# *Not an average casino – providing insight in how online casinos operate by means of a simple application*

A report for *R&D Project*

**4 June 2021**

by Matteo Gamberoni, Casper de With, Ege Sarı and Colin de Jong.

# Table of contents

# Introduction

The presented software is a brief simulation of an online casino, using tricks and manipulative probability to keep the user playing. Once the main game has been finished once, the program offers another game mode where you can see all statistics going on behind the scenes.

This application was created because of the publication of articles describing how online casinos were keeping their users addicted to the game while logged-in, making them increase their bets and constantly lose money.

Online casinos are online servers, powered by their algorithms and sustained by their users. Because of this, those betting platforms need a method to profit from their users' losses while keeping their audience attracted and focused on the game. This happens by engaging users with funny games, captivating sound effects and, most importantly, a false hope of winning money using deceptive tricks.

Online casinos are different from physical casinos. In the online ones computations of results happen through a code that players do not have access to. This makes it easy for the casino to have the odds strongly against the player. Even though there are laws that oblige online casinos to have a disclaimer explaining the odds of winning, this disclaimer is often very badly readable and in cryptic language[1].

Our application is focused on showing some of those tricks and it aims to raise awareness about the biased probabilities of games offered by the online casinos. It is easy to blame the individuals that are addicted to online gambling, but the bigger problem is the fact that the casinos get away with it. By making an accessible and harmless app, we hope to achieve this.

The reason why players should use our app is intrinsic in the malevolent methods used by the online casinos, so to help players to reduce, or stop, betting. The nature of betting is complex and it might result in a gambling disorder. Because of this, users of our product need to be aware of the situation they are in and be actively committed in understanding what happens behind the curtains of online casinos' scripts.

Our application is not particularly funny and  engaging, this is partly to remove those distracting factors, and partly because we didn't have enough time to build an entire realistic

[1] see Warning labels for online gambling 'ineffective and too difficult to find' new study concludes (medicalxpress.com)

casino in JavaFX. This does however makes users focus on what actually matters: the currency they are betting – no useless distractions.

## Product justification

The Casino application we have developed is unique in its kind. This is the case because the final goal of our system is genuinely the opposite of traditional casino applications: instead of getting a profit by stimulating people to bet, we encourage them to stop and there is no profit involved.

With four simple games: coin-toss, single dice roll, spinning suit block and five dice roll, we let users feel a basic casino experience, but, at the end of it, statistics and probabilities are shown. This last part reveals the idea behind online casinos' devious tricks and here lays the innovation of our application.

Because of the above our application can be seen as a didactic game, in which the goal is deeper than pure entertainment and it consists in educating the players. We believe this to be an honorable and just cause, which deserves attention and a proper implementation.

This being said, there are online casino apps already available, “BETMGM”, “DraftKings” and “BET365” are a few. However all of those applications fall in the standard “casino” category, those do share the gambling subject with ours, but the prospective is completely different.

The following sections are about the program itself.

## Sub-games: flow and properties

The casino application is a series of sub-games and when the system is started, the first one of the sequence is presented to the user.

The sub-games share some common attributes, but each sub-game has some different properties as well. We will go into them in the following section.

The common attributes shared between sub-games can be summed up into three categories:

- Variables and layout
- Transition to next game
- Saving and loading

### Variables and layout

Every sub-game is different, but they all have a few things in common. Those are the amount of currency the player has, the amount the player bet, the choice the player made, the results the game produced, and a menu.

The screen is separated into an upper and a lower part. The lower part is meant for user input, whereas the upper part is for the results and game elements. This simulates casinos: all buttons of a slot machine are on the bottom and the big rotating wheels are on top. Furthermore, it makes the interface organised and intuitive.

The initial amount of currency variable is set to 100. This variable serves as a trigger to automatically proceed to the next game once a certain limit has been reached. The sequence of games is designed to have the odds be slightly in favor of the player in the first games, while gradually decreasing the expected values in the following games. This makes the player feel they're winning at the start, making them comfortable and confident as their accumulated money slowly decays.

This is in line with the bet amount. Each sub game has an increment button which raises the bet amount of a predefined constant. This constant is smaller in the games favorable to the player and higher in the games against him. By forcing higher bets, we incentivise people to play with more risk. A "decrease bet" button has been omitted on purpose, to incentivise higher betting and risk taking, which makes us, the evil casino owners, win more money.

In order to allow a smaller bet in subsequent rolls, the increment button is set to 0 every time the round is over. Out of playtesting, we thought it would have been useful to also have a “Reuse last bet” button since constantly mashing the “Increase bet” button after each roll was tiring.

## Transition to next game

The first games are very profitable to the player, allowing them to rack up currency for the next games. Evidently, we cannot allow them to permanently stay on these rewarding games. That’s why we implemented a trigger that automatically moves to the next game once a certain number of rounds in that game have been played or a certain amount of currency has been reached. These amounts differ per sub-game and they control the flow of the game, so as to prevent it from being too boring or too profitable for the player. It is therefore impossible to dwell on the earlier games, where winning is easy.

## Saving and loading

In the corner of the screen in every game, there are two buttons: to save your progress and to load an earlier save file. A few key variables, among which the amount of rounds played and the amount of currency the player has, are saved to allow people to continue the game at a later point.

# Results and statistics

The second part of the casino application is where the educational aspect of the system lies.

Once the game is completed, the user is directed to a new menu with links to see the results and statistics behind each sub-game. In this final screen it is also possible to close the application or to start the game from the beginning.

By making these statistics visible, the player can gain insights into why the game seemed to be simple in the beginning, but hard near the end.

## Layout

The approach used for the layout is identical for each sub-game: a new screen displays on a background image the results of the game in the upper section, and some information about the algorithm used to determine the outcome in the lower section of the screen. The background image for each screen depicts the sub-game that the statistics are about.

In each sub-game the displayed results are the total amount of currency that has been bet and received, the amount that has been lost and won, the total number of games played and the amount of victories.

The button to increment the bet is linked to the available currency; every time this button is pressed the bet amount increases and the available currency immediately decreases by the same amount.

