“Lincoln” card game development

This paper describes whole application structure planning and realization. Moreover, it has a completed checklist initially provided and describes application testing process. Also, this paper describes OOP features used in code and their location.

Table of Contents

LIST OF FIGURES

LIST OF ABBREVIATIONS

BVA	Boundary Value Analysis
ECP	Equivalence Class Partitioning
FSM	Finite-state machine
OOP	Object-oriented programming
UT	Usability Testing

Application structure

Planning

To make any application with complex behavior it should be described in the details. To make easier to develop such app, the author used problem decomposition principle. To facilitate the control of application even more, FSM (Brilliant.org, 2021) and Abstract Factory (Erich Gamma, et al., 1994, 87-95) patterns were used.

To visualize whole application architecture and different classes interaction, author have created UML-diagram (see Figure 1).

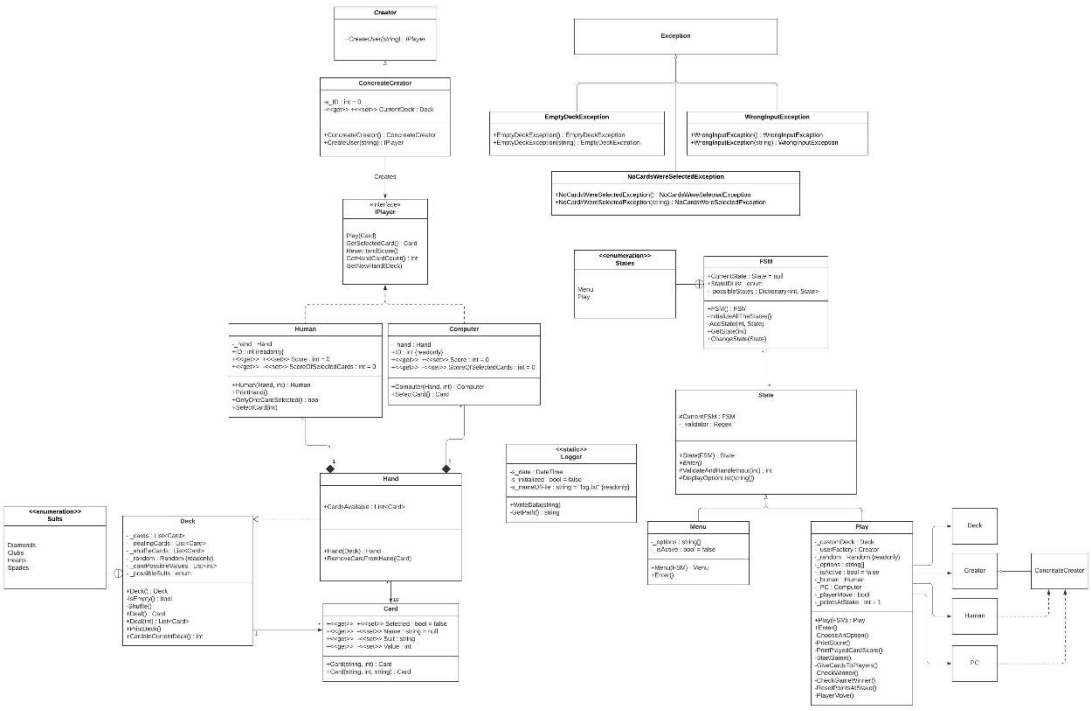

Figure 1 UML diagram of application

In total, author come up with following structure:

Deck – class responsible for handling such deck logic as shuffling and dealing one or more cards.

Card – class responsible for representation card object with suit, value, and name (for improving visual aesthetics of card representation).

Hand – class responsible for handling available for playing cards for user.

EmptyDeckException – exception which is thrown when deck is empty

NoCardsWereSelectedException – exception which is thrown when user tries to select card while another is selected or if user “confirms” the selected cards, but there is more than 1 selected.

WrongInputException – exception which is thrown when user tries to put invalid input.

