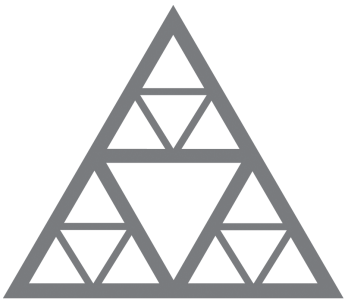

École des Ponts

ParisTech

TDLOG PROJECT: BLACKJACK

Arthur BRESNU, Émilie PIC, Guillaume HENON-JUST, Gauthier MERLEN

20. January 2023

1 Introduction

The purpose of this paper is to present a Blackjack game coded in Python. The game is a card game in which the objective is to be dealt cards with a higher count than the dealer's, without exceeding 21. The dealer can use a single deck of 52 cards or multiple decks from a device called a shoe. In the game, aces are worth 1 or 11, face cards are worth 10, and other cards are worth their face value.

To complete this project, several objectives were set. The main objectives were: to code the overall operations and architecture of the game to make it work for human players without bets, to create a graphical user interface, to add the ability to place bets, to code an AI that plays the game using counting methods, to test these counting methods, and to add additional features. To accomplish these goals, the game's architecture was assigned to Arthur and Guillaume, while Emilie and Gauthier were responsible for the graphical aspect of the game. The two parts were combined through collaboration. These two parts have been linked through joint work and have been divided into 4 main files and 1 file for the analysis of the AI data named analysis_proba. The 4 files are: game and model to describe the architecture of the game and its operations, display for the graphical part and main which gathers these files, initializes the parameters of the game and loops on the rounds of the game.
Throughout this report all the name of classes will be written like **This** and methods or functions like this.

Contents

2 Game modeling (back office)

2.1 Choice of the rules

The game can be played in various ways, and the following decisions have been taken into account. First, the game include the option to split cards. This decision is crucial for our project as the split function was one of the challenging parts to code to ensure it worked correctly. Additionally, we decided to include classic features such as burning cards, folding, and placing bets. Furthermore, the dealer can have between 4 and 8 decks. This is also a significant decision to observe how the AI's decisions change with the number of decks. The insurance rule has not been incorporated into our game. Finally, players have the option to double their bet face down.

2.2 Model: elementary blocks of the game

In this part, the elementary blocks of the game will be explained by describing the class of the file model and all the methods of it.

Class around Card First of all, there are the 2 classes which inherit of the class Enumerate. The first one is the **Color** for the color of the card and the second one is the **Rank** for the rank of the Card.
Then, the class **Card** has as inputs : a color, a rank and a value. This value encode the fact that Queen, King and Jack are worth 10 and As is worth 1 or 11.
To conclude with cards, we have the class **Deck** which contains a list of card, the number of deck (between 4 and 8) and the stop index (which is where the red card is put in the deck to know when re-shuffling the deck). There are other methods for this class like draw that draws a card, and if the stop index is reached shuffle the deck and generate another index (with the method associate).

Class around Player In this paragraph, the classes for the players are implemented. The first class is **Player** which has as inputs : a name, a hand (list of cards), an amount of money, a bet, a number of hands and an owner (we will precise it later). Other methods are : reset which resets all the attributes of the class to each round, draw which calls the draw method for the deck and add the card to the hand. This method also shows the hand with the method associated. There are also methods which analyse the hand of the player like:

- pair: returns true if the player has a pair
- value: gives the value of the player's hand
- double: refreshes the money account and bet if the player decides to double the bet
- even money and win money: refreshes the money and return the string of the result

This class has several child classes : **Dealer**, **HumanPlayer** and **AI**. The class **Dealer** assigns by default the name *dealer* to the dealer and the value to the money. This class also contains a method play which plays as a dealer in term of Blackjack rules. Another method is used to show the hand of the player without showing the second card (useful for all the round).
Then the class **HumanPlayer** takes as input the name of the player. The method show possibilities is used to get the choice of the player (stand, hit, double or split if he can).
In addition, the class **AI** codes the plays of an AI. The method choose option ai cheat will be discussed in the dedicated part.
Finally, **AliasPlayer** is a child class inherited of **HumanPlayer** and **AI**. This type of inheritance actually is a diamond architecture as we saw during the course. This class has been implemented for the split feature : a player can have several hands. Each hand will be hold by an alias player with an index of hand and an owner who can be a either a human or an AI (i.e. the real owner of the hand).

