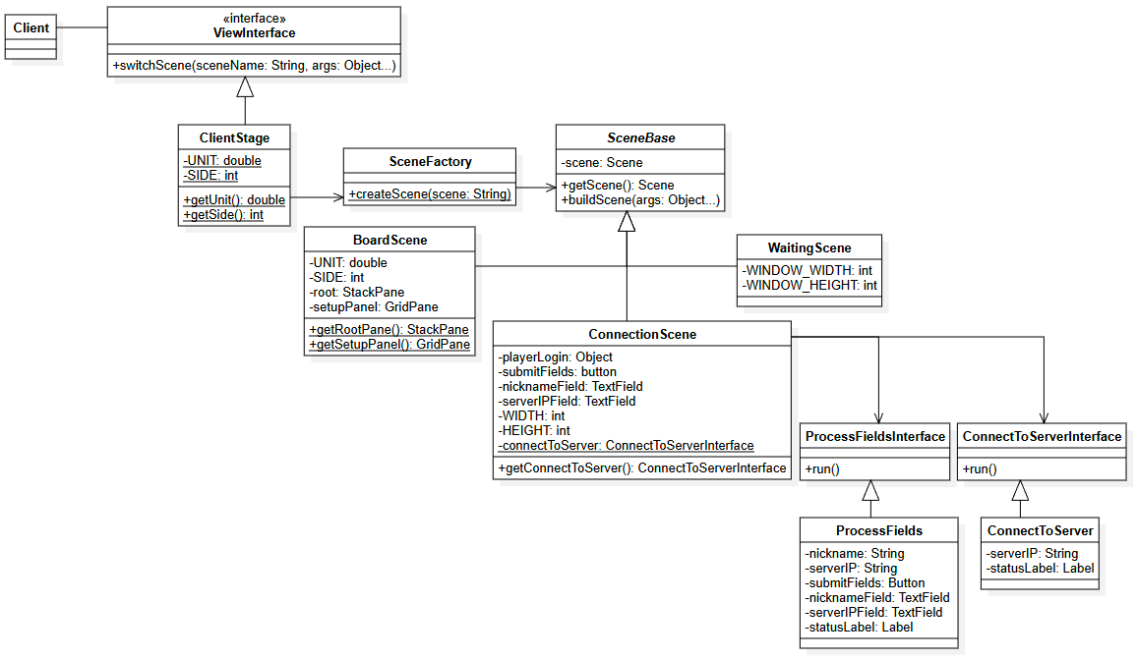


Ilustración 9: Resultado de la aplicación de los patrones Factory Method y Template Method sobre las vistas

3. Evolución

Como segunda parte de la práctica se nos encomendó realizar ciertas funciones adicionales pensadas por nosotros y previamente confirmadas con los profesores. Para ello estuvimos viendo el funcionamiento en sí del sistema y lo comparamos con otras aplicaciones de juegos 1 vs 1, como el chess.com.

Las principales deficiencias las encontramos en la falta de una posible comunicación entre los jugadores, ya que al final este programa se puede jugar desde diferentes lugares al hacer uso de IP, la inexistencia de una pantalla final que cierre los procesos cliente y el del servidor correctamente y la capacidad de posicionar nuestras piezas de manera aleatoria si el jugador quiere, es poco experimentado u otra razón.

Por último, se realizó una mejora ya planteada por los desarrolladores originales, pero no funcional, siendo la utilización real de la dirección IP escrita por los usuarios en la primera ventana. Esto ya que estaba programado en la clase *ClienSocket* pero nunca se le pasaba el valor introducido usando siempre localhost. Por ello se han relacionado los atributos posibilitando conectar un cliente al servidor desde dos maquinas distintas, tanto el puerto 4212 (la aplicación en sí) como el puerto 12345 (donde ase ha implementado el chat de texto).

A continuación, se explican dichas evoluciones en el sistema.

3.1. Chat de texto

Esta funcionalidad fue pensada para mejorar la experiencia social del juego permitiendo a los jugadores comunicarse y comentar la partida en tiempo real y aumentar la inmersión haciendo el juego más dinámico y real.

Para llevarla a cabo primero añadimos el espacio que tendrá el chat dentro de la pantalla del tablero, creamos un *VBox* llamado *chatPane* que contiene el área de texto que mostrara los mensajes y debajo de este un campo de entrada y un botón de envió, configuramos su ancho para que ocupen un tercio de la pantalla y lo alineamos al borde derecho del layout principal.

A continuación, hicimos la lógica de nuestro chat que sigue un modelo cliente-servidor, dentro de la carpeta *common* creamos la carpeta chat con 5 clases:

- *ChatClient*
  Representa al cliente de chat integrado en la interfaz del juego, esta clase establece conexión con el servidor.
- *ChatServer*
  Levanta el servidor de sockets para gestionar múltiples clientes, escuchas conexiones entrantes en el puerto 12345, instancia un *ClientaHandler* por cada nuevo cliente y lo lanza en un hilo independiente, por último, mantiene una lista sincronizada con todos los clientes conectados al chat.
- *ChatController*
  Controlador que conecta la interfaz gráfica (*TextArea* y *TextField*) con la lógica del *ChatClient*, gestionando la conexión, envío y recepción de mensajes para actualizar la UI.
- *ClientHandler*
  Se encarga de gestionar la comunicación individual por cada cliente conectado, gestionando los mensajes del cliente y difundiéndolos a todos los clientes conectados excepto a el mismo, también gestiona la desconexión de un cliente del chat.
- *ClientHandlerInterface*
  Interfaz que define los métodos básicos de un cliente conectado.

Por último, en el Server.java se lanza el servidor del chat en un hilo independiente mientras el resto del servidor continúa funcionando, antes de hacer la fase de configuración del tablero se inicializa el controlador del chat. Desde la interfaz gráfica del chat se realiza la conexión entre los clientes y el servidor del chat. De esta manera, se integra un chat completamente funcional para los usuarios durante la partida.

Ilustración 10: Chat integrado en el tablero

3.2. Pantalla final

La segunda funcionalidad aplicada se consideró necesaria por el equipo al no tener una pantalla final que te permitirá tanto conocer de manera explícita el ganador y el perdedor, como terminar los procesos clientes correctamente avisando al servidor.

Para ello se implementó la clase *FinalScene* encargada de construir la ventana usando JavaFX, siendo muy sencilla su implementación gracias al patrón **Tempate Method** aplicado en la lógica que se encarga de aplicar las pantallas, explicada en el principio DIP.

Una vez construida esta ventana, mostrada a continuación, faltaba saber cuándo se la llamaría para ser aplicada, para ello toco revisar el código hasta encontrar dentro de la clase *GameLogic* donde y como se termina la partida. Una vez dentro del método *playGame* se llama a los métodos necesarios para construir la pantalla pasándole el nombre del ganador, el del perdedor y el objeto de la clase *ServerConnection* para posteriormente, al pulsar el botón de “cerrar” se le notifique al servidor y se cierren los clientes correctamente, y el servidor normal y el servidor del chat lo sepan, pero siguen esperando nuevas sesiones.

Por último, se aplicaron 10 segundos de espera entre que se revelan todas las piezas y se muestra la última pantalla para poder comprobar o hacer una captura de pantalla sobre las piezas o chatear al final con el contrincante.

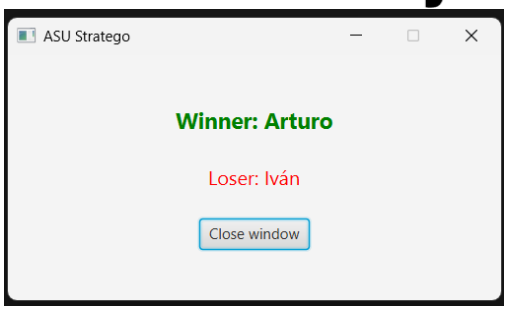


Ilustración 11: Pantalla final

3.3. Selección de fichas aleatoria

Como equipo pensamos que sería de interés darles la posibilidad a los jugadores de elegir de forma aleatoria el posicionamiento de sus piezas, con el objetivo de hacer las cosas más "picantes".