Logger – static class responsible for log file generating. File contains data such as: action of user, ID of users, cards being played, score of each user etc.

Creator – abstract class, responsible for getting “interface” for creating different types of users.

ConcreateCreator – class which inherits Creator and is responsible for creating different types of users.

IPlayer – interface, which handles such logic as playing a card by user, getting selected card, reset hand score, getting number of cards in hand, and getting new hand.

Computer – class which implements IPlayer interface. Represents a computer type player, who choose card based on random.

Human – class which implements IPlayer interface. Represents a human type player, who can choose card based on user input.

FSM – class responsible for initializing and handling changing logic of the states.

State – abstract class which validates user input by using Regex and displays options available to user.

Menu – class which inherits State and is responsible for handling the logic of application in main menu.

Play – class which inherits State and is responsible for handling the play logic of the application.

Realization

Deck

Deck class has 4 lists, which are responsible for handling all the cards available in deck, dealing multiple cards, temporary storing shuffled deck and handling card possible values respectively (see Figure 2). Also, a `Random` object was created for dealing random card to user (see Figure 3). For generating all the possible cards with different suits one of previously mentioned lists was used in cooperation with enumerator.

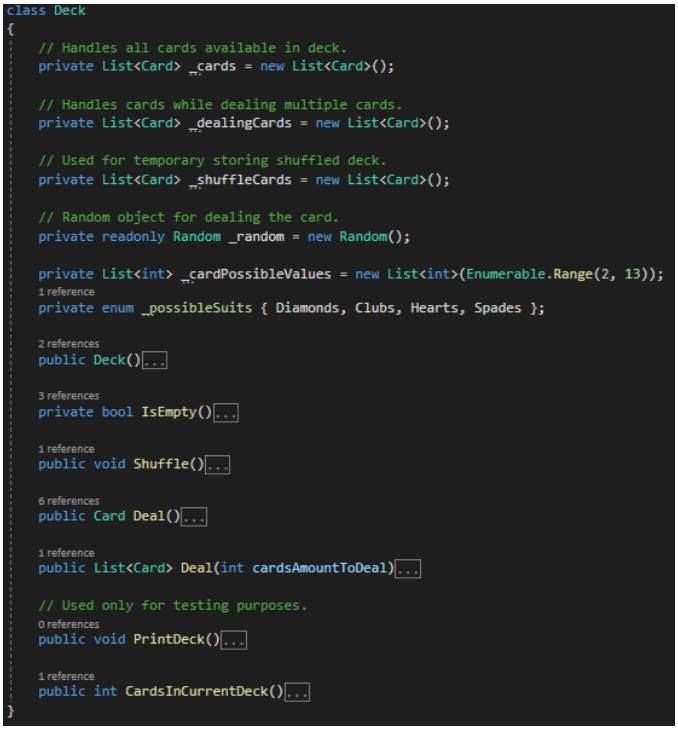

Figure 2 Deck class

```
// Get random card from deck
int randomIndex = _random.Next(0, _cards.Count);
Card randomCard = _cards[randomIndex];
_cards.RemoveAt(randomIndex);
```

Figure 3 Get random card from deck

Card

`Card` class has 4 auto-properties fields with appropriate access modifier as well as its constructor overloading in case the card should have custom name (see Figure 4).

```
class Card
{
    7 references
    public bool Selected { get; set; } = false;
    5 references
    public string Name { get; private set; } = null;
    7 references
    public string Suit { get; private set; }
    23 references
    public int Value { get; private set; }

    1 reference
    public Card(string requstedSuit, int requestedValue)
    {
        Suit = requstedSuit;
        Value = requestedValue;
    }

    4 references
    public Card(string requstedSuit, int requestedValue, string newName)
    {
        Suit = requstedSuit;
        Value = requestedValue;
        Name = newName;
    }
}
```

