Project Documentation
Project: Durak

Azamat Zarlykov

Table of Contents

User’s Guide

Introduction

The purpose of the project is to create an online game - **Durak.** This is one of the famous, well-known, Russian games; it would hardly be an exaggeration to say that almost all citizens of post-Soviet Union countries are familiar with it. The game itself is very entertaining and strategic at the same time; people like to play it because of the different variations and strategies that it has.
“**Durak**” means fool. The aim of the game is to get rid of all one’s cards. The player who is left with cards after everyone else has run out is a loser (fool or durak).

In this implementation, players can play with each other. In other words, it doesn’t support any computer bot that can play the game, yet. The number of players that can take part is 2-6. This game implements “Podkidnoy Durak” and “Durak with Passports” variations, which stand for “Classic” and “Passport” respectively in this program. Besides that, the project supports multiple attacking types: one side, neighbors, and all sides. For any variation and type, the game is played fairly. In other words, trying to play the wrong card and hoping to be not noticed by other players is not possible (this variation is also famous in some countries). For a detailed description of the rules check the next section - Rules Description.

The game is played online. Using the *Chrome* web browser, players can join and play Durak together. Since the project is not deployed yet and does not run on any domain, the game is played locally. So if you want to play with your friends, you need to start the server and go to `localhost:5000`, and your friends can join the same page with the same URL from another tab (more about it in the ‘How to compile` section).

Rules Description

In different countries, the game is played in different ways. However, it still preserves the main objectives and rules. The following link describes the "Podkidnoy Durak" very well: https://www.pagat.com/beating/podkidnoy_durak.html. Nevertheless, some parts in this implementation were adapted. The following list of things will explain what changes were made regarding the rules described in the link above in each variation.

Variation: Classic (Perevodnoy)

- **Players:** No option to play with a team. The users play individually.
- **Dealing:**
    - As an online game, there is no option for players to deal the cards to other players. When the game starts, it automatically distributes the cards to participants. The rest of the cards, if any left, are put on the side with one face-up card that represents the trump. In case all six players participate, the trump card is randomly picked by a computer.
    - Additionally, there is no need to show the lowest trump among players' hands to determine whose turn is to attack; the game will determine it automatically while dealing.
- **Attack and Defence:**
    - Conditions for the attack described in this section are implemented with the following change on the second condition of the rule page (ii. The total number of cards … never exceeds six) - the total number of cards played by the attacker is unlimited as long as the defender has cards. (There are cases when the defender, as well as the attacker, can have more than 6 cards)
    - Consequently, the defending player will succeed in defending the move if he manages to beat all of the cards on the table (opposed to the rule described - 'ii the defender succeeds in beating six attacking cards').
    - As it was mentioned before, while creating the game, the user can select the type of attack: one side, neighbors, all sides.
        - **Neighbors attack.** The principal attacker (the one who begins the attack) can decide to continue attacking or to Pass. In case of passing, the next opponent of the defender (left neighbor) starts the attack if they have the cards on the table. Once the current attacker is done, they will indicate it by pressing the Done button after which, the principal attacker can add more cards if they have any. Once two attackers are done, the bout is done.

        - **All sides attack**. The concept of attacking is the same as in the ‘neighbor attack’, but involves all the players (except the defender).
- **Drawing from the Talon:** Everything described in the URL applies, with the exception of exchanging the lowest trump with the trump in the talon that is face up.
- **The sequence of Play:** The player, once the cards are dealt, goes first if they have the lowest trump in the possession. In case no trumps are possessed by the players, the game decides randomly.

Other than the points noted above, everything stays the same.

For **Passport** variation, the general rules described above (Classic variation) are applied. However, it has its own differences that make the game more interesting as well as complicated.

Variation: Passport

- One round of the game is considered to be a full game itself described in “Podkidnoi Durak”.
- In the first round, every player **has** a passport of value six. That does not mean players physically own the card, but that if they, in any way, obtain one with this value during the game, they can (and must) use it to upgrade to the next passport.
- A player wins the game only if they manage to upgrade to the passport of value Ace and win a round using that passport.
- The following are the passport values: 6(starting round), 10, Jack, Queen, King, Ace(final round)
- So, in general, if a player has a card of value X (X being the value of their current passport) they are not allowed to defend or attack using that card. The only exception is if the card is of the trump suit, in which case they can use it to defend themselves. However, then they might lose their passport if they don't possess more than one card of value X (sometimes it might be worth doing so).
- The way to obtain the card X is either when dealing or taking the cards from the talon. One more way is if you get attacked by the card of value X. In such a case, you are not allowed to defend yourself, but you have to take the card, as well as any other that you might have been attacked with previously, in the same bout.
- **When do we use the passport?** In a simple case you will get attacked and manage to defend the attack successfully, and, by doing so, get rid of any other card in your hand that isn't a passport. In this situation, it is your turn to attack the next player, but since you only have passports left, you simply show them to the other players and hence, leave the round and upgrade to the next passport successfully (in case your passport was Ace, you win the overall game).
  After this, other players will continue to play the round, until only one player is left. That player cannot transfer to the next passport value, so in the next round played they will have the passport of the same value.
- In a different situation, you might not manage to obtain the passport during the round (this is unfortunately mostly based on luck), so, once you get rid of your cards, you stop playing that round, but without the ability to transfer to the passport of the next value.