Por lo que en la pantalla para conectarse al servidor hemos añadido un nuevo botón que permite inicializar tu tablero con las piezas de forma completamente aleatoria. Esta será una decisión que tomará cada usuario de forma independiente. Por lo que puede haber usuarios que quieran jugar de forma aleatoria contra usuarios que juegan de forma convencional.

Para realizar esos cambios, además de la lógica de añadir el botón, se añadió una variable de configuración publica en el *ClientGameManager*, esta variable establece si se desea o no jugar de forma aleatoria.

El botón en el *connectionPanel* cambiara el valor del booleano según la decisión del jugador. Y este booleano, a su vez, cuando intente iniciar el tiempo de colocar las piezas, será chequeado y en caso de querer tener una distribución aleatoria, el timer será establecido a 0 para ese jugador. Es decir, quedará a la espera del otro, con todas sus piezas colocadas de forma aleatoria.

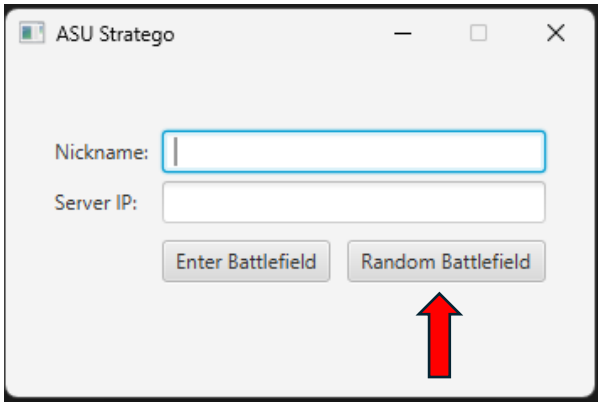


Ilustración 12: Imagen con el nuevo botón para permitir campo de batalla aleatorio

4. Conclusión

Durante el desarrollo de esta práctica se ha realizado un proceso de transformación y actualización un sistema heredado, particularmente el código fuente de Stratego. Gracias al análisis previo de la estructura del programa mediante un diagrama UML se logró establecer una base sólida para poder empezar a plantear los pasos de mantenimiento y evolución del sistema.

En cuanto al mantenimiento, al aplicar los principios SOLID se logró reducir el acoplamiento entre los diferentes componentes y aumentar la cohesión entre las clases, esto podría facilitar a futuro el hacer testing y la extensión del código. Para aplicar los principios fue necesario el uso de múltiples patrones de diseño tales como Strategy, Mediator, Factory Method, Templete Method lo cual hizo que la arquitectura del sistema fuese más robusta y favoreció flexibilidad ante modificaciones futuras.

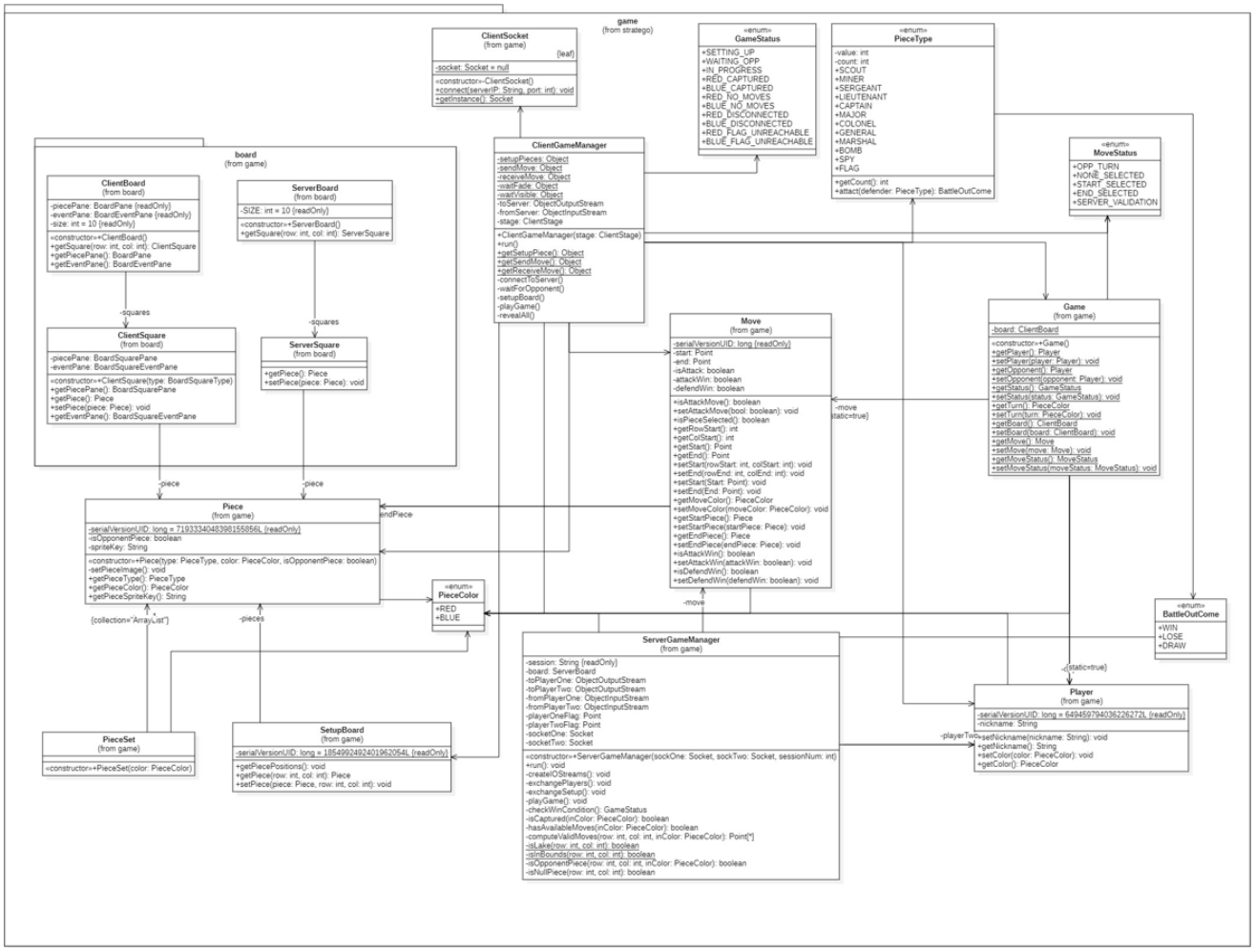
En la fase de evolución, fueron añadidas funcionalidades con el fin de mejorar la experiencia del usuario, las cuales no fueron consideradas en el diseño original, como la inclusión de un chat de texto disponible durante toda la partida, una pantalla de cierre al acabar la partida y la opción de colocar de manera aleatoria las piezas sin estar forzado a que el contador llegue a cero. El desarrollo de estas mejoras no solo mejoró el producto final sino que también validó la extensibilidad lograda gracias a la etapa de mantenimiento puesto que el desarrollo fue mucho más manejable.