The amount won is equal to the amount received minus the amount bet. If the player receives currency after a roll, there is an animation that increases the amount they have. Even when the amount returned is equal to the bet, they will still see that animation and feel like they've won, even though nothing has changed. By emphasising wins and largely ignoring losses, we give the player a sense of satisfaction and this encourages gameplay.

The lower part of the screen contains a brief explanation of the algorithm used to determine the outcome result; favourable to the player in the first games, but more unfair as the game progresses.

## Specifications of sub-games

In this section a detailed description of the application will be presented. A detailed description of each screen will allow a clear understanding of the system and its behaviour.

The core idea behind the system is expressed in a case diagram, which shows the two main cases of the application and the way players and system interact.

### Single coin toss

The first sub-game is a coin toss simulator. The player is asked to pick between either heads or tails by means of radio buttons. They can increase their bet by repeatedly clicking the "Increment bet +1" button.

When the player is finished, they can click "Roll" and the coin is flipped. If they predicted the outcome correctly, they receive 2× their bet; otherwise, they get nothing.

In this game the chances are biased in favour of the player; the probability of winning is a favourable 17/32 instead of a fair 1/2. This makes the expected value higher than the bet, so players will keep winning more than they lose.

### Suit block

The suit block game consists of a slot machine with one column that returns one suit out of the four standard card suits: ♣, ♥, ♠, ♦.

The player must select one suit. They also have the option to switch between high risk and distributed risk: there's a switch with the options "4" and "3+1". If the player chooses option "4", they will receive 4× their bet if the chosen suit matches the result, and nothing otherwise; if the player chose option "3+1", they will receive 3× their bet if it matches, and 1× their bet if only the colour matches. This last option is for people who want to play it safe.

> *For instance,* the player chooses "4" and "♣", bets 10 and clicks Roll. The outcome is ♠, so the player loses their bet. If they had chosen "3+1" however, they would have received 10.

The layout of the scene is akin to the previous ones and the "Increment bet" button increases the bet by +2.

This is another aspect in which the casino encourages risk, this happens again due to the algorithm producing the result.

The RNG is made so that there is a 33/64 chance of getting the same colour as the picked suit, and there is a $(33/64)^2$ chance of getting the exact same suit. So it is slightly more favourable than 1/4.

## Single die

The second game in the sequence is the single die roll.

Instead of +1, now the increment button increases the bet by +3. These numbers keep rising, forcing the players to bet more.

The player has to choose between the numbers 1–6 and then click Roll. Now the die returns a value from 1 to 6 – this time it is totally fair. This is where it becomes interesting: if the player predicted correctly, then they receive n× the amount they bet, where n is their choice. Furthermore, if the result is higher than the choice, they receive 1× their bet, so basically a refund. This choice allows players to determine how much risk they want to take – a higher number is a higher risk.

It is also possible to place a bet on 1, but that would always return the same bet amount, so it does nothing. We kept it for completeness.

## Five dice game

This game revolves around five dice that are all rolled at once.

The player must bet – which is only possible in multiples of 5 – and then click Roll. This rolls all five dice. Once they stop, the player is given the choice to select any number of dice that will be rerolled. After the selected dice have been rerolled, the outcome is definitive. This outcome will determine the amount the player receives. In the following table, the return values for each configuration are displayed. Note that this is in multiples of 5 (the bet increase amount); if you have a full house for example, you receive 6/5 of your input. The reason it is this way is because of clarity.

| **outcome** | **return** |
|---|---:|
| five of a kind | 20 |
| four of a kind | 10 |
| three of a kind | 3 |
| full house | 6 |

| | |
|---|---:|
| two pairs | 2 |
| large straight | 10 |
| small straight | 5 |

The outcomes are just as in the game Yahtzee; they are well-defined and well-known, so we'll skip over their definition here.

If the final result isn't in the list, this will count as "junk" and it will yield no reward.

Coming up with the statistics for this game was an enormous challenge. The details can be found later in the document.

# Design

This section is concerned with the technical design and the implementation of the system.

Our application has been designed for desktop use and it runs only on PC.

The resulting system is a single package containing images and Java code, presented to the user within a graphical user interface created with the library *JavaFX16*.

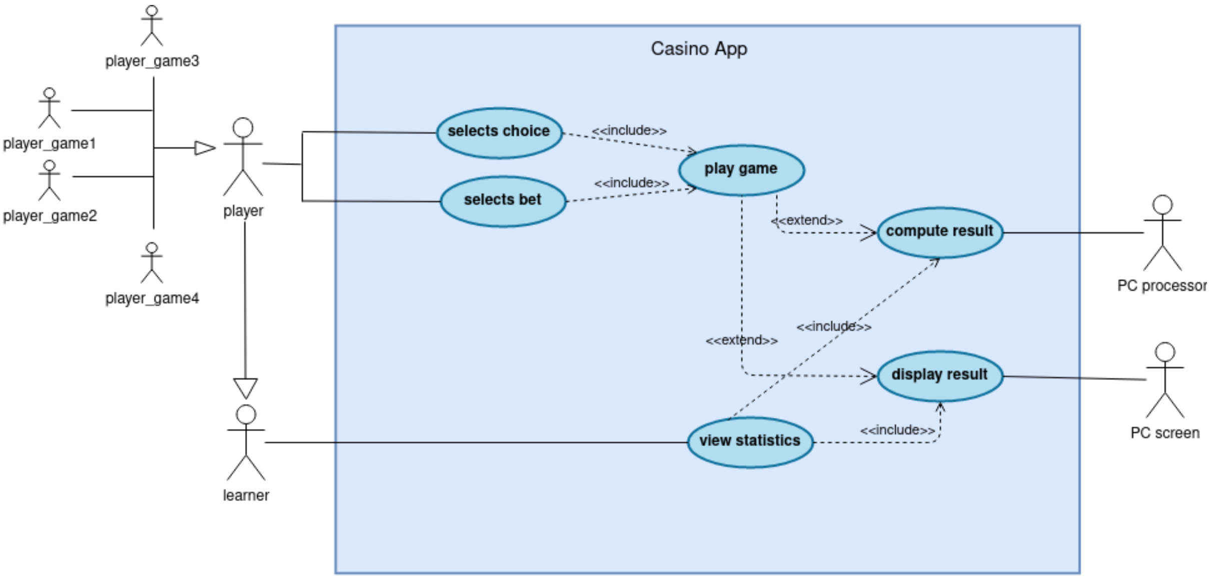
The Java language has been selected because of its object-oriented style and its property of combining pieces of software together. The OOP (Object Oriented Programming) technique resulted to be the proper choice for modeling big applications with reference to reality. Because of this technique, it was possible to define and instantiate each sub-game as an independent component. The Java language also offers the *JavaFX* library, a nice tool to create and organize graphical user interfaces.

