Blackjack Player

Final report

Submitted for the BSc in

Computer Science

April 2020

by

Tom Shaw

Word count: 8834

Abstract

The purpose of this project was to create a blackjack playing program, that will consistently use the best possible strategy to win games and amass chips. The program features the functionality to carry out full games of blackjack, with multiple players, and betting capability, using either a manual or computer-controlled dealer.

This project's intent was to research the techniques used to play blackjack well, with an aim to improve the skills of beginner and intermediate level players, by playing practice games and situational drills. It would act as a platform to train these players at blackjack, reinforcing good habits and betting tactics, without the risk/distraction of real gambling.

In its current state, the program is mostly complete. It allows multiple human or computer-controlled players, supports manual or regular dealing modes, and features a completely implemented betting system.

The computer-controlled player logic is complete. They will follow basic blackjack strategy, to perform the optimal action for each situation, and will make strategic bets based on the amount of chips they have left.

To reach this level of completeness, a lot of features from the original specification had to be cut.

The original aim was to include neural-network-based card recognition software, which would read card numbers and suits, through a webcam or live screen capture. This would then be input through the program’s manual card input. However, this feature could not be effectively implemented in such a short time scale.

Game modes were an additional secondary feature. A poker-style betting mode, where players can increase their bet between turns in a round, was developed, but it couldn’t be implemented. This is because the game mode switching interface was not completed. Players can still use custom decks, with the manual input mode, to play some custom modes though.

Unfortunately, the GUI is incomplete, so in its current state, the program is CLI only. The GUI specification and design is complete. However, it could not be implemented satisfactorily in time. The completed CLI is stable and allows access to all of the program’s features, it’s just a little bit inaccessible to less computer-savvy users.

Due to the incremental model used in the project, testing and bug fixing was completed throughout development, during the prototyping of each feature, but was not well documented. Towards the end of the project, a set of tests was created to ensure all functionality worked as intended.

Most tests were successful straight away, and the failing tests were made to pass with minimal difficulties.

The project succeeded in its main goals, but sacrificed some of its most interesting features along the way. The resulting program is a stable Blackjack simulator, with a variety of options. It can capably help players to improve their skills, and reinforce their strategies.

Contents

[Abstract i](#_Toc38987889)

[Acknowledgements ii](#_Toc38987890)

[1 Introduction 3](#_Toc38987891)

[1.1 Background to the project 3](#_Toc38987892)

[1.2 Aims and objectives 3](#_Toc38987893)

[1.3 Research question 4](#_Toc38987894)

[2 Literature review 5](#_Toc38987895)

[3 Requirements 8](#_Toc38987896)

[3.1 Product requirements 8](#_Toc38987897)

[3.2 Functional requirements 8](#_Toc38987898)

[3.2.1 Interfaces 8](#_Toc38987899)

[3.2.2 Functional Capabilities 8](#_Toc38987900)

[3.2.3 Performance Levels 8](#_Toc38987901)

[3.2.4 Data Structures/Elements 8](#_Toc38987902)

[3.2.5 Safety 8](#_Toc38987903)

[3.2.6 Reliability 8](#_Toc38987904)

[3.2.7 Security/Privacy 8](#_Toc38987905)

[3.2.8 Quality 8](#_Toc38987906)

[3.2.9 Constraints and Limitations 8](#_Toc38987907)

[3.2.10 Performance requirements 8](#_Toc38987908)

[3.3 Design constraints 8](#_Toc38987909)

[4 Design 9](#_Toc38987910)

[4.1 Software design 9](#_Toc38987911)

[4.2 Experimental design 9](#_Toc38987912)

[5 Implementation and testing 10](#_Toc38987913)

[5.1 Implementation 10](#_Toc38987914)

[5.2 Testing 10](#_Toc38987915)

[6 Evaluation 11](#_Toc38987916)

[7 Conclusion 12](#_Toc38987917)

[References 13](#_Toc38987918)

[Appendix A – Interesting but not vital material 14](#_Toc38987919)

[Appendix B – Other things which may be useful 15](#_Toc38987920)

# Introduction

## Background to the project

### Game Concept and Rules

Blackjack is a simple card game, where players aim to collect a hand with a value closest to 21, without going over it. The highest possible score being 21.

Each player is initially dealt two cards, and each turn, they can choose whether to be dealt another card (called hitting) or stay with their current hand (called standing). Until every player has chosen to stand.

Jacks, Queens and Kings are all worth 10 and number cards 2-10 are worth their value. However, Aces can be worth either 1 or 11, making them very versatile.

There are two additional actions that a player can choose:

Doubling down is where a player doubles their bet after receiving their first two cards, and is dealt one extra card, after which they stand. House rules vary significantly around doubling, due to the game’s evolution. Originally, you could only double with a hand worth 9, 10, or 11.

Splitting is when a player splits a hand into two separate hands, with two separate bets, and plays them individually. This can only be done on the first turn, with a hand containing two cards of the same value.

### Origin and Popularity

The origin of blackjack is disputed, but it is accepted that is has roots in France, Spain, and potentially Rome. It is confirmed to have been played in French casinos, under the name ‘Vingt-et-Un’ (Twenty-One), around 1700. Whilst a similar variant: ‘One and Thirty’ was played at a similar date, in Spain. There are unconfirmed rumours that it originated in Roman times, where a variant was played with wooden blocks. The game has stood the test of time with both casual players, and gamblers, due to its simplicity and short rounds. Remaining popular in casino, on betting websites, and in casual play.

Blackjack is extremely popular with gamblers, because it is simple, and has a low reliance on chance. This combination gives it the lowest house advantage (when using optimal strategy), meaning it offers the best opportunity to consistently win money. (Anon., n.d.) On top of this, due to it being so simple, the optimal strategy is easy to learn and memorise.

## Aims and objectives

The aim of this project is to create a program which will play blackjack at a high level. Utilizing strategies to reduce risk, but not relying on an unlimited memory of dealt cards. It should be able to play with a number of different players and react to a variety of situations.

The primary objective is an all in one desktop application which includes: a basic blackjack game, the blackjack playing program (known as BJP or Blackjack Player) and a dealer program. It will allow up to four human or BJP players to compete in a game of blackjack. There will be two modes: a manual input mode, and a non-manual input mode.

The manual input mode will allow the user to choose the value of each card dealt, or deal a random card. It’s a simple feature, but has a lot of potential for experimentation, and extending the uses of the basic program.

It can be used as a means to play with a physical deck, which would prevent any issues with computer-based shuffling, mentioned later, as well as allowing the use of custom decks, for different variants/game modes. On top of this, manual mode can be used to set up scenarios, or drills, to test strategies and improve a player’s skills.

The non-manual mode will involve a computer-controlled dealer, using the fairest (reasonable) shuffling algorithm available. This mode will allow full games of blackjack to be played within the program, featuring multiple human and computer-controlled players.

As a secondary objective, a betting system will be implemented, allowing both computer-controlled, and human players to bet simulated chips. An expansion of this would be to have the BJP players bet safely, regularly and aggressively, based on how many chips they have.

Another secondary objective is the addition of extra game modes and house rules. These would flesh out the program, giving it a lot more versatility for, as well as re-playability for casual players.

## Research question

"What is the best strategy for a computer-controlled blackjack player to implement, to achieve the best and most consistent results? How can this program be utilized to help human players?"

# Literature review

There is a variety of applications designed to give players an edge in online blackjack. These programs can use a manual input, or gather game information a website’s HTML. (S, 2014) They can advise the user on the ideal move, or even play autonomously.