2.3 Game: interweaving of all the blocks

After this implementation of elementary blocks we have to link them all together, and this is exactly what the class **Game** does. It takes as attributes a list of player (players), a dealer, a deck and other attributes which will be discussed in the upcoming sections (as all the methods counting cards). As method we have:

- choose bet: asks for each player his bet and updates it
- first distribution: distributes cards as in casino with all the Player methods associated
- reset: resets all the game between each round and calls all the methods named reset of the other classes
- results: computes and returns the string of the result for each player
- play player: Firstly, it displays the player's hand. Then, if it is a human player, this latter chooses his action. On the contrary, if it is an AI player, a function determines his action. Finally, the function updates player's attributes according to the chosen action.
- play round: Each player chooses his bet, then receives his cards and plays (with the function above). After that, the dealer plays and it finally displays the results.

2.4 Focus on a technical aspect: Split

We thought it would be interesting to focus on the split feature since it brought some algorithmic complexity into our code. We implemented it this way :

In the play_player function, if the player decides to split, we remove the current player from the *players* list. Then, a new variable is created *index*. It represents the number of times the players split, that is to say the number of hands added to the game. To do so, we modified the play_player function to be a recursive one.
If the current player is a human or an AI : his number of hands is increased by one and his money is updated to money minus his current bet. Two AliasPlayers which represent his two new hands are created (with as owner the current player). They each receive the split cards as their new hand and we insert them at the right position in the players list. Then, the index is increased by the value of the play_player function for each of them (here it's the recursive nature of the function). After all, the index is returned (keep in mind that if there is no split this function returns 0 because no hand has been added).
If the current player already is an AliasPlayer, all the same as above is done except the current player is replaced by his owner (player.owner).

In the play_round function, we create an *nb_hand_added* variable keeping in track the difference between the number of players and the number of playing hands initialized at 0. Then, instead of going through the original players list, we go through a copy of the initial list (since the original one will be modified during the loop if some AliasPlayers are added to it). During this loop, *nb_hand_added* is increased by the value returned by the play_player function. It takes as arguments the current player and his index in the original players list ($i + nb\_hand\_added$).

3 Graphical interface (front office)

The ultimate goal of our project is to create a user-friendly interface for one or more people to play the game easily. Once the model has been coded, it was possible to play in the terminal. The objective of this phase was to connect the model to the display module, so that the player can interact with a graphical interface instead of the terminal. This is illustrated in the architectural diagram we could see in Fig11.

3.1 Graphical tools

We have chosen to use the *Pygame* library in Python. First, we focused on developing the necessary functions and classes for the display itself.

Three classes were created :

- **Rectangle** :it contains functions that allows displaying a rectangle of a given size and color, at a specified position and with a given text.
- **Button** (Rectangle) : it is a rectangle on which one can click.
- **Image** : it allows to load, display and resize an image.

We also implemented a function that simulates the motion of an object on the screen.

Finally, we learnt how to manage the events with the *Pygame* library, e.g. when the player clicks on the screen or on the close button, or also when he enters text.

3.2 Link between the model and the graphical interface

Once we knew how to display objects and manage events, we had to link the model and the graphic interface. We proceeded as follows :

$\Longrightarrow$ First, we get the information on the state of the game from the model.

$\Longrightarrow$ Then, we display the state of the game.

$\Longrightarrow$ Afterwards, the player does something and we translate the event to modify the model.

$\Longrightarrow$ Finally, we do it again.

The game is divided in three parts :

- the initialization of the game : the choice of the number of players, the type of each player (AI or human) and his name if it is a human.
- the game : each payer chooses what he wants to do according to his hand (stand, split, hit, double) and then the dealer plays.
- the end of the game : it shows which players won and which players lost and proposes to continue the game or not for the human players who still have some money.