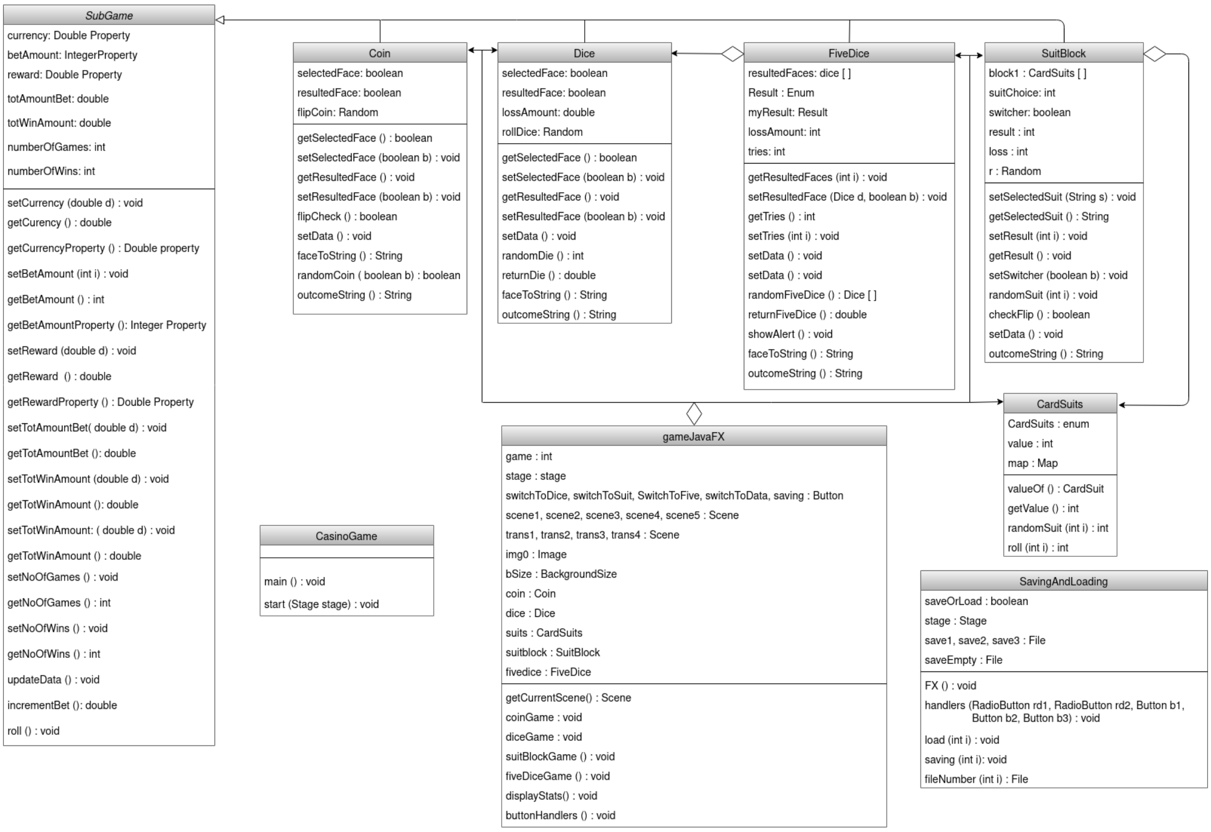
The concept behind the application is to have a fixed structure for common elements and then implement each component in this structure.

As in previous parts of the report, this section first presents a broad view of the system and then a deeper analysis of components worth mentioning.

## Global design

The global design of the application and the interactions between components is presented in the use case diagram and the class diagram below.





The class diagram shows name, attributes and methods of the classes in the system and it also defines the relationships between them.

The design of the system is divided in the model, made of sub-games and statistics, and graphical user interface.

In the model, all sub-games share common attributes and methods. Those commonalities are expressed in the abstract class SubGame. This class can not be initialized at compile time and all its abstract methods must be overridden by any class extending it. This is because those elements do not constitute a game in themselves, but are necessary information needed in every one. This class contains all variables needed to manage currency and winning rate: `currency`, `betAmount`, `reward`, `totAmountBet`, `totWinAmount`, `noOfGames` and `noOfWins` are the attributes of this class, which constitutes the structure of each sub-game. Every sub-game requires those attributes and hence every sub-game extends the `SubGame` class.

Classes used for sub-games are `Coin`, `Dice`, `SuitBlock` and `FiveDice`. The differences in those are in the methods used to calculate and check the results and assign rewards. Because of those divergences, different types of variables and data structures are used.

There are also similar games. Those are usually implemented by the use of the simpler game of the kind as an attribute of the more complex one. An instance of this is the five dice roll game, in which a list of dice is used.

Another similar case is the one of the suit block game; instead of using a different game as attribute, an enumeration class has been defined: the `CardSuits` class.

The graphical user interface displays the game in user-friendly screens. Those screens are all organized with the same layout and are meant to be easily understood.

The graphical elements are organized in modules: by the use of JavaFX it is possible to select specific nodes and group them together. JavaFX has an hierarchical organization in which elements compose a tree of graphical components , which are then contained in panes and added to a scene. The scene is the screen displayed and it is contained in a stage.

The grouping of graphical modules to be put on panes followed a logical reasoning: what needs to be displayed together or elements related to the same aspect of the sub-game were added to each other.

This allowed quick edits to the interfaces and made different options possible to explore.

## Detailed design

This part presents the technical details of the classes used in the system. Here a description of variables, data structures and methods is given. Since the design of the application is analyzed, arguments and some justifications are also expressed in this part.