One of the methods used by these programs involves consulting a strategy sheet, to decide the best possible move, considering the cards in play. (Anon., n.d.) However, some applications utilize more unfair methods, such as card counting, or cracking the pseudo-random seed that the casino bases its deck shuffling on. (Elgabry, 2017)

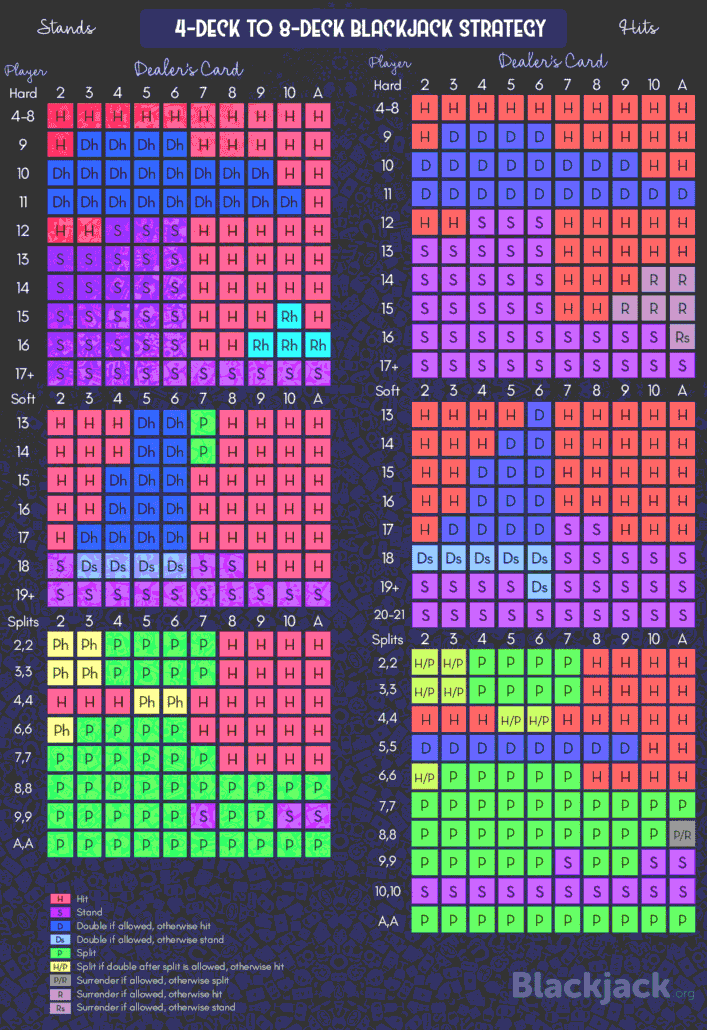
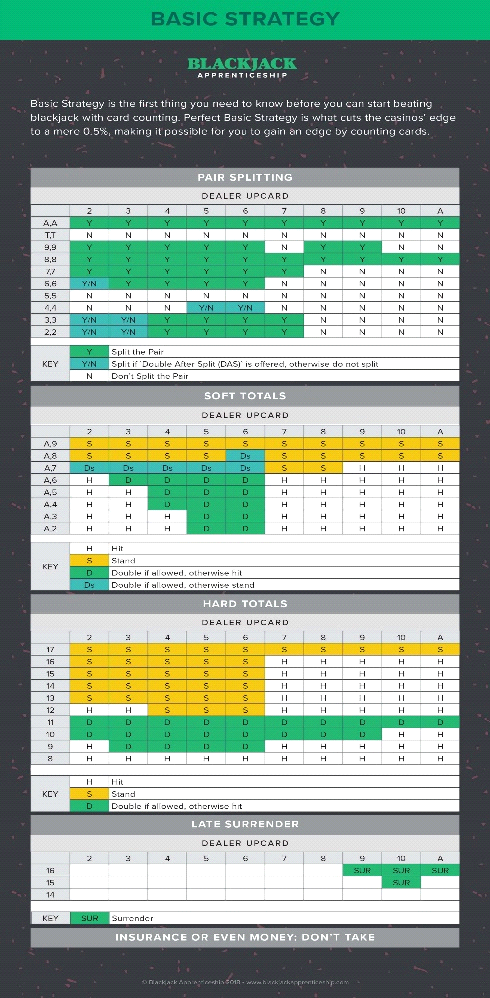
 

Figure 1 and 2 - Blackjack Strategy Sheets (Anon., n.d.) (Anon., 2018)

Card counting is a blackjack playing method which involves assessing the ratio of high and low cards left in the deck. This allows a player to determine the likelihood of the next card being high or low, and play accordingly, giving an edge over a regular player. Casinos have led the public to believe that card counting is not legal. However, without the use of an external device, it is a legal strategy. This was proven by famous card counter Ken Uston, who took the casinos of Atlantic City to court, where the judge ruled that the casinos are unable to ban skilled players from entering their premises. (Stanley, 1982)

By working out the algorithm which an online casino uses to shuffles its cards, and the seed it starts with, a program can create a deck shuffled in the exact same way. This allows the player to know the exact order of the cards to be played. A seed is a large randomly generated string, the seed is manipulated by an algorithm to return a set of 52 numbers, which each correspond to a card. The result is the order of the deck. (Elgabry, 2017)

Using this information, the player has an even more in-depth advantage that a card counting player, and can completely avoid going bust. This method is illegal, as it can only be performed with the assistance of an external device, with the knowledge of which algorithm the casino is using. Some casinos switch between selection of algorithms or use multiple at the same time.

Online casinos have combatted this is by using a variety of methods:

Physical random number generators utilize creative solutions to generate provably random numbers to create a unique deck layout. An interesting example involves firing a laser beam at a semi-opaque mirror, it results in 0 if the beam bounces off, or 1 if it goes through. (PokerStars, 2015)

Live dealers, these are real people dealing cards via webcam. These dealers use continuous shuffling machines (CSMs) which shuffle the cards continuously throughout the game, and distributes them fairly. This completely eliminates the need for an algorithm, and also prevents card counting. (Stevens, 2020)

The basic blackjack strategy is a culmination of the best move, mathematically, for each given situation. Commonly displayed as a chart/spreadsheet showing the optimal action to take, based on the dealer’s card, and the players hand. These charts are slightly different, depending on the house rules, but the strategy mostly remains the same. By memorising the basic strategy, a player can achieve a minimal house edge, making it more likely for them to profit.

House edge is the metric that shows the casino’s advantage over the player, derived from the ratio of the average loss to the initial bet. It can be used to determine how much a game will pay out over time, essentially showing how profitable a game is. (Anon., n.d.)

A high house edge comes from a large amount of randomness, and the amount of control a casino has over skewing a game in their favour. Blackjack is a simple and fairly predictable game, so by using the optimal strategy, this house edge can be negated.

The house edge for players using the basic blackjack strategy is 0.5, which is one of the lowest house edges out of all casino games. However, this can rise to 2% if a player is using incorrect strategy. If a player combines basic strategy with other techniques, such as card counting, they can even achieve a negative house edge. Meaning that they will statistically gain money over time. (Anon., n.d.)

There are a variety of different house rules, that alter the gameplay of blackjack, sometimes as much to be a new game mode entirely.

One of the more prominent house rules is the s17 rule. This determines whether the dealer should hit or stick, when they have a soft hand (a hand containing an ace) that totals 17. It is a small difference, but if a dealer hits on a soft 17, the house edge actually rises by .22%. (Anon., 2015)

Another adaptation involves allowing the players to peek at the dealer’s face down card, if their up card is either an ace, 10, or equivalent. If the dealer has blackjack when the player peeks, the dealer wins automatically. This sounds negative for the player, however it can actually prevent them from losing as much money, because they won’t have the chance to double or split. Which would increase their bet. (Anon., 2017)

Some versions of blackjack allow the player to increase their bet between turns, similarly to poker, or make a side bet separate to the game. For example, the in-between side bet is a wager that the dealer’s card will be between the player’s two cards. So, if a player is dealt a 4 and a 10, any dealer card from 5-9 will win the side bet. (Anon., n.d.)

The takeaway from this research is as follows:

This project is intended to play good poker, but will not be utilizing unfair methods to gain any advantage. The program will follow strategy sheets to play safely and efficiently.

Different shuffling methods will be explored to ensure fair play, minimising the potential for card counting and algorithm reverse-engineering. However, this is not the focus of the project, so will not be covered in a huge amount of depth.

Multiple game modes will be investigated and implemented, to extend the functionality of the program. The program will include a natural 21 check method, meaning any player with a natural 21 will immediately win, streamlining gameplay. As well as this, it will use the S17 rule (stand on soft 17), because it’s the most widely used version.

# Requirements

## Product requirements

The software will allow games of blackjack to be played between multiple human and computer- controlled players, using either manually inputted cards, or a computer shuffled deck.

