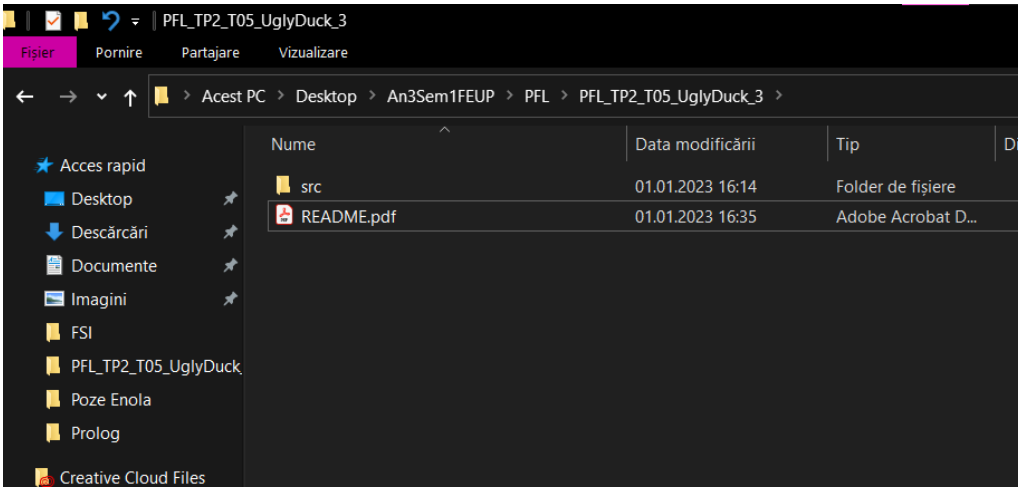
Ugly Duck Game Report

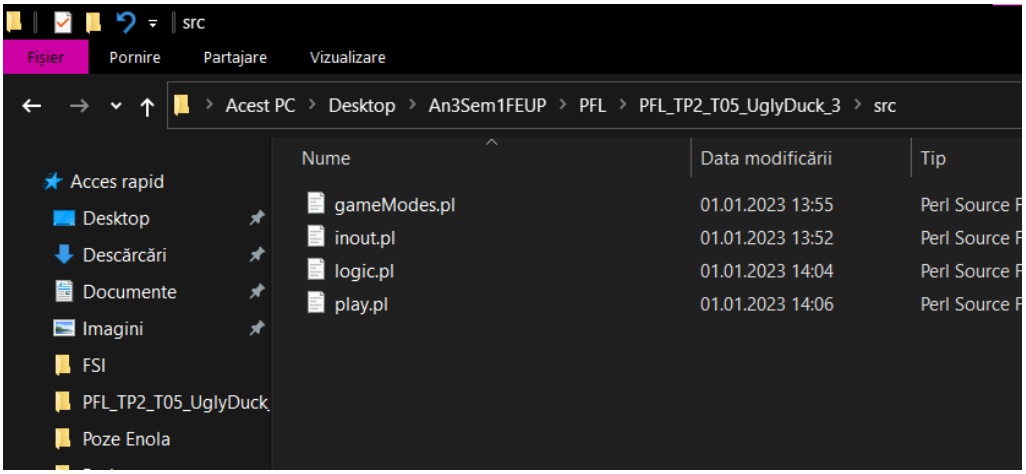
- **Game**: Ugly Duck
- **Group**: 3LEIC05, UglyDuck_3
- **Students**:
    - Raducanu Alexandru: up202202991
    - Serban Emilia-Bianca: up202202992

- **Work Percentage**

    Both students worked equally on this project (each 50%)

- **Installation and Execution both on Windows and Linux (Steps):**
    1. Open SICStus Prolog
    2. Click “File” -> “Consult”. Go to the location where you saved the directory PFL_TP2_T05_UglyDuck_3 and enter the directory.
    3. Enter the “src” directory which contains all the Prolog files (.pl) and double click on the “play.pl” file which is the main file of this project.