The biggest challenges presented in the development of the system were fundamentally four.

The first was in the definition of the biased random generator and the possible best strategies. Since the results were mostly dependent on user actions, a generator able to reward best strategies and penalise bad ones was needed.

The second one was about choices of specific variables and algorithms in sub-games, to make those easily connected to other parts of the application and to avoid repetitions. The problems here were about the selection of a multitude of options and a deep valuation of the best ones.

The third challenge concerned the implementation of a graphical user interface in which data and elements needed to be intertwined as least as possible.

The last obstacle was the implementation of the saving and loading functions.

### Variables and data structures

The sub-games of the application are all extending the `SubGame` class. This gives a common structure to the game and makes storing and forwarding data easier. The `SubGame` class is a collection of getters, used to obtain, and setters, used to modify the general values of each sub-game. The `SubGame` class also contains abstract methods that are overridden and customized in each individual sub-game class. Those individual classes are the components initialized by the view, which uses those to display the game.

Each of those components, so each sub-game, has two constructors: one is used to a normal flow of the game and the designed succession of sub-games, while the second one is utilized by the loading function to restore the saved values.

All the sub-game classes contain a variable of type `Random`. This is used to compute a random value with the `Java.util` package and then determine the results according to the sub-game.

### `Coin` class

The `Coin` class is straightforward: a boolean variable is used to store the outcome and another variable of the same kind is used to store the player's choice. If after the toss those two variables are identical, then the player wins the round and receives 2× the bet amount. The statistics are updated by incrementing the wins' count by 1 and by adding the reward to the total amount of won currency. The available currency is updated based on the result of the reward.

This class also offers different methods to return strings. Those are used to print the value of the outcome and a related message; "Congratulations! You guessed it right! You win." is printed if the player wins; "How unlucky, you guessed it wrong. You lose your bet." otherwise.

### SuitBlock class

The SuitBlock class has a similar structure to the previous sub-games, but it differs from those in two aspects. This class also uses an enumeration type as variable, named `block1`, which consists of an array containing the four possible card suits. However the enumeration itself is not defined in this class, as in both dice games, but in a separate one called `CardSuits`. The choice of dividing those parts lies in the Java implementation of `Enum`. Those can be created separately, given attributes, a static constructor and methods. Since there are properties typical of the card suits but not of the suit block game, those two have been treated as different components. The main implemented difference is the property of connecting any `enum` variable to an integer (or any other type in Java) value. This has been done with the use of `HashMap`; a data structure that extends the `Java.util.Map` package, which gives to every suit a integer value used to determine results.

This differentiation results in a possible advantage: if other games concerning cards or suits need to be implemented, the suit enumeration can already be used.

The second difference is the presence of a boolean variable used to determine the risk mode of the game: the switch variable, allowing you to choose between "3+1" and "4", as described earlier. The technical aspect of those modes is in the handlers of the radio buttons used to present the choice between modes. Each handler sets a specific value for the variable switcher, making the algorithm used to determine the reward follow different branches in conditional statements.

The `Statistic` part of the application is the only part that has no hidden secrets: it uses simple methods from the `SubGame` class, and overridden in every sub-game, to access the stored values of a game and it prints it to screen.

The explanations at the bottom of the page consist of predefined text loaded in a label by another abstract method in `SubGame`.

### Dice class

The `Dice` class follows a similar structure of the `Coin` one, having similar methods and attributes. Choice and result are stored in an integer value instead of a boolean since six values can not be mapped with only a boolean variable. The variable lossAmount is introduced to calculate correct results due to the fact that this game allows a bet refund when the specified choice is lower than the outcome.

This class also contains an enum declaration, named Result. It is used to carry the result from the game and it can have three values: WIN, where the player guessed the number correctly, NO_W_L, where the player guessed the number lower than the result and LOST, where the player lost the game.

The enum is used for the variable myResult, which has type Result. It is not initialized. The enum type attribute is used so that we can construct all the reward calculations based on these results easily. At the beginning we were doing the comparison between the resulting face and the selected face. When the system prints out the message result, such as "You win!", it used to make the comparison check for both reward calculation and the total win rate. With this enum attribute the comparison happens once and myResult is set to the computed result. This makes the implementation more portable and efficient.

The returnDie() method returns the reward based on comparison between the values of selectedFace and resultedFace. It has a simple algorithm:

- If the selected face is equal to the resulting one then myResult is set to WIN, it updates the total win amount with reward, it increases total number of wins by 1 and it returns the reward.
- If the resulting face is bigger than the selected face, then myResult is set to NO_W_L, it updates the total win amount with reward and it returns the reward.
- If none of the above conditions hold, then myResult is set to LOST and lossAmount is updated with the bet.

The other methods are similar to the ones in class Coin and are used to return strings about the outcome and reward.

This class is also the base class for the five dice sub=game. We have tried to make the code readable for future developers. For this purpose, we have used methods for every need, even to increment the bet. This class can be implemented in a more basic way, but we also wanted to implement it in an elegant way, for clarity.

### FiveDice class

The FiveDice class is a sub-game extension of the single dice roll sub-game and hence this class contains instances of the Dice class as elements. This resulted in the attribute of the FiveDice class resultedFaces, which is an array of dice.

A similar approach to the dice game has been also used to define the results: those are contained in another enum, named result as well. In addition to the other classes, the five dice sub-game contains an integer variable named tries. This variable is used to check that the user does not roll any single dice more than three times in a row. The idea behind this is explained in the analysis of the returnFiveDice() method and the rules behind the game.

The most important method in the class is returnFiveDice(). It returns the reward as a double type. It is also responsible for updating the attribute myResult. Here is an explanation of the algorithm and data structure behind it.

At the first step, the method checks that resultedFaces has five results. If it has more or less results than 5 it throws an exception to the console. Otherwise the algorithm starts. Firstly, a histogram named h is created. This is an int array with size 6 at which the frequencies of results are assigned. Initially, all the indices are assigned to 0, then the histogram is filled by cycling through the results and incrementing the index in the array that matches the number. This yields a histogram: for every value the number of occurrences.