Its target audience is blackjack players with an interest to improve their basic strategy, card counting and betting tactics. However, the program also caters to a casual audience, who just want to play a game of blackjack.

Its goal is to facilitate the training of blackjack players, through drills and practise matches.

Additional rules:

As mentioned in the literature review, the dealer will stand on a soft 17.

A player can only double with a hand totalling 9, 10 or 11, and will receive one card upon doubling, after which they must stand.

A player can double after splitting their hand.

If a player’s starting hand totals 21, they immediately win.

Bets are only taken at the start of the round (before any cards are drawn).

Side pots will not be included.

Extra game modes will be investigated, and implemented if there is time.

Figure 3 is the strategy chart calculated based on these rules. This is the strategy that the computer-controlled players will adhere to.

## Functional requirements

### Interfaces

Initially, the program will be designed for use with a command line interface. Once the project is suitably complete, a GUI will be designed for the program and included as a launch option. Allowing users to choose between the two interfaces.

The CLI will need to be easy to understand, and not cumbersome to make choices in.

The GUI will need to look respectable, show all the necessary information, and fit the theme of blackjack. Whilst remaining intuitive to use and uncomplicated. It will also need to not be resource intensive.

Figure 3 – Blackjack Strategy Chart (Anon., 2010)

### Functional Capabilities

The software will allow games of blackjack to be played between multiple human and computer- controlled players, using either manually inputted cards, or a computer shuffled deck.

There may be support for additional game modes/house rules, dependant on timescale.

### Performance requirements

The software should easily run on low level hardware, in both CLI and GUI modes.

### Data Structures/Elements

The program will make efficient use of data types and data structures, to minimize its resource usage.

### Safety

This software will be programmed following good coding practise. It will not use any hacky or dangerous implementation that could have an adverse effect on hardware.

### Reliability

The program should not crash, or break unexpectedly. Any exceptions should be handled, and clearly conveyed to the user.

### Security/Privacy

There will be no opportunity for the program to access any information, other than user input. So, there will be chance for it to breach a user’s privacy.

### Quality

There should be no half measures in terms of quality. All features should be implemented cleanly, and efficiently.

### Constraints and Limitations

Time – Limited time will likely result in most of the secondary objectives remaining incomplete. Unfortunately, the program will probably lack the quality of life features it needs to stand out.

Human Resources - As a result of being a one-man team, the project scope is rather small. This means that the project will not be able to explore additional functionality that could make it more interesting and useful.

Technical – As a result of the program being developed in C#, it will not run on Apple devices. Unless a virtual machine is used, this reduces the size of the available target audience.

# Design

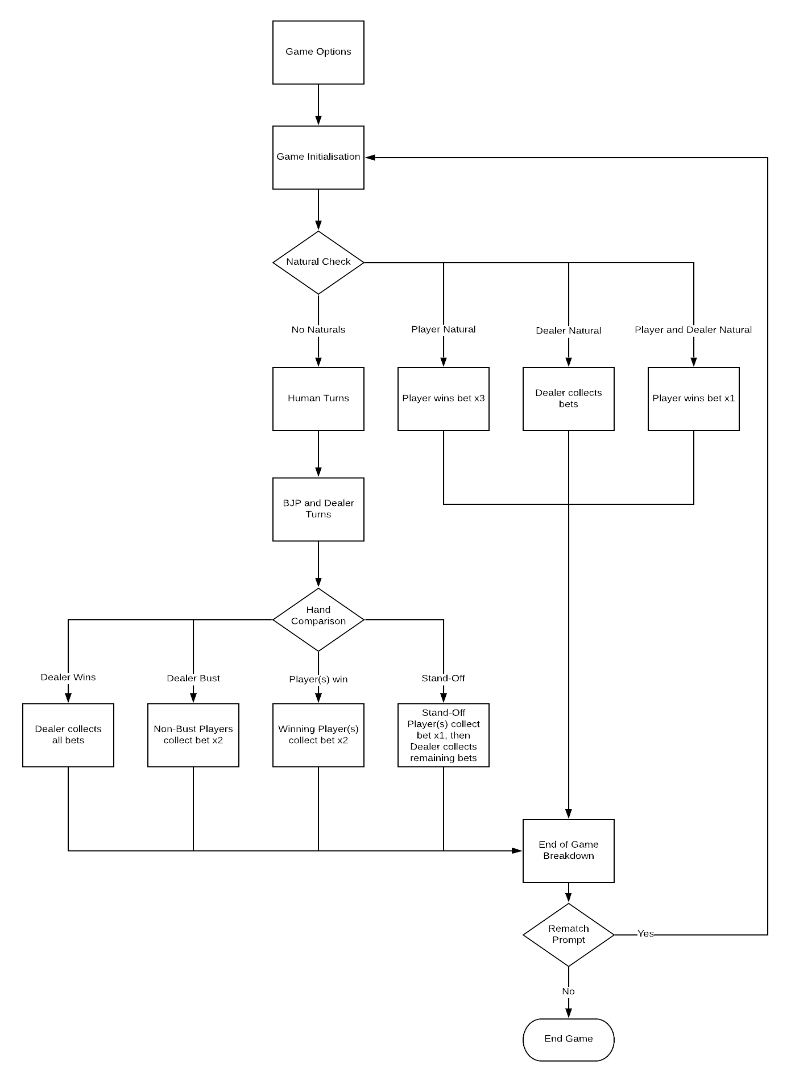
## Methodology

This project was based on the incremental methodology. This methodology involves designing, developing, testing and implementing the software in a segmented manner. By developing in this way, prototypes are available from the early stages of development, allowing you to build up functionality, whilst ensuring the software is stable and bug free.

One of the advantages of the incremental methodology is that it is very flexible. Meaning that the project’s scope can be easily reassessed during development, allowing it to be scaled up or down, depending on timescale, resources, etc.

## Software design

### Process Breakdown

Figure 4 shows the processes that comprise the program’s design, each process will be detailed in this section:

Game Options – The user is presented with various game modes and settings for them to configure. This includes card input mode, player names, number of decks, etc.

Game Initialisation – This process involves setting up the game’s components, so that the game can start. This set up comprises of: creating the Player objects; loading and shuffling the deck; taking bets; and dealing hands.

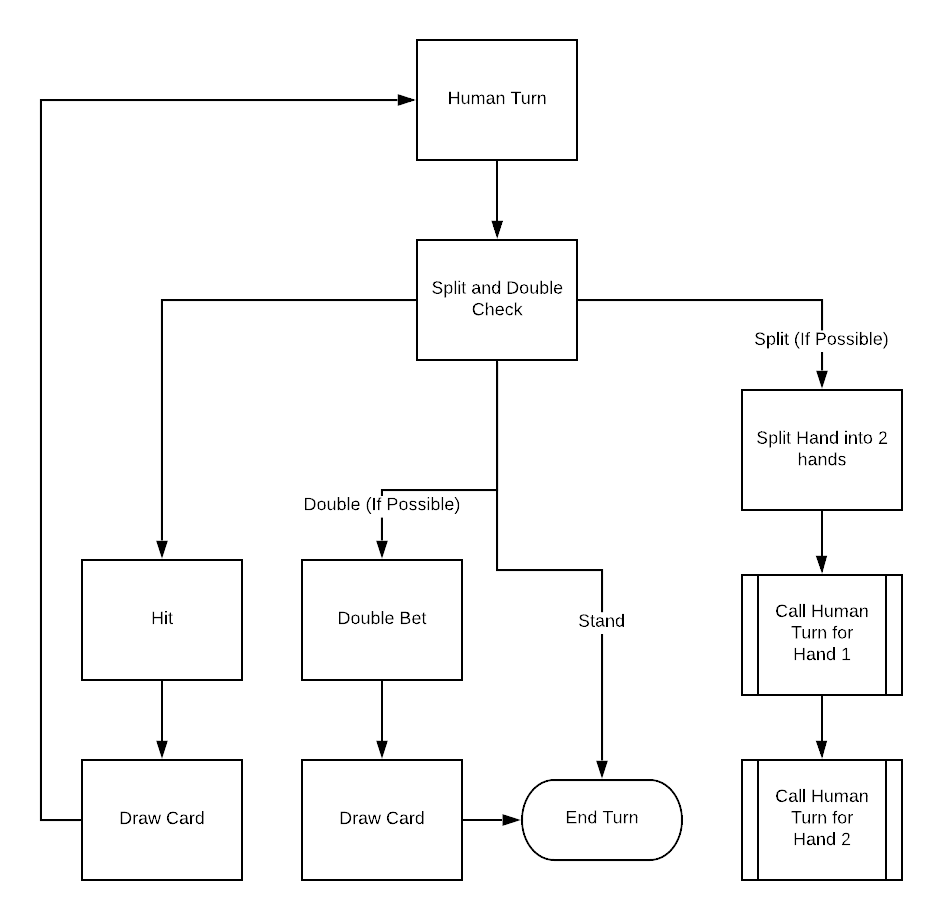
Natural Check – Natural 21 hands immediately end the game, so they need to be checked for next. If any Player(s) and/or the Dealer have a natural, they receive their winnings, and the game ends.