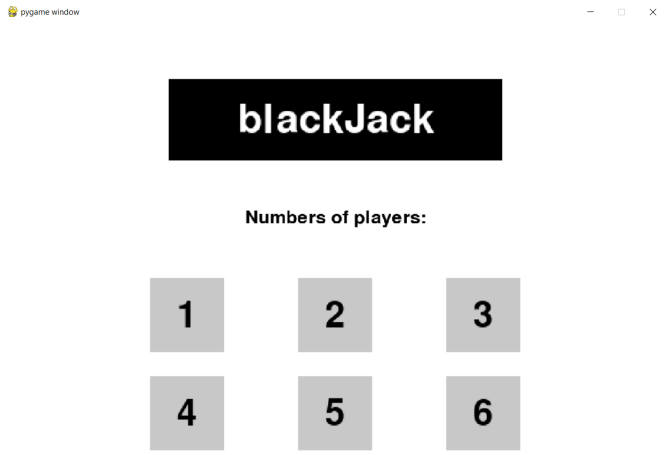

Figure 1: Home page

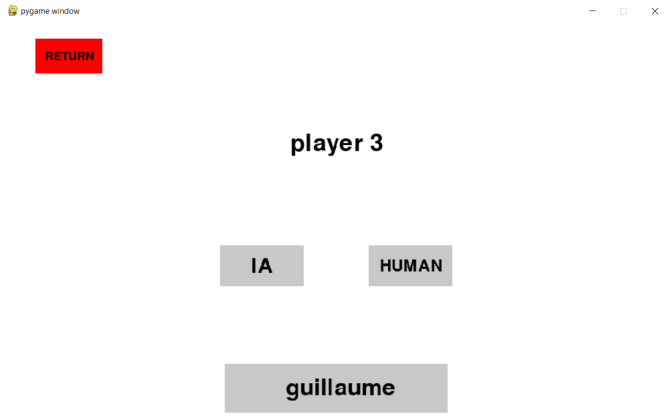

Figure 2: Characteristics of a player

Figure 3: Choose bet

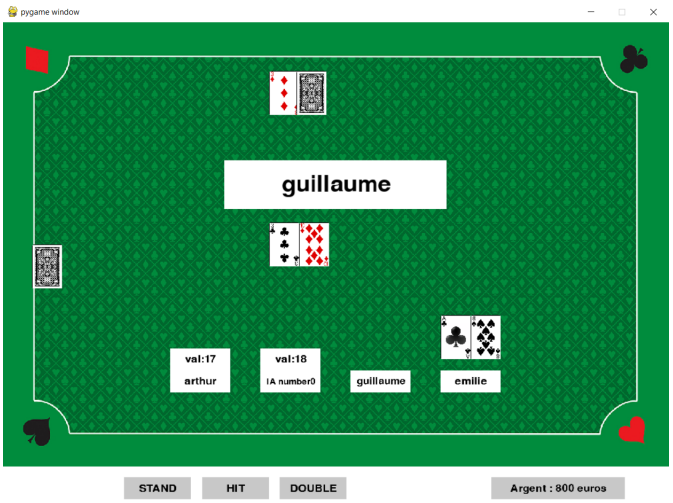

Figure 4: Main game

4 AI

4.1 Motivation and choice of counting methods

To go further, we decided to implement an AI to play against. We first implemented a naive AI following the dealer's game-play : to always hit if our hand value is smaller than 17, to stand otherwise. To improve this basic AI, we implemented a card counting system and we made our AI play following usual Blackjack card counting methods. The idea was to estimate the efficiency of these methods by simulating an important number of games with these AI and comparing their results.

Firstly, we have researched these different methods and decided to implement the following ones :

- Hi-Lo :

2,3,4,5,6	count +1
7,8,9	count 0
10,J,Q,K,A	count -1

- KO :

2,3,4,5,6,7	count +1
8,9	count 0
10,J,Q,K,A	count -1

- Omega II :

2,3,7	count +1
4,5,6	count +2
8	count 0
9	count -1
10,J,Q,K,A	count -2

Counting cards allows the player to keep in mind the order of magnitude of the number of high cards in the deck of cards. Thus, if the global count is high, the player will have interest in hitting because he will have little chance to draw a high value card that would make him burn. On the contrary, if the global count is low, the player should pass.

