CMP5344 Coursework - Farkle Application

David Cottrell (18152465)

March 2021

Contents

1 Project Summary and Initial Plans

This logbook will document the stages of development I took while developing my project of choice - the Farkle dice game. It will also include reflective comments for each section, using the Gibbs Reflective Cycle (Gibbs, 1988). This application should have the ability to mimic the action of rolling a set of dice and produce a score based upon Farkle's standard scoring convention. Players within the application should be presented with options based upon the dice they rolled. Once a player has reached a score of at least 10,000, they have won and the game should end.

1.1 Environment Setup

This project will be developed on a machine running the Ubuntu distro of Linux, using Visual Studio Code (VS Code) with the Ionide extension installed. Visual Studio Code is available from: https://code.visualstudio.com/ and the Ionide extension can be installed within VS Code via the extensions menu.

Once the dotnet SDK and runtime was installed, I setup the project directory by following the steps found within the Microsoft documentation found here: https://docs.microsoft.com/en-us/dotnet/fsharp/get-started/get-started-command-line.

After creating the project directory, the below command was executed which creates a library that will be used later in development.

```
dotnet new classlib -lang "F#" -o src/Library
```

A reference to the library was added to project solution:

```
dotnet sln add src/Library/Library.fsproj
```

The main application template was then created:

```
dotnet new console -lang "F#" -o src/FarkleApp
```

A reference to the previously created library was added to the main application:

```
dotnet add src/FarkleApp/FarkleApp.fsproj reference src/Library/
Library.fsproj
```

Finally, a reference to the main application was added to the project solution:

```
dotnet sln add src/FarkleApp/FarkleApp.fsproj
```

1.2 Task Choice Comparison

This project was chosen out of a list of 3 project options, being an implementation of either **Poker**, **Connect Four** or **Farkle**.

1.2.1 Poker

At a minimum, the implementation of **Poker** would require the ability to play a game with the Five-card Draw rules. The rules for a game of Five-card Poker are:

- The game should be played with 2 to 6 players and a standard 52 card deck.
- “Chips” should be used to make bets with.
- An “Ante” should be added to the “Pot” at the start of the round. The “Pot” is the collection of bets made by each player and the “Ante” is an amount determined at the start of the round each player must add to the Pot.
- Each player is then dealt 5 cards at the start of the round.
- A subsequent round of betting is then played where each player can choose to fold, check, bet, call or raise.
- If more than one player remains after the round of betting, a round of drawing will take place.
- Each player in the drawing round can choose choose to replace however many cards in there hand with cards drawn from the deck.
- Another round of betting then takes place.
- If more than one player remains after the final round of betting, the player with the best hand wins.
- The score of each hand follows the same pattern as Texas hold’em, ranging from a “High card” as the lowest scoring hand to the “Royal Flush” as the highest.

1.2.2 Connect Four

The minimum implementation of **Connect Four** would require the ability to play the game with its standard rules. These rules would be:

- Played with 2 people.
- A player is chosen at random to play first.
- Each player takes it in turn dropping a counter of their colour into a grid with 7 columns and 6 rows.

- If a player is able to drop 4 of their counters in a pattern that forms a vertical, horizontal or diagonal line without containing any counters of the opposing player, that player wins.

1.2.3 Farkle

At a minimum, the implementation of **Farkle** would require the ability to play Farkle using the standard scoring convention and rules. The rules for a standard game of Farkle are:

- A game is played with six six-sided dice.
- A player is chosen at random to play first, from which point play continues clockwise around the group of players.
- Each player will take it in turn to role 6 dice.
- The player must remove at least one rolled dice that forms a scoring combination.
- The player can then choose to roll the remaining dice again to attempt to score more points, however if the remaining dice rolled fail to score any points, the player looses all points gained in that round.
- The player can also choose to pass there turn on once they have scored points, allowing the next person's turn to start.

Dice	Score
Each 1	100
Each 5	50
Three 1s	300
Three 2s	200
Three 3s	300
Three 4s	400
Three 5s	500
Three 6s	600

Table 1: Farkle standard scoring convention.

1.3 My Choice

I chose the Farkle project as I believed this would be the most interesting to implement out of the three possible options. Given that I also wanted to avoid building a Graphical User Interface, I felt this project would fit a text-based interface well.

1.4 Initial Plans

I plan to implement the ability to play with the standard scoring convention (as shown in Table 1) from a Command-Line Interface.

An existing application similar to the goal of this project is Farkle (PlayOnlineDiceGames, N.D.), available here: `http://www.playonlinedicegames.com/farkle`. This solution allows a user to either take it in turn with another player using the same physical computer, or one player to play against an AI player. The application developed in this project however will aim to allow 2 or more players play against each other on the same machine via a locally installed program with a CLI.