Figure 4 – Game Structure Flowchart

Human Turns – A check is made to determine what moves the Player can make. Then they are given the option to Hit, Stand, Double (if available), or Split (if available). This is repeated until the Player’s hand(s) are either bust or standing. See Figure 5.

BJP Turn – The BJP will choose the action that is designated for their current situation, until their hand(s) are standing or bust. Their situation is defined as a combination of: their hand value, the Dealer’s up card, whether they hold an ace, and if both of their cards are the same.

Figure 5 – Human Turn Flowchart

Dealer Turn – The Dealer will reveal their face down card to the Players. Then they will hit if their hand is worth less than 17, otherwise, they’ll stand.

Hand Evaluation – This section will check whether the dealer is bust, check if any Players have beat the Dealer, and check if any Players are in a stand-off with the Dealer. Winnings will then be awarded to the relevant Players.

End of Turn Breakdown – The round’s outcome, and winnings will be displayed, as well as the current standings. Afterwards, the user will be asked whether they would like another game, and the game will either exit, or loop back to the Game Initialisation phase.

### Class and Object Design

There will be classes for the Player, Dealer, and BJP, as well as the main program. The Player, Dealer and BJP will all be Player objects. Shared methods, such as gets and sets, will be stored in Player, whilst individual methods will be stored in their respective class.

The Dealer class will be responsible for handling the Deck, such as loading, shuffling and dealing. As well as holding the logic for the Dealer Player object, which will handle the down card reveal, and whether they hit or stand.

The Player class will hold the Player object specification, and constructor. In addition to, gets and sets, human turn logic, the hand assessment method, and any other methods that need access to Player member variables.

The BJP class will include methods related to the algorithm that selects a BJP Player’s action. This will require an AceCheck method and a PairCheck method, to select the correct Hard, Soft, or Pair method. Which will select the designated optimal action, based on the BJP’s card and the up card.

The Program class handles the flow of the game. It will mostly consist of calls to either the Player or Dealer class to store, collect, or operate on data, for executing turn and bet logic. However, it will also feature a significant amount of logic to determine the current game situation. Including: whether Player’s can double or split, if any Players have a natural 21, and if there is a stand-off between any Player(s) and the Dealer. It will also include all of the logic for the game’s options and setup. Such as: number of Players, manual or regular card input, and amount of decks.

# Implementation and testing

## Program - Flow of Game

Note: all methods and data members are explained with more detail in the following sections.

When the program is launched the user is prompted to set the game’s options, with a set of while loops. The available options are:

* Amount of CPU/BJP Players
* Amount of Human Players
* Manual Input or Regular Mode
* Amount of Decks

The amounts are stored as ints, and the mode is stored as a bool.

The user is then prompted to name their human Players, which are then created by calling the human Player constructor. Followed by the BJPs and Dealer, using their respective constructors.

A rematch bool is set as false, and a do while loop is opened. If the user later chooses to play again, this bool is set to true.

Next, the Initialise method is called, to read in and shuffle the deck, then an array is populated with all the Players.

After this, the user is prompted to enter a bet for each human Player. ChipCheck is used to determine if the Player has enough chips to make the bet, then InitialBet is called to set the bet.

Following this, BJPBet is called on each BJP Player. It checks their chips, and will set a bet as either 1/10, 1/5 or 1/3 of their chips, depending on how many they have.

Each Player is given their initial hand of 2 cards, using Draw and AddToHand. These hands are then printed in console, using PrintAllHands.

The next step is to check for natural 21s. There are 4 possible outcomes: no naturals, Player natural, Dealer natural or Player and Dealer natural. The int natCheck is used to hold the number that corresponds to the current situation, and the String List natList holds any Players who have a natural. All of the hands are evaluated using HandAssess, to check if any are natural 21s. In the event of a natural, the relevant Player is added to the natList, and natCheck’s state is changed.

There is then a switch statement on the value of natCheck:

Any Players that have a natural 21, receive their bet + 2x their bet/splitBet. The program then continues to the end of game breakdown.

In the event of a Dealer natural, the Dealer collects all of the Player’s bets, and the program continues to the end of game breakdown.

If both the Dealer and at least one Player has a natural, the Player(s) will receive their bet + 1x their bet/splitBet. Then the dealer collects any remaining bets, and the program continues to the end of game breakdown.

If no one has a natural, the game continues as normal.

A 3 state int: spdbCheck is used to determine if a Player is allowed to split, double, or both, based on their hand. This is done using the methods DoubleCheck, and SplitCheck. HumTurn is then called, passing spdbCheck as a parameter. This method will then call the relevant turn method, which will prompt for and execute the Player’s choice of action, until their turn is complete.

This is followed by the BJPTurn method being called for all computer-controlled players. After which, the DealerTurn method is executed.

Now that all Players have completed their turns, their hands are assessed, and then their bets evaluated. First, a check is made, to see if the Dealer’s hand is over 21. If it is, all Players win their bets, and the program moves to the end of game breakdown.

Next, two lists are created, stdoffList holds Players who have a hand equal to the Dealer’s, and winnerList holds Players with a higher hand than the Dealer.

Each Player is looped through, and placed in the relevant list, if they meet the criteria, and aren’t bust.

If there are no winners, a message is printed, and the program moves to the end of game breakdown.

If any Player wins, they win their bet/splitBet, then the end of game breakdown proceeds.

In the event of a stand-off, the relevant Player(s) win 1x their bet, and the Dealer collects the rest. Followed game breakdown.

Finally, all hands are printed to console, and the user is prompted if they want to play again. If they select yes, RematchReset is called, which sets every Player’s bust tokens and bets to their default values, and empties their hands. Then, NatCheck is reset, and a new game is started, with the same options.

## Deck

The deck is a list of strings, that each represent a different card. The format for these strings is ‘value’’suit’, so for example: a 2 of hearts is ‘2H’ and a queen of spades is ‘QS’.

Any methods related to the deck are stored in the Dealer class.

The Initialise method will read all of the cards in from a csv file called deckInput, using a StreamReader, and then add each one to the deck. It will loop this an amount of times equal to the supplied deckNo parameter. Next, it calls the Shuffle method on the, now populated, deck.

The Shuffle method works by swapping a card from the deck, in a random position, with another card from the deck, in an iterated position, until the iterator reaches 0. This is not truly random, because C# Randoms relies on the computer’s system clock, which is predictable. However, it is random enough for this game’s needs.

The Draw method is used to give players their initial hands, and extra cards if they choose to hit. It simply places the card in the top position of the deck into a temp value, removes the card from the deck, and returns the card in the temp value. However, if the manual input mode is being used (checks static manInput bool), the method will prompt the user for a card input, and if it matches the format, it is returned. Alternatively, they can input ‘card’ to use the non-manual input instead.

## Player – Object

This object is used to represent each of the players within the game, this includes the ‘Dealer’, ‘Human’ Players and the computer controlled ‘BJP’ Players.