Gameplay

Once the game started, the user is given an option to either: a) Create the game or b) Join the game (Figure 1). Based on the decision, UI varies.

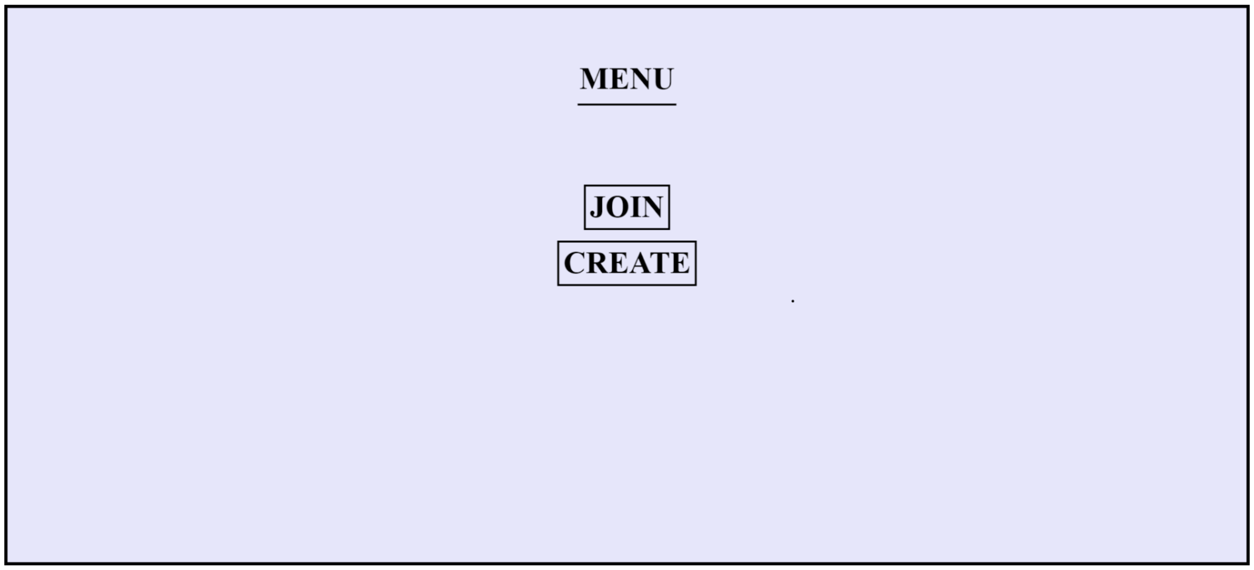


Figure 1. - Main Menu.

To create the game, you need to select the "CREATE" button. This action will move the game to the next, creating state (Figure 2). The user can select from each row one of the desired options for the game. The first row determines the *variation* of the game - classic or passport (check the **Rules Description** section to get familiar with the rules of the game). The second row indicates that the game is *fair* (some Durak games have variations with cheating, which this game does not include, so to avoid confusion this row indicates that the game is fair). The third row defines *type* - one side, neighbors, or all sides attacking. By default, the first items from each row are selected. Once ready to proceed, a respective button should be pressed.