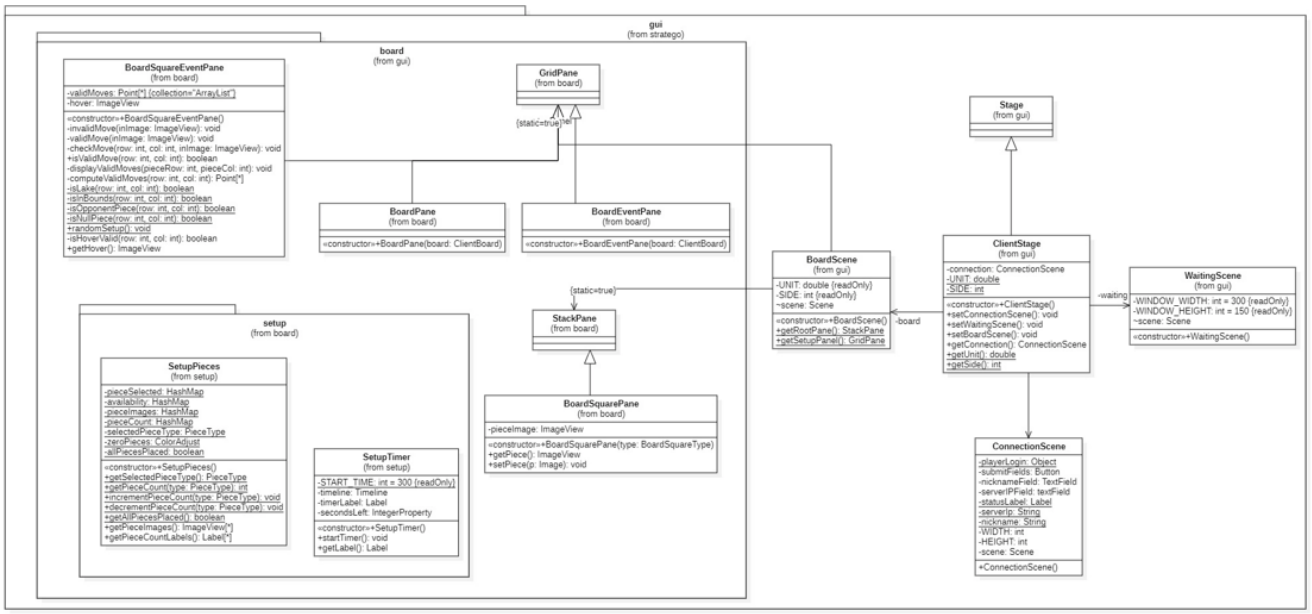
Debido a problemas con la herramienta de ingeniería inversa proporcionada por StarUML, mostraremos el estado final del programa utilizando la herramienta ApiDocs. Los resultados pueden consultarse en el Anexo 2 o en el *apidocs.zip* dentro de la carpeta *"documentation"* en la raíz del repositorio.