The player object has 10 data members:

playerID and playerName are used to identify, and represent the player.

humTok is a bool to show whether a player is human or computer-controller. This is used in the main program to separate Players in the list, and call the relevant Human, BJP or Dealer methods for the Player.

hand and splitHand are lists of strings, used to hold the values of cards that the Player holds. A Player can split a pair hand (where both cards have the same value) into two hands, in this scenario, the splitHand is used to hold the new, second hand. All of the following split-related methods will operate on this ‘splitHand’, rather than the standard ‘hand’.

bustTok and splitBustTok are bools that show whether a player’s hand, or split hand is over 21. This is the metric used to determine whether a Player’s hand, or split hand, is out. It is used during hand evaluation, to check which Player(s) win the current round.

chips, is an int that holds a player’s current chip-count. Chips are the currency used to bet on a round, before it begins. Methods are operated on this data member to set the bets on a Player’s hands and splitHands.

bet and splitBet are used to represent the bet a player has placed on their hand and split hand. These members are operated on by methods which give chips to players, based on the outcome of a round. Player – List

The Player list is used to smoothly loop through, and operate on all of the Players in a game. In combination with the humTok, it is used to single out the BJP, or Human players, to initiate methods that are relevant to them.

## Player – Methods

The Player class contains a large amount of methods related to the Player object, Player List, and the turn logic for Humans, BJPs and the Dealer. (BJP and Dealer Logic in following sections)

AddPlayer – Adds specified Player to playerList.

GetPlayers – returns playerList as an array.

AddToHand/AddToSHand – Adds card to specified hand or splitHand.

PrintHand/PrintSHand – Returns string containing a Player’s name, bet and hand.

PrintDealerHand – Returns string containing Dealer’s up card.

PrintAllHands – Writes to console the hands of all Players, and Dealer, using previous methods.

PrintAllHandsEnd – Same as previous, except for Dealer’s hand. Used before down card reveal.

InitialBet – Sets Player’s bet as the supplied int, and removes the same amount of chips.

DoubleBet/DoubleSBet – Removes chips equal to bet, and then doubles Player’s bet.

SplitBet – Removes chips equal to bet, then sets splitBet to the same value as bet.

StandOffBet/StandOffSBet – Returns a Player’s bet to them. Used in the event of a stand-off.

WinBet/WinSBet – Adds 2x bet to chips.

GetBet/GetSBet – Returns value of a Player’s bet or splitBet

ChipCheck – Checks whether a Player has enough chips to bet the amount they have chosen.

GetChips (overloaded) – Returns chips as int or string.

SplitCheck – Checks that Player has sufficient chips, and compares cards in hand. If they’re the same, and they have enough chips to Split, return true, else false.

DoubleCheck/SDoubleCheck – Checks if a Player’s hand totals 9, 10 or 11, and that the Player has enough chips to Double their bet.

SplitHand – Sets splitHand[0] as hand[1], removes hand[1], and calls splitBet.

GoBust/SplitGoBust- Sets bustTok or splitBustTok to true.

BustCheck/SplitBustCheck – Checks whether bustTok or splitBustTok are true.

RematchReset – Sets bet and splitBet to 0. Next, sets bustTok to false, and splitBustTok to true. Then, removes all cards from hand and splitHand.

HandAssess/SplitHandAssess – Evaluates hand value, with handling for Aces and picture cards.

HumTurn – Calls BasicTurn, SplitTurn, DoubleTurn, or SplitDouble turn, depending on the value of int parameter spdbCheck. Which is set based on the actions are available to the Player.

BasicTurn/BasicSTurn – Loops asking Player whether they want to hit or stand, through console.

On hit, calls Draw and AddToHand, then HandAssess. If handValue > 21, Calls GoBust.

On stand, exits loop.

Else, prints invalid input message to console.

DoubleTurn/DoubleSTurn – Same as BasicTurn, with the additional option to double.

On Double calls DoubleBet, Draw and AddToHand, then HandAssess. If handValue > 21, Calls GoBust. Then exits loop.

SplitTurn – Same as BasicTurn, with the additional option to split.

On Split, call SplitPhase.

SplitDoubleTurn – Combination of SplitTurn and DoubleTurn.

SplitPhase – Sets splitBustTok to false and calls SplitHand. Then checks if either hand is eligible to double, using DoubleCheck and SDoubleCheck. Finally calls either DoubleTurn or BasicTurn as well as DoubleSTurn or BasicSTurn.

## Dealer – Player

The Player object used to represent the game’s dealer is almost identical to a regular player’s Player object. It is created using a slightly different constructer, which gives it a false humTok: so that the turn methods for human players don’t operate on it. As well as a large amount of chips, so that it cannot run out.

The Dealer is the last Player object to be created, so it is the last Player in the list. So, it is very accessible, which allows it to be easily ignored from loops, or operated on individually, when needed.

## Dealer – Methods

There is only one method specific to the Dealer: DealerTurn. This method involves revealing the Dealer’s face down card to the Players, then calling the handAssess method to check the value of the Dealer’s hand. If their hand is worth 17 or above, the dealer stands, otherwise they hit. If their hand is then worth more than 21, the goBust method is called, which changes their bustTok to true. This bustTok is checked during the Hand Evaluation phase, to determine which Player(s) win the round.

## BJP Logic

### Bet Logic

First, a new char: betSize is initialised, its possible values are h, m or l (high, mid, low) which represent how large a bet the BJP will place. This value is determined by the BJPBetSize method, which returns ‘l’ if the BJP has 1-24 chips, then ‘m’ if they have 25-40 chips, or ‘h’ if they have more than 40 chips.

A low bet is 1/10 of their current chips, a mid-sized bet is 1/5 of their current chips, and a high bet is 1/3 of their current chips. A switch statement on the result of BJPBetSize will calculate the resulting bet, and place it into a temporary int variable. Finally, InitialBet is called to set the BJPs bet as the temporary int.

### Turn Logic

After this, A new string: dcard is set as the value of the dealer’s up card, and a new char: ‘decision’ is set as ‘ ‘. Then, three new bools: pairBool, splitBool, and softHand are initialized. pairBool is used to check if both cards in a hand are worth the same. splitBool is used to represent whether BJP has decided to split. softHand is used to check if a hand contains an ace.

The splitCheck method is used to determine the value of pairBool. It removes the suit from the end of the card string, and checks whether the remaining strings are equal. If so, pairBool is true, otherwise its false.

This is all the data necessary to decide on the best move for the BJP to make.

The next section is looped, using a while, until either:

decision is set as ‘s’ (stand), decision is set as ‘p’ (pair), or the BJPs hand goes bust, and the loop is broken.

The BJP’s hand is looped over, and if it contains an ace, softHand is set to true.

If pairBool is true, it is set to false, and the bjpPair method is called.

If softHand is false, the bjpHard method is called, otherwise, the bjpSoft method is called.

MoveLogic is called to execute the logic for the action specified by decision. In addition, if decision equals ‘p’, split bool is set to true.

If bustTok is true, break out of the loop.

bjpHard, bjpSoft and bjpPair return ‘h’, ‘s’, ‘d’ or ‘p’, which correlate to hit, stand, double or pair, the four actions available to the BJP (‘p’ is unique to bjpPair). They all switch on the player’s hand value, and then, if necessary, the dealer’s card (dcard) to determine the optimal move. Which is based on the generated blackjack strategy sheet (Figure 3). This code is minimised by stacking cases (Figure 6).

A check is made to ensure that, if ‘d’ or ‘p’ are selected, the BJP has enough chips to Double or Split. If they don’t, decision is changed to ‘h’.

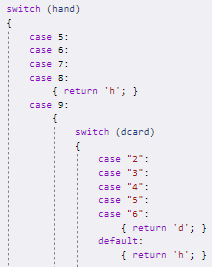


Figure 6 – Stacked case statements.

The result of this is that, once the chosen action has been returned:

‘s’ will result in the while loop terminating.

‘h’ will result in a card being added to the BJP’s hand, and the loop will continue, if the BJP does not go bust.

‘d’ will result in the BJP’s bet being doubled, followed by a card being added to their hand. Then decision will be set to ‘s’, which will terminate the while loop.

‘p’ will result in the BJP’s splitBustTok being set to false, meaning the program will check its split hand during hand evaluation. Next, the SplitHand method will be called. It will copy the second card from the BJP’s hand into the first position in their splitHand, and then remove the second card of the first hand. After which, it will call SplitBet, which assigns the value of BJP’s bet, to their splitBet Finally, the loop will continue.