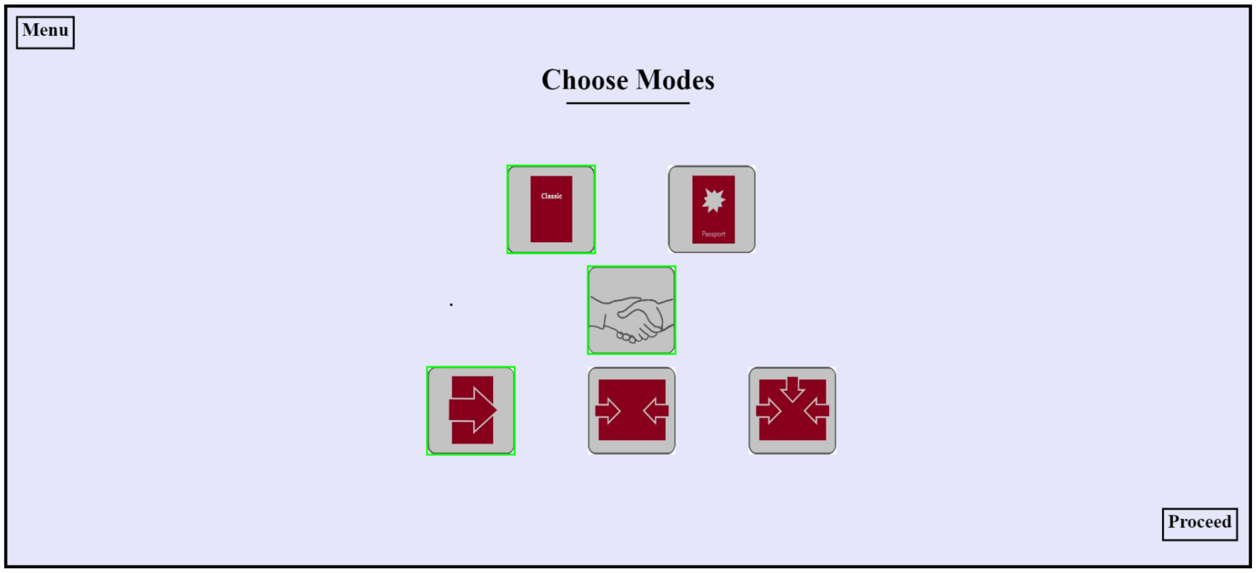


Figure 2. - Creating the game.

To join the game, press “JOIN”. The user can do so if one was created (only one game can be created at the time). Once joined, players can customize their profile i.e setting their names and selecting the icon (Figure 3). When the player is ready they need to press “Create”. This will move them to the Waiting Room.

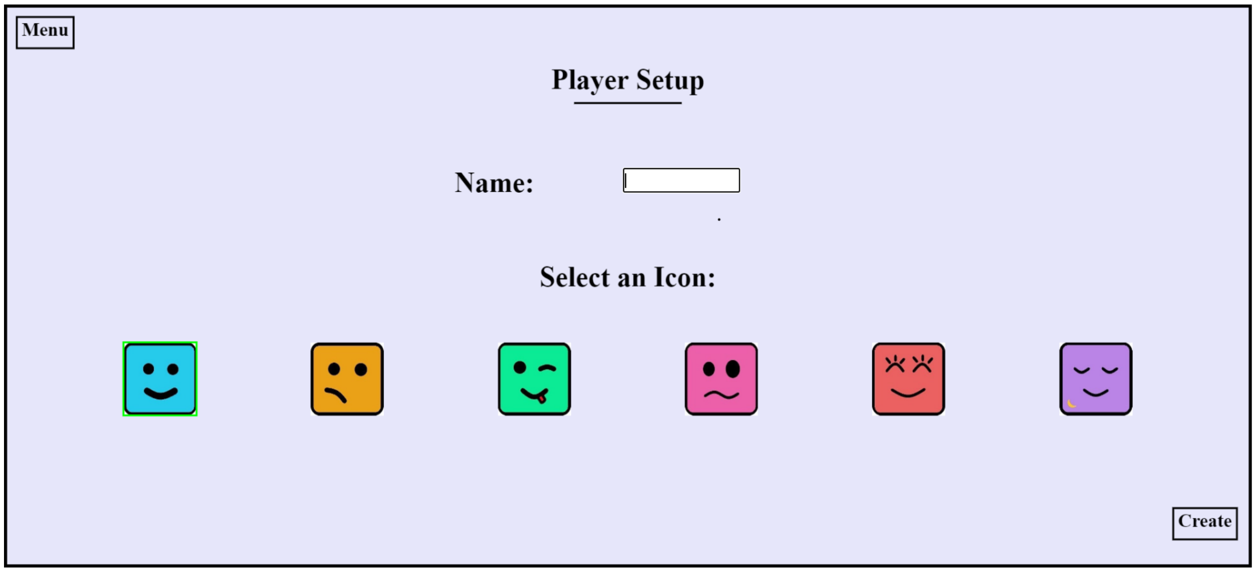
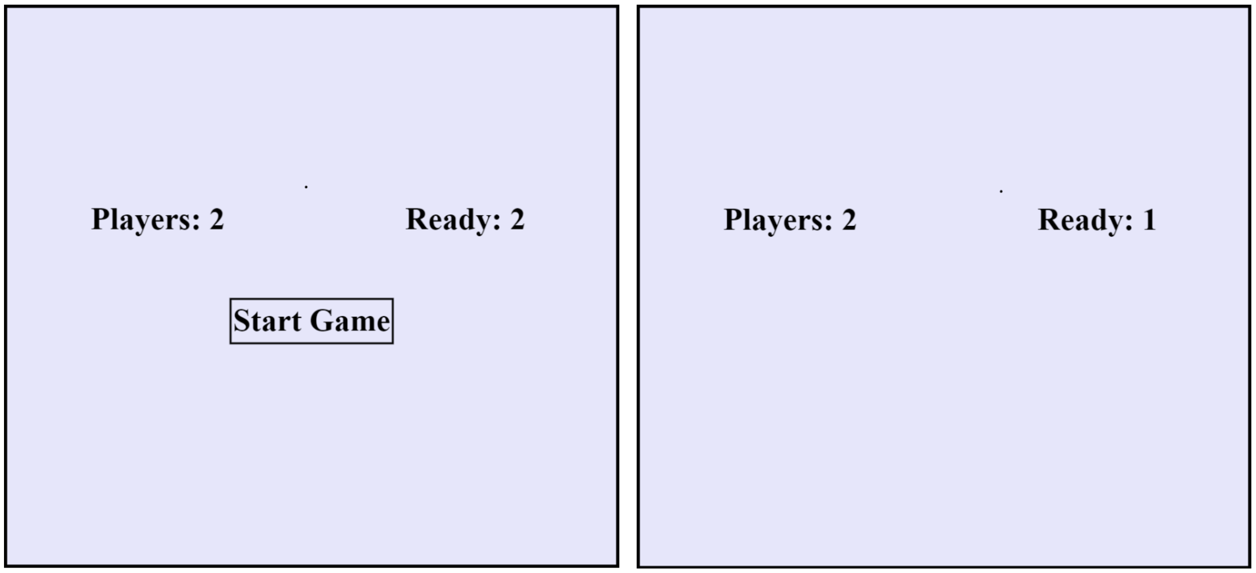