To implement these methods, we add to the **Game** class a field count, keeping in track the count during the whole game. Each time a card is drawn, we call the increaseCount() method. Depending on the card counting method chosen beforehand, this function updates the total count. When the dealer reaches the red card in the card shoe and shuffles back all the cards, the count has to be reset to 0.

4.2 Analysis of results and critics

We didn't manage to find a serious scientific paper on Blackjack counting methods demonstrating and measuring their efficiency, but we were expecting that an AI playing with these counting methods would be more successful than a naive AI or even a human player.
To confirm or refute these expectations, we simulated a large number of games for each counting method and observed the results. Then, we implemented a first function to plot these games and to observe the evolution of the player's money :

Here, the sample is obviously too small to make any conclusion. However, we can observe that the games seem to be longer with counting methods, and that the player reaches higher level of money owned during the game.

To measure the efficiency of these methods, we chose to count on an important number of games the ratio of the game where the player exceeds its initial stake multiplied by a pre-determined ratio. This computation corresponds in the code to the counting_method_efficiency_test() function in the data module. Note that we keep a track of all the evolutions of these games in a JSON file. The results we obtained are a bit disappointing : with all of these methods, the AI only exceeds 1.5*times its initial stake 15% of the time. With the naive gameplay, this figure drops to only 12%. The gap does not seem large enough to prove the effectiveness of these methods.

5 Tests

We implemented some global test to ensure that our code continues to work as it evolves and as we make new implementations. To do so, we implemented 3 tests functions :

- global_game_test() : This function globally tests the files model.py and test.py, running a game with 1 AI, 1 Human and a game with 4 Human players and 4 AI on 10 rounds. Then, we successively test each counting method with an AI. The program should first successfully run, and then the programmer can check on the terminal if the players properly played, following rules.

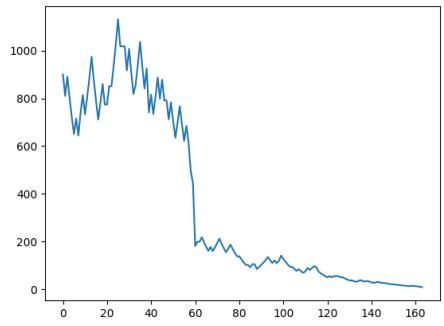

Figure 5: Naive game-play

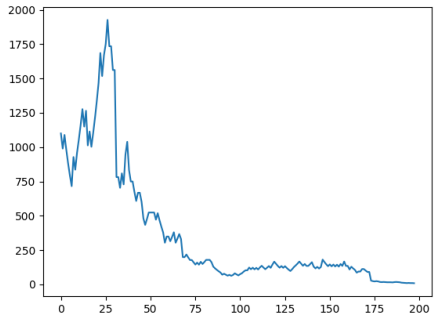

Figure 6: Hi-Lo method

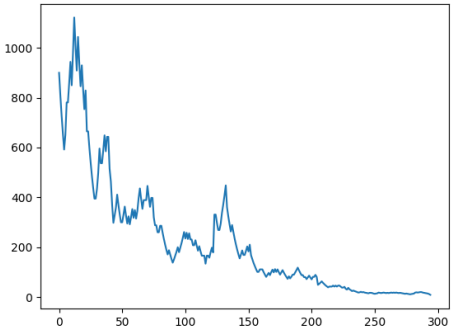

Figure 7: KO method

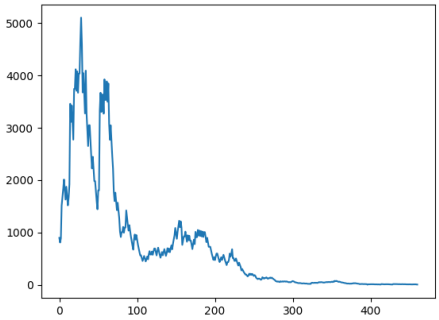

Figure 8: Omega II method

- split_test() : This function tests the split feature, in extreme cases where players split a lot of times their hand. It tests it on AI and Human Players. NB : players stop splitting when they reach 20 hands. We chose arbitrarily 20 hands since it is highly unlikely and should be sufficient to detect any bug.
- counting_method_efficiency_test() : This function returns the frequencies (12% and 15%) described in the previous section.