After either standing or going bust with the first hand, another while loop is executed, which copies the previous while loop, but uses the split-variant methods. Meaning that the turn will have been completed for both hands.

## Class Relations

Most of the class relation details have already been explained, so this section will be a brief overview.

The software includes 3 classes:

Dealer holds methods to load, shuffle, and draw from the deck.

Player holds all of the methods and constructers related to the Player object and Player List, with variants for splitHands. This includes: gets and sets for hands and bets; bust check and go bust methods; checks for if a player can split or double during their turn; and methods to assess a hand’s value. As well as the turn logic related to Human, Dealer, and BJP Players, for every possible game situation.

The Program class is the entry point of the program. It holds the logic that handles the flow of the blackjack game. It will call the Player or Dealer class to store, collect, or operate on data.

## GUI Plan and Design

### Requirements

The interface needs to be simple, and easy to understand at a glance.

Any text, buttons and cards need to be clear and readable.

The design needs to be eye-catching, but not distracting.

### Original Design Specification and Inspiration

The GUI will be designed based on blackjack tables, taking cues from classic casino styling. It will feature a colour scheme of deep greens, golds and browns, which are traditionally present on blackjack tables. Appendix 1 contains images used as inspiration.

The menu and options screens will share a background, featuring a blackjack table from a distance.

The main game and replay screens will share a background of a blackjack table from a birds-eye view.

The main game will be designed to look as if it is being played on a blackjack table. The cards in play will be centrally located, with buttons and scores placed unobtrusively around the exterior of the screen.

### Original Design and Evaluation

The GUI draft was entirely created in Visual Studio, in Windows Forms. This draft’s purpose was to show the program’s layout, so visual quality was not a priority. However, the colour-scheme and backgrounds were replicated, as a proof of concept. Appendix 1 shows images of the GUI draft.

The draft’s layout was sufficient; however, it was decided that the styling needed more flair. The solution was to further simplify, and modernize the design. This modernisation would take the form of geometric line patterns, which would be featured in the updated table and cards. The menu and option screens would now also use this table as a background, to better fit the new house style.

### Final Design Specification and Inspiration

The GUI will be designed to look like a blackjack table, inspired by classic casino styling, but with a modern twist. It will feature a colour scheme of deep greens, golds and browns, which are traditionally present on blackjack tables. Appendix 1 contains images used as inspiration.

The menu, options, replay, and game screens will all use the same background.

The main game will be designed to look as if it is being played on a blackjack table. The cards in play will be centrally located, with buttons and scores placed unobtrusively around the exterior of the screen.

To make the program look unique, and give it some flair. A clean, modern, geometric line pattern will be applied to the board and cards.

### Final Design and Evaluation

The final design was also implemented in Visual Studio, in the same way as the draft. It was originally going to be completed with functional buttons, to move between forms. So that it could act as a prototype. However, unfortunately this wasn’t possible, due to time constraints.

The design is uncluttered, and easy to understand. It stands out, and suits the program well. It has i’s own style, but is still very recognisable as Blackjack. Overall, the design meets and exceeds expectations, it’s a shame that it couldn’t be fully implemented in time. Appendix 1 shows images of the Final GUI design.

## Implementation Reflection

Originally, the project was supposed to include an additional real-time image recognition program, which would be trained by a neural network, to recognise playing cards. It was planned that, using the manual input mode, the software could be used to play blackjack through a webcam or using a live screen capture. Research into the area showed that similar projects had been done before, and the selected program, YOLOv3 seemed to be capable, as well as easy to use. However, after installation difficulties, and recommendations to scale the project back, the feature was dropped. It was an interesting idea, but it seemed like it might be overengineering the project.

CLI was a noticeable design constraint. The focus of the project was implementing as much game functionality as possible, so there was no guarantee of a complete GUI. This meant that the program needed to support a CLI, which resulted in bulky code, with some awkward loops.

One regret is that the Player object wasn’t abstracted. Considering the 3 types of Player all use separate constructers and methods, they may as well have had their own class. Initially, they did have their own classes, but a way to access data members from the Player class, without compromising security, couldn’t be found. So, they were all reduced into Player, making it extremely bloated. Unfortunately, by the time abstraction was realised to be a solution, the change was too large a job, for too little gain. So, it wasn’t a priority.

On top of this, the Player turn logic should have been kept separate from the Player methods, this would have further reduced the bloatedness of the Player class.

In addition, it has been realised that there was a much simpler and cleaner solution to dealing with splitting hands. Splitting could have created a new Player object, that would be connected to the original object as a data member. This way, the split Player could use all the regular methods for its turns, bets etc, which would greatly reduce the amount of code, and remove the need for all the awkward workarounds that have been put in place.

In contrast, even though the code could be reduced by these methods, the existing code is clean, and well designed. Even the workarounds are nowhere near hacky. Every useful function has been created as a method, so all that is required of the logic is to single out each possible situation, with loops or switch statements, and make the necessary method calls. In addition, it is organised and thoroughly commented, to make it easy to read and understand.

# Testing

## Test Overview

Throughout development, features were tested whenever they were completed, or significantly updated. In accordance to the incremental model. However, this testing was informal, and not documented.

Towards the end of the project, tests were created, to ensure that all of the main features and functionality work as intended. Each test was to be completed manually, with the use of the manual input mode, where necessary. Using debug mode in Visual Studio, to verify results Any failed tests would be noted, and fixed, then re-tested, until working (time dependant).

Due to the nature of these tests, many can be completed within one game. So, the process was not as long as the large test table suggests.

## Test Plan

Figure 7 – Test Plan

# Evaluation

## Project Completeness

Currently, the project is mostly complete, compared to the re-scoped aims and objectives. The only missing elements are additional game-modes, which were a minor secondary objective, and the completed GUI.

## Current Functionality

In its current state, the program supports multiple Human and Computer-Controlled Players, multiple decks and manual or regular card input. Hit, Stand, Double and Split are all available actions, dependant on the Player’s hand. The game can end in a multitude of ways: natural 21 for Player, Dealer, or both; Player win, Dealer win, Dealer bust or stand-off. Betting is completely implemented, with differing rewards for each of these eventualities.

BJP Players bet based whether they have a large or small chip count. In addition, they will consistently select the best actions available to them, whether they have a hard, soft or pair hand. The Dealer will follow the S17 rule, by standing on any hand 17 or above, and hitting on any other hand.

The CLI allows for option values and action choices to be input easily, although there is no handling for similar/imperfect inputs. In addition, it clearly shows all of the necessary information, such as hands, chip totals, and each action a Player takes; as well as how a Player wins, or if they stand-off with the Dealer, and how many chips each Player wins.

## Testing Evaluation

A majority of the tests worked as expected first time. However, there was a fair amount of logic alteration, and additional code required to make all tests pass. This table shows all of the bugs/issues encountered, and their solutions.

|  |  |  |
| --- | --- | --- |
| Issue | Cause | Resolution |
| Hands, bets, bustToks and the nat check remained the same on rematch. | No method in place to reset values. | RematchReset method created to reset hands bets and bustTok. NatCheck set to 0, and RematchReset called when rematch = true. |
| HandAssess not reading strings correctly | Tryparse and case in handassess checking wrong string. | Swap logic to operate on trim string, instead of input string. |
| HandAssess valued Ace to be 1, in situation where it should have been 11 | Less than instead of greater than in Ace handling. | Swap if(hand + 11 < 21) to  if(hand + 11 > 21) |
| All Players being added to winnerList and standOffList | !() was expected to function as ‘does not equal true’, but did not | Changed if!(bustCheck) to if(bustCheck) !=true |
| splitHands not being evaluated correctly | Evaluation methods were checking regular hands, not splitHands. | Created split variants of standoffList and WinnerList, to use in checks for splitHands |
| Doubling not available during second turn | No DoubleCheck after Hit or Split | Added if(DoubleCheck) {doubleTurn} after hit or split choices |
| Players can choose Double or Split when they don’t have enough chips for the bet | No chipcheck before double or split offered | DoubleCheck and SplitCheck edited to only return true if chips >= bet |
| BJP can Double or Split when they don’t have enough chips for the bet | No chip checks in BJP logic | Additional logic created to check if BJP has enough chips to Double or Split, if they have chosen to. If they don’t, their action is changed to Hit. |
| Dealer Natural does not trigger nat check | Dealer’s Player skipped in nat check | Removed -1 from for(players.count-1) |

# Conclusion

## Project

All of the project’s aims and objectives have been achieved, with the exception of a GUI, and game-modes. It’s in an almost complete state, any additional functionality would be a luxury, rather than a necessity.