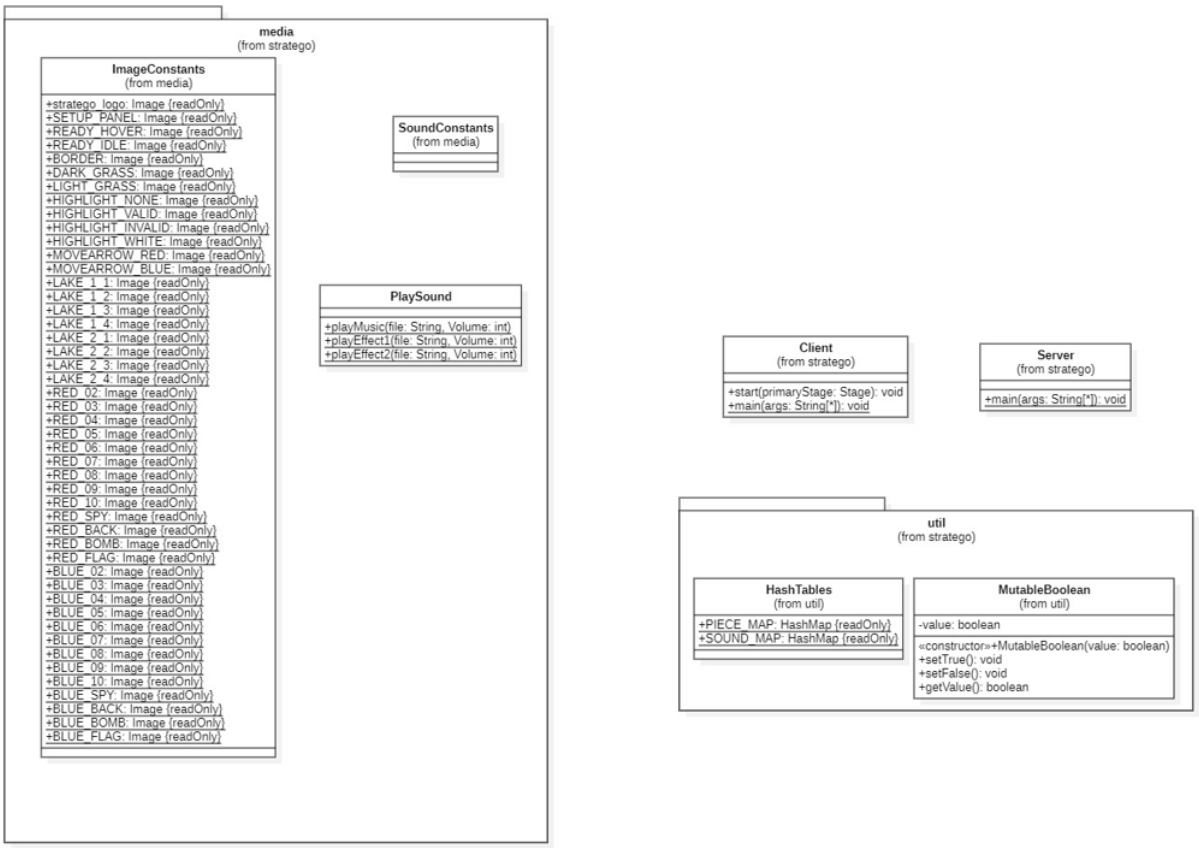
Anexos

Anexo 1

Diagrama de UML inicial, dividido en los paquetes principales


game
(from stratego)
ClientSocket
(from game)
{leaf}
-socket: Socket = null
«constructor»-ClientSocket()
+connect(server/IP: String, port: int): void
+getInstance(): Socket
«enum»
GameStatus
+SETTING_UP
+WAITING_OPP
+IN_PROGRESS
+RED_CAPTURED
+BLUE_CAPTURED
+RED_NO_MOVES
+BLUE_NO_MOVES
+RED_DISCONNECTED
+BLUE_DISCONNECTED
+RED_FLAG_UNREACHABLE
+BLUE_FLAG_UNREACHABLE
«enum»
PieceType
-value: int
-count: int
+SCOUT
+MINER
+SERGEANT
+LIEUTENANT
+CAPTAIN
+MAJOR
+COLONEL
+GENERAL
+MARSHAL
+BOMB
+SPY
+FLAG
+getCount(): int
+attact(defender: PieceType): BattleOutCome
board
(from game)
ClientBoard
(from board)
-piecePane: BoardPane {readOnly}
-eventPane: BoardEventPane {readOnly}
-size: int = 10 {readOnly}
«constructor»+ClientBoard()
+getSquare(row: int, col: int): ClientSquare
+getPiecePane(): BoardPane
+getEventPane(): BoardEventPane
ServerBoard
(from board)
-SIZE: int = 10 {readOnly}
«constructor»+ServerBoard()
+getSquare(row: int, col: int): ServerSquare
ClientGameManager
-setupPieces: Object
-sendMove: Object
-receiveMove: Object
-waitFade: Object
-waitVisible: Object
-toServer: ObjectOutputStream
-fromServer: ObjectInputStream
-stage: ClientStage
+ClientGameManager(stage: ClientStage)
+run()
+getSetupPiece(): Object
+getSendMove(): Object
+getReceiveMove(): Object
-connectToServer()
-waitForOpponent()
-setupBoard()
-playGame()
-revealAll()
«enum»
MoveStatus
+OPP_TURN
+NONE_SELECTED
+START_SELECTED
+END_SELECTED
+SERVER_VALIDATION
-squares
-squares
ClientSquare
(from board)
-piecePane: BoardSquarePane
-eventPane: BoardSquareEventPane
«constructor»+ClientSquare(type: BoardSquareType)
+getPiecePane(): BoardSquarePane
+getPiece(): Piece
+setPiece(piece: Piece): void
+getEventPane(): BoardSquareEventPane
ServerSquare
(from board)
+getPiece(): Piece
+setPiece(piece: Piece): void
Move
(from game)
-serialVersionUID: long {readOnly}
-start: Point
-end: Point
-isAttack: boolean
-attackWin: boolean
-defendWin: boolean
+isAttackMove(): boolean
+setAttackMove(bool: boolean): void
+isPieceSelected(): boolean
+getRowStart(): int
+getColStart(): int
+getStart(): Point
+getEnd(): Point
+setStart(rowStart: int, colStart: int): void
+setEnd(rowEnd: int, colEnd: int): void
+setStart(Start: Point): void
+setEnd(End: Point): void
+getMoveColor(): PieceColor
+setMoveColor(moveColor: PieceColor): void
+setStartPiece(): Piece
+setStartPiece(startPiece: Piece): void
+getEndPiece(): Piece
+setEndPiece(endPiece: Piece): void
+isAttackWin(): boolean
+setAttackWin(attackWin: boolean): void
+isDefendWin(): boolean
+setDefendWin(defendWin: boolean): void
Game
(from game)
-board: ClientBoard
«constructor»+Game()
+getPlayer(): Player
+setPlayer(player: Player): void
+getOpponent(): Player
+setOpponent(opponent: Player): void
+getStatus(): GameStatus
+setStatus(status: GameStatus): void
+getTurn(): PieceColor
+setTurn(turn: PieceColor): void
+getBoard(): ClientBoard
+setBoard(board: ClientBoard): void
+getMove(): Move
+setMove(move: Move): void
+getMoveStatus(): MoveStatus
+setMoveStatus(moveStatus: MoveStatus): void
-piece
-piece
Piece
(from game)
-serialVersionUID: long = 7193334048398155856L {readOnly}
-isOpponentPiece: boolean
-spriteKey: String
«constructor»+Piece(type: PieceType, color: PieceColor, isOpponentPiece: boolean)
-setPieceImage(): void
+getPieceType(): PieceType
+getPieceColor(): PieceColor
+getPieceSpriteKey(): String
endPiece
-move
{static=true}
«enum»
PieceColor
+RED
+BLUE
«enum»
BattleOutCome
+WIN
+LOSE
+DRAW
{collection="ArrayList"}
-pieces
ServerGameManager
(from game)
-session: String {readOnly}
-board: ServerBoard
-toPlayerOne: ObjectOutputStream
-toPlayerTwo: ObjectOutputStream
-fromPlayerOne: ObjectInputStream
-fromPlayerTwo: ObjectInputStream
-playerOneFlag: Point
-playerTwoFlag: Point
-socketOne: Socket
-socketTwo: Socket
«constructor»+ServerGameManager(sockOne: Socket, sockTwo: Socket, sessionNum: int)
+run(): void
-createIOStreams(): void
-exchangePlayers(): void
-exchangeSetup(): void
-playGame(): void
-checkWinCondition(): GameStatus
-isCaptured(inColor: PieceColor): boolean
-hasAvailableMoves(inColor: PieceColor): boolean
-computeValidMoves(row: int, col: int, inColor: PieceColor): Point[*]
-isLake(row: int, col: int): boolean
-isInBounds(row: int, col: int): boolean
-isOpponentPiece(row: int, col: int, inColor: PieceColor): boolean
-isNullPiece(row: int, col: int): boolean
{static=true}
Player
(from game)
-serialVersionUID: long = 6494597940362262721 {readOnly}
-nickname: String
+setNickname(nickname: String): void
+getNickname(): String
+setColor(color: PieceColor): void
+getColor(): PieceColor
-playerTwo
PieceSet
(from game)
«constructor»+PieceSet(color: PieceColor)
SetupBoard
(from game)
-serialVersionUID: long = 1854992492401962054L {readOnly}
+getPiecePositions(): void
+getPiece(row: int, col: int): Piece
+setPiece(piece: Piece, row: int, col: int): void

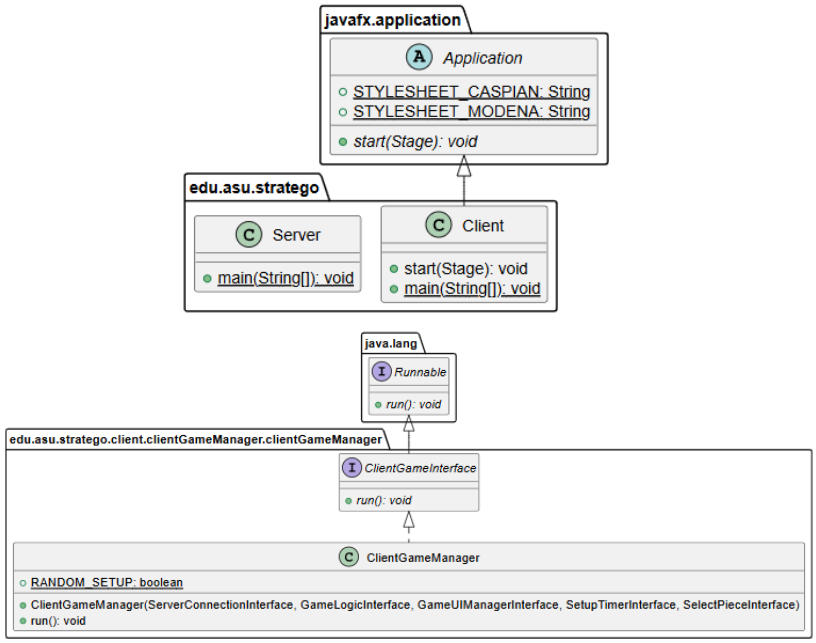

gui
(from stratego)
board
(from gui)
BoardSquareEventPane
(from board)
-validMoves: Point[*] {collection="ArrayList"}
-hover: ImageView
«constructor»+BoardSquareEventPane()
-invalidMove(inImage: ImageView): void
-validMove(inImage: ImageView): void
-checkMove(row: int, col: int, inImage: ImageView): void
+isValidMove(row: int, col: int): boolean
-displayValidMoves(pieceRow: int, pieceCol: int): void
-computeValidMoves(row: int, col: int): Point[*]
-isLake(row: int, col: int): boolean
-isInBounds(row: int, col: int): boolean
-isOpponentPiece(row: int, col: int): boolean
-isNullPiece(row: int, col: int): boolean
+randomSetup(): void
-isHoverValid(row: int, col: int): boolean
+getHover(): ImageView
GridPane
(from board)
{static=true}
BoardPane
(from board)
«constructor»+BoardPane(board: ClientBoard)
BoardEventPane
(from board)
«constructor»+BoardEventPane(board: ClientBoard)
Stage
(from gui)
BoardScene
(from gui)
-UNIT: double {readOnly}
-SIDE: int {readOnly}
~scene: Scene
«constructor»+BoardScene()
+getRootPane(): StackPane
+getSetupPanel(): GridPane
-board
ClientStage
(from gui)
-connection: ConnectionScene
-UNIT: double
-SIDE: int
«constructor»+ClientStage()
+setConnectionScene(): void
+setWaitingScene(): void
+setBoardScene(): void
+getConnection(): ConnectionScene
+getUnit(): double
+getSide(): int
-waiting
WaitingScene
(from gui)
-WINDOW_WIDTH: int = 300 {readOnly}
-WINDOW_HEIGHT: int = 150 {readOnly}
~scene: Scene
«constructor»+WaitingScene()
setup
(from board)
SetupPieces
(from setup)
-pieceSelected: HashMap
-availability: HashMap
-pieceImages: HashMap
-pieceCount: HashMap
-selectedPieceType: PieceType
-zeroPieces: ColorAdjust
-allPiecesPlaced: boolean
«constructor»+SetupPieces()
+getSelectedPieceType(): PieceType
+getPieceCount(type: PieceType): int
+incrementPieceCount(type: PieceType): void
+decrementPieceCount(type: PieceType): void
+getAllPiecesPlaced(): boolean
+getPieceImages(): ImageView[*]
+getPieceCountLabels(): Label[*]
SetupTimer
(from setup)
-START_TIME: int = 300 {readOnly}
-timeline: Timeline
-timerLabel: Label
-secondsLeft: IntegerProperty
«constructor»+SetupTimer()
+startTimer(): void
+getLabel(): Label
{static=true}
StackPane
(from board)
BoardSquarePane
(from board)
-pieceImage: ImageView
«constructor»+BoardSquarePane(type: BoardSquareType)
+getPiece(): ImageView
+setPiece(p: Image): void
ConnectionScene
-playerLogin: Object
-submitFields: Button
-nicknameField: TextField
-serverIPField: textField
-statusLabel: Label
-serverIp: String
-nickname: String
-WIDTH: int
-HEIGHT: int
-scene: Scene
+ConnectionScene()