Figure 4 Card class

Hand

Hand class has a list of cards available with represents the hand itself. In constructor of Hand object requests 10 cards and appends them to the list of available cards (see Figure 5).

```
class Hand
{
    public List<Card> CardsAvailable = new List<Card>();


    public Hand(Deck currentDeck)
    {
        try
        {
            CardsAvailable.AddRange(currentDeck.Deal(10));

            if (CardsAvailable.Count < 10)
            {
                throw new Exception("Not enought cards were dealt!");
            }
        }
        catch (Exception)
        {
            // Handle the exception.
        }
    }

    2 references
    public void RemoveCardFromHand(Card cardToDelete)
    {
        CardsAvailable.Remove(cardToDelete);
    }
}
```

Figure 5 Hand class

Exceptions

Each of previously described exceptions have similar structure. Firstly, it inherits from Exception class, then have constructor in case throwing exception without any message and otherwise have a constructor where developer may pass the message for the user (see Figure 6).

```
class EmptyDeckException : Exception
{
    0 references
    public EmptyDeckException()
    {

    }

    1 reference
    public EmptyDeckException(string message) : base(message)
    {
        Console.WriteLine(message);
        Console.WriteLine("");
    }
}
```

Figure 6 Exception class example

Logger

Static class checks if it is not initialized and file exists, in this case such file is not valid and is being deleted (see Figure 7). After that, logger gets current date and append passed text to a file with time stamp (see Figure 8).

```
static class Logger
{
    private static DateTime s_date;
    private static bool s_initialized = false;
    private static readonly string s_nameOfFile = "log.txt";


    public static void WriteData(string text)
    {
        if (!s_initialized && File.Exists(GetPath()))
        {
            File.Delete(GetPath());
            s_initialized = true;
        }

        s_date = DateTime.Now;

        File.AppendAllText(GetPath(), "["+s_date.ToString("F")+"]" +"\t"+ text+"\n");
    }


    private static string GetPath()
    {
        return Path.Combine(Directory.GetCurrentDirectory(), s_nameOfFile);
    }
}
```

Figure 7 Logger class

```
[15 May 2021 12:29:48]  FSM Initialized
```

Figure 8 Time stamp example

Creator

`Creator` class has only one abstract method for working with corresponding factory (see Figure 9).

```
abstract class Creator
{

    public abstract IPlayer CreateUser(string type);
}
```

Figure 9 Creator abstract class

ConcreateCreator

To create a different types of user `ConcreateCreator` was created. It also handles deck object to pass it to hand constructor. Depending on input string, new type object with unique deck and ID will be created (see Figure 10).

```
class ConcreateCreator : Creator
{
    private static int s_ID = 0;

    public Deck CurrentDeck { private get; set; }


    public ConcreateCreator() { }


    public override IPlayer CreateUser(string type)
    {
        switch (type)
        {
            case "PC":
                s_ID++;
                return new Computer(new Hand(CurrentDeck), s_ID);

            case "Human":
                s_ID++;
                return new Human(new Hand(CurrentDeck), s_ID);

            default: throw new ArgumentException("Invalid Type", type);
        }
    }
}
```

Figure 10 ConcreateCreator class

IPlayer

`IPlayer` interface (see Figure 11) describes all the methods used by different types of players as `PC` and `Human`.

```
interface IPlayer
{

    void Play(Card card);


    Card GetSelectedCard();


    void ResetHandScore();


    int GetHandCardCount();

    4 references
    void GetNewHand(Deck deck);
}
```

Figure 11 IPlayer interface

Human

This class object has custom hand and ID (see Figure 12). Also, it has field of hand score (`Score`) and score of played cards (`ScoreOfSelectedCards`).

```
class Human : IPlayer
{
    private Hand _hand;
    public readonly int ID;


    public int Score { get; set; } = 0;


    public int ScoreOfSelectedCards { get; private set; } = 0;

    1 reference
    public Human(Hand newHand, int newID)...

    7 references
    public void Play(Card cardToPlay)...

    4 references
    public void GetNewHand(Deck deck)...

    3 references
    public Card GetSelectedCard()...

    6 references
    public void ResetHandScore()...

    1 reference
    public void PrintHand()...

    11 references
    public int GetHandCardCount()...

    2 references
    public bool OnlyOneCardSelected()...

    1 reference
    public void SelectCard(int userInput)...
}
```