Figure 3. - Player Setup.

In this stage, the participants can see how many are in the game and how many are ready. The player who created the game (the one who chose the modes), can press the “Start Game” button that appears only when all of them are ready (Figure 4 below).



When the “Start Game” is pressed, all the players are moved to the table where they can start to play. However, the gameplay and view can vary based on the number of participants and settings that were chosen by the creator (in the following example the options are: Classic, one side attacking, and, as always, fair play).

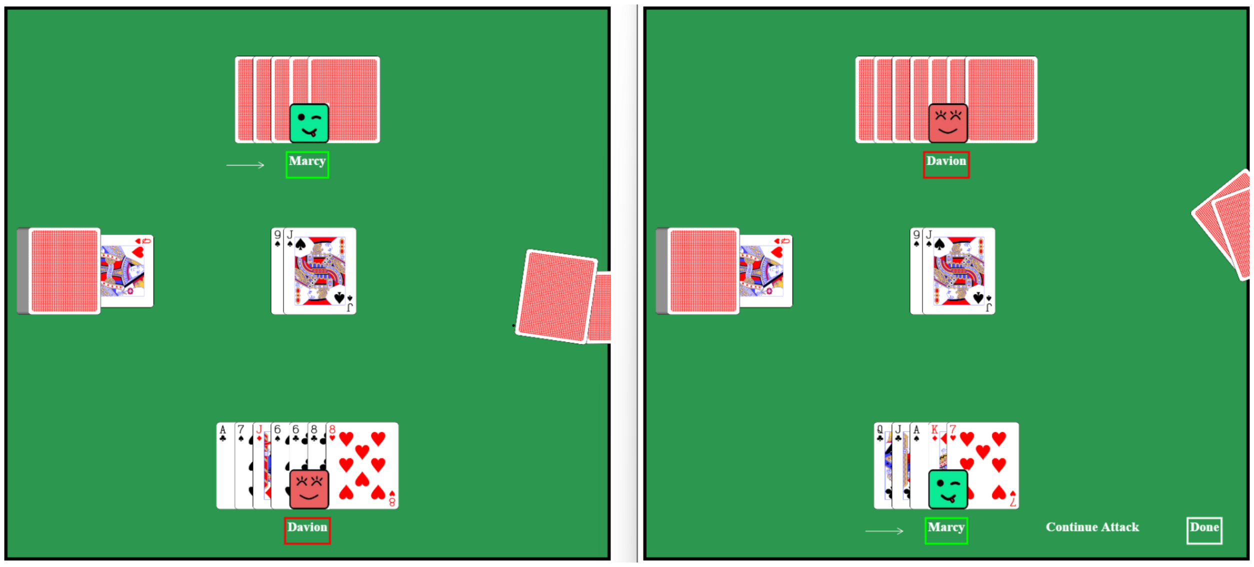


Figure 5. - Game.

In Figure 5, you can see the UI of the game table. For each turn, the program, with an arrow, indicates who will go next. Additionally, the colors show, as a helper, who is defending/attacking. The green frame around the name of the player implies that they are attacking. The red color, on the other hand, marks the defending one. Depending on the role on the right side of the players' names, the user can see options for a current turn (for the attacking player, it is the "Continue Attack" label and "Done" button).

Once the game is over, the users can witness the following state (Figure 6). The creator of the game can press the "Back To Lobby" button and move all the players back to the main menu state (Figure 1). Where the game can be created again.