1.5 Reflections

Initially I had planned to implement a Graphical User Interface for the game using Avalonia FuncUI and spent a lot of time researching the best way to implement one using F#. However I was becoming overwhelmed with work from other modules and a programming internship, so I decided to commit to producing work I knew was more achievable. Although that time could have been used more productively, I feel that this was a valuable experience with time management that I will likely be able to apply to future projects.

2 Planning and Design

2.1 Stakeholders

As users will interact with this product via a Command-Line Interface, the potential audience may be restricted to more technically confident users, likely with a a preexisting knowledge of how to play Farkle. Because of this, players will likely be in a more mature age range.

The system should still remain easy and intuitive to use however, as user experience is vital to keeping the game enjoyable to play.

2.2 Interface

The user will interact with the system by entering a number as input. This will represent a choice as described by a menu displayed by the system. For example, a given menu might be:

```
Enter:
1) to roll again with selected dice
2) to bank points
>_
```

2.3 Requirements

The requirements as identified from the task sheets are that the game should:

- Allow 2 or more players to be added into the game and given turns in order
- Allow these players to roll dice in an attempt to score points
- Calculate points for a given roll based on the standard scoring convention
- Give the player the option to bank their points, roll some non-scoring dice again or roll all their dice again, depending on the result of their roll
- Cause the player to loose their points and move to the next player's turn if the player fails to score any points for a given roll
- Allow a player to win if they achieve a score of at least 10,000

I also plan to ensure the game is intuitive by:

- Displaying the scores of all players after each turn
- Providing a break-down of the scores and dice that did and did not score after each roll

2.4 Gherkin User Story Specifications

To help document the above requirements in more detail we were instructed to use User Story Specifications using the Gherkin framework. Gherkin user story specifications aims to help document all of the planned user's needs and interactions with the system. They include a "Feature" which describes the part of the system to be documented, which can be described using the traditional user story format "As a", "I want", "So that". Within each feature are multiple "Scenarios" which describe the events that could occur for that "Feature". This is done with the Gherkin format "Given", "When", "Then". Orienting planning around the actions the user could take is called Behavior-driven Development. "Gherkin is a structured approach to writing behavioral tests, also called Behavior Driven Development (BDD)" ... and "seek to follow true user workflow" (Werner, 2017).

```
Feature: Add players
As a user
I want to add players to the game
So that I and others can play the game

Scenario: User enters less than two players
Given no players have yet been added to the game
When the user enters either one or no player information
And indicates the system should start the game
Then the game should display an error message
And prompt the user to enter more player's into the system

Scenario: User enters at least two players
Given at least two players have been added
When the user indicates the system should start the game
Then the game should store the player information
And start the game

Feature: Roll dice
As the current player
I want to roll a set of dice
So that I can attempt to score points

Scenario: Score Points
Given it is the start of the current player's turn
When the player rolls some specific individual dice or a set of
three dice of the same number
Then the player should score the appropriate amount of points as
described by the scoring convention
And be presented with the option to bank the points or roll again
```

```
Scenario: Score No Points (Farkled)
Given it is the start of the current player’s turn
When the player fails to roll any dice that match the scoring convention
Then the player should loose all scored points for that round
And the next player's turn should start

Scenario: Bank Points
Given it is the current player’s turn
When they score points within the current roll
Then the player should have the option to add the points earned
from that round to their total
And the next player's turn should start

Scenario: Roll Again
Given it is the current player’s turn
When they score points within the current roll
Then the player should have to choose one or more scoring dice to keep
And roll the remaining dice

Scenario: Scoring With All Dice (Hot Dice)
Given it is the current player’s turn
And it is the start if their turn
When all six dice rolled score points
Then the score from all six dice are added to the current turn total
And the player is able to roll all six dice again

Feature: Win the current game
As a player
I want my total score to be stored and checked against the
winning amount 10,000 after each roll
So that I am able to win the current game

Scenario: Obtain a Total >= 10,000
Given it is the current player’s turn
When they score points within the current roll that cause
their total to be equal to or greater than 10,000
Then they should win the game
And the game should end
```

2.5 Data Model Specifications

2.5.1 Dice and Scoring

Once some number of dice are rolled, the user can either score points from specific individual dice or sets of three dice of the same number. To initialize a player's first turn, a list of 6 random numbers should be generated. This represents the physical act of rolling the dice.

An example of a given roll may be rolling a 5, 5, 1, 5, 5 and a 2. From this list the 1, a single 5 and the three remaining 5s should be identified by the system as scoring dice.

To model this, types should be declared that represent:

- A six-sided dice (***Dice***)
- A list of six-sided dice to represent a roll or dice combination (***DiceList***)
- The score achieved from a given set of 3 within the roll (***SetTotal***)
- The dice that scored a given set of 3 within the roll along with their score (***SetCombination***)
- The score achieved from a given individual scoring dice within the roll (***RemainderTotal***)
- The dice that scored a given number of individual scores within the roll along with their score (***RemainderCombination***)
- A group of the score for either a set of 3 or individual scoring dice along with the list of the dice that scored that set or individual score (***ScoreResult***)
- A list of these groups to represent all the scores and their respective dice for a given roll (***ScoreResults***)

An example of how the final type in the above list (***ScoreResults***) would be represented using the previous example of rolling 5, 5, 1, 5, 5 and a 2, would be:

[([1], 100), ([5], 50), ([5,5,5], 500)]

Where each list element is a tuple of either the list of dice for the scoring set or some scoring individual dice along with their respective score.

