Contributions

I completed every step of the project alone.

Statement of Completion

All the requirements have been completed.

The system caters for the following play modes: Human vs Human, Computer vs Human and Computer vs Computer(IMoveStrategy).

The system can check the validity of moves made by players (IGameLogic). A computer player selects a valid move at random.

The game can be played from end to finish, can be saved and loaded. The system can recreate the players involved in a saved game, the dimension of the board and all move played. From that the system calculates the score. Moves can be undone, the only limitation being a move made by the computer will be instantly replaced by a new one. The moves are saved in the text file.

The game rules are displayed before starting a new game, the instructions are displayed before each move.

Final Design Overview

My design has changed quite a bit since the preliminary design. The reason being I wanted to learn as much as could and perfect my programming skills. Implementing the functionalities revealed a lot of shortcomings of my previous design. I enjoyed thinking about how to make the program the more OO possible, and reading books and resources online gave me a lot of ideas. My system can be split into different parts:

The Userinterface or pregame display:

This group of class serves to present the various options available for selecting and launching the game. Menus act as the interface for showcasing the games, start a new Game or, if a game needs to be loaded, display the available saves. The command design pattern was employed to include flexibility to the menus, making it easy to add future games for selection and present the saves in a convenient way. Given the requirement for multiple menus, the composite design pattern was adopted. Commands were used as leaves and menu as components.

The Game Creator

The Match class was introduced to facilitate the scenario where victory requires winning multiple games. This addition streamlined the implementation of essential functionalities, including text display and the management of save and load options.

The Game Components

The game is an easily customable abstract class. GameComponent is the cornerstone class, representing game elements. Chosen as base for the board, Tile enables the use of the decorator design pattern. In turn, this allows the utilization of TileWithPiece. This can be used for different games, tile can be decorated with different kinds of GameComponent(Piece in our design) and can be use for multiple kind of future games. The board object holds the tiles and displays the board. An abstract class unlocks the ability to have boards with different shapes (circular, hexagonal, etc.).

The Game Actions

Responsibility for crafting moves now rests with the players. To facilitate this, the Strategy design pattern was introduced, featuring the IMoveStrategy interface and its makeMove method. This inclusion greatly streamlined the incorporation of diverse logic for both human and computer players. Moreover, this blueprint welcomes future expansions with additional strategies for varying games or AI complexity levels.

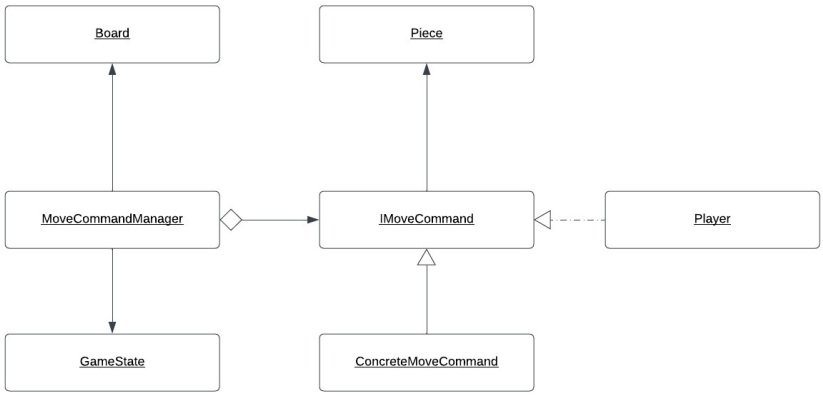
A novel entrant, the MoveCommandManager class, has taken on the mantle of executing move commands and updating scores. In tandem, the Scorer assumes the duty of quantifying points accrued through moves. Concurrently, the IGameLogic orchestrates the determination of move validity and game culmination. Subsequent iterations may witness the migration of other functions to this class.

This fresh design approach champions greater dependency injections, fostering a more modular system. It's acknowledged that this design could have been more straightforward, yet the driving force was the pursuit of enhanced programming proficiency through experimentation with diverse design strategies.