For the first function to run, it is necessary that all the methods of the src module runs properly, since this function simulates all possible cases. Then, to ensure that the rules are properly followed, the programmer has to check in the JSON files if the game goes well according the rules.
The second function makes sure that the split implementation, which can lead to complications due to the creation of aliases, works correctly.
The last function ensures that the counting methods are well implemented which means that an AI playing with these methods has better results than the AI with naive game-play.

6 To go further

6.1 Features to add

Overall, we achieved the goals we set at the beginning for our project. However, we have now the intention to pursue it with some features despite the fact that we know clearly that some of them will be hard to implement in this short amount of time. Here is a small list of features which could be great to add:

* Sound: We all know that the user's experience on a game is visual but also auditory. That is why we want to add game sounds like home page sound, background sound, click sound for each button, sound when a card is draw, different sounds for results...

* Token: For now, a click on a token for a bet is definitive. In fact, the bet is showed as a string and we want to change that. We want the token to slide in the middle when you click on it and slide on the other way to remove it, if the player made a mistake.

* Gold ticket: The goal of this ticket is to capt the user on our game. He will win a gold ticket for each Blackjack or for each series of 5 wins. This ticket will provide him the tips of our card counting AI on his own round. It will be like a boost for the user.

* Online mode: This is just an idea because we doubt about our ability to make it before the deadline. But we think it could be great to have a leader board which allows a player to see at the beginning of a party who went the farthest (same amount of money at the beginning, rush the best amount).

6.2 Code and project review

When we look backward and we take hindsight we could see as positive point in our project:

+ The react window depends on the width of the window so the window is adapted to any kind of monitory.

+ The ability we all get in term of clean coding and all the knowledge we put in practice to handle this project.

+ Our capacity to make a project with a standard structure and architecture. Also, we learned about the creation of all setting parameters for a package and a generation of a HTML documentation's page with Sphinx.

+

Here are some negative points we could improve (maybe during the gap between now and the deadline):

– A complex part of the project was to bring the architecture and graphics together. We probably should have worked more together from the beginning to know our needs (input and output of functions), our used functions...

– Due to how we managed this project, we didn't think at first about how we could discuss with the user efficiently. So we have for the graphic part almost only functions and no real architecture as we have for the game. To fix that we think about a class **App** which has a party and a window as fields. This window would be an instance of a class **Window** which is composed of all window parameters and as method all the functions we have in the file display.

– Maybe we should have done a Monte-Carlo AI with reinforcement learning to improve our model because counting method does not seem really efficient.

– For the tests, we found it hard to test all the small functions in our game because a lot of them are interconnected or extremely basic. This is why at least we just globally tested our program but it's not a solution so we will have to fix that for the deadline.

7 Conclusion

In conclusion, this project provided valuable experience in various aspects of software development. We were able to learn and use new tools such as Trello, which helped us to manage the project more effectively. We also gained experience working with the Pygame library and using Sphinx for documentation. Additionally, this project helped us to learn how to structure a project, work in a group on a long-term project, assign tasks, use Github in an organized manner, create a schedule and set goals. Overall, it was a challenging and rewarding experience that helped us to improve our skills in software development. Finally, we would like to come back to this project personally :

- Gauthier : Being the only one in the Industrial Engineering sector of the group, I have more difficulty in terms of coding. Despite this, I learned a lot with my classmates who gave me simpler tasks but which allowed me to understand little by little how to code properly in Python, how to use classes well, and how to help them to make the project progress. Moreover, my classmates did not hesitate to help me understand their code parts, to code, to explain to me what they had learned in other courses that I had not participated to. I've learn a lot with Emilie concerning pygame library for the graphical part and also with Guillaume for the sphinx file.