Figure 12 Human class

Computer

Computer class implementation is like Human, but only with one method edit – `SelectCard` (see Figure 13). Instead of selecting card depending on user input, computer selects it based on random.

```
public Card SelectCard()
{
    int cardIndex = _random.Next(0, _hand.CardsAvailable.Count);
    return _hand.CardsAvailable[cardIndex];
}
```

Figure 13 SelectCard method in Computer class

FSM

FSM (see Figure 14) have reference to current state as well as all possible states which being saved in Dictionary based on state ID as a key and state object itself as a value. After FSM object is created, it initialises all the possible states, by calling according method.

```
class FSM
{
    public State CurrentState = null;

    6 references
    public enum StateIDList
    {
        Menu,
        Play
    }

    private Dictionary<int, State> _possibleStates = new Dictionary<int, State>();

    1 reference
    public FSM()
    {
        InitializeAllTheStates();
        Logger.WriteData("FSM Initialized");
    }

    1 reference
    private void InitializeAllTheStates()
    {
        AddState((int)StateIDList.Menu, new Menu(this));
        AddState((int)StateIDList.Play, new Play(this));
    }

    2 references
    private void AddState(int stateID, State state)
    {
        _possibleStates.Add(stateID, state);
    }

    4 references
    public State GetState(int stateID)...

    4 references
    public void ChangeState(State newState)...
}
```

Figure 14 FSM class

State

Each of the states handles FSM object to be able to change the state as well as regex to validate the user input (see Figure 15). Each state should have Enter method where all state behaviour should be described. `DisplayOptionList` method prints to user possible options to choose from based on provided array of strings. Last, method called `ValidateAndHandleInput` is responsible for parsing the user input value and return error if any problems noticed (see Figure 16).

```
abstract class State
{
    protected FSM CurrentFSM;

    // Regex for validation of user input.
    protected Regex Validator = new Regex(@"^[1-9]\d*$");

    2 references
    public State(FSM fsm)
    {
        CurrentFSM = fsm;
    }

    3 references
    public abstract void Enter();

    3 references
    protected int ValidateAndHandleInput(int maxValue)...

    // Display options from provided list.
    2 references
    protected void DisplayOptionList(string[] optionsToDisplay)...
}
```

Figure 15 State class

```
protected int ValidateAndHandleInput(int maxValue)
{
    string userInput = Console.ReadLine();
    Console.WriteLine("");

    // Validate input with regexp.
    if (!Validator.IsMatch(userInput))
    {
        throw new WrongInputException("USER ERROR: Passed wrong option!");
    }

    // If integer provided is too large - handle the exception.
    int result;
    try
    {
        result = Int32.Parse(userInput);
    }
    catch (OverflowException)
    {
        result = 0;
    }

    // Validate based on number of available options.
    if (result > 0 && result <= maxValue)
    {
        return result;
    }
    else
    {
        throw new WrongInputException("USER ERROR: Passed wrong option!");
    }
}
```

Figure 16 Validate user input method

Menu State

To create infinite loop author have create `_isActive` variable, which is going to be set to false, only after switching the states or exiting the application. To show user options available (see Figure 17), new list of options was created and previously mentioned method `DisplayOptionList` is used.

```
private string[] _options = {
    "Start new game",
    "Exit"
};

private bool _isActive = true;
```

Figure 17 Menu fields

When user enters the app, menu state is set as starting point for user. After options being printed, application is waiting for valid input (see Figure 18). After that depending on its value it enters the play state or exits the application (see Figure 19).

```
public override void Enter()
{
    Logger.WriteData("User entered the menu");
    _isActive = true;

    while (_isActive)
    {
        Console.WriteLine("");

        int inputValue = -1;

        // Get good input.
        while (inputValue == -1)
        {
            try
            {
                DisplayOptionList(_options);
                inputValue = ValidateAndHandleInput(_options.Length);
            }
            catch (WrongInputException)
            {
                // Handle the exception.
            }
        }
```