Identification of design principle and patterns

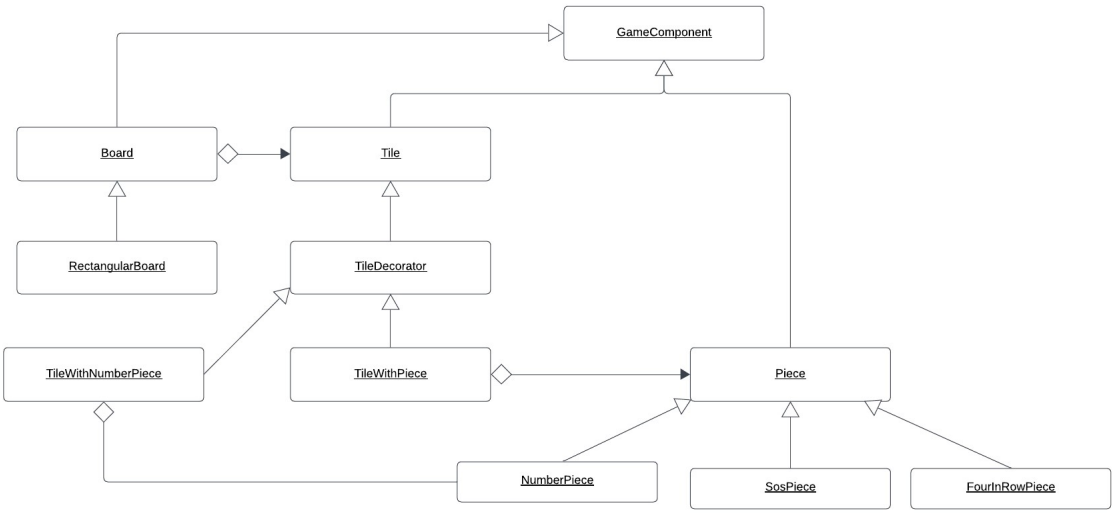
The command design pattern:

Enables the storage of moves in a list. To undo a move, we remove the last move from the list. All commands are executed between each turn to update the score and the board. The Execute method adds a game component on the board. Moves have coordinates, the player who created the move, and the piece (GameComponent).



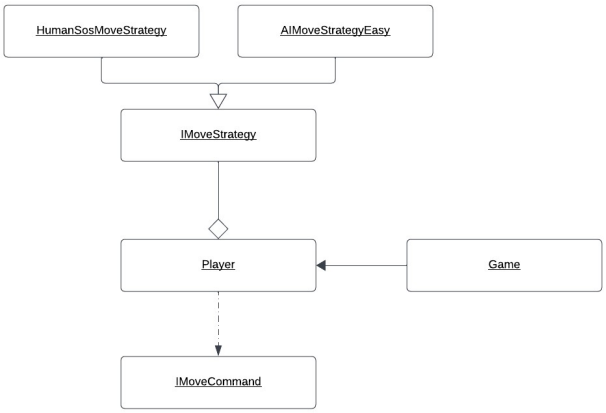
The Decorator Design Pattern

The tiles are the subunit of the board. For both games the tiles can have a piece object. The Decorator design pattern allows the program to dynamically add features to the tiles. A tile can hold a gameComponent. TileDecorator are tiles, other decorator could be added for other games without having to change a lot of the code. As a proof of concept, TileWithNumber have been added to demonstrate how tiles can be modifies dynamically.



The strategy Design Pattern

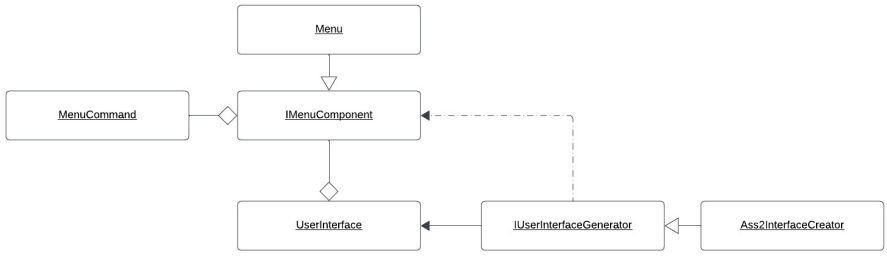
Human player and Computer player have different ways to make a move. The computer players select a move at random but the human player needs to select a valid move. The concrete IMoveStrategy object are injected in the player objects. The GameLoop() method from the class object asks the player for a move.