- Emilie : In this project, I was in charge of the graphical interface with Gauthier. It was a good expérience because we learnt how to use a new library that we did not knew anything about. It was also interesting to work in groups. We learnt how to construct a project, divide the tasks and then merge all of our works. I think this latter, that is to say linking the model and the graphical interface, was the most difficult for us and this is the point I would have improve if we had more time.

- Guillaume & Arhur: In this project, we mainly focused on the back-office part, implementing the basic structure of the game, adding the split feature, implementing the cards counting methods and analysing the AI performances. It was really enriching for our because we discovered a more structured way of coding. Moreover, it was our first big group IT project, which raised new organizational challenges. All this allowed us to discover new very useful tools such as sphinx, UML and Trello.

Appendix

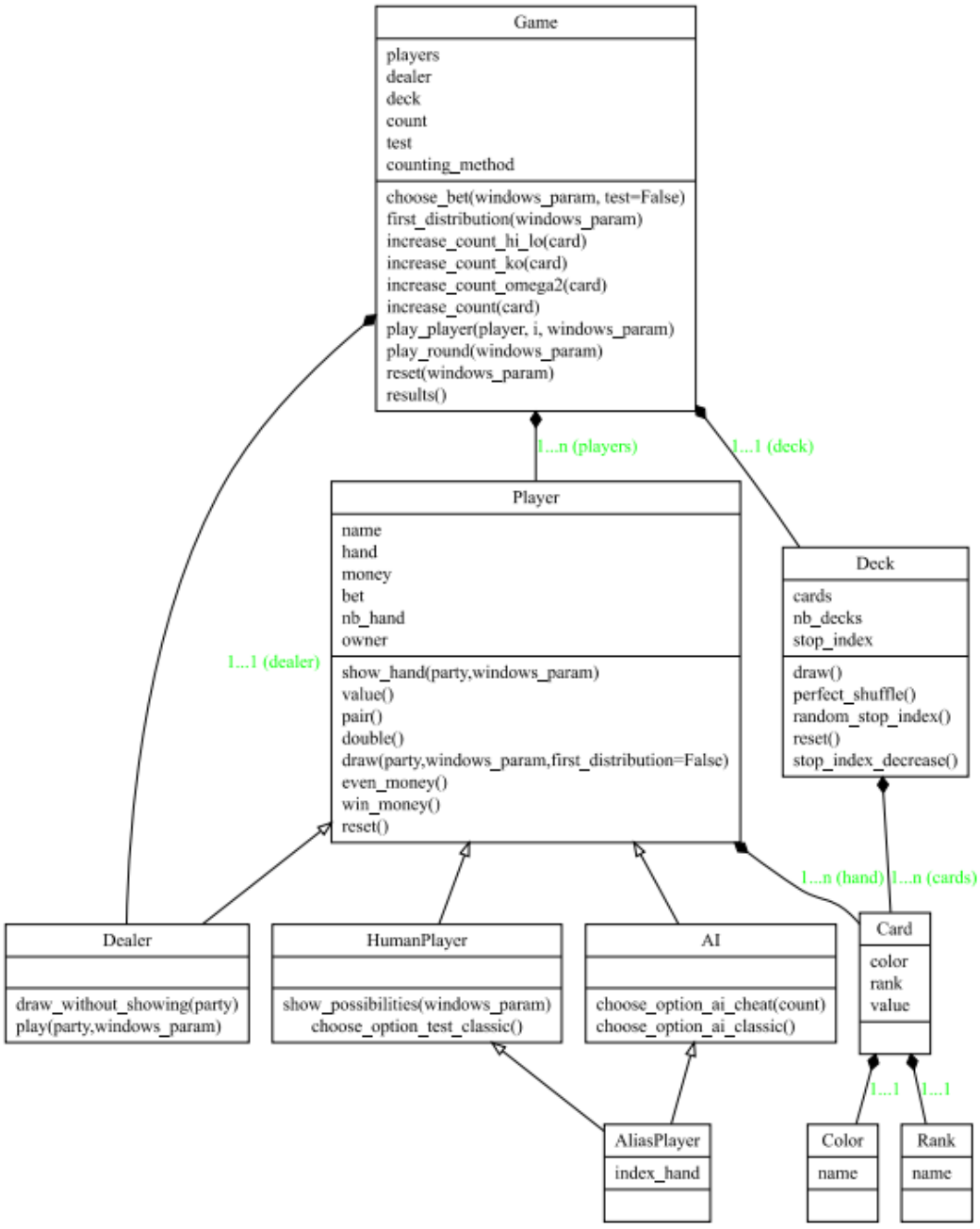

Figure 9: Class diagram for structure of the game

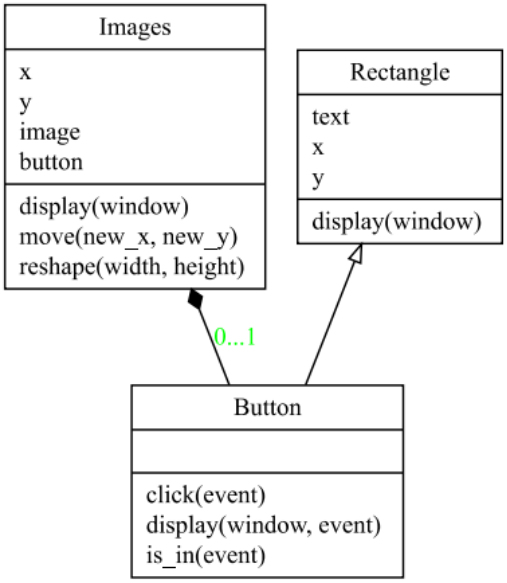

Figure 10: Class diagram for graphic of the game

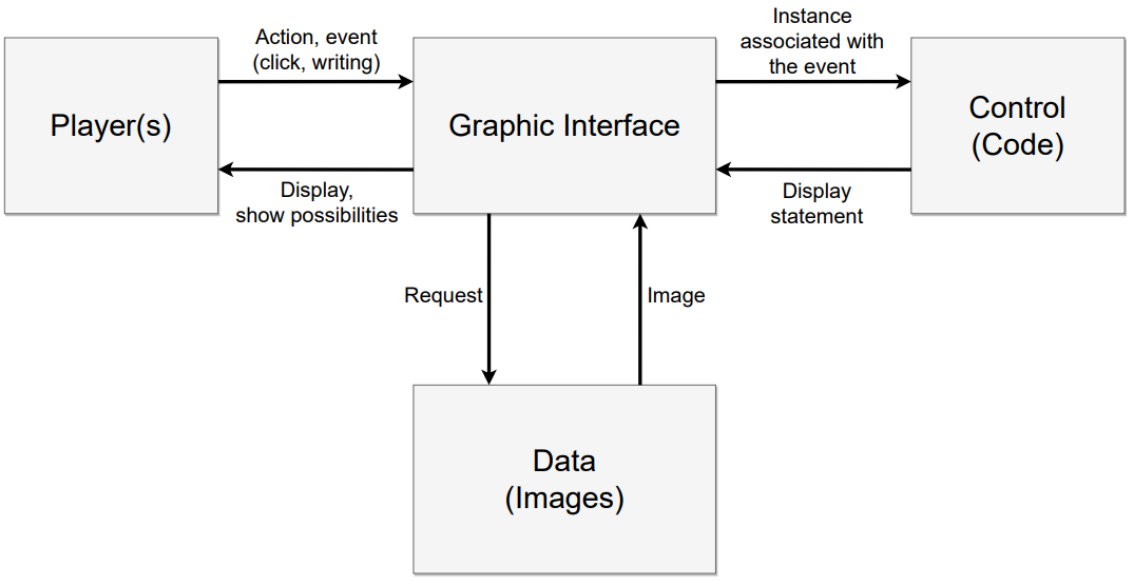

Figure 11: Architecture diagram of the game