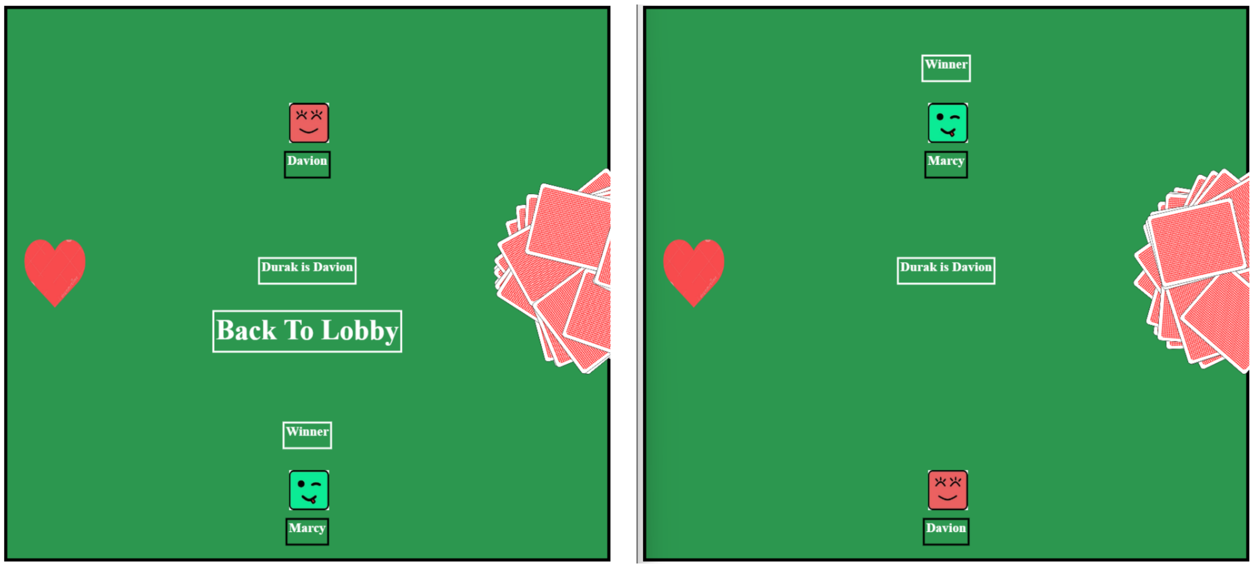


Figure 6 - EndGame.

Developer Documentation

How to Compile

Prerequisites:

1. ASP.NET 5.0
2. TypeScript 4.3 and up

Setup:

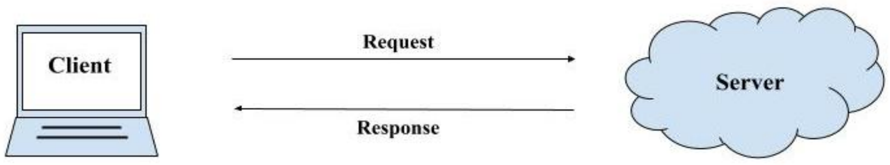
1. Clone the project from GitHub
2. Go to `DurakGame.Server` directory
3. Run the program using the `dotnet run` command (this will start the server)
4. Open the browser and type in `localhost:5000`

Since ASP.NET is cross-platform, the server of the game runs on any operating system.

Description of the System

Communication between client and server

To maintain the game with multiple users, the program uses the client-server architecture for communication with each side. The server is written with C# language and uses ASP.NET 5 for its implementation. The Client-side, on the other hand, uses TypeScript to handle all functionalities of the client's browser and represents the view using the canvas.



There is a two-way communication between the client and the server that uses the WebSocket HTTP. Once the connection is established the server-side keeps listening for any requests from the client.

The messages between two entities are exchanged using JSON format. Each message, exchanged between the client and the server, has an identifier “Message”, based on which certain actions are performed from either side.
The following table represents which actions, done by the user, trigger the client-side to send their respective JSON object to the server:

Description	JSON
Player decided to create the game	Client: {"Message": "CreateGame"}
Player chose the modes. The “GameSetting” key holds the value as an array. The indices represent the rows and the values represent which item/column.	Client:{ "Message":"GameSetup", "GameSetting":[1,0,1] }
Player setted up the profile. “From” indicates the player ID. “Name” and “Icon” indicate the settings picked.	Client:{ "From":0, "Message":"PlayerSetup", "Name":"Davion", "Icon":4 }
Player (creator) started the game.	Client: {"Message": "StartGame"}
Attacking player attacked with the card.	Client: {"Message": "Attacking", "Card":2}
Defending player defends with the card.	Client: {"Message":"Defending","Card":4}
Defending player takes the cards.	Client: {"Message": "Take"}
Attacking player stops the attack.	Client: {"Message": "Done"}
Player (creator) resets the game. This happens at the end of the game.	Client: {"Message": "ResetGame"}
Player displays the passport.	Client: {"Message": "Show Passport"}
Player (creator) starts next round of passport game.	Client: {"Message": "Next Round"}

The server registers all the requests from the client and processes them accordingly. For the purpose of deserialization of the JSON object from the client-side, the middleware namespace contains the `ClinetMessage` class. It stores the information that is sent from the clients such as message, from, card selected, game settings, icon, and the name selected.