The Composite Design Pattern

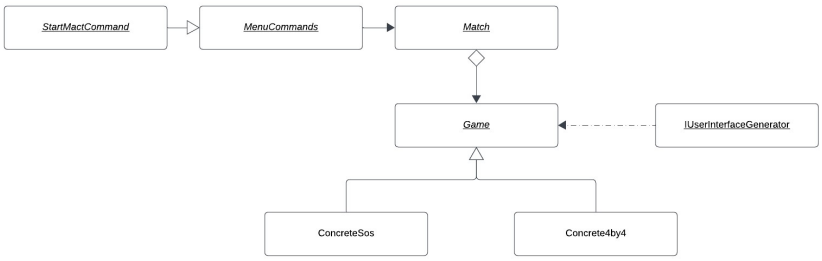
The user needs a way to select a game to play, then choose if a new game should start and if not, showcase the available saved games. Menus are stored in a list in a userInterface object. Each menu holds a dictionary with a menuEntry for key and a IMoveCommand for value. The

IUserInterfaceGenerator can be customized to create the appropriate menu and commands for the userInterface.



The template Design Pattern

A abstract class defining the basic methods that outline the sequence of steps of the game was used. Then each Game concrete class defines specific implementation for the abstract method. This structure facilitates the implementation of future games.



Design Principles

It's still a work in progress, but I tried to implement the principle of Single responsibility, limiting the scope of class, for example the class GameLoader is tasked to load and save games exclusively. It has not been completed for the game class, who have too much responsibilities. The liskov substitution was use to make most class work with any ConcreteGame(a Game), the commands, the board. The dependency inversion was used for example in the Match class: a game object needs to be added and the class interacts with the Game object. An other example could be TileDecorator class: object of this class can hold an object of class implementing the GameComponent interface (Piece, etc) and interact.

t with it. Overall, I tried to follow the SOLID design principle.

Reusable Classes/interface

System.Text.StringBuilder= »Board,LoadManager

System.Text.FileStream and StreamWriter=>LoadManager

Class Diagram

Objects Diagram

Sequence Diagram

The following diagram shows the methods called and object created once the user selects a game starts a new game. The game will be initialize, the players will be created and the game with start by launching the gameLoop. Note the call to executeCommands, it’s use is to play the saved move in case a game have been loaded from a save.