Let's consider the following example with the results of the five dice being 2, 3, 3, 4, 1. The values stored in the histogram are:

h[0] = **1** because there is **one** 1

h[1] = **1** because there is **one** 2.

h[2] = **2** because there are **two** 3s.

h[3] = **1** because there is **one** 4.

h[4] = **0** because there are **no** 5s.

h[5] = **0** because there are **no** 6s.

At this point another array of size 5 is created: an histogram of histogram h. This second array was named hh, and elements are initialized to 0. Now for every value at the indices of h, we update hh. This is done in order to create an array of occurrence values. Every index of h is checked and, if it is bigger or equal than 1, then hh[h[k]-1] is incremented. This results in an overview of the groups of identical numbers.

> For instance, the arrangement “full house” consists of a group of 3 and a group of 2, so the third and second index of `hh` will both have value 1: `{0, 1, 1, 0, 0}`. The arrangement “two pairs” consists of two groups of 2 and one group of 1, so the resulting `hh` will be `{1, 2, 0, 0, 0}`.

The combination of `h` and `hh` is needed to assign the result and reward based on the different combinations of outcomes.

An example of this is shown below. Let’s consider the results `2, 3, 3, 4, 1` again.

> `hh[0] = 3` because there are **three** separate unique values: 1, 2, and 4.
>
> `hh[1] = 1` because there is **one** pair of 3s.
>
> `hh[2] = 0` because there is **no** number occurring three times in the outcome.
>
> `hh[3] = 0` because there is **no** number occurring four times in the outcome.
>
> `hh[4] = 0` because there is **no** number occurring five times in the outcome.

In conclusion, `hh` is an histogram of occurrence times of the values, used to determine the exact result.

At this point, after `h` and `hh` are determined, the rules of the game are used to assign values to result and reward.

The variable `myResult` will be assigned any of the following values:

- The result `FIVE_KIND` occurs when the same number occurs five times, so `hh = {0,0,0,0,1}`.
- The result `FOUR_KIND` occurs when the result is four of a kind, so `hh = {1, 0, 0, 1, 0}`.
- The result `FULL_HOUSE` occurs when a face occurs three times and a different one occurs two times, so `hh = {0, 1, 1, 0, 0}`.
- The result `THREE_KIND` occurs when the same face appears three times and the other two faces can not be used to create better combinations, so `hh = {2, 0, 1, 0, 0}`.
- The result `TWO_PAIRS` occurs when there are two pairs and an individual, so `hh = {1, 2, 0, 0, 0}`.
- The result `BIG_FLUSH` occurs when there are five consecutive numbers. This result can only occur in two specific cases, namely `1, 2, 3, 4, 5` and `2, 3, 4, 5, 6` (in any order), and those situations have been hard coded as follows:
  `(h[0] != 0 && h[1] != 0 && … && h[4] != 0) ||`

`(h[1] != 0 && h[2] != 0 && … && h[5] != 0)`
If this condition of BIG_FLUSH holds, myResult is set to it and the calculated reward is returned.

- The result SMALL_FLUSH occurs when four consecutive numbers appear in the results. Here the possibilities are three, 1-2-3-4-x, 2-3-4-5-x and 3-4-5-6-x. Like the previous case, those have been hard coded. The algorithm is:
  ```
  (h[0] != 0 && … && h[3] != 0) ||
  (h[1] != 0 && … && h[4] != 0) ||
  (h[2] != 0 && … && h[5] != 0)
  ```
- If none of the previous results occur, then the player has obtained no relevant result and loses the amount they bet. In this case myResult is set to JUNK and the reward of 0 is returned.

## Graphical user interface

The graphical user interface has been realized with the use of JavaFX and images, some of which have been created specifically for this application. The images are used for the background and as icons inserted in buttons.

The layout of the game is implemented in a separate class that relates the elements of the game and the different screens.

This allowed the script to be organized in methods inside this class, methods in which the layout and values of previous graphical objects were changed in order to adapt to the new part. This saves implementation time expressed in Big-O notation (so number of steps executed) and the amount of space allocated to the data at the end of the Random Access Memory stack.

The idea behind the graphical user interface is standard: implementing handlers in order to accomplish specific actions and bind properties to have a homogeneous and natural flow.

All handlers have been implemented by the use of Lambda expressions, which allow a direct code in imperative style easy to see and comprehend. A possible alternative is the use of anonymous inner classes. Since any specific import or extension is needed for the handlers, this alternative has been discarded as an exaggeration of code in comparison to the task.

## Design justification

This section is dedicated to justify the design choice. Some of those have been given throughout the report and those and others will be summed up in this part.

The core idea of the system design is, from one hand, to implement a base structure on which extra functionalities are built and, on the other hand, to include those functionalities in a graphical user interface that keeps model and view separated.

This creates a stable and logical project that can be decomposed and analyzed from different perspectives almost independently. This makes maintenance quick and edits possible.

Instead of this SubGame abstract class, a Java interface would have been possible. An abstract class permits you to make functionality that sub-classes can implement or override whereas an interface only permits you to state functionality but not to implement it. A class can extend only one abstract class while a class can implement multiple interfaces. The abstract class has been chosen because of the specification of functionalities.

In a more technical aspect, several parts could have been coded differently. Some instances are the boolean variables in the coin game, the use of enumeration types and the user interface itself.

In the coin game the choice of boolean variables for a binary result seemed the most logical one, however casting those values to strings often happens and a possible alternative would have been the use of strings in the first place. This might have resulted in less lines and methods, but it would appear less elegant and logical.

The use of enumeration types gives a declaration of constants that can be used when needed. This is convenient and saves repetition of code.