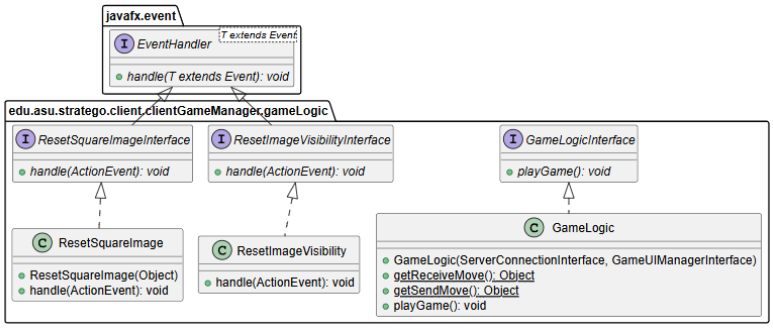


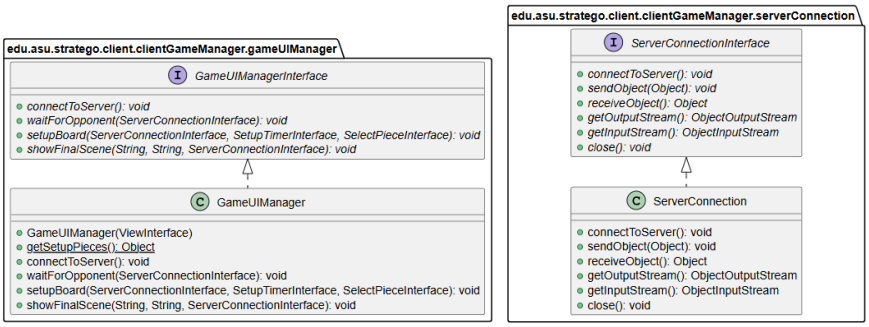
Anexo 2

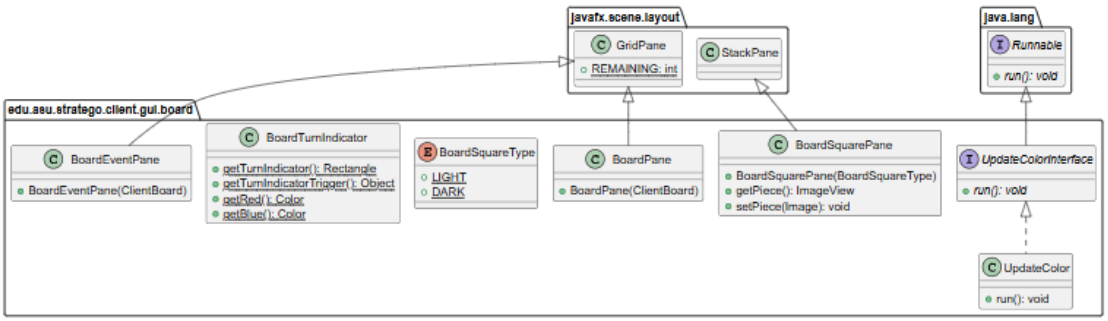
Diagrama de UML final, dividido en los diferentes paquetes

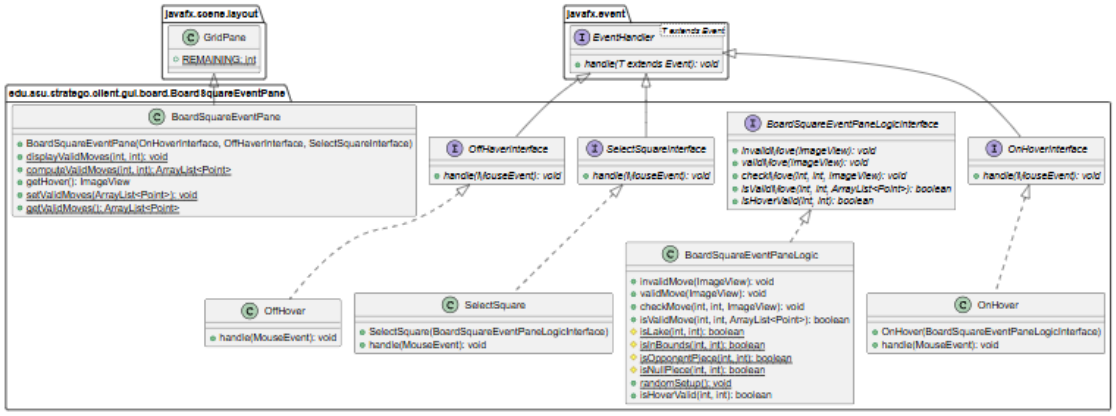
Client

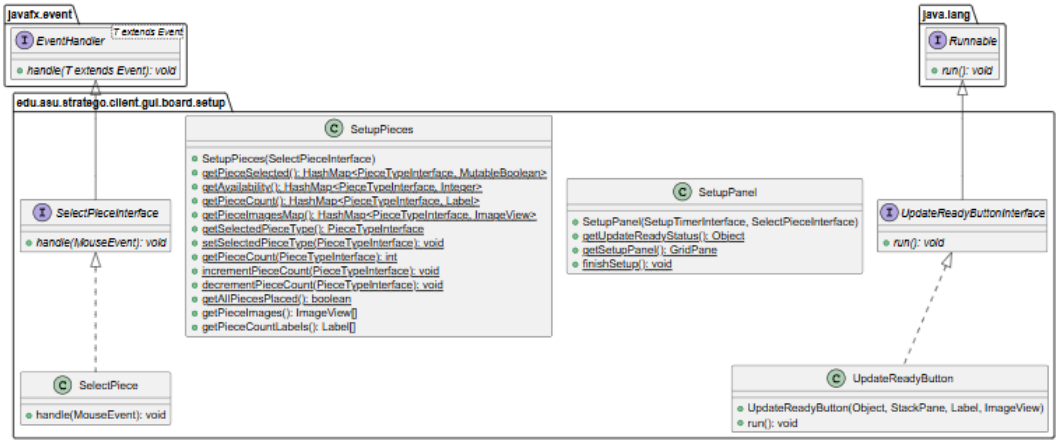


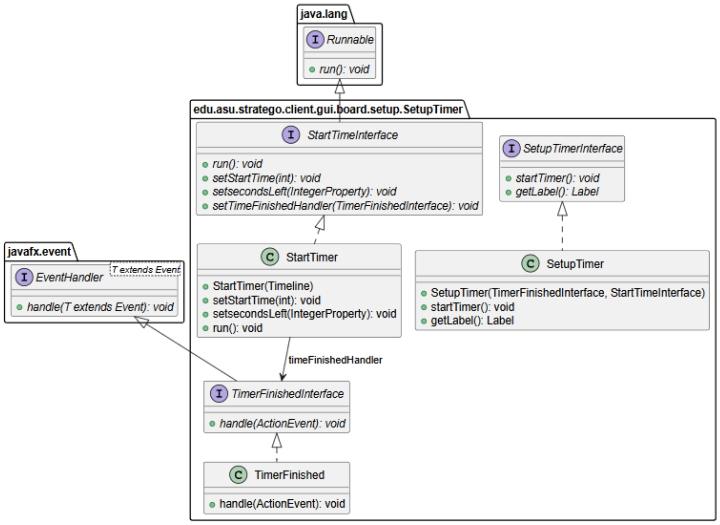

javafx.event
EventHandler
T extends Event
handle(T extends Event): void
edu.asu.stratego.client.clientGameManager.gameLogic
ResetSquareImageInterface
handle(ActionEvent): void
ResetImageVisibilityInterface
handle(ActionEvent): void
GameLogicInterface
playGame(): void
ResetSquareImage
ResetSquareImage(Object)
handle(ActionEvent): void
ResetImageVisibility
handle(ActionEvent): void
GameLogic
GameLogic(ServerConnectionInterface, GameUIManagerInterface)
getReceiveMove(): Object
getSendMove(): Object
playGame(): void