Launching the program

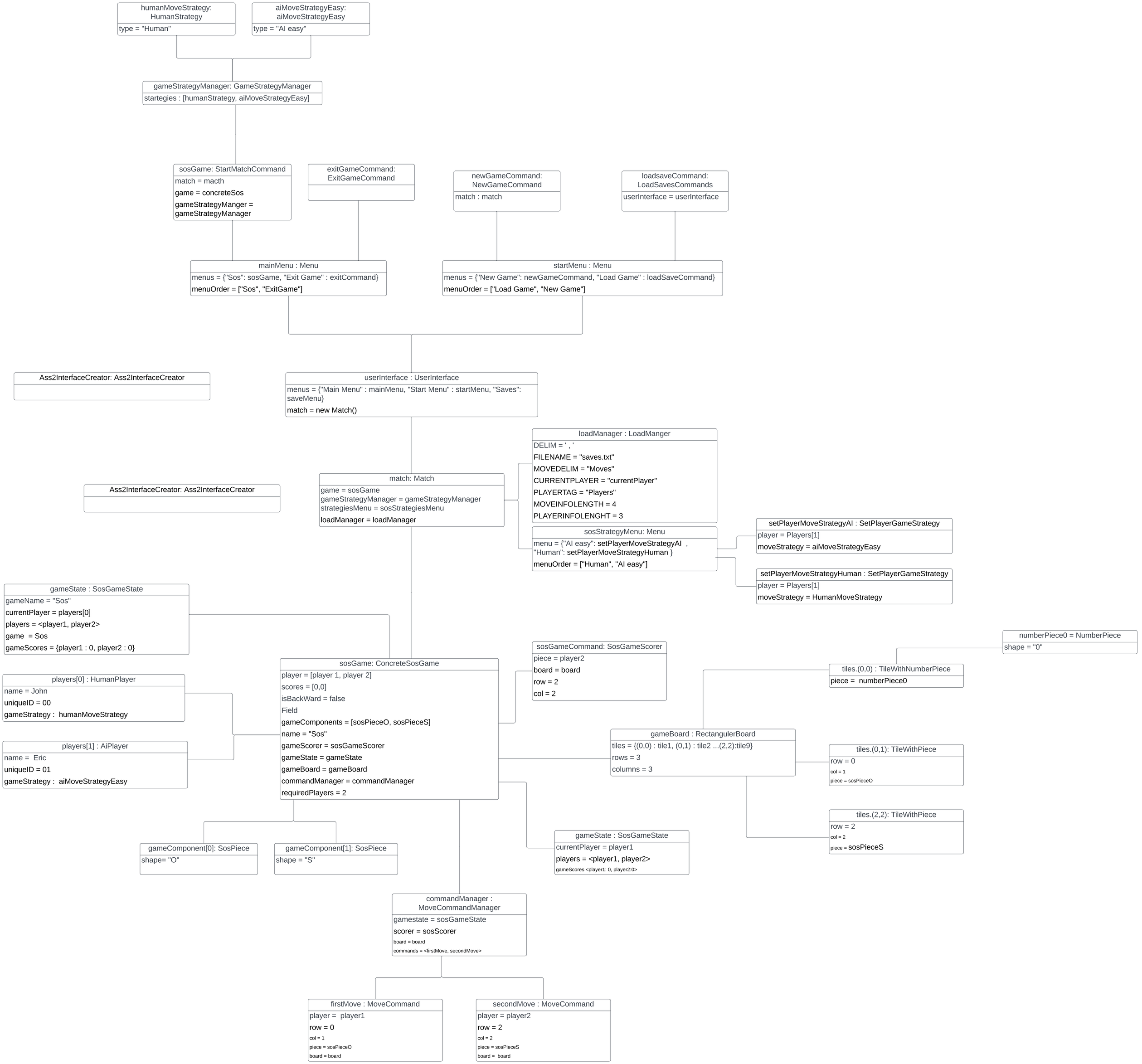
To launch the game, simply compile and run it. You can achieve this by opening your terminal, navigating to the folder containing the project file, and using the following commands:

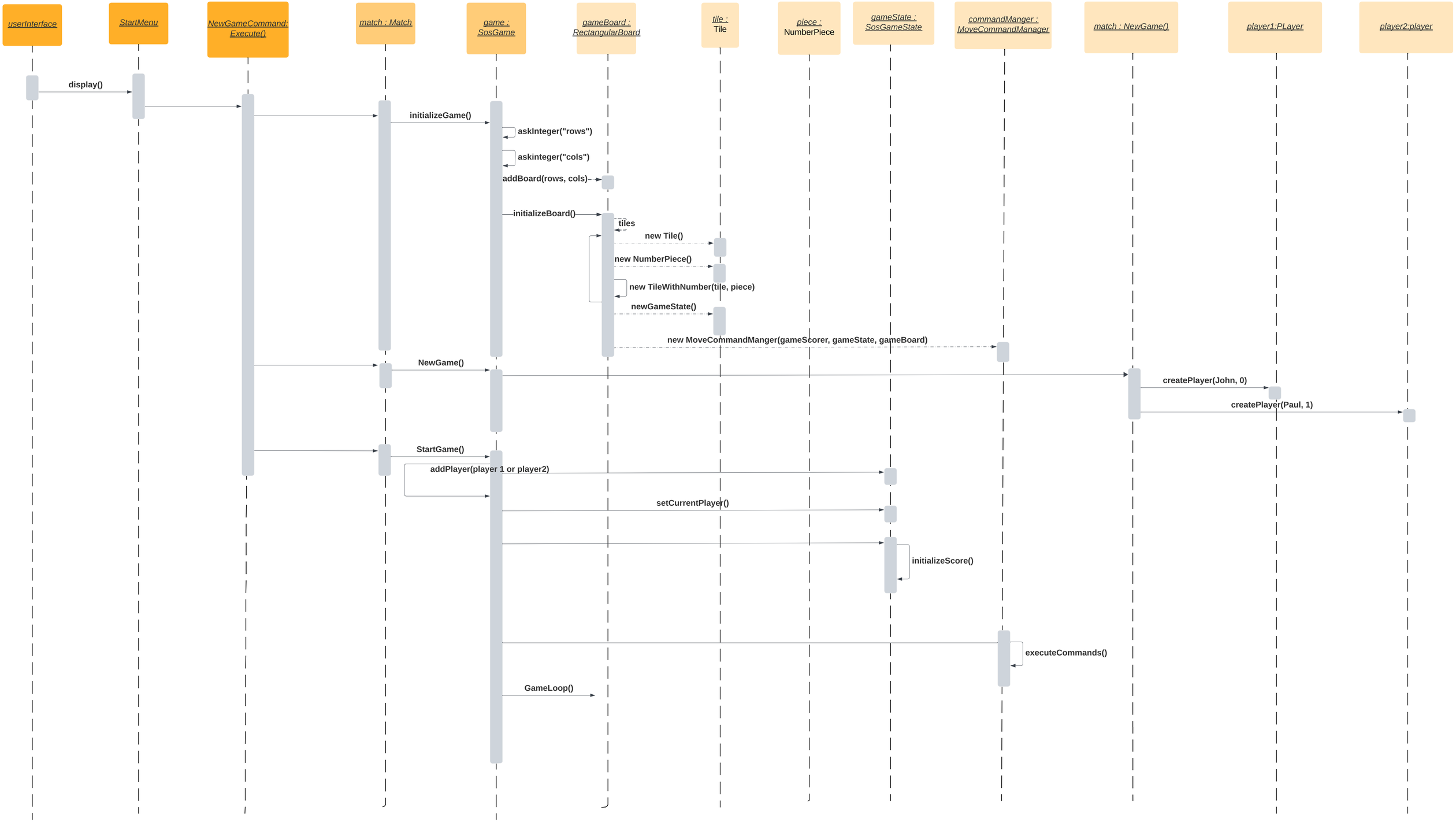
1. First, build the project with dotnet build.
2. Then, run the game with dotnet run.