The program functions as a capable Blackjack simulator, and contains powerful game customisation options. Which can be used effectively to achieve its purpose, to train and reinforce optimal strategies and betting habits.

It is a useful, but niche, piece of software. A lot of the software that could be seen as ‘competing’ is aimed at cheating or automating games of Blackjack, and seems to only be used to make money from gambling websites. In contrast, this project aims to train users so that they don’t need to rely on software to win games.

Some features were stripped from the original specification. A feature of particular interest was the webcam card recognition. It seemed to have a lot of possibilities and expansions, such as a mobile app, or screen capture card recognition. In addition, it would work using neural networks, a very interesting and useful technology to explore. However, as to not over-engineer the project, keeping the focus on betting and strategy, the feature needed to be dropped.

If the project were to be expanded, continuing as a Blackjack training program, the key additional features would be:

A tutorial section, including card counting tips, strategy sheets and strategy memorisation tips.

A custom strategy sheet generator, with a variety of settings and rules.

However, the project also has potential for a mobile application, or an autonomous Blackjack tool, as well as the previously mentioned card recognising program.

## Process

The incremental methodology suited my workstyle, and I would use it again. It allowed me to easily revisit and update features, without having to rework or re-test the entire project. My only regret is that I had planned my tests and recorded my results. However, I made effective use of notes and logging to ensure any issues or bugs wouldn’t be forgotten, so that they could be fixed promptly.

In the beginning, my time planning was good, I was following my Gantt chart effectively, and work was consistently being completed. However, throughout the project, this organisation slipped, as other deadlines took priority.

Did you complete the primary and secondary objectives, and if not then why?

After stripping unnecessary features large from the original plan, most of the objectives were very completable. So, it was relatively easy to implement the majority of them.

The GUI was put off due to limited experience, whilst focusing on core game features. Until it got to the point that it wouldn’t be completable in time.

Similarly, game modes were put off, but because they were a minor, ‘good-to-have’ feature. I’m not hugely disappointed that they are missing from the project.

If I could re-do the project, I would like to abstract the Player class, to remove clutter. As well as removing all the split methods, and instead create a new Player, when someone splits. These two improvements would vastly reduce the amount of code, as well as making it cleaner and more readable.

# References

Anon., 2010. *Wizard of Odds.* [Online]   
Available at: https://wizardofodds.com/games/blackjack/strategy/calculator/  
[Accessed 2020].

Anon., 2015. *Casinos Online.* [Online]   
Available at: https://www.casinosonline.com/articles/soft-17-blackjack-rules-s17-vs-h17/  
[Accessed 2020].

Anon., 2017. *Blackjack.com.au.* [Online]   
Available at: https://www.blackjack.com.au/games/21-peek/  
[Accessed 2020].

Anon., 2018. *BlackJack Apprentiship.* [Online]   
Available at: https://www.blackjackapprenticeship.com/wp-content/uploads/2018/08/BJA\_Basic\_Strategy.jpg  
[Accessed 01 2020].

Anon., n.d. [Online]   
Available at: https://www.gamblingsites.net/wp-content/uploads/2016/05/blackjack-table-at-casino.png  
[Accessed 2020].

Anon., n.d. [Online]   
Available at: https://img.thedailybeast.com/image/upload/c\_crop,d\_placeholder\_euli9k,h\_1439,w\_2560,x\_0,y\_0/dpr\_2.0/c\_limit,w\_740/fl\_lossy,q\_auto/v1492177299/articles/2015/08/27/i-played-blackjack-with-the-world-s-best-card-counter/150826-strochlic-cards-tease\_cxl3af  
[Accessed 2020].

Anon., n.d. [Online]   
Available at: https://thumbs.dreamstime.com/z/vector-seamless-pattern-modern-line-geometric-background-texture-lines-grid-rhombus-nodes-63440786.jpg  
[Accessed 2020].

Anon., n.d. [Online]   
Available at: https://media.istockphoto.com/vectors/modern-simple-geometric-vector-seamless-pattern-with-gold-line-on-vector-id1018223248  
[Accessed 2020].

Anon., n.d. *Blackjack.org.* [Online]   
Available at: https://www.blackjack.org/blackjack-calculator/#  
[Accessed 2020].

Anon., n.d. *Blackjack.org.* [Online]   
Available at: https://www.blackjack.org/blackjack-strategy/  
[Accessed 2020].

Anon., n.d. *Casino.org.* [Online]   
Available at: https://www.casino.org/features/house-edge/  
[Accessed 2020].

Anon., n.d. *Casino.org.* [Online]   
Available at: https://www.casino.org/features/house-edge  
[Accessed 2020].

Anon., n.d. *Counting Edge.* [Online]   
Available at: https://www.countingedge.com/the-in-between-blackjack/  
[Accessed 2020].

Anon., n.d. *Online Gambling.* [Online]   
Available at: https://www.onlinegambling.com/casino/blackjack/house-edge/  
[Accessed 2020].

Elgabry, O., 2017. *Medium.* [Online]   
Available at: https://medium.com/omarelgabrys-blog/shuffling-35fb8e303f7c

PokerStars, 2015. *Youtube.* [Online]   
Available at: https://www.youtube.com/watch?v=-DkHzOUzDjc  
[Accessed 2020].

S, J., 2014. *Stack OverFlow.* [Online]   
Available at: https://stackoverflow.com/questions/25027339/how-to-get-data-from-inspect-element-of-a-webpage-using-python  
[Accessed 26 07 2020].

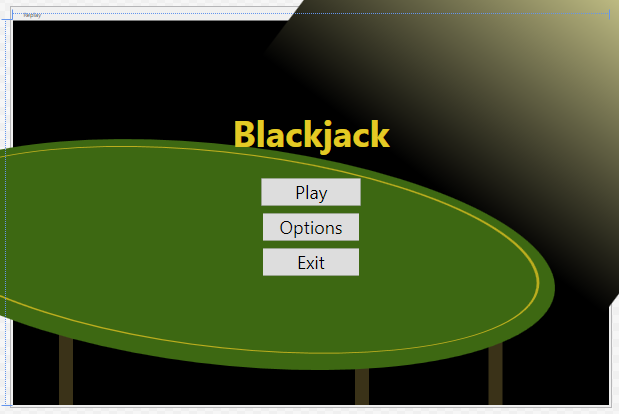
Stanley, T., 1982. *Justia.* [Online]   
Available at: https://law.justia.com/cases/new-jersey/supreme-court/1982/89-n-j-163-0.html  
[Accessed 2020].

Stevens, M., 2020. *GamblingSites.org.* [Online]   
Available at: https://www.gamblingsites.org/blog/can-you-count-cards-with-continuous-shuffling-machine/  
[Accessed 2020].