I plan on creating a function that takes a given list of dice as input, applies Farkle's standard scoring convention to it and produces this list of tuples as an output. This would then allow the system to easily determine which dice should be kept behind when the user chooses to roll again, along with keeping data neatly grouped and organised.

Below are the previously described types represented in Mathematical notation:

$$\boldsymbol{Dice} == \{x : \mathbb{N} \mid x \geq 1 \wedge x \leq 6\}$$

$$\boldsymbol{DiceList} == \mathit{seq\ Dice}$$

$$\boldsymbol{SetTotal} \in \{200, 300, 400, 500, 600\}$$

$$\boldsymbol{RemainderTotal} \in \{50, 100, 200\}$$

$$\boldsymbol{SetCombination} == \mathit{DiceList}\ \times\ \mathit{SetTotal}$$

$$\boldsymbol{RemainderCombination} == \mathit{DiceList}\ \times\ \mathit{RemainderTotal}$$

$$\boldsymbol{ScoreResult} == \{\mathit{SetCombination}\} \cup \{\mathit{RemainderCombination}\}$$

$$\boldsymbol{ScoreResults} == \mathit{ScoreResult\ list}$$

$\boldsymbol{SetTotal}$ is limited to the set of $\{200, 300, 400, 500, 600\}$ as those are the only unique set scores defined in the scoring convention (Table 1). The following are the only dice combinations that could result in a value for $\boldsymbol{RemainderTotal}$:

$$\{5\} == 50$$

$$\{5, 5\} == 100$$

$$\{1\} == 100$$

$$\{1, 1\} == 200$$

And so $\boldsymbol{RemainderTotal}$ is limited to the set of $\{50, 100, 200\}$.

2.5.2 Players and Choices

I will also need to store some information on the players within a given game. This information will consist of their name along with their current total score. Given that there could be many players within a single game, each piece of individual information for a given player should be stored within a list.

To model all of the players for the given game of Farkle, I plan on defining the following data type:

$$\boldsymbol{Players} = \mathit{Player\ list}$$

Where:

$$\boldsymbol{Player} = \mathit{Name}\ \times\ \mathit{Score}$$

$$\boldsymbol{Name} = \mathit{String}$$

$$\boldsymbol{Score} \in \mathbb{N}$$

Each list element within Players is an individual player, represented by a tuple of their name and their current score.

The system should also allow the user to make decisions based on the result of their roll. To model this, the following data type should be defined:

$$\boldsymbol{Choice} == \{RollAgain\} \cup \{RollAllAgain\} \cup \{BankPoints\}$$

Where ***RollAgain*** represents the user rolling some selected dice again after scoring, ***RollAllAgain*** represents the user rolling all their dice again after getting "Hot Dice", ***BankPoints*** represents the user choosing to finish their turn, adding the points scored in that round to their total.

2.6 Program Task Specifications

As shown in figure 1 below, the game will take the following steps per player's turn:

1. Roll some number of dice
2. Calculate the scoring sets and/or remainders their totals
3. If the current player's score + round total + roll total >= 10,000, that player wins and execution stops.
4. If the player failed to score any points, move play to the next player.
5. Otherwise display the player's scoring dice.
6. Display their available options and get their choice.
7. Complete the appropriate tasks for that choice
8. Finally return to step 1 either as the current player - allowing them to roll again, or move to the next players turn - adding the previous players points to their total score if necessary.