The function `MessageHandle (ClientMessage route, WebSocket socket)` is responsible for handling the requests. The process of handling the specific request involves calling the function from the model that performs the changes in the game.

Once the request was processed and respective changes were made, the middleware provides the responses in the form of JSON. But instead of the “Message” keyword, the server provides a “command” identifier.

The table below shows responses from the server to the clients' requests with the corresponding JSON.

Description	JSON

Player connected to the game. This command is sent to all the players to set the total players currently in the game and game status. Besides, it gives an indication if this user will be playing the game.	SERVER: {
	"command": "SetTotalPlayers",
	"totalPlayers":1,
	"gameStatus":0,
	"isPlaying":true
	}

Player pressed create the game. This command indicates to all the players to change the provided fields and shows that players can start joining the game.	SERVER: {
	"command":"JoinGame",
	"gameStatus":1,
	"playerID":1,
	"sizeOfPlayers":2,
	"isPlaying":true,
	"isCreator":true
	}

Player gets to the setting up profile stage. availableIcons show which icons are available.	SERVER:{
	"command":"UpdateAvailableIcons",
	"availableIcons":[true,true,true,true,true,true]
	}

Player finished setting up profile. playerSetupOk indicates to the client if the username selected is unique.	SERVER:{
	"command":"UpdatePlayerSetup",
	"playerSetupOK":true
	}

Player (creator) started the game. The server sends the gameView object that represents the state of the game for the current player.	SERVER: {
	"command":"StartGame",
	"gameView": {...},
	"playerUserNames": [“Name1”, “Name2”],
	"takenIcons": [3, 2]
	}

Player attacked/defended/pressed Done button.	SERVER: {
	"command":"UpdateGameProcess",
	"gameView": {...}
	}

Player (defender) took the cards	SERVER: {
	"command":"TakeCards",
	"gameView": {...}
	}

Player (creator) resets the game (back to lobby)	SERVER: {"command":"TakeCards"}
Player disconnected from the game	SERVER: {"command":"Terminate"}

There are other commands that are sent to the client-side to indicate the errors, such as: player played the wrong card, the card was selected when it was not the player’s turn, selected card is a passport, etc.

As an update of the game round, besides the command identifier, the server sends the `gameView` object. This object is an instance of the `GameView` class that carries information about the current state of the game.
This is how `gameView` object is represented:

```
PlayerView {
        numberOfCards: number
        isAttacking: boolean
        allCardsPassport: boolean
        playerState: PlayerState
        Passport: PassportCard
}

GameView{
        playerID: number
        playersHand: Card[]
        trumpCard: number
        gameStatus: GameStatus
        playersTurn: number
        …
        playersView: PlayerView[]
        …
        attackingCards: Card[]
        defendingCards: Card[]
}
```

Stability of the game

All the necessary information regarding the game is kept in the model class `Durak.cs` (players, their hands, turns, etc.). Every time the game state changes, the `GameView` class reads from the model information used for each player. Therefore, rewriting the client code and trying to cheat the game would be hard since the GameView state sent, is relevant to each individual player.

Besides sending information about the game safely, the system maintains a persistent connection between the server and the client. Since the game requires all the players to be present during a game, situations like leaving are handled by the termination of the current game. In such a case, players, who are still playing, are moved back to the main menu since the game aborted.

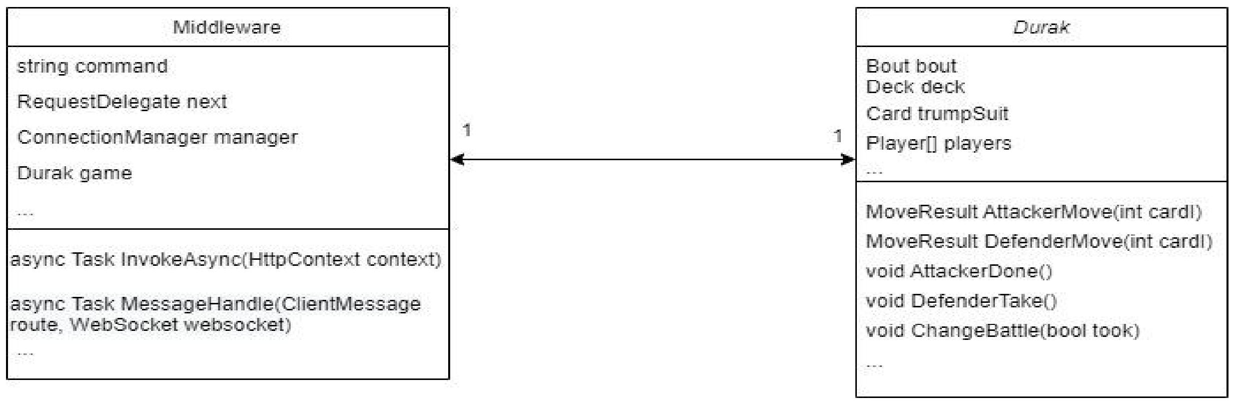
Regarding the simultaneous connection or access of information, the program is not thread-safe. During the process of writing the project, the decision was made that the game will not be accessed at the same time by multiple users. However, as further improvement, the thread can be assigned to each one separately, and whenever any action is made, locks can be assigned to vulnerable fields (that are shared between the threads).