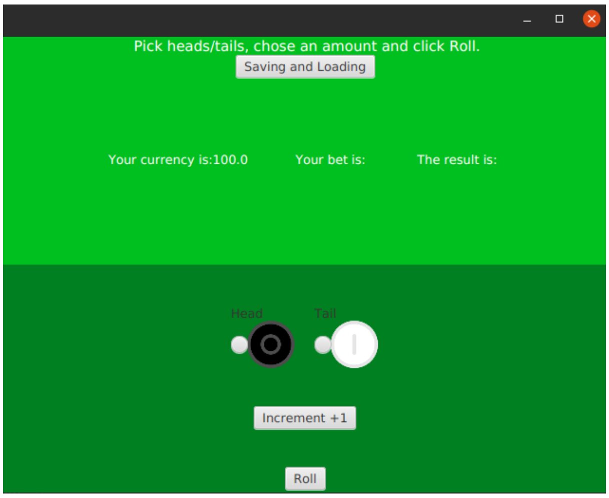
A lot could have been done differently in the graphical user interface. First of all the choice of JavaFX and the OOP orientation. A solid alternative to this would have been the use of JavaFX Scene Builder and then incorporating the resulting FXML file in the script. This was avoided because of a lack of knowledge about Scene Builder and the related XML-based markup language. The results with this tool would have probably been more satisfactory, but, due to a lack of time, this was not possible to implement.

Based on the adopted system, other possible design alternatives are in the implementation of the start method and in the relations between elements.

The start method was initially very large, confused and disorganized. Creating a separate class for the GUI elements, defining methods to give the game a order resulted in an easier understanding of the components used in the interface and a better layout of the start method.

## Screenshots

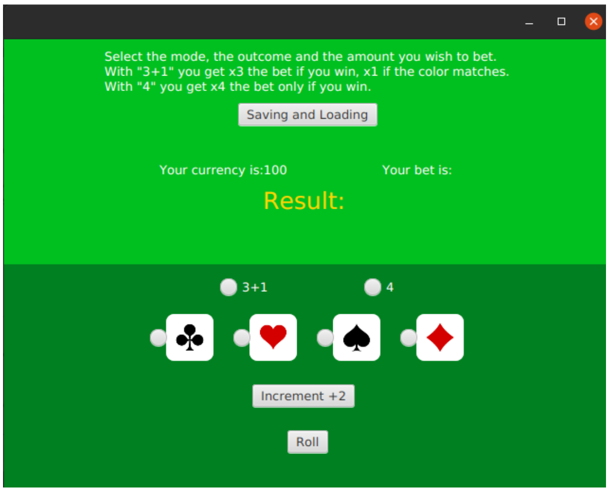
Here are a few screenshots of the game, as well as an original concept sketch with comments.



Pick heads/tails, chose an amount and click Roll.

Your currency is:100.0 Your bet is: The result is:

*The flipping coin game.*

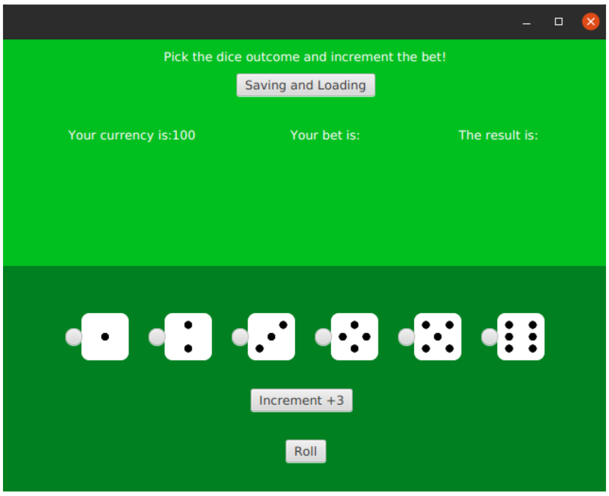


Select the mode, the outcome and the amount you wish to bet.
With "3+1" you get x3 the bet if you win, x1 if the color matches.
With "4" you get x4 the bet only if you win.

Your currency is:100 Your bet is:

Result:

*The suit game.*



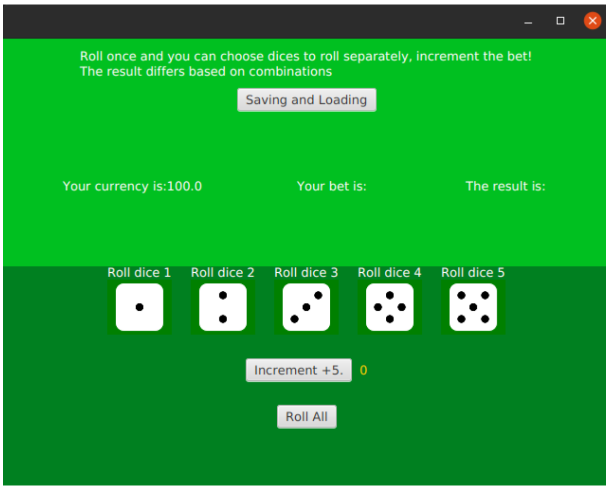
Pick the dice outcome and increment the bet!

Your currency is:100

Your bet is:

The result is:

*The dice game.*



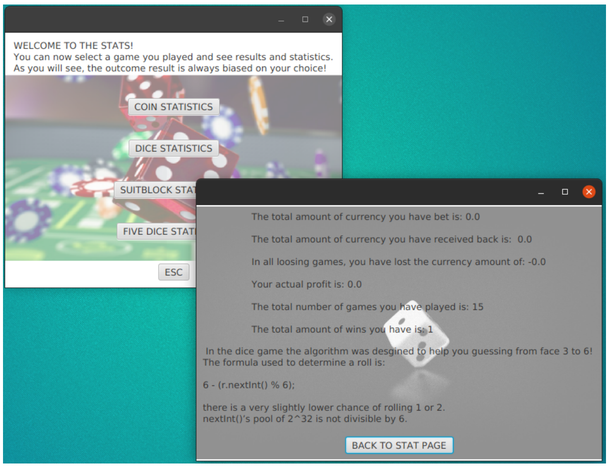
Roll once and you can choose dices to roll separately, increment the bet!
The result differs based on combinations

Your currency is:100.0

Your bet is:

The result is:

*The five dice game.*



WELCOME TO THE STATS!
You can now select a game you played and see results and statistics.
As you will see, the outcome result is always biased on your choice!