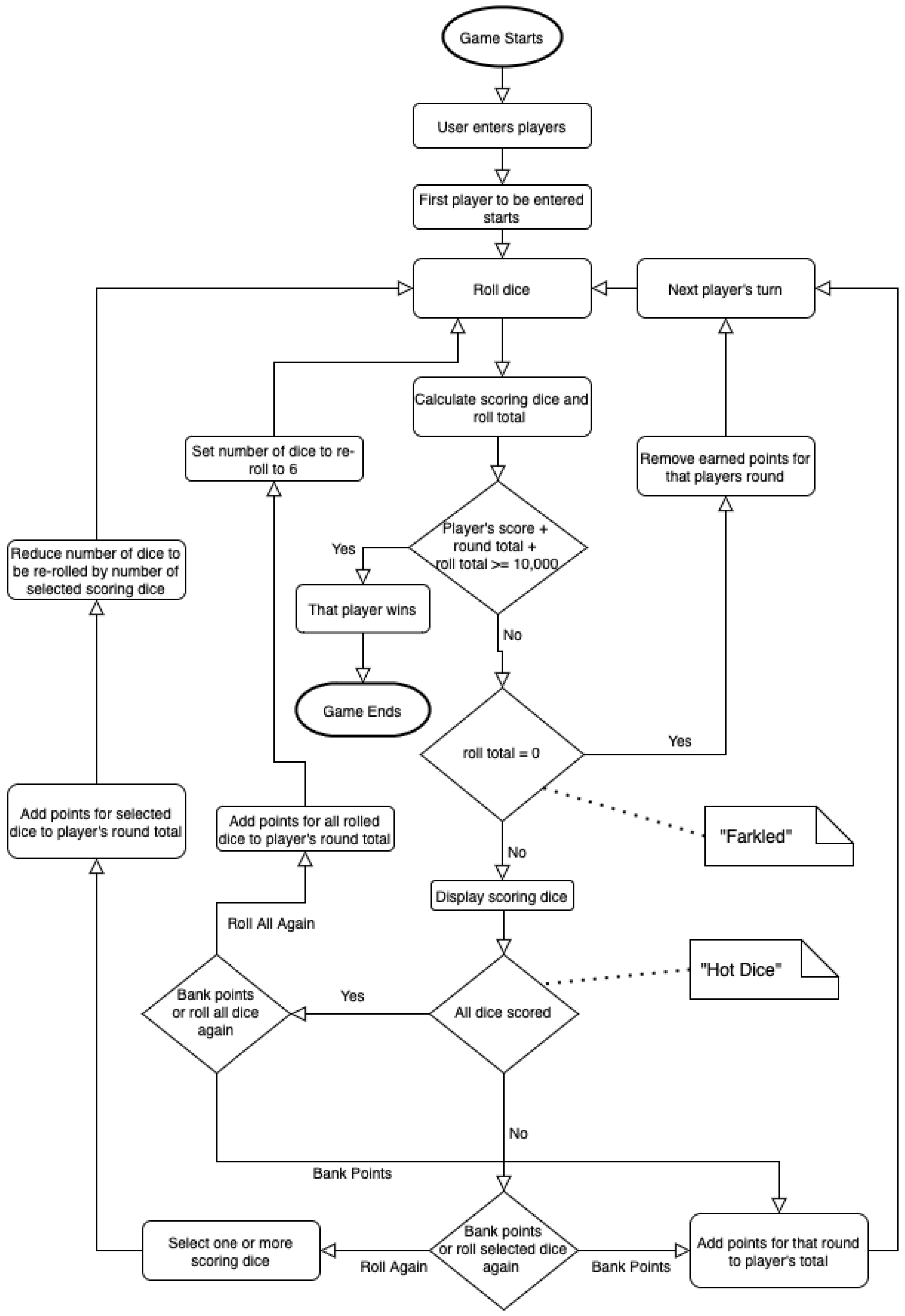

Game Starts
User enters players
First player to be entered starts
Roll dice
Next player's turn
Calculate scoring dice and roll total
Set number of dice to re-roll to 6
Remove earned points for that players round
Reduce number of dice to be re-rolled by number of selected scoring dice
Player's score + round total + roll total >= 10,000
Yes
That player wins
No
Game Ends
roll total = 0
Yes
"Farkled"
Add points for selected dice to player's round total
Add points for all rolled dice to player's round total
No
Display scoring dice
Roll All Again
"Hot Dice"
Bank points or roll all dice again
Yes
All dice scored
No
Bank Points
Select one or more scoring dice
Bank points or roll selected dice again
Add points for that round to player's total
Roll Again
Bank Points

Figure 1: System Flowchart

Relating the flow of the system described above to the previously defined data models, the system will need to start with a ***DiceList*** (the rolled dice in step 1) and end with a ***Choice*** (step 6) that determines the last steps of the main game loop.

Therefore the specific tasks that should be completed to evaluate a ***Choice*** from a given ***DiceList*** are shown below.

$$\begin{aligned} &\mathit{roll}\colon (\mathit{numOfDice} : \mathit{int}) \to \mathit{Result}\langle\langle \mathit{rolledDice} : \mathit{DiceList}\rangle\rangle \\ &\downarrow \mathit{rolledDice} \\ &\mathit{scoreRoll}\colon (\mathit{rolledDice} : \mathit{DiceList}) \to \mathit{Result}\langle\langle \mathit{scoringDice} : \mathit{ScoreList}\rangle\rangle \\ &\downarrow \mathit{scoringDice} \\ &\mathit{getChoice}\colon (\mathit{scoringDice} : \mathit{ScoreList}) \to \mathit{Result}\langle\langle \mathit{choice} : \mathit{Choice}\rangle\rangle \end{aligned}$$

"scoreRoll" will need to take a given ***DiceList*** and produce a ***ScoreResults***, as with the example below:

$$[6,\ 5,\ 6,\ 1,\ 6,\ 6]$$

$$\downarrow$$

$$[\ \ ([6,\ 6,\ 6],\ 600),\ \ ([5],\ \ 50),\ \ ([1],\ \ 100)\ ]$$

To achieve this, "scoreRoll" should have a sub-function "countOccurrences":

$$\mathit{countOccurrences}\colon (\mathit{diceToFind} : \mathit{Dice}, \mathit{roll} : \mathit{DiceList}) \to \mathit{Result}\langle\langle \mathit{numOfOccurrences} : \mathit{int}\rangle\rangle$$

This function should take a dice along with the list of dice the player rolled and return the number of times that dice is present within the roll.