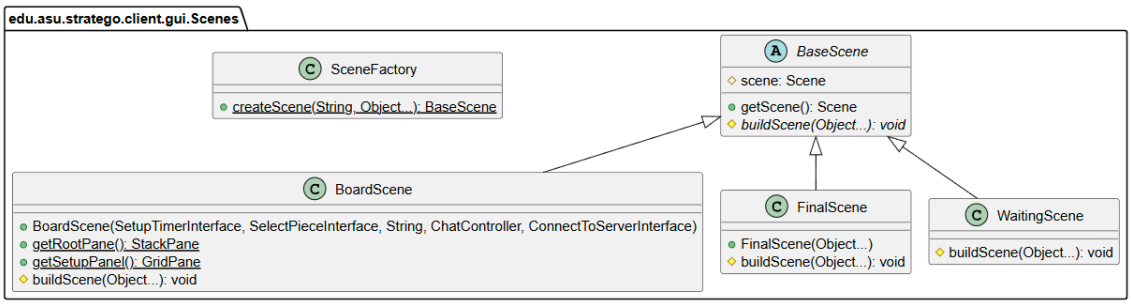

edu.asu.stratego.client.clientGameManager.gameUIManager
GameUIManagerInterface
connectToServer(): void
waitForOpponent(ServerConnectionInterface): void
setupBoard(ServerConnectionInterface, SetupTimerInterface, SelectPieceInterface): void
showFinalScene(String, String, ServerConnectionInterface): void
GameUIManager
GameUIManager(ViewInterface)
getSetupPieces(): Object
connectToServer(): void
waitForOpponent(ServerConnectionInterface): void
setupBoard(ServerConnectionInterface, SetupTimerInterface, SelectPieceInterface): void
showFinalScene(String, String, ServerConnectionInterface): void
edu.asu.stratego.client.clientGameManager.serverConnection
ServerConnectionInterface
connectToServer(): void
sendObject(Object): void
receiveObject(): Object
getOutputStream(): ObjectOutputStream
getInputStream(): ObjectInputStream
close(): void
ServerConnection
connectToServer(): void
sendObject(Object): void
receiveObject(): Object
getOutputStream(): ObjectOutputStream
getInputStream(): ObjectInputStream
close(): void

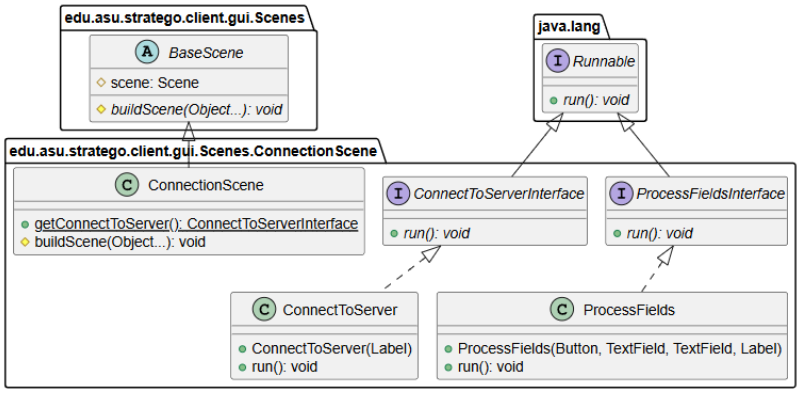

javafx.scene.layout
GridPane
REMAINING: int
StackPane
java.lang
Runnable
run(): void
edu.asu.stratego.client.gui.board
BoardEventPane
BoardEventPane(ClientBoard)
BoardTurnIndicator
getTurnIndicator(): Rectangle
getTurnIndicatorTrigger(): Object
getRed(): Color
getBlue(): Color
BoardSquareType
LIGHT
DARK
BoardPane
BoardPane(ClientBoard)
BoardSquarePane
BoardSquarePane(BoardSquareType)
getPiece(): ImageView
setPiece(Image): void
UpdateColorInterface
run(): void
UpdateColor
run(): void


javafx.scene.layout
GridPane
REMAINING: int
javafx.event
EventHandler
T extends Event
handle(T extends Event): void
edu.asu.stratego.client.gui.board.BoardSquareEventPane
BoardSquareEventPane
BoardSquareEventPane(OnHoverInterface, OffHaverInterface, SelectSquareInterface)
displayValidMoves(int, int): void
computeValidMoves(int, int): ArrayList<Point>
getHover(): ImageView
setValidMoves(ArrayList<Point>): void
getValidMoves(): ArrayList<Point>
OffHaverInterface
handle(MouseEvent): void
SelectSquareInterface
handle(MouseEvent): void
BoardSquareEventPaneLogicInterface
invalidMove(ImageView): void
validMove(ImageView): void
checkMove(int, int, ImageView): void
isValidMove(int, int, ArrayList<Point>): boolean
isHoverValid(int, int): boolean
OnHoverInterface
handle(MouseEvent): void
OffHover
handle(MouseEvent): void
SelectSquare
SelectSquare(BoardSquareEventPaneLogicInterface)
handle(MouseEvent): void
BoardSquareEventPaneLogic
invalidMove(ImageView): void
validMove(ImageView): void
checkMove(int, int, ImageView): void
isValidMove(int, int, ArrayList<Point>): boolean
isLake(int, int): boolean
isInBounds(int, int): boolean
isOpponentPiece(int, int): boolean
isNullPiece(int, int): boolean
randomSetup(): void
isHoverValid(int, int): boolean
OnHover
OnHover(BoardSquareEventPaneLogicInterface)
handle(MouseEvent): void

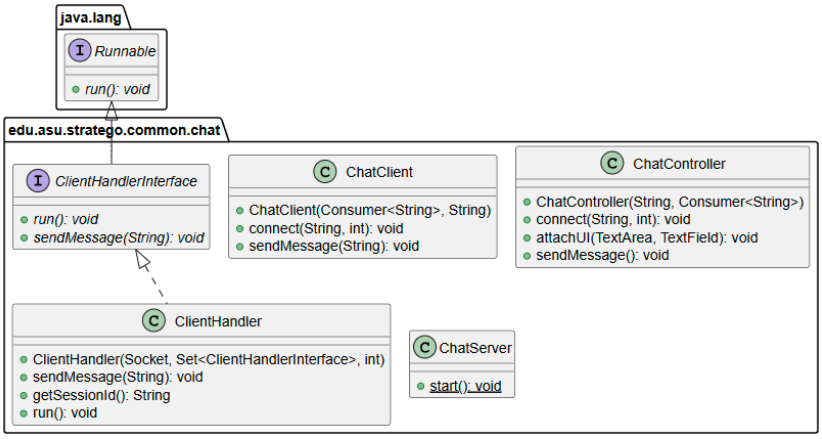

javafx.event
EventHandler
handle(T extends Event): void
java.lang
Runnable
run(): void
edu.asu.stratego.client.gui.board.setup
SetupPieces
SetupPieces(SelectPieceInterface)
getPieceSelected(): HashMap<PieceTypeInterface, MutableBoolean>
getAvailability(): HashMap<PieceTypeInterface, Integer>
getPieceCount(): HashMap<PieceTypeInterface, Label>
getPieceImagesMap(): HashMap<PieceTypeInterface, ImageView>
getSelectedPieceType(): PieceTypeInterface
setSelectedPieceType(PieceTypeInterface): void
getPieceCount(PieceTypeInterface): int
incrementPieceCount(PieceTypeInterface): void
decrementPieceCount(PieceTypeInterface): void
getAllPiecesPlaced(): boolean
getPieceImages(): ImageView[]
getPieceCountLabels(): Label[]
SetupPanel
SetupPanel(SetupTimerInterface, SelectPieceInterface)
getUpdateReadyStatus(): Object
getSetupPanel(): GridPane
finishSetup(): void
SelectPieceInterface
handle(MouseEvent): void
UpdateReadyButtonInterface
run(): void
SelectPiece
handle(MouseEvent): void
UpdateReadyButton
UpdateReadyButton(Object, StackPane, Label, ImageView)
run(): void