This is a short description of the .pl files:

- **gameModes.pl**: here you can find the functions we have created for the game modes “Player vs Player”, “Computer vs Computer” etc.
- **inout.pls**: the purpose all functions found here is for getting input from the user (e.g. to read the coordinates of a move) or to display the output(e.g. to display the board after a move). Also, there are some auxiliary functions that we created for making our work easier (e.g. trans).
- **logic.pl**: this file contains all functions that made this game possible. This is where the logic behind the game is implemented.
- **play.pl:** the starting point of the game.
- **bot.pl:** contains functions regarding the Greedy robot.

4. To start playing Ugly Duck, type “play.” in SICStus Prolog.

- **Game description**:

“*Ugly Duck is played on a **5x5 board.***
DUCK:

- ➔ *A duck moves one cell orthogonal and diagonal forward.*
- ➔ *Ducks capture diagonal forward. Captures are not mandatory.*
- ➔ *A duck reaching the last row is promoted to a swan.*

SWAN:

- ➔ *A swan moves one cell orthogonal and diagonal backward*
- ➔ *Swans capture diagonal backward. Captures are not mandatory.*

***GOAL** - Wins the player who first moves a swan into his first row.”*

These rules have been taken from the website:
https://www.di.fc.ul.pt/~jpn/gv/uglyduck.htm

- **Game logic:**
  The starting point of the game is the function **play/0**. It displays the Menu, which contains multiple options for playing the game.

➔ **Internal representation of the state of the game**

```
initialState(
    [
        [
        [wd,wd,wd,wd,wd],
        [e,e,e,e,e],
        [e,e,e,e,e],
        [e,e,e,e,e],
        [bd,bd,bd,bd,bd]
        ],
        w
    ]
).
```

Initial State of the game

As shown above, a state of the game represents a list with 2 elements: a list made of five lists which represents the **board** and a constant which is the **next player** that is about to move.
The board is made of five lists, each list representing a line of the board. Each line can have some of the following elements:

- **wd = white duck**
- **bd = black duck**
- **ws = white swan**
- **bs = black swan**
- **e = empty space**

In the file logic.pl, we have created 3 predicates that return a state of the game: **initialState(-State), intermediateState1(-State) and intermediateState2(-State)**.

➔ **Game state view:**

```
/*--- displayGame(+GameState)
Prints the Board and "announces" whose turn it is (White or Black).
*/
displayGame([Board, w]):-
                    displayBoard(Board),
                    format('~p, it is your turn!~n', ['White']).

displayGame([Board, b]):-
                    displayBoard(Board),
                    format('~p, it is your turn!~n', ['Black']).
```

displayGame(+GameState) prints the board by calling the function displayBoard(Board) and prints a message for the player that is about to move, depending on the second element of the state (the state is a list made of 2 elements: the board and the next player).
The output would look like the one below:

```
      |  a |  b |  c |  d |  e |
 ---------------------------------
   1  | wd |    | wd | wd | wd |
 ---------------------------------
   2  |    |    | ws | bd |    |
 ---------------------------------
   3  | bd |    |    |    |    |
 ---------------------------------
   4  |    |    |    |    | bd |
 ---------------------------------
   5  |    |    |    |    |    |
 ---------------------------------
 White, it is your turn!
```

```
displayBoard(Board):-
            format('~n      |  ~p |  ~p |  ~p |  ~p |  ~p |~30|~n', ['a','b','c','d','e']),
            format('~`-t~33|~n', []),
            trans(Board, NewBoard),
            displayBoardAux(NewBoard, 1).
```

```
displayBoardAux(_, 6):-!.

displayBoardAux(Board, N):-
                    N =< 5,
                    nth1(N, Board, Line),
                    format('  ~p   | ~p | ~p | ~p | ~p | ~p |~n', [N|Line]),
                    format('~`-t~33|~n', []),
                    NewN is N + 1,
                    displayBoardAux(Board, NewN).
```

```
trans([] , [] ).

trans([H | T], [H2 | T2]):-
    transAux(H , H2),
    trans(T, T2).

transAux( [] , [] ).
transAux( [e | T ], ['  '| T2]):-
    transAux(T,  T2).
transAux( [H | T], [H | T2]):-
    H \= e,
    transAux(T, T2).
```

displayBoard(+Board) prints the letter of the columns of the board, transforms the board (see t**rans** function) and then uses **displayBoardAux(+Board, +N)** which actually displays the board line by line, adding before it the number of the line. The 2nd function is first called with the argument N = 1 because it starts printing the board from the 1st line and stops when N = 6, as the board has 5 lines.
trans(+Board, -NewBoard) transforms the board which contains ‘e’ as empy space into a board containing ‘ ‘ (double space). We made this only for a more clear view of the board.