The number of sets of 3 dice can be represented by the integer value of the number of multiples of 3 within numOfOccurrences, or mathematically:

$$\mathit{numOfSets} = \frac{\mathit{numOfOccurrences} - (\mathit{numOfOccurrences} \bmod 3)}{3}$$

And the number of remainders can be represented as the remainder after the division of the numOfOccurrences by 3, or mathematically:

$$\mathit{numOfRemainders} = \mathit{numOfOccurrences} \bmod 3$$

If the number of times each number a dice could be (1 - 6) is calculated to appear in the provided ***DiceList***, the number of sets and remainders of that number of dice can be calculated using the above formulae.

For example, given the previous ***DiceList***: [6, 5, 6, 1, 6, 6] and checking for the number of occurrences of 6:

$$numOfOccurrences = 4$$
$$numOfSets = \frac{4 - (4 \bmod 3)}{3} = 1$$
$$numOfRemainders = 4 \bmod 3 = 1$$

Once the number of sets and remainders are calculated, these results should be passed to a function that determines the score, given the dice and its respective number of sets and reminders:

$$\begin{aligned} parseDice &\colon (dice : Dice, numOfSets : int, numOfRemainders : int) \\ &\to Result\langle\langle SetTotal * RemainderTotal\rangle\rangle \end{aligned}$$

These results can then be calculated per dice number (1 - 6), and returned from “scoreRoll” once the scores for each dice number have been calculated as a ***ScoreResults***.

2.7 Reflections

Discussing this section with friends from within the module helped me to come to the decision that the dice scoring part of the system could be split into the three functions: “roll”, “scoreRoll” and “getChoice”. I then created the System Flowchart (figure 1) and documented these functions as shown above. Those discussions were very valuable as this idea seemed to have suited the problem well.

3 Implementation

3.1 Roll Scoring

I started my development of the application by adding the roll-scoring data models previously using mathematical notation to the application in F#:

```
type SetTotal = SetTotal of int
type RemainderTotal = RemainderTotal of int

type Dice = Dice of int
type DiceList = Dice list

type ScoreResult =
    | SetCombination of (DiceList * SetTotal)
    | RemainderCombination of (DiceList * RemainderTotal)

type ScoreResults = ScoreResult list
```

I then created a function that would generate and return a list of dice with random values and of length defined by the parameter passed to the function, as in the future the number of dice thrown will need to be reduced if the user chooses to roll again.

```
// Returns a given amount of random numbers (Dice)
let rollDice (numOfDice:int) : DiceList =
    let rand = Random()
    List.init numOfDice (fun _ -> Dice (rand.Next (1, 7)))
```

From there I implemented the “scoreRoll” function:

```
// Returns a list of the scoring combinations along with their score
let scoreRoll (currentRoll:DiceList) =
    let mutable scoreList:ScoreResults = []
    for currentDice in 1..6 do
        // Get the number of times the current dice appears within the current roll
        let count = countOccurrences (Dice currentDice) currentRoll
        if count > 0 then
            // Get number of sets for the current dice within the roll
            let numOfSets = (count - (count % 3)) / 3
            // Get number of times the dice occurs excluding the sets of 3
            let numOfRemainders = count % 3
            // Get the score for the current dice number in the roll
            let mutable score = parseDice currentDice numOfSets numOfRemainders
            // Add each scoring sets of three for the current dice number
            //to the score list (if present)
            if fst score > SetTotal 0 then
                scoreList <- scoreList
                    @ (getSetDice numOfSets (Dice currentDice) (fst score))
            // Add each individually scoring dice for the current dice number to
            //the score list (if present)
            if snd score > RemainderTotal 0 then
                scoreList <- scoreList
                    @ (getRemainderDice numOfRemainders (Dice currentDice) (snd score))

    scoreList
```

As discussed within the task specification, I needed to implement a way of determining the dice that make up a score, along with the score that they equate to. This is done by first calculating the amount of times each number a dice could be (given they range from 1 to 6) occurs in the given ***DiceList***:

```
for currentDice in 1..6 do
    // Get the number of times the current dice appears within the current roll
    let count = countOccurrences (Dice currentDice) currentRoll
```

Where “countOccurences” is:

```
// Returns the number of times a given dice appears within a DiceList
let countOccurrences (diceToFind:Dice) (roll:DiceList) =
    let mutable count = 0
    for dice in roll do
        if diceToFind = dice then count <- count + 1
    count
```