Figure 18 Menu Enter method input validation part

```
try
{
    switch (inputValue)
    {
        // Start the game.
        case 1:
            _isActive = false;
            CurrentFSM.ChangeState(CurrentFSM.GetState((int)FSM.StateIDList.Play));
            break;

        // Exit.
        case 2:
            _isActive = false;
            Logger.WriteData("User exists the application");
            Console.WriteLine("Thanks for using our Software!");
            break;

        default:
            throw new WrongInputException("USER ERROR: Passed wrong option");
    }
}
catch (WrongInputException)
{
    // Handle exception.
}
```

Figure 19 Menu Enter method selection part

Play State

Similar to menu state, play state (see Figure 20) also have list of options and `_isActive` field. Moreover, play state handles user factory for creating new users, random object for deciding who move first, `_playerMove` field to save random returned value, `_human` and `_PC` player objects and field called `_pointsAtStake` which represents points winner of next hand will get.

```
class Play : State
{
    private Deck _customDeck;
    private Creator _userFactory;

    private Random _random = new Random(Guid.NewGuid().GetHashCode());

    // Options to choose from.
    private string[] _options = ...;

    // State status.
    private bool _isActive = true;

    // Players objects.
    private Human _human;
    private Computer _PC;

    // Determines if player should move first.
    private bool _playerMove;

    // How many hands are at stake.
    private int _pointsAtStake = 1;
```

Figure 20 Play class

OOP

Most of the OOP features as classes, inheritance, abstraction, polymorphism etc. which were used in this project and their location might be observed in Table 1.

Table 1 OOP feature location

OOP feature	Location
Classes	DeckStructure/Deck.cs
Abstract class	UserFactory/Creator.cs
Static class	Logs/Logger.cs
Object instantiation	Program.cs
Encapsulation	Logs/Logger.cs
Methods	FiniteStateMachine/FSM.cs
Data abstraction	FiniteStateMachine/States/Play.cs

Inheritance	FiniteStateMachine/States/Play.cs
Virtual/abstract methods	FiniteStateMachine/States/State.cs
Static methods	Program.cs
Interface	UserFactory/Users/IPlayer.cs
Static polymorphism	DeckStructure/Hand.cs
Dynamic polymorphism	FiniteStateMachine/States/Menu.cs

Testing

Black box

Black box testing approach is based on that user does not know internal structure of application. The main objective of black box testing is to check what functionality is working and what is not.

For more complex and accurate testing following techniques are used: Equivalence Class Partitioning (ECP), Boundary Value Analysis (BVA) and Usability Testing (UT).

Equivalence Partitioning idea is to separate inputs and group them by similar behavior. Hence selecting one input from each group to design the test cases (Rajkumar, 2018).

Main two groups created by author are selecting options in any of the menus and selection of the card. Firstly, to test selecting option category author have created multiple tests with different inputs and expected outcomes (see Table 2).

Table 2 ECP option input testing

File Path	File Input	Expected Outcome	Outcome confirmed (Y/N)
SelectionTest/ECP/test1.txt	-99999	Exception Message	Y
SelectionTest/ECP/test2.txt	999999	Exception Message	Y
SelectionTest/ECP/test3.txt	2	Print rules	Y

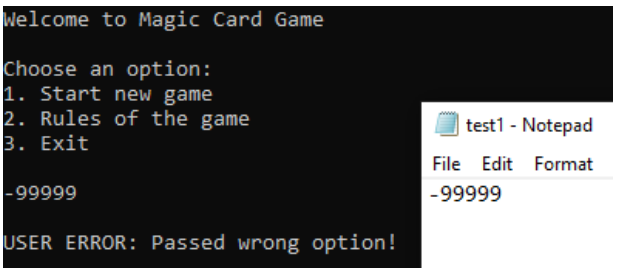

Figure 21 ECP option input testing example

After successful testing of options selection (see Figure 21), author have created tests for selection of cards assuming user is entered that state already (see Table 3).