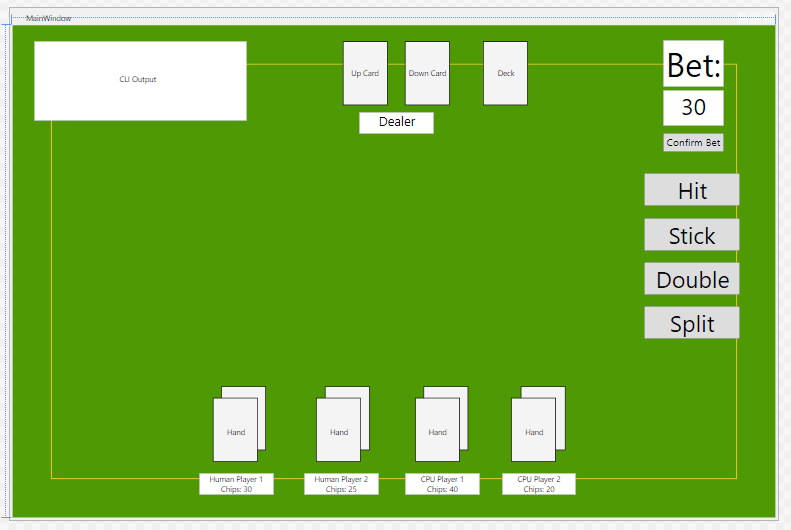
Appendix 1 – GUI Design Process

## GUI Draft Design

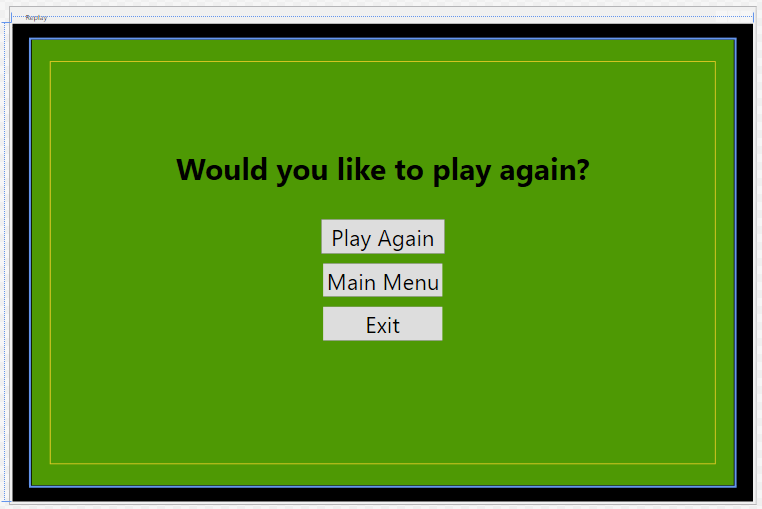
### Menu Screen



### Game Board



### Replay Screen



## Inspiration

### Blackjack Tables

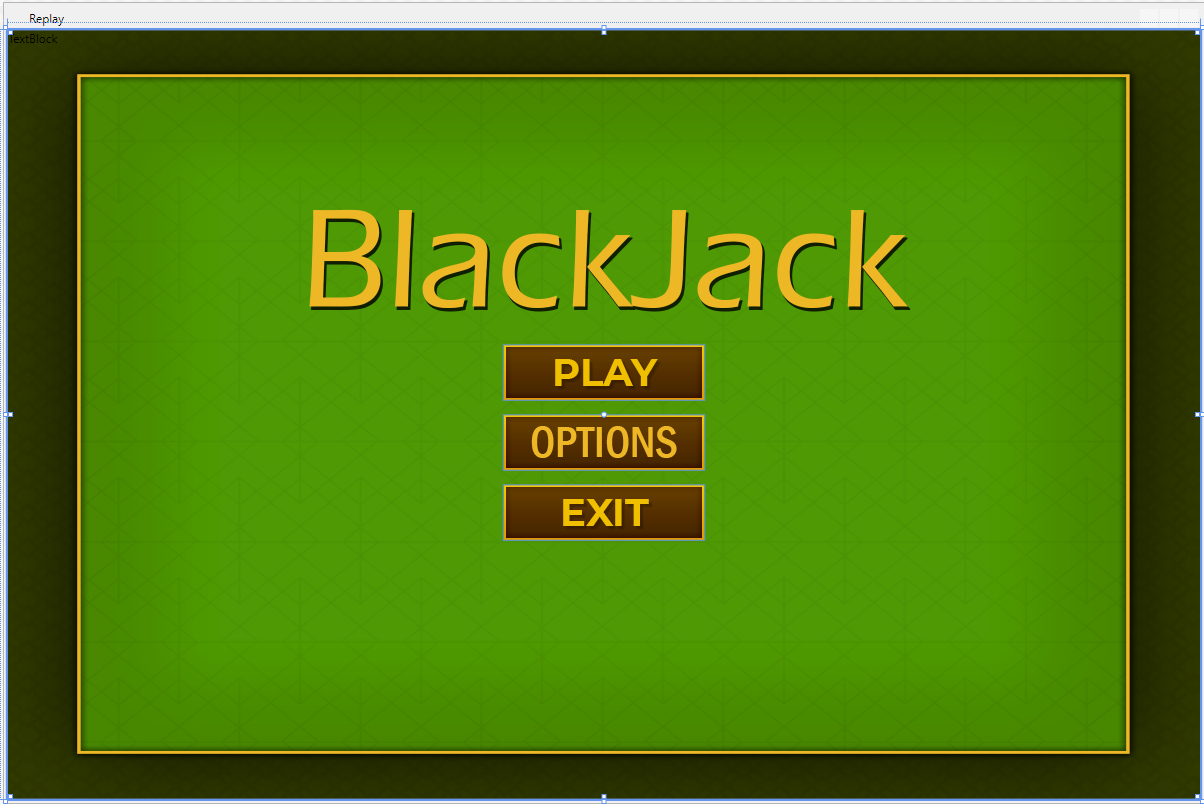
(Anon., n.d.) (Anon., n.d.)

### Geometric Line Patterns

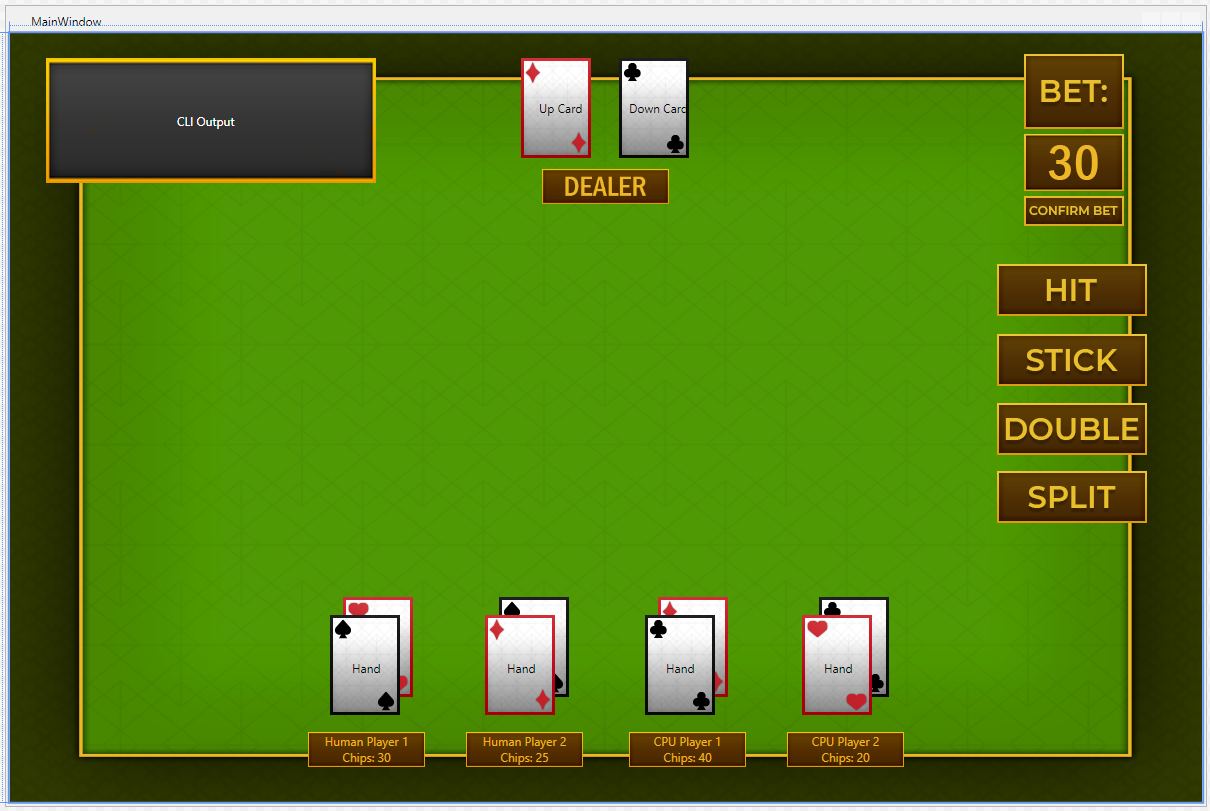
(Anon., n.d.) (Anon., n.d.)

## Final GUI Design

### Menu Screen



### Game Board



## Replay Screen