- **Menu system**
  After displaying the menu,the program waits for the user to write a number in the console, then it follows the choice of the user.

```
| ?- play.

******************** Menu **********************
*                                              *
*                  Options                     *
*                                              *
*         1          Player vs Player          *
*         2         DumbBot vs DumbBot         *
*         3          Player vs DumbBot         *
*         4          DumbBot vs Player         *
*         5        SmartBot vs SmartBot        *
*         6         SmartBot vs Player         *
*         7         Player vs SmartBot         *
*         8         DumbBot vs SmartBot        *
*         9         SmartBot vs DumbBot        *
*        10          Game Instructions         *
*         11           Exit the Game           *
*                                              *
*                                              *
************************************************
Choose your option. Please put a dot (.) after your choice.
```

If the user chooses a number from 1 to 9, then the game will begin. Otherwise, they can view the Instructions by typing ‘10.’ or they can exit the game by typing ‘11.’.

```
****************************************** Instructions ******************************************
*                                                                                                *
*                                Ugly Duck is played on a 5x5 board                              *
*                                                                                                *
*                  DUCK - A duck moves one cell orthogonal and diagonal forward                  *
*                  Ducks capture diagonal forward. Captures are not mandatory.                   *
*                        A duck reaching last row is promoted to a swan.                         *
*                                                                                                *
*                 SWAN - A swan moves one cell orthogonal and diagonal backward                  *
*                  Swans capture diagonal backward. Captures are not mandatory.                  *
*                                                                                                *
*                GOAL - Wins the player who first moves a swan into his first row                *
*                                                                                                *
**************************************************************************************************
Type "back." to go back to the menu or type "exit." to exit the game.:
```

Now, the program waits for the choice of the user: to go back or to exit the game.

```
******************** Menu *********************
*                                             *
*                   Options                   *
*                                             *
*         1          Player vs Player         *
*         2         DumbBot vs DumbBot        *
*         3          Player vs DumbBot        *
*         4          DumbBot vs Player        *
*         5        SmartBot vs SmartBot       *
*         6         SmartBot vs Player        *
*         7         Player vs SmartBot        *
*         8         DumbBot vs SmartBot       *
*         9         SmartBot vs DumbBot       *
*        10          Game Instructions        *
*         11           Exit the Game          *
*                                             *
*                                             *
***********************************************
Choose your option. Please put a dot (.) after your choice. 11.
Game closed!
yes
```

If the user decides to exit the game, the program displays the message “Game closed!” and stops.
There are multiple functions that we have created which use input/output predicates such as congratsMsg/0, readInputCoord(-SourceLine, -SourceColumn, -DestLine, -DestColumn) etc.

- **Input Validation**
  Because it was requested from us to use get_char/1, instead of read/1, we did our best in trying to do so.
  The 3 cases in which the user has to give input to the program are when:
  1. Choosing an option from the Menu.
  2. Choosing to go back or to exit the game while they are in the Instructions.
  3. Writing the coordinates of their moves.

  For 1 and 2, we managed to implement the program in such way that the user is announced when they are doing a mistake (writing a choice that the menu does not have).
  As far as 3 is concerned, we did not manage to announce the user of their mistake, but luckly, our program does not consider the wrong input as a valid move, therefore, the board will be displayed again as it was before the wrong move. We want to mention that we were not able to manage this due to the fact that get_char(?Char) messes up with the buffer. We created a function that clears the buffer, but, because of backtracking, the program will ask for '\n' uncontrollably. It is definitely something we would like to improve in the future.
  Our problem would have been solved using read/1, but we wanted to respect the requirements of our laboratory teacher.
  The main idea is that our program is robust against any invalid input.