Table 3 ECP card select input testing

File Path	File Input	Expected Outcome	Outcome confirmed (Y/N)
PlayTest/ECP/test1.txt	-99999	Exception Message	Y
PlayTest/ECP/test2.txt	999999	Exception Message	Y
PlayTest/ECP/test3.txt	2	Select Card	Y

PlayTest/ECP/test4.txt	2 11	Select and play a card	Y
PlayTest/ECP/test5.txt	11	Exception Message	Y

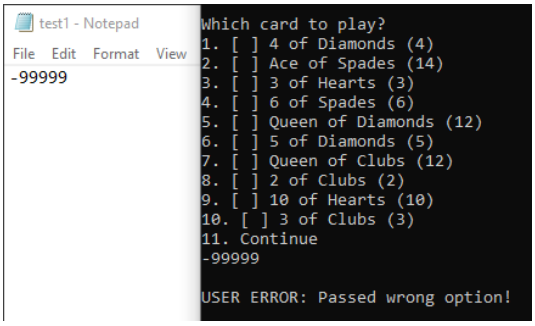

Figure 22 ECP play card testing example

After ECP testing (see Figure 22), next was BVA which is based on ECP but is more focused on the values at boundaries by determine a range accepted by system. Mostly BVA technique is useful where input is within certain ranges (Guru99, 2021).

After determining boundaries in application, following values are usually being tested by BVA:

- Minimum-1
- Minimum
- Minimum+1
- A nominal value
- Maximum-1
- Maximum
- Maximum+1

As in application option menus there is insufficient amount of options, some values tested by BVA were skipped (see Table 4 and Figure 23).

Table 4 Option selection BVA testing

File Path	File Input	Expected Outcome	Outcome confirmed (Y/N)
SelectionTest/BVA/test1.txt	0	Exception Message	Y
SelectionTest/BVA/test2.txt	1	Enter Play State	Y
SelectionTest/BVA/test3.txt	2	Print rules	Y
SelectionTest/BVA/test4.txt	3	Exit the application	Y
SelectionTest/BVA/test5.txt	4	Exception Message	Y

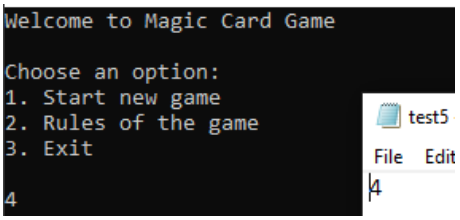

Figure 23 Option select BVA testing example

Same as ECP testing, BVA was used to test card selection and play actions (see Table 5 and Figure 24).

Table 5 BVA card select input testing

File Path	File Input	Expected Outcome	Outcome confirmed (Y/N)
PlayTest/BVA/test1.txt	0	Exception Message	Y
PlayTest/BVA/test2.txt	1 1	Select and deselect card	Y
PlayTest/BVA/test3.txt	2 11	Select and play card	Y
PlayTest/BVA/test4.txt	5 11	Select and play card	Y
PlayTest/BVA/test5.txt	10	Exception Message	Y
PlayTest/BVA/test6.txt	11	Exception Message	Y
PlayTest/BVA/test7.txt	12	Exception Message	Y

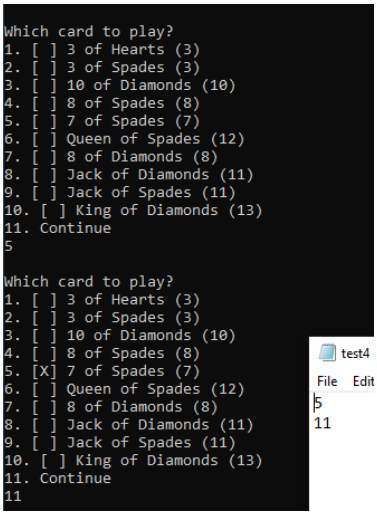

Figure 24 BVA play card testing example

Moreover, another testing technique was used - usability testing. The main idea of such is to test a usability of the application by end-user (Usability.gov, n.d.). For usability testing accuracy author invited non-developer person: Edwards Strazd (edwardstrazd@gmail.com).