Major Modules

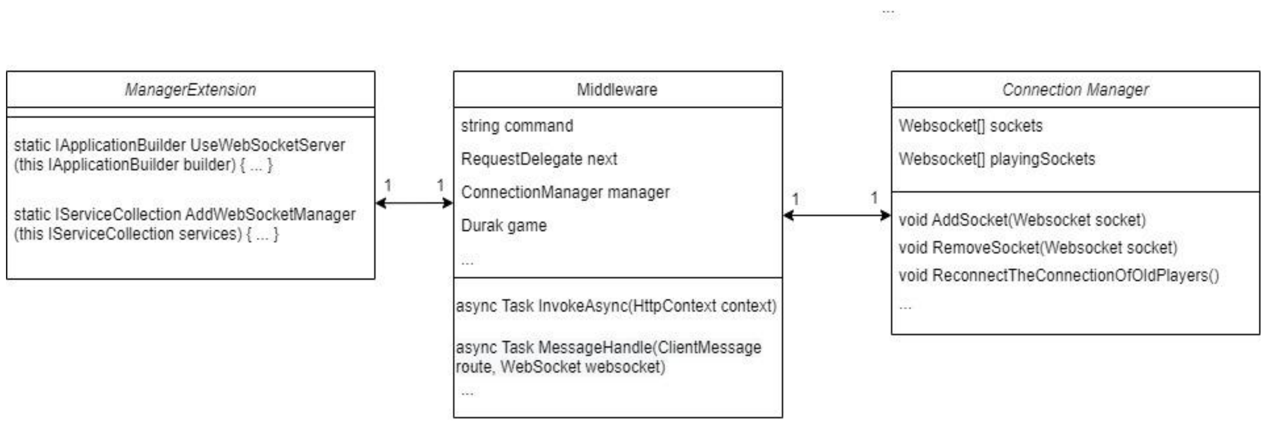
It is worth mentioning that the project uses MVC (Model-View-Controller) methodology to implement Durak where the Model is the logic of the game, the View represents the client-side code and the Controller part is written as the Middleware that takes care of handling the requests.

Server

The server side is separated into server and model (the game itself). It is a one-to-one relationship because the project allows running only one game at a time. This UML diagram describes the relationship between the two classes and depicts important fields that they have.