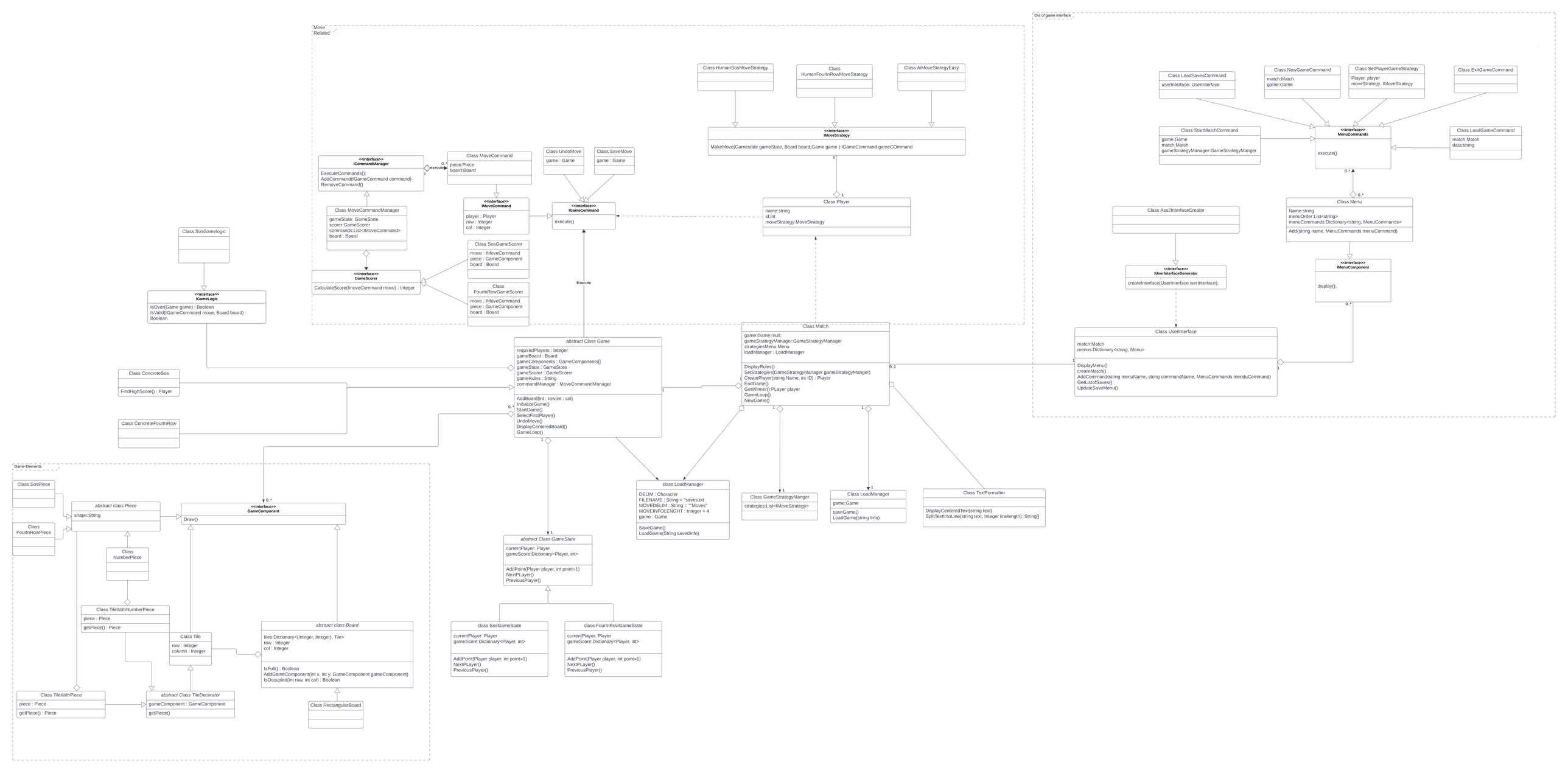
Alternatively, if you're using Visual Studio, you can open the program there and run it from the integrated development environment.

The entry point of the program is in the Program file, and the interface generator handles the creation of the necessary user interface elements. Please note that the program does not support fullscreen mode, as it doesn't anticipate the console size changing with different monitors.

When launching the game, the player should use arrow keys to select the game they want to play, press enter, and then choose "New Game." A prompt will appear, asking for the size of the game board, player names, and the type of game (human vs. human or human vs. AI). Once these settings are configured, the game will begin, and the first player, determined randomly, will be required to make their move. If the first player is AI, the AI's move will be automatically displayed on the board. To play a move, enter the number of the desired tile. -2 will save the game and -3 exit the program