As fast as the empty document with all the expected behavior and executable were ready, they were sent to the tester.

Table 1 Black box testing requirement specification table

Functionality to test	Works? (Y/N)	Other notes (how to improve)
User is greeted	Y	Short rule explanation would be great
User can see the menu	Y	
User can enter play mode	Y	
User gets 10 cards in hand	Y	
User can see hand of cards	Y	
User can choose card to play	Y	It's confusing that you can pick multiple cards at the same time
User can play a card	Y	
User sees played card values	Y	
User can win	Y	
User can lose	Y	
User can make draw by total score	N	
User can make draw by hand score	Y	
After getting a draw, winner of next hand gets both hand	Y	
User sees the hand score	Y	
User sees game winner when leaving the game	Y	
In case of draw by hand score on last hand – users see random card being played	Y	If the random card is the same the game needs a message that the second random card will be drawn
User can exit the application	Y	

Other notes (what to improve):

When having a draw with the last cards, the explanation of further events and score changes are confusing and not described.

Figure 25 Black box testing outcome from Edward Strazd

After receiving a feedback (see Figure 25), author have made several changes to the code accordingly to it (see Figure 26, Figure 27, Figure 28 and Figure 29).

```
Choose an option:
1. Start new game
2. Rules of the game
3. Exit

2

Lincoln card game rules:
1. Each from 2 players gets 10 cards from deck. Each card have it value from 2 up to 14.
2. Players takes moves by playing 1 card each(2 in total)
3. Player with highes sum of card values wins the round and makes move first on the next round.
4. If totals are the same, continue to the next hand. Winner of that gets both rounds.
5. If the number of hand wins are the same, draw a random card from the remaining cards - highest wins.
6. If the final hands are the same value, draw a random card from the remaining cards - highest wins the hand.
7. Player with highest number of hand wins, wins the game
```

Figure 26 Added rules

```
Which card to play?
1. [X] 2 of Spades (2)
2. [ ] 8 of Clubs (8)
3. [ ] 5 of Clubs (5)
4. [ ] 2 of Clubs (2)
5. [ ] 4 of Hearts (4)
6. [ ] 8 of Hearts (8)
7. [ ] 6 of Spades (6)
8. [ ] 9 of Spades (9)
9. [ ] 5 of Hearts (5)
10. [ ] 6 of Hearts (6)
11. Continue
3

Any other card already selected!
```

Figure 27 Prevent user from selecting more than 1 card

```
public void SelectCard(int userInput)
{
    userInput--;

    if (_hand.CardsAvailable[userInput].Selected)
    {
        _hand.CardsAvailable[userInput].Selected = false;
    }
    else
    {
        if (!OnlyOneCardSelected())
        {
            _hand.CardsAvailable[userInput].Selected = true;
        }
        else
        {
            throw new NoCardsWereSelectedException("Any other card already selected!");
        }
    }
}
```

Figure 28 Select no more than 1 card code fix

```
if (humanCard.Value < PCCard.Value)
{
    winnerFound = true;
    _PC.Score++;
}
else if (humanCard.Value > PCCard.Value)
{
    winnerFound = true;
    _human.Score++;
}
else
{
    Console.WriteLine("Random picked cards were equal! Picking another random cards...");
    Logger.WriteData("Random picked cards were equal! Picking another random cards...");
}
```

Figure 29 Picked random cards were equal message added

White box

White box testing is based on developer and testers feedback who knows internal code structure. Main objective of white box testing is to improve code and remove unused code.

White box testing was performed by the author and for more accuracy another developer: Finlay Robb (25195541@students.lincoln.ac.uk).