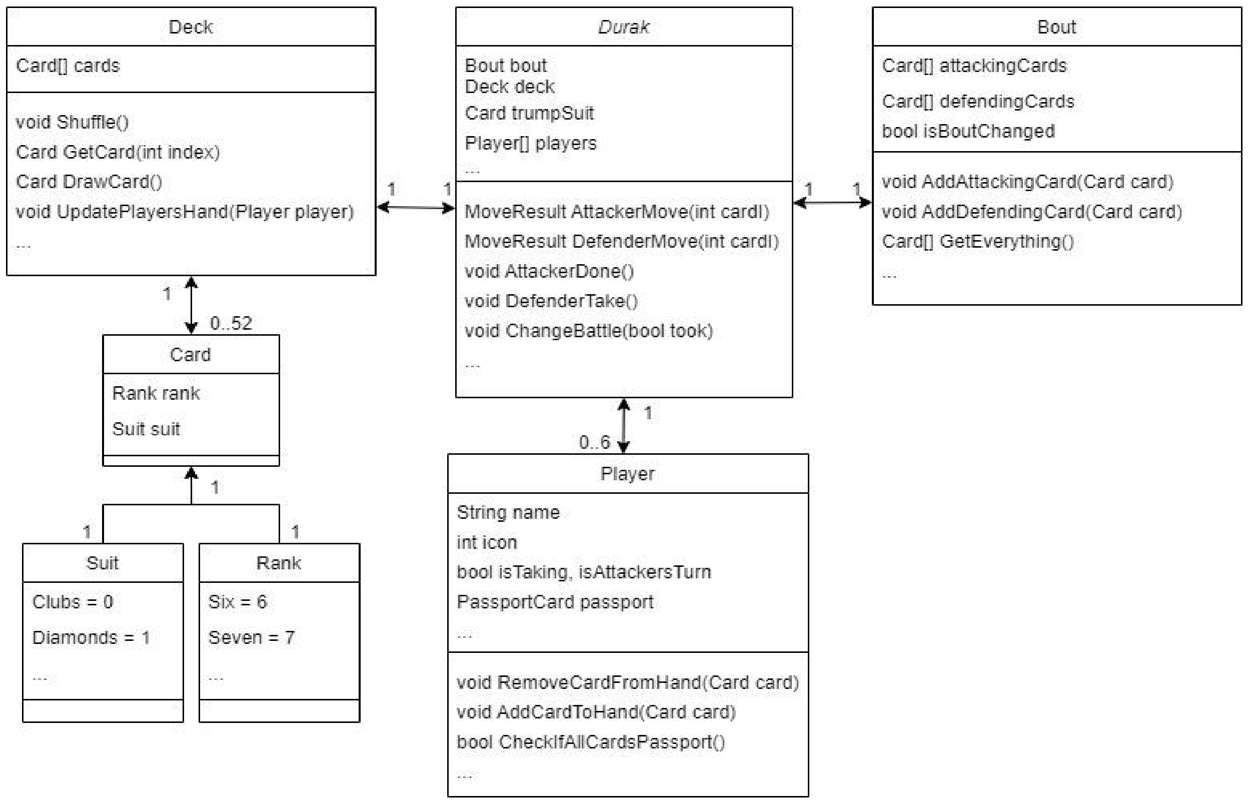

Middleware
string command
RequestDelegate next
ConnectionManager manager
Durak game
...
async Task InvokeAsync(HttpContext context)
async Task MessageHandle(ClientMessage route, WebSocket websocket)
...
1
1
Durak
Bout bout
Deck deck
Card trumpSuit
Player[] players
...
MoveResult AttackerMove(int cardI)
MoveResult DefenderMove(int cardI)
void AttackerDone()
void DefenderTake()
void ChangeBattle(bool took)
...

The server-side, which supports the communication with the client, has a namespace **Middleware** that captures classes such as **Middleware**, **ConnectionManager**, and **MiddlewareExtensions**. Its sole responsibility is to assemble a pipeline that handles requests and responses. This UML diagram illustrates the relationships between the classes inside Middleware namespace:


...
ManagerExtension
static IApplicationBuilder UseWebSocketServer (this IApplicationBuilder builder) { ... }
static IServiceCollection AddWebSocketManager (this IServiceCollection services) { ... }
1
1
Middleware
string command
RequestDelegate next
ConnectionManager manager
Durak game
...
async Task InvokeAsync(HttpContext context)
async Task MessageHandle(ClientMessage route, WebSocket websocket)
...
1
1
Connection Manager
Websocket[] sockets
Websocket[] playingSockets
void AddSocket(Websocket socket)
void RemoveSocket(Websocket socket)
void ReconnectTheConnectionOfOldPlayers()
...

The model, on the other hand, has many major components: **Durak**, **Bout**, **Card**, **Deck**, **GameView** (discussed in the "communication between client and server" part), and **Player**. They are separated into namespaces (eg. the class **Card** is within **DurakGame.Model.PlayingCards**).

As one of the important parts of the game, the Durak namespace, which encapsulates the Durak class, also includes essential Enum classes to describe the state of the game. Such enums are **AttackType**, **Variation**, **MoveResult**, **GameStatus**, etc. To illustrate the relationship between the classes in the model, take a look at the following UML diagram:



Client

Similar to the server-side, the client-side has a controller, **socket.ts,** that listens to the responses from the server. It allows identifying the messages sent and, consequently, calling the appropriate functions to display. Because of that, the controller has an instance of the view class (**GameView**) that draws the game based on the given state from the server.

The client-side is written as a single-page application. In other words, the source code uses the DOM to get the canvas from the HTML and draws the window on it. I decided to go with this option rather than working with multiple pages because I had more experience with a desktop rather than web applications. This was the main reason for my decision and I am quite happy with it, regardless of some difficulties encountered during the drawing/displaying process.

Different states of the game are drawn on the canvas. For that, as mentioned before, the program has the instance of the **GameView** that is responsible for drawing the whole client-side:

- Displaying different states of the game
    - States: menu, creating the game, setting up the profile, waiting room, and game table.
- Displaying messages
    - Error, Helper, and Information messages

Besides the above mentioned class, it also has some helper classes such as **MousePos** and **Button** that help to work with canvas.
The **MousePos** one is used whenever the user is selecting the buttons or cards. With the positions of the mouse click saved, the view can correctly differentiate what was selected.
The **Button** class facilitates the process of displaying them and registering the presses. It stores the information of the name and sizes and has a method that draws it.
Besides the view classes, the client-side has the interfaces and enums declared to process the objects sent from the server. Such declarations include replicating the objects from the server. For example, **GameStatus, PlayerState, Rank, Suit, GameViewInfo** (GameView from server), etc.

Even though the view part is wrapped in the class, the code could be more object-oriented.