If that dice exists in the ***DiceList***, the scoreRoll calculates how many sets and remainders of that dice there are:

```
// Get number of sets for the current dice within the roll
let numOfSets = (count - (count % 3)) / 3

// Get number of times the dice occurs excluding the sets of 3
let numOfRemainders = count % 3
```

The score for the given dice is then calculated with “parseDice”:

```
// Get the score for the current dice number in the roll
let mutable score = parseDice currentDice numOfSets numOfRemainders
```

This function takes the current dice and the previously calculated number of sets and remainders there are of it within the current roll as parameters and returns the score for its sets and/or remainders:

```
// --- Standard Scoring Convention Logic ---
let parseDice dice numOfSets numOfRemainders =
    match dice, numOfSets with
    | 1, _ ->
        SetTotal (300), RemainderTotal (100 * numOfRemainders)
    | 5, _ ->
        SetTotal (500), RemainderTotal (50 * numOfRemainders)
    | _, _ when numOfSets > 0 ->
        // Any dice other than 1 or 5 with at least one set
        SetTotal (dice * 100), RemainderTotal 0
    | _, _ ->
        SetTotal 0, RemainderTotal 0
```

This function implements the standard scoring convention, which:

- Checks if the passed dice is 1
  - Returns 300 and the (number of remainders of 1) $\times$ 100
- Checks if the passed dice is 5
  - Returns the 500 and the (number of remainders of 5) $\times$ 50
- Checks if the passed dice (that isn’t 1 or 5) has more than 1 set
  - Returns the number of the dice $\times$ 100 and None for the remainder total
- Returns None for both the set and remainder scores if none of the previous checks are true

Once the score(s) for the current dice are calculated, the dice that form that score can then determined. This is done by using the score(s) returned by "scoreRoll" with:

```
// Add each scoring sets of three
//for the current dice number to the score list (if present)
if fst score > SetTotal 0 then
    scoreList <- scoreList
        @ (getSetDice numOfSets (Dice currentDice) (fst score))

// Add each individually scoring dice
//for the current dice number to the score list (if present)
if snd score > RemainderTotal 0 then
    scoreList <- scoreList
        @ (getRemainderDice numOfRemainders (Dice currentDice) (snd score))
```

These two if statements check that the score returned from "parseDice" included a score for a set and/or remainder(s). The number of dice for each type of score is then added with "getSetDice" and "getRemainderDice".

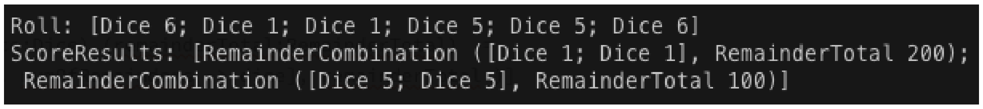
As shown below, "getSetDice" returns a list consisting of either one or two of the ***SetCombination*** type - depending on how many sets were scored for the given dice. This list can then be appended to the full list of scores within the "scoreRoll" once returned as shown above.

```
let getSetDice numOfSets (dice:Dice) (setTotal:SetTotal)  =
    let mutable scoreList:ScoreResults = []
    for _ in 1..numOfSets do
        scoreList <- scoreList @ [SetCombination ([dice; dice; dice], setTotal )]
    scoreList
```

The "getRemainderDice" is similar to "getSetDice", although only one list of remainders will need to be returned and the ***RemainderCombination*** type is used instead.

```
let getRemainderDice numOfRemainders (dice:Dice) (remainderTotal:RemainderTotal)  =
    [RemainderCombination ([for _ in 1 .. numOfRemainders -> dice], remainderTotal )]
```

Once this processes has been completed for each dice within the given roll, a complete list of scoring dice and their respective scores (***ScoreResults***) will have been calculated and returned by the function. Figure 2 below shows the printed output of passing a ***DiceList*** with 6 random dice to the function.



```
Roll: [Dice 6; Dice 1; Dice 1; Dice 5; Dice 5; Dice 6]
ScoreResults: [RemainderCombination ([Dice 1; Dice 1], RemainderTotal 200);
 RemainderCombination ([Dice 5; Dice 5], RemainderTotal 100)]
```

Figure 2: scoreRoll example

3.2 Gameplay

Now that I have developed a system that successfully calculates the score for a given list of dice, I could work on building the game logic and the user interactions. To begin, I implemented the data types defined previously for storing player data and user choices in F#:

```
type Name = String
type Score = int
type Player = (Name * Score)
type Players = Player list

type Choice =
    | BankPoints
    | RollAgain
    | RollAllAgain
```

Once the program starts, the system asks the user to enter the names of each player within the game:

```
let mutable players:Players = []
players <- getPlayers
```

Where “getPlayers” is:

```
// Asks user to input the names of at least two players
let getPlayers : Players =
    let mutable keepAdding = true
    let mutable players = []
    printfn "- Add players -"
    printfn "Enter player's name or type 'stop' to stop."
    while keepAdding do
        printf "> "
        let name = System.Console.ReadLine()
        if name = "stop" then
            if List.length players >= 2 then
                keepAdding <- false
            else
                printfn "Please add at least two players."
        else
```

```
            if not (name = "") then players <- players @ [(name, 0)]
    players
```

This allows the user to add at least two names of players until they enter "stop" to indicate that the last user has been added.