As the result of authors white box application testing author removed 1 method called “DrawLastHand” in Play mode because of code repetition and editing logic so to replace it with CheckWinner method. Moreover, author have replaced all variable names according to C# naming convention (see Figure 30).

Figure 30 Variable renaming

As the result of another developer white box testing - some code issues were found (see Figure 31), as “in State class Validator have protected access modifier, but it is not being used in child classes.”, “PrintDeck method in Deck class is not being used at all” and “Shuffle method in Deck class should have private access modifier.”.

Improvements- In State class Validator have protected access modifier, but it is not being used in child classes. PrintDeck method in Deck class is not being used at all. Shuffle method in Deck class should have private access modifier. Mostly the code is very well formatted and handles all possible

Figure 31 Whitebox testing improvements suggestions

After getting such feedback author have fixed almost all mentioned errors (see Figure 32, Figure 33 and Figure 34) except PrintDeck method. Instead of removing it, author have added comment explaining this method is used for testing purposes.

```
// Regex for validation of user input.
private Regex _validator = new Regex(@"^[1-9]\d*$");
```

Figure 32 Edited access modifier

```
// Used only for testing purposes.
```

Figure 33 Added comment

```
private void Shuffle()
```

Figure 34 Edited access modifier

Checklist

Pass standard might be observer in Table 6:

Table 6 Pass standard

Requirement	Implemented (Y - yes/N - no)
The code compiles and runs	Y
Cards are shuffled, 10 cards each are dealt to players, players can play 2 cards per round	Y
Some errors are captured, such as (but not limited to), cards are not shuffled, players are dealt more or less than 10 cards	Y
Class definitions and object instantiation evident	Y
Method calls to methods in the same class as 'Main'	Y

2:2 standards might be observed in Table 7:

Table 7 2:2 standard

Requirement	Implemented (Y - yes/N - no)
The rules of the card game as specified in the brief are implemented	Y
Application repeats or quits the game gracefully according to player choice	Y
Method calls from 'Main' to methods in other classes	Y
Exception handling is evident	Y
Class definitions show encapsulation	Y

Requirements for 2:1 standard might be observed in Table 8:

Table 8 2:1 standard

Requirement	Implemented (Y - yes/N - no)
Interfaces are used	Y
Static polymorphism (eg. method/operator overloading)	Y
Inheritance showing a class hierarchy (the one shown in the brief, or your own design)	Y
public/private access control in classes, abstraction evident.	Y

Last, first standard requirements are structured in Table 9:

Table 9 1st standard

Requirement	Implemented (Y - yes/N - no)
Custom exceptions are defined and used	Y

Dynamic polymorphism (eg. method overriding)	Y
Use of virtual/abstract methods	Y
protected access control is used in classes	Y

Result

In total, application architecture was described using UML-diagram and text description of each component was added. Moreover, patterns and application realization were described and screenshots of it attached. Furthermore, paper describes 4 different types of testing done (white box testing included) by the author as well as OOP features used in code and their location in files. Last, checklist of requirements was added and filled accordingly.

References

Moore, K., Gupta D. (2021) *Finite State Machines.* Available at: https://brilliant.org/wiki/finite-state-machines/ [Accessed 17 05 2021].

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software.* Massachusetts: Addison-Wesley.

Guru99 (2021) *Boundary Value Analysis and Equivalence Partitioning Testing.* Available at: https://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html [Accessed 17 05 2021].

Madhurihammad (2021) *Equivalence Partitioning Method.* Available at: https://www.geeksforgeeks.org/equivalence-partitioning-method/ [Accessed 17 05 2021].

Rajkumar (2018) *Equivalence Partitioning Test Case Design Technique.* Available at: https://www.softwaretestingmaterial.com/equivalence-partitioning-testing-technique/ [Accessed 17 05 2021].

Usability.gov (2013) *Usability Testing.* Available at: https://www.usability.gov/how-to-and-tools/methods/usability-testing.html [Accessed 17 05 2021].