The total amount of currency you have bet is: 0.0

The total amount of currency you have received back is: 0.0

In all loosing games, you have lost the currency amount of: -0.0

Your actual profit is: 0.0

The total number of games you have played is: 15

The total amount of wins you have is: 1

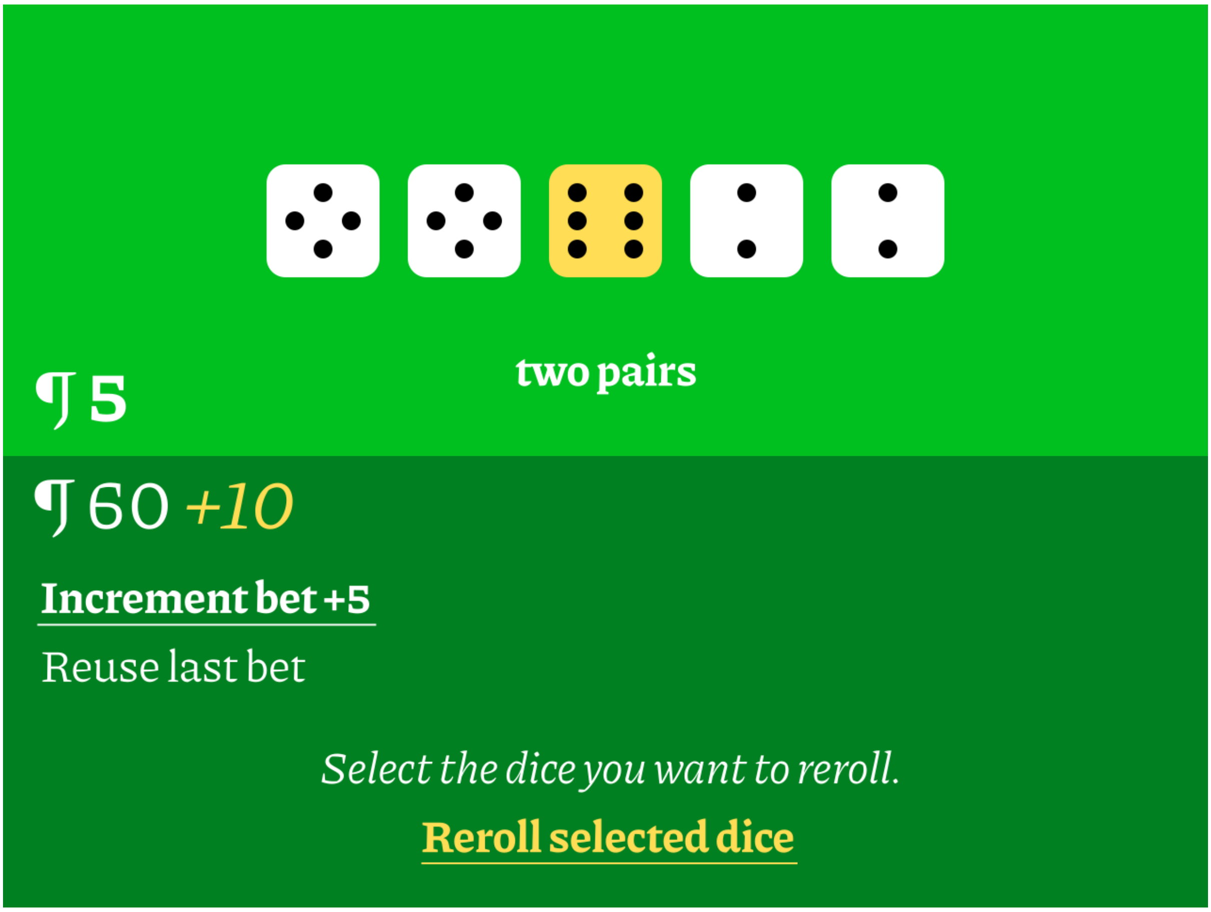
In the dice game the algorithm was desgined to help you guessing from face 3 to 6!
The formula used to determine a roll is:

6 - (r.nextInt() % 6);

there is a very slightly lower chance of rolling 1 or 2.
nextInt()'s pool of 2^32 is not divisible by 6.

*The statistics screen.*

*The transition screen.*



*The original concept sketch. The simple interface is a screen separated into two parts: controls at the bottom and results at the top. There is an "Increment bet +5" button, a "Reuse last bet" button that allows for easier long gameplay, a brief description and call-to-action: "Select the dice you want to reroll" and a yellow button, matching the yellow selected die. On the left side, the currency you have (below) can be bet on the table (above) – it goes away from the "controls" section since the money is no longer yours. Notice the symbols for the 2 and 4 dice: they are different from the conventional diagonal-2 and corners-4. This is to make it easier to tell them apart; the regular 4 and 5 dot configurations only differ by one dot, and the same holds for the diagonal 2 and 3 configurations. By making them like this, they are more distinctive and better readable.*

## Statistics of five dice sub-game

This part is dedicated to the determination of return values in the five dice sub-game. These needed to be chosen carefully, such that the expected value neither isn't too high that the players beat the casino, nor too low that the game becomes impossible.

### The challenge

The big challenge arose from the fact that the player has a choice. When the five dice have been rolled, the player can select any dice and reroll them once. Then the result is definitive. Calculating the probabilities was very hard, because the optimal player strategy depends on the return values (you might take more risk to aim for something with a high return value), and the return values need to be computed based on … the player strategy. This is a self-referring problem and it requires quite some mathematical skill to deal with.

So to start off, we just put in some placeholder return values. Later, we would refine them.

### Determining the optimal strategy

What we did is create a spreadsheet with all possible hh values, and the expected value based on the optimal strategy, and took a weighted average with the chance of that hh situation occurring. Now we have the expected value for the entire sub-game. However, how do you implement an optimal strategy in one spreadsheet cell?

The way we tackled this, is to have in each cell the following structure:

```
[chance of occurring sub-situation] * MAX (
        [expected value with strategy 1];
        [expected value with strategy 2];
        … ;
) + [chance of occurring of different sub-situation] * MAX (…), etc.
```

All expected values for the different strategies are computed based on the sum of return values. For most situations, determining the possible strategies was very simple.

For instance, if you have five of a kind, the maximum return value, you obviously reroll nothing. If you have four of a kind or two pairs, you reroll the single outlier. If you have a full house or a large straight, you also reroll nothing. In all these cases, rerolling anything would make a good situation vanish, and the risk is never worth it. This was determined with some easy probability calculations.