Once the players have been added, the program clears the player entry dialog and declares two constants. These are "gameOver" - which controls whether the main loop should continue, and "winningScore" - which defines the score the user has to achieve to win the game.

```
System.Console.Clear()
let mutable gameOver = false
let winningScore = 10000
```

The main loop of the program is shown below. This allows the program to loop while no players have achieved the winning score of 10,000. The for loop within the main loop will iterate over each player, giving each player a turn.

```
// Loop while game isn't over
while not gameOver do
    // Give each each player their turn
    for i in 0 .. (List.length players) - 1 do
```

For each iteration of the player loop, a score board will be displayed along with declaring 3 variables. These variables store the state of the players current turn.

```
if not gameOver then displayScoreBoard players
let mutable diceCount = 6
let mutable roundTotal = 0
let mutable turnOver = false
```

As shown below, "displayScoreBoard" displays the list of players and their current scores. This function is used after each turn.

```
let displayScoreBoard (players:Players) =
    printfn "\nScore board"
    printfn "----------------------"
    for player in players do
        printfn "- %s has %d points" (fst player) (snd player)
    printfn "----------------------\n"
```

Below is the beginning of the loop that iterates while it is the current player's turn and the game isn't over. This rolls the set amount of dice they currently have, determines the score for those dice and determines if they have achieved the winning score.

```
// While it's the current players turn and the game isn't over
while not turnOver && not gameOver do
    printfn $"- Current player: %s{fst players.[i]} -"
    // Roll some number of random dice
    let roll = rollDice diceCount
    printfn $"\n%s{fst players.[i]} rolled %A{diceCount} dice: %A{roll}"
    // Get set and remainder score combinations from the roll
    let diceScores:ScoreResults = scoreRoll roll
    let rollTotal = calcTotalScore diceScores
    let hasWon = (snd players.[i] + roundTotal + rollTotal) >= winningScore
```

Once that information has been calculated, the path that the system should go can then be determined.

```
// If the current player has achieved the winning amount (won the game)
if hasWon then
    displayGameOver players.[i]
    turnOver <- true
    gameOver <- true
// If they failed to score any points in the current roll
else if rollTotal = 0 then
    System.Console.Clear()
    printfn $"-%s{fst players.[i]} was farkled!-"
    roundTotal <- 0
    turnOver <- true
else
    printfn $"- Current round total: %d{roundTotal}"
    // Display the scores for the dice they roll
    displayScores diceScores diceCount rollTotal
    // Show the available choices based on the score from the roll
    let choice = getChoice diceScores diceCount
    // ---- Act on chosen choice ----
    match choice with
    | RollAgain ->
        let (newDiceCount, newRoundTotal) = rollAgain diceScores diceCount roundTotal
        diceCount <- newDiceCount
        roundTotal <- newRoundTotal
    | BankPoints ->
        roundTotal <- roundTotal + rollTotal
        players <- bankPoints players roundTotal i
        turnOver <- true
    | RollAllAgain ->
```

```
        System.Console.Clear()
        roundTotal <- roundTotal + rollTotal
        diceCount <- 6 // All dice can be re-rolled on Hot Dice
```

If the player won, their name will be displayed and the game will end. Otherwise it will check if they scored any points, if they didn't their turn will end and their points will be lost, if they did the scores for the dice they rolled and their available choices are displayed with "displayScores" and "getChoice" functions respectively.

"displayScores" shows the scoring dice along with their scores.

```
let displayScores scoringDice diceCount rollTotal =
    let numOfScoringDice = getNumOfScoringDice scoringDice
    // Display scoring dice and the roll total with option numbers
    printfn $"\n%d{numOfScoringDice}/%d{diceCount} dice scored. Scoring dice:"
    let mutable count = 1
    for score in scoringDice do
        match score with
        | RemainderCombination (x, y) ->
            printfn $"%d{count}) %A{x}, with a total of %d{remainderTotalToInt y}"
        | SetCombination (x, y) ->
            printfn $"%d{count}) %A{x}, with a total of %d{setTotalToInt y}"
        count <- count + 1
    printfn $"\n- Current roll total: %d{rollTotal}"
```