➔ **Moves execution:**

Our team decided to encode a **Move** as a list of 4 elements. The first 2 elements represent the column and the line of the piece that is going to be moved and the last 2 elements represent the column and the line of the field on which the piece will be moved. **As an example, if you want to move your duck from column 'a' line 3 to the field on column 'b' line 4, the move will be [a,3,b,4] .**

We have created a few predicates that help us manage the validation and the execution of a move on the table. We have:

1) validMove/3 which receives a Board, a Player (w or b) and a Move. This function will succeed if the Move is a valid one. We can also use this function to generate all the valid moves from a Board.
2) makeMove/3 receives a Board, a Move and returns a NewBoard. The NewBoard is the result of executing the move on the given Board. This function does not do any checking for validation and correctness of the move.

3) **move(+GameState, +Move, -NewGameState).** Using the 2 functions described above, we managed to implement the requested predicate “move/3”.

➔ **List of Valid Moves:**

validMoves(+GameState, -ListOfMoves).

For this predicate, we used the “validMove/3” predicate described above. With a simple “findall” we managed to get all the valid moves from a given state of the game.

Note! *We did not use the Player as an argument, as it is already present on the GameState.*

➔ **End of the Game:**

Based on the rules, we decided to call a game over when there is a Swan located on the “home” line of the player. Also, the game is over when a player does not have any available pieces on the board. These 2 scenarios were approached in the **gameOver(+ GameState, - Winner)** predicate. There are more details in the commentaries in the source code.

➔ **Board Evaluation:**

Note! *For the following lines it is important to define what is a “home” line. The “home” line is the line on which all of the 5 ducks start the game for a player. Thus, for the “White” player, the “home” line is the line number 1. For the “Black” player, the “home” line is the line number 5.*

Every board configuration can be translated into a value from 0 to 1000.

We decided to evaluate the board in this way because we believed it is the most clear and concise way to do it.

Thus, a board that represents a win for the “White” player will have the maximum value, 1000.

A board that represents a win for the “Black” player will have the minimum possible value, 0.

If a board has the value 500 it means that there are no advantages for either of the 2 players. If the value is greater than 500, it means that “White” has an advantage. On the other hand, the “Black” has an advantage if the value of the board is lower than 500.

How to compute the value of the board?

After managing the extreme cases described above, we decided to evaluate a board based on the number and type of pieces each player has. The position of the piece is also important, as a Swan that only has 1 more move to make in order to win is much more valuable than a Swan that is farther away from the “home” line.

Thus, we decided to compute the values of the pieces based on the following table:

TYPE OF PIECE	POSITION	VALUE
Duck	“Home” line (line 1 for “White” player / line 5 for “Black” player)	**10 points**
	2nd line	**13 points**
	3rd line	**16 points**
	4th line	**19 points**
	5th line	**It is impossible to have a duck on the opponent’s “home” line. It will be automatically converted into a Swan.**
Swan	Opponent’s “Home” line (a Duck has just transformed in a Swan)	**30 points**
	4th line	**45 points**
	3rd line	**60 points**
	2nd line	**90 points**
	“Home” line	**It is impossible to have a Swan on the “home” line. It means that you have already won the match.**

Now, we have to sum up all the points for the “white” player and the “black” player. After that, we make the difference of the 2 sums and add 500. Now we have the value of any table computed.

➔ **Computer move:**

We have implemented 2 computer bots that are able to make valid moves on a board.
We have the first bot, the Dumb Bot, that randomly chooses a move from the list of valid moves and executes it. For this bot we have the predicates: **chooseMove/2**.
The second bot is the Smart Bot. It uses the evaluation method described above to choose the best move in every situation. For this bot we implemented the predicates: **pickBest/2, randomBestMove/2.** Every predicate is described in detail in the source code.

- **Conclusions**

There are a lot of ways we improved our knowledge while working on this project. We learned to use the Logical Programming Paradigm in many situations, such as: defining loops, managing input and output and using extra-logical predicates.

Concerning the game, we think that there are some ways in which we can improve our program:

1) Finding a better evaluation of the board, based on the positions of “allied” pieces.
2) Making the Smart Bot look “further” in the game tree. This can be useful because the best immediate move does not necessarily put you in the best position globally.
3) Making the game board look more intuitive and appealing.
4) Adding the option to change the size of the board.