It was slightly more difficult to come up with the “three of a kind” situation, since you can either go for a straight if that’s beneficial, or you can reroll the two individuals and go for three/four/five of a kind or a full house. So we determined the expected values of getting a straight for every [6 choose 3] situations, which was easy but tedious, and then for each situation compare them with the expected values of getting many-of-a-kind. The formula worked.

Also for the case where all of the numbers are different, we found an easy strategy: don’t bother with many-of-a-kind, but just fully go for straights. The way you maximise your chances of getting a straight, is to reroll the outliers of the smallest group. It works like this:

> Since all five dice are different and there are six possible values, there is exactly one value not present.

- If this missing value is 1 or 6, you already have a large straight, and you shouldn’t reroll anything.
- If this missing value is 2 or 5, you have a small straight. You can reroll the outlier of the smallest group (a group of 1, consisting of either the number 1 or 6), and get a large straight with a 1/6 chance and a small straight otherwise.
- If this missing value is 3 or 4, you have a group of two consecutive and a group of three consecutive. Reroll the group of two and hopefully, one or both of them will result in a small or even a large straight. The best you can otherwise get, is two pairs or three of a kind.

Now remains the ultimate challenge. It was by far the hardest situation to deal with, taking hours and hours of sketching, calculating, erasing, typing and executing tedious tasks systematically. Behold!

## The situation of one pair

This occurs when you have one pair and three different individual dice.

The hh = {3, 1, 0, 0, 0} case is very complicated and difficult to compute. There are tens of possible sub-arrangements, many choices, and even more outcomes. Furthermore, it’s by far the most probable situation with a near 1/2 chance, so we cannot simply ignore it; we ought to find an optimal strategy for it.

To determine what the best strategies are for each of the possible sub-situations, we used a big formula in a spreadsheet, comprising all fifteen (= 6 choose 4) distributions of numbers –

of which 9 unique; the symmetrical pairs are allocated a 2/15 chance as opposed to 1/15 – and within each is a MAX function determining what choice gives the maximum return. The possible choices that the MAX function can pick between, are:

- **Reroll one die and go for a probable straight;**
- **Reroll two dice and go for a less likely straight – with two pairs or three of a kind as a fallback;**
- **Reroll three dice (such that the pair remains), and go for anything but a straight.** *(a small straight would still be possible, but unlikely with a 1/18 expected chance: the average of 1/36, 2/36 and 3/36; determined by the value of the occurring pair (1|6, 2|5 and 3|4 respectively)).*

Some other choices are also possible (such as: "keep two different dice", or "reroll two dice such that a pair and a different single die remain"), but if you compare their expected value (sum) formulas with the other choices, each individual part of the sum was always lower than another choice's expected sum, so it would never be a profitable choice. One of these choices' functions, "keep pair, and one different individual", is found below. In similar cases, where something inside of a MAX function could just never become the maximum, we removed it.

The cells in column G were the return values of the different situations.

Once the individual parts were finished, it turned out that some sub-formulas happened to be identical to others. So as to reduce redundancy, we merged them into the same sub-formula with a larger probability. Also, the "reroll three dice such that the pair remains" was used so often, we put the formula in a separate cell, **J10**, and referred to that cell to reduce more redundancy. The spreadsheet is attached with the project zip, so have a look around if you want to.

Following are the expected value formulas of different strategies. The "straight" calculations have all been calculated separately, since they are the only result that depend on the value of the dice. For other results, like many-of-a-kind, it doesn't matter what the value is, only what the grouping is. Therefore, the formula is structured like this: for each of the nine unique distributions, the chance of a straight with the three possible strategies is weighed against the always identical chance of a many-of-a-kind or full house: **J10**. That's why the **J10** cell occurs as a last argument of every MAX.

(unused) `Keep pair + one different individual: (G6/36) + (G8/18) + (G7/9) + (G9 * 2/9)` + small straight [depends on case]

Keep one pair: **J10** = (G5/216) + (G6 * 5/72) + (G7 * 5/9) + (G8 * 5/36) + (G9 * 5/18) + (G11/18)

Reroll one die for straight: (G10/6 + G11 * 5/6)

Keep three different: (G7/12) + (G9/6) + straight [depends on case]

## The abomination of a formula for the one pair case

=

2/15 * (G10/6 + G11 * 5/6) + // 1234 3456

1/15 * (G10/3 + G11 * 2/3) + // 2345

4/15 * **MAX** (G10/6; ((G10/9) + (G11 * 7/36) + (G7/12) + (G9/6)); J10) + // 1235 2456 1245 2356

5/15 * **MAX** (((G10/18) + (G11/9) + (G7/12) + (G9/6)); J10) + // 1236 1456 1246 1356 1346

1/15 * **MAX** (((G10/18) + (G7/12) + (G9/6)); J10) + // 1256

2/15 * **MAX** (G10/6; ((G10/9) + (G11/2)); J10) // 1345 2346

// (comments for the distributions they apply to)

All in all, it was enormously rewarding once the formula didn't raise errors and came with a realistic result. All that was left to do was to come up with some nice-looking numbers for the return values, and the expected value changed automatically every time a return value was edited. We increased the numbers of the return value slightly every time, until a big – but not too big – house edge was achieved: if you pay 5, you receive on average 4,73; the casino will get on average 5,3 % of your money. Even though it is massive, some online casinos have edges of as high as 10 %.

# Evaluation

The final result of this project was quite different from the idealistic image we had in our mind in the beginning. The first idea involved more sub-games, a better interface,  sound effects and proper animations.

At first sight, this idea seemed easy to concretise and doable within the schedule. This ended up not being the case because of the amount of time used to finalise the design and make the graphical user interface was way more than expected.

The presented result is not particularly engaging and quite repetitive. Nonetheless, the application compiles and runs smoothly, doing what it's supposed to do and conveying the message.

We believe that there is big room for improvement and with more time the application could really result in something valuable. In any case, we hope that the public, through one way or another, will stop blaming individuals for being addicted, but instead address the larger issue of corporations allowing it to happen; and our casino application is just one step in
that direction.