humanMoveStrategy: HumanStrategy
type = "Human"
aiMoveStrategyEasy: aiMoveStrategyEasy
type = "AI easy"
gameStrategyManager: GameStrategyManager
startegies : [humanStrategy, aiMoveStrategyEasy]
sosGame: StartMatchCommand
match = macth
game = concreteSos
gameStrategyManger = gameStrategyManager
exitGameCommand: ExitGameCommand
newGameCommand: NewGameCommand
match : match
loadsaveCommand: LoadSavesCommands
userInterface = userInterface
mainMenu : Menu
menus = {"Sos": sosGame, "Exit Game" : exitCommand}
menuOrder = ["Sos", "ExitGame"]
startMenu : Menu
menus = {"New Game": newGameCommand, "Load Game" : loadSaveCommand}
menuOrder = ["Load Game", "New Game"]
Ass2InterfaceCreator: Ass2InterfaceCreator
userInterface : UserInterface
menus = {"Main Menu" : mainMenu, "Start Menu" : startMenu, "Saves": saveMenu}
match = new Match()
loadManager : LoadManger
DELIM = ' , '
FILENAME = "saves.txt"
MOVEDELIM = "Moves"
CURRENTPLAYER = "currentPlayer"
PLAYERTAG = "Players"
MOVEINFOLENGTH = 4
PLAYERINFOLENGHT = 3
Ass2InterfaceCreator: Ass2InterfaceCreator
match: Match
game = sosGame
gameStrategyManager = gameStrategyManager
strategiesMenu = sosStrategiesMenu
loadManager = loadManager
sosStrategyMenu: Menu
menu = {"AI easy": setPlayerMoveStrategyAI , "Human": setPlayerMoveStrategyHuman }
menuOrder = ["Human", "AI easy"]
setPlayerMoveStrategyAI : SetPlayerGameStrategy
player = Players[1]
moveStrategy = aiMoveStrategyEasy
setPlayerMoveStrategyHuman : SetPlayerGameStrategy
player = Players[1]
moveStrategy = HumanMoveStrategy
gameState : SosGameState
gameName = "Sos"
currentPlayer = players[0]
players = <player1, player2>
game = Sos
gameScores = {player1 : 0, player2 : 0}
sosGame: ConcreteSosGame
player = [player 1, player 2]
scores = [0,0]
isBackWard = false
Field
gameComponents = [sosPieceO, sosPieceS]
name = "Sos"
gameScorer = sosGameScorer
gameState = gameState
gameBoard = gameBoard
commandManager = commandManager
requiredPlayers = 2
sosGameCommand: SosGameScorer
piece = player2
board = board
row = 2
col = 2
numberPiece0 = NumberPiece
shape = "0"
tiles.(0,0) : TileWithNumberPiece
piece = numberPiece0
players[0] : HumanPlayer
name = John
uniqueID = 00
gameStrategy : humanMoveStrategy
players[1] : AiPlayer
name = Eric
uniqueID = 01
gameStrategy : aiMoveStrategyEasy
gameBoard : RectangulerBoard
tiles = {(0,0) : tile1, (0,1) : tile2 ...(2,2):tile9}
rows = 3
columns = 3
tiles.(0,1): TileWithPiece
row = 0
col = 1
piece = sosPieceO
tiles.(2,2): TileWithPiece
row = 2
col = 2
piece = sosPieceS
gameComponent[0]: SosPiece
shape= "O"
gameComponent[1]: SosPiece
shape = "S"
gameState : SosGameState
currentPlayer = player1
players = <player1, player2>
gameScores <player1: 0, player2:0>
commandManager : MoveCommandManager
gamestate = sosGameState
scorer = sosScorer
board = board
commands = <firstMove, secondMove>
firstMove : MoveCommand
player = player1
row = 0
col = 1
piece = sosPieceO
board = board
secondMove : MoveCommand
player = player2
row = 2
col = 2
piece = sosPieceS
board = board


userInterface
StartMenu
NewGameCommand: Execute()
match : Match
game : SosGame
gameBoard : RectangularBoard
tile : Tile
piece : NumberPiece
gameState : SosGameState
commandManger : MoveCommandManager
match : NewGame()
player1:PLayer
player2:player
display()
initializeGame()
askInteger("rows")
askinteger("cols")
addBoard(rows, cols)
initializeBoard()
tiles
new Tile()
new NumberPiece()
new TileWithNumber(tile, piece)
newGameState()
new MoveCommandManger(gameScorer, gameState, gameBoard)
NewGame()
createPlayer(John, 0)
createPlayer(Paul, 1)
StartGame()
addPlayer(player 1 or player2)
setCurrentPlayer()
initializeScore()
executeCommands()
GameLoop()