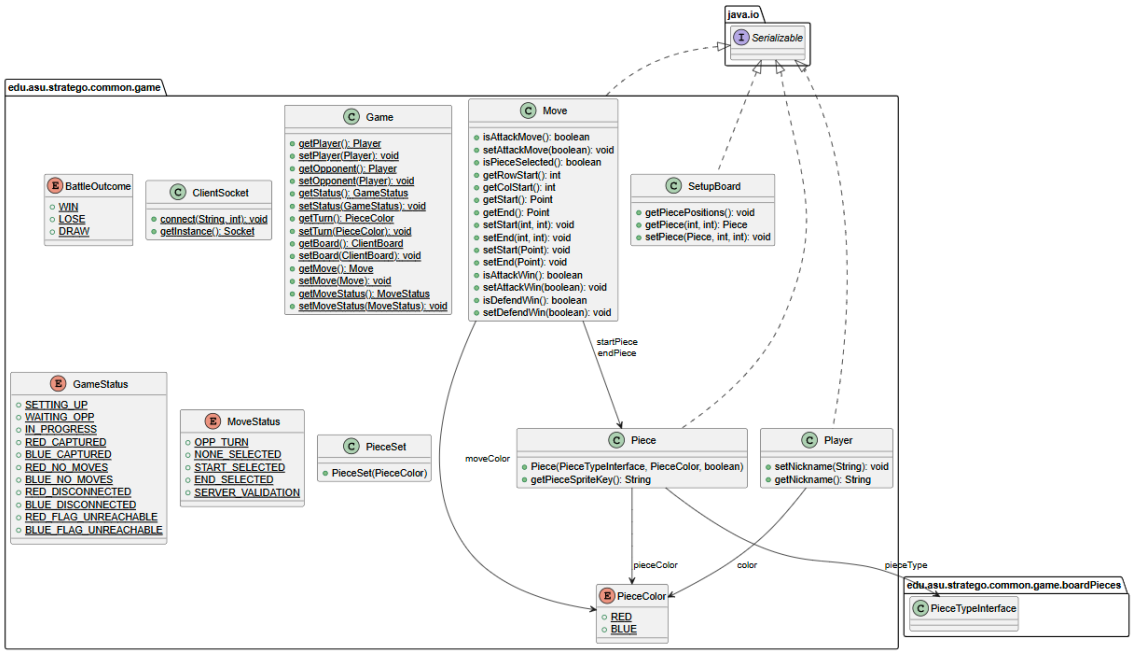

java.lang
Runnable
run(): void
edu.asu.stratego.client.gui.board.setup.SetupTimer
StartTimeInterface
run(): void
setStartTime(int): void
setsecondsLeft(IntegerProperty): void
setTimeFinishedHandler(TimerFinishedInterface): void
SetupTimerInterface
startTimer(): void
getLabel(): Label
javafx.event
EventHandler
handle(T extends Event): void
StartTimer
StartTimer(Timeline)
setStartTime(int): void
setsecondsLeft(IntegerProperty): void
run(): void
SetupTimer
SetupTimer(TimerFinishedInterface, StartTimeInterface)
startTimer(): void
getLabel(): Label
timeFinishedHandler
TimerFinishedInterface
handle(ActionEvent): void
TimerFinished
handle(ActionEvent): void

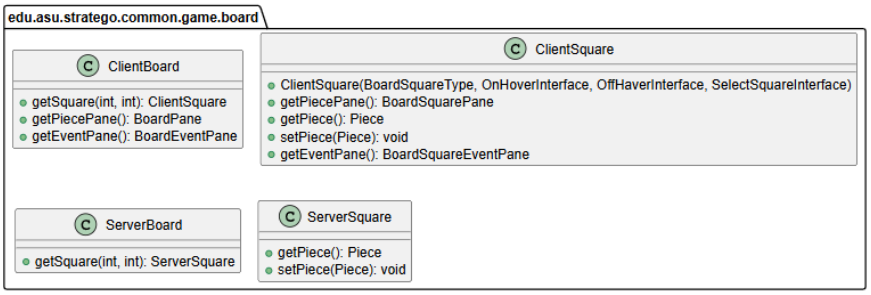

edu.asu.stratego.client.gui.Scenes
SceneFactory
createScene(String, Object...): BaseScene
BaseScene
scene: Scene
getScene(): Scene
buildScene(Object...): void
BoardScene
BoardScene(SetupTimerInterface, SelectPieceInterface, String, ChatController, ConnectToServerInterface)
getRootPane(): StackPane
getSetupPanel(): GridPane
buildScene(Object...): void
FinalScene
FinalScene(Object...)
buildScene(Object...): void
WaitingScene
buildScene(Object...): void

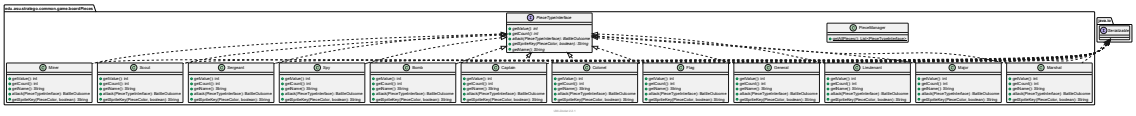

edu.asu.stratego.client.gui.Scenes
BaseScene
scene: Scene
buildScene(Object...): void
java.lang
Runnable
run(): void
edu.asu.stratego.client.gui.Scenes.ConnectionScene
ConnectionScene
getConnectToServer(): ConnectToServerInterface
buildScene(Object...): void
ConnectToServerInterface
run(): void
ProcessFieldsInterface
run(): void
ConnectToServer
ConnectToServer(Label)
run(): void
ProcessFields
ProcessFields(Button, TextField, TextField, Label)
run(): void

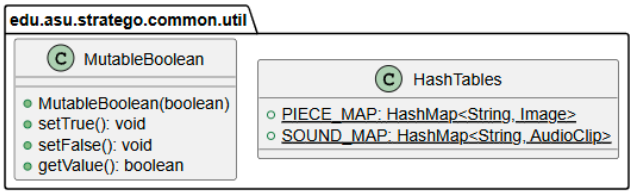
Common


java.lang
Runnable
run(): void
edu.asu.stratego.common.chat
ClientHandlerInterface
run(): void
sendMessage(String): void
ChatClient
ChatClient(Consumer<String>, String)
connect(String, int): void
sendMessage(String): void
ChatController
ChatController(String, Consumer<String>)
connect(String, int): void
attachUI(TextArea, TextField): void
sendMessage(): void
ClientHandler
ClientHandler(Socket, Set<ClientHandlerInterface>, int)
sendMessage(String): void
getSessionId(): String
run(): void
ChatServer
start(): void


java.io
Serializable
edu.asu.stratego.common.game
Game
getPlayer(): Player
setPlayer(Player): void
getOpponent(): Player
setOpponent(Player): void
getStatus(): GameStatus
setStatus(GameStatus): void
getTurn(): PieceColor
setTurn(PieceColor): void
getBoard(): ClientBoard
setBoard(ClientBoard): void
getMove(): Move
setMove(Move): void
getMoveStatus(): MoveStatus
setMoveStatus(MoveStatus): void
Move
isAttackMove(): boolean
setAttackMove(boolean): void
isPieceSelected(): boolean
getRowStart(): int
getColStart(): int
getStart(): Point
getEnd(): Point
setStart(int, int): void
setEnd(int, int): void
setStart(Point): void
setEnd(Point): void
isAttackWin(): boolean
setAttackWin(boolean): void
isDefendWin(): boolean
setDefendWin(boolean): void
BattleOutcome
WIN
LOSE
DRAW
ClientSocket
connect(String, int): void
getInstance(): Socket
SetupBoard
getPiecePositions(): void
getPiece(int, int): Piece
setPiece(Piece, int, int): void
startPiece
endPiece
GameStatus
SETTING_UP
WAITING_OPP
IN_PROGRESS
RED_CAPTURED
BLUE_CAPTURED
RED_NO_MOVES
BLUE_NO_MOVES
RED_DISCONNECTED
BLUE_DISCONNECTED
RED_FLAG_UNREACHABLE
BLUE_FLAG_UNREACHABLE
MoveStatus
OPP_TURN
NONE_SELECTED
START_SELECTED
END_SELECTED
SERVER_VALIDATION
PieceSet
PieceSet(PieceColor)
moveColor
Piece
Piece(PieceTypeInterface, PieceColor, boolean)
getPieceSpriteKey(): String
Player
setNickname(String): void
getNickname(): String
pieceColor
color
pieceType
PieceColor
RED
BLUE
edu.asu.stratego.common.game.boardPieces
PieceTypeInterface

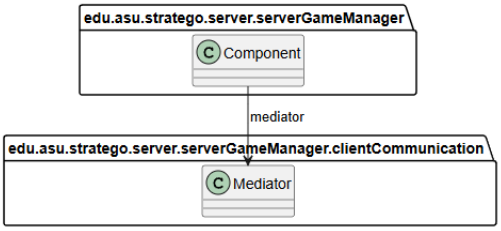

edu.asu.stratego.common.game.board
ClientBoard
getSquare(int, int): ClientSquare
getPiecePane(): BoardPane
getEventPane(): BoardEventPane
ClientSquare
ClientSquare(BoardSquareType, OnHoverInterface, OffHaverInterface, SelectSquareInterface)
getPiecePane(): BoardSquarePane
getPiece(): Piece
setPiece(Piece): void
getEventPane(): BoardSquareEventPane
ServerBoard
getSquare(int, int): ServerSquare
ServerSquare
getPiece(): Piece
setPiece(Piece): void

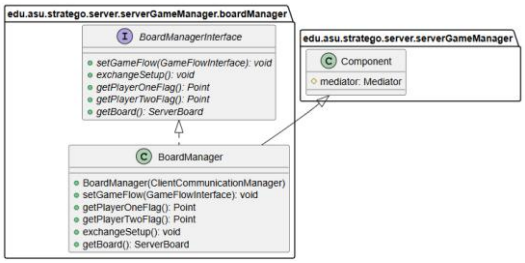


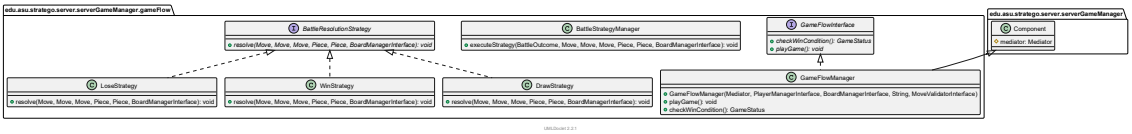

edu.asu.stratego.common.util
MutableBoolean
MutableBoolean(boolean)
setTrue(): void
setFalse(): void
getValue(): boolean
HashTables
PIECE_MAP: HashMap<String, Image>
SOUND_MAP: HashMap<String, AudioClip>

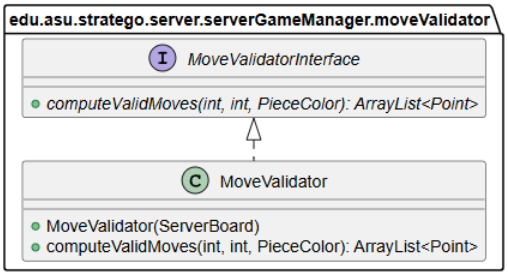
Server

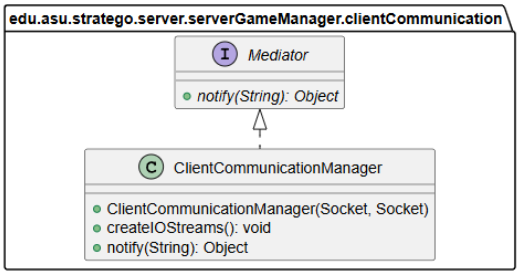

edu.asu.stratego.server
GameSessionFactory
createSession(Socket, Socket, String): ServerGameInterface

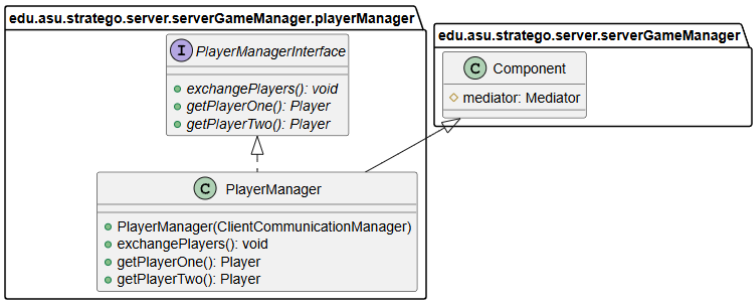

edu.asu.stratego.server.serverGameManager
Component
mediator
edu.asu.stratego.server.serverGameManager.clientCommunication
Mediator

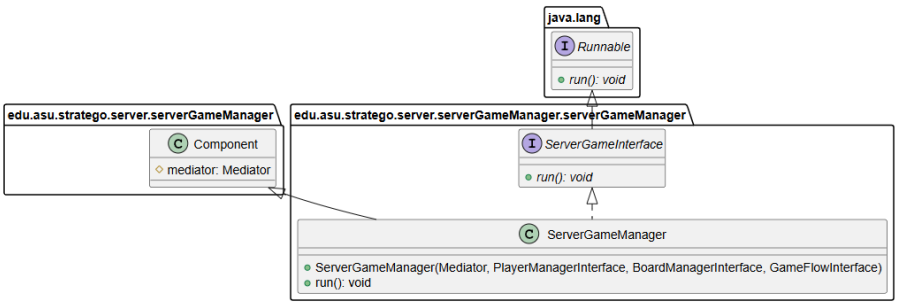

edu.asu.stratego.server.serverGameManager.boardManager
BoardManagerInterface
setGameFlow(GameFlowInterface): void
exchangeSetup(): void
getPlayerOneFlag(): Point
getPlayerTwoFlag(): Point
getBoard(): ServerBoard
BoardManager
BoardManager(ClientCommunicationManager)
setGameFlow(GameFlowInterface): void
getPlayerOneFlag(): Point
getPlayerTwoFlag(): Point
exchangeSetup(): void
getBoard(): ServerBoard
edu.asu.stratego.server.serverGameManager
Component
mediator: Mediator


edu.asu.stratego.server.serverGameManager.gameFlow
BattleResolutionStrategy
BattleStrategyManager
GameFlowInterface
LoseStrategy
WinStrategy
DrawStrategy
GameFlowManager
edu.asu.stratego.server.serverGameManager
Component
mediator: Mediator


edu.asu.stratego.server.serverGameManager.moveValidator
MoveValidatorInterface
computeValidMoves(int, int, PieceColor): ArrayList<Point>
MoveValidator
MoveValidator(ServerBoard)
computeValidMoves(int, int, PieceColor): ArrayList<Point>


edu.asu.stratego.server.serverGameManager.clientCommunication
Mediator
notify(String): Object
ClientCommunicationManager
ClientCommunicationManager(Socket, Socket)
createIOStreams(): void
notify(String): Object


edu.asu.stratego.server.serverGameManager.playerManager
PlayerManagerInterface
exchangePlayers(): void
getPlayerOne(): Player
getPlayerTwo(): Player
PlayerManager
PlayerManager(ClientCommunicationManager)
exchangePlayers(): void
getPlayerOne(): Player
getPlayerTwo(): Player
edu.asu.stratego.server.serverGameManager
Component
mediator: Mediator


java.lang
Runnable
run(): void
edu.asu.stratego.server.serverGameManager
Component
mediator: Mediator
edu.asu.stratego.server.serverGameManager.serverGameManager
ServerGameInterface
run(): void
ServerGameManager
ServerGameManager(Mediator, PlayerManagerInterface, BoardManagerInterface, GameFlowInterface)
run(): void