Move Related
Out of game interface
Game Elements
Class HumanSosMoveStrategy
Class HumanFourInRowMoveStrategy
Class AiMoveStategyEasy
<<interface>> IMoveStrategy
MakeMove(Gamestate gameState, Board board,Game game):IGameCommand gameCOmmand
Class LoadSavesCommand
userInterface: UserInterface
Class NewGameCommand
match:Match
game:Game
Class SetPlayerGameStrategy
Player: player
moveStrategy: IMoveStrategy
Class ExitGameCommand
Class StartMatchCommand
game:Game
match:Match
gameStrategyManager:GameStrategyManger
<<interface>> MenuCommands
execute()
Class LoadGameCommand
match:Match
data:string
<<interface>> ICommandManager
ExecuteCommands();
AddCommand(IGameCommand command)
RemoveCommand()
execute
Class MoveCommand
piece:Piece
board:Board
Class UndoMove
game : Game
Class SaveMove
game : Game
Class MoveCommandManager
gameState: GameState
scorer:GameScorer
commands:List<IMoveCommand>
board : Board
<<interface>> IMoveCommand
player : Player
row : Integer
col : Integer
<<interface>> IGameCommand
execute()
Class Player
name:string
id:int
moveStrategy:MoveStrategy
Class Ass2InterfaceCreator
Class Menu
Name:string
menuOrder:List<string>
menuCommands:Dictionary<string, MenuCommands>
Add(string name, MenuCommands menuCommand)
<<interface>> IGameScorer
CalculateScore(ImoveCommand move) : Integer
Class SosScorer
move : IMoveCommand
piece : GameComponent
board : Board
Class FourInRowScorer
move : IMoveCommand
piece : GameComponent
board : Board
Class SosGamelogic
<<interface>> IGameLogic
IsOver(Game game) : Boolean
IsValid(IGameCommand move, Board board) : Boolean
Execute
<<interface>> IUserInterfaceGenerator
createInterface(UserInterface iserInterface);
<<interface>> IMenuComponent
display();
Class UserInterface
match:Match
menus:Dictionary<string, Menu>
DisplayMenu()
createMatch()
AddCommand(string menuName, string commandName, MenuCommands menduCommand)
GetListofSaves()
UpdateSaveMenu()
abstract Class Game
requiredPlayers : Integer
gameBoard : Board
gameComponents : GameComponents[]
gameState : GameState
gameScorer : GameScorer
gameRules : String
commandManager : MoveCommandManager
AddBoard(int : row,int : col)
InitializeGame()
StartGame()
SelectFirstPlayer()
UndoMove()
DisplayCenteredBoard()
GameLoop()
Class Match
game:Game=null;
gameStrategyManager:GameStrategyManager
strategiesMenu:Menu
loadManager : LoadManager
DisplayRules()
SetStrategies(GameStrategyManager gameStrategyManger)
CreatePlayer(string Name, int ID) : Player
EndGame()
GetWinner():PLayer player
GameLoop()
NewGame()
Class ConcreteSos
FindHighScore() : Player
Class ConcreteFourInRow
class LoadManager
DELIM : Character
FILENAME : String = "saves.txt
MOVEDELIM : String = ""Moves"
MOVEINFOLENGHT : Integer = 4
game : Game
SaveGame()
LoadGame(String savedInfo)
Class GameStrategyManger
strategies:List<IMoveStrategy>
Class LoadManager
game:Game
saveGame()
LoadGame(string Info)
Class TextFormatter
DisplayCenteredText(string text)
SplitTextIntoLine(string text, Integer linelength): String[]
Class SosPiece
abstract class Piece
shape:String
<<interface>> GameComponent
Draw()
Class FourInRowPiece
Class NumberPiece
abstract Class GameState
currentPlayer: Player
gameScore:Dictionary<Player, int>
AddPoint(Player player, int point=1)
NextPLayer()
PreviousPlayer()
Class TileWithNumberPiece
piece : Piece
getPiece() : Piece
Class Tile
row : Integer
column : Integer
abstract class Board
tiles:Dictionary<(Integer, Integer), Tile>
row : Integer
col : Integer
IsFull() : Boolean
AddGameComponent(int x, int y, GameComponent gameComponent)
IsOccupied(int row, int col) : Boolean
class SosGameState
currentPlayer: Player
gameScore:Dictionary<Player, int>
AddPoint(Player player, int point=1)
NextPLayer()
PreviousPlayer()
class FourInRowGameState
currentPlayer: Player
gameScore:Dictionary<Player, int>
AddPoint(Player player, int point=1)
NextPLayer()
PreviousPlayer()
Class TileWithPiece
piece : Piece
getPiece() : Piece
abstract Class TileDecorator
gameComponent : GameComponent
getPiece()
Class RectangularBoard