"getChoice" prompts the user to either bank their points, roll some selected dice again or roll all 6 dice again if they get "hot dice" (all 6 dice scored).

```
let getChoice (scoringDice:ScoreResults) (diceCount:int) : Choice=
    let numOfScoringDice = getNumOfScoringDice scoringDice
    let rollChoice =
        if numOfScoringDice = diceCount then RollAgain
        else RollAgain
    if numOfScoringDice = diceCount then
        printfn "Would you like to bank your points (1) or roll all 6 dice again? (2)"
        printf "> "
    else
        printfn "Would you like to bank your points (1) or roll again? (2)"
        printf "> "
    let mutable validChoice = false
    let mutable choice = Unchecked.defaultof<Choice>
    while not validChoice do
        let success, choiceInt = System.Int32.TryParse(System.Console.ReadLine())
        match success, choiceInt with
        | true, 1 ->
            choice <- BankPoints
            validChoice <- true
        | true, 2 ->
            choice <- rollChoice
            validChoice <- true
        | _, _ -> printfn "Invalid input, please try again."
    choice
```

“chooseRollAgainScores” prompts the user to choose which of the scoring dice they want to keep (at least one).

```
// Allows the user to choose which dice they want to keep if they choose to re-roll
let chooseRollAgainScores (scoreList:ScoreResult list) =
    let mutable parsedChoices = []
    printfn "Enter the dice you would like to keep from the above
        list (separate choices with spaces)"
    let mutable valid = false
    while not valid do
        printf "> "
        let choices = System.Console.ReadLine().Split [|' '|] |> Array.toList
        // Choice validator function
        let validateScoreChoice (value:String) =
            let (success, num) = System.Int32.TryParse(value)
            if success && num > 0 && num <= (List.length scoreList) &&
               List.length choices <= List.length scoreList then num
            else -1
        // Validate each individual choice
        parsedChoices <- List.map (fun choice -> validateScoreChoice choice) choices
        if List.exists ((=) -1)
            parsedChoices then printfn "Invalid input, please try again."
        else valid <- true
    let mutable chosenScores = []
    for choice in parsedChoices do
        // Get the chosen scores from the list of choices
        chosenScores <- chosenScores @ [scoreList.[choice - 1]]
    System.Console.Clear()
    chosenScores
```

Finally, the choice returned from “getChoice” is used within the loop for the current player’s turn.

```
match choice with
| RollAgain ->
    let (newDiceCount, newRoundTotal) = rollAgain diceScores diceCount roundTotal
    diceCount <- newDiceCount
    roundTotal <- newRoundTotal
| BankPoints ->
    roundTotal <- roundTotal + rollTotal
    players <- bankPoints players roundTotal i
```

```
        turnOver <- true
    | RollAllAgain ->
        System.Console.Clear()
        roundTotal <- roundTotal + rollTotal
        diceCount <- 6 // All dice can be re-rolled on Hot Dice
```

3.3 Reflections

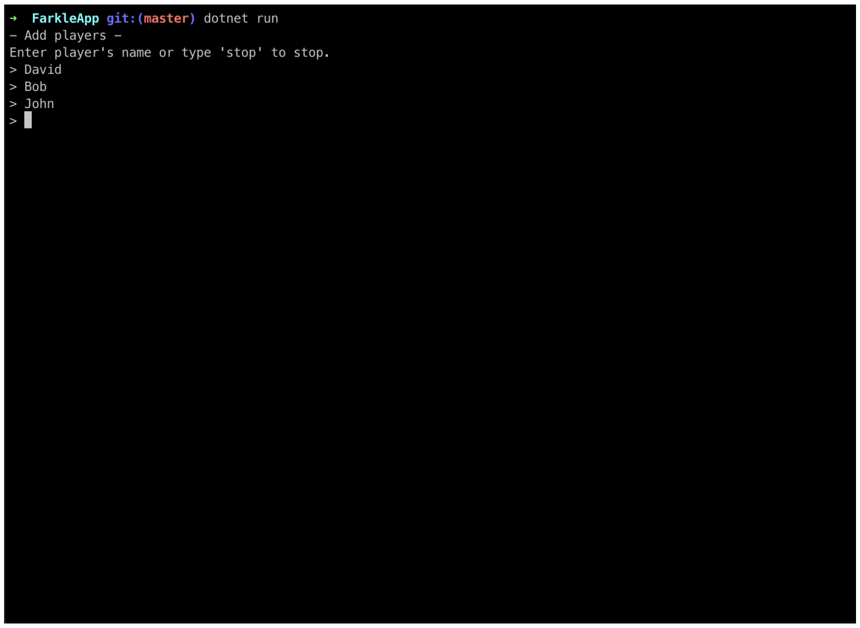
As stated previously, I developed my ability to use F# in large part from the development of this project. This was stressful initially as there also seemed to be a lack of tutorials and answers to problems online, however once I spent more time developing the project I felt comfortable enough with the language for the lack of online resources to be less of an issue. In the future I would likely aim to engage more with support sessions, as these would've likely been helpful with answering some of the questions I had during development.

4 Testing

To test that the system is working now that all of the requirements have been met, I started by completed some demo runs of the game myself which both valid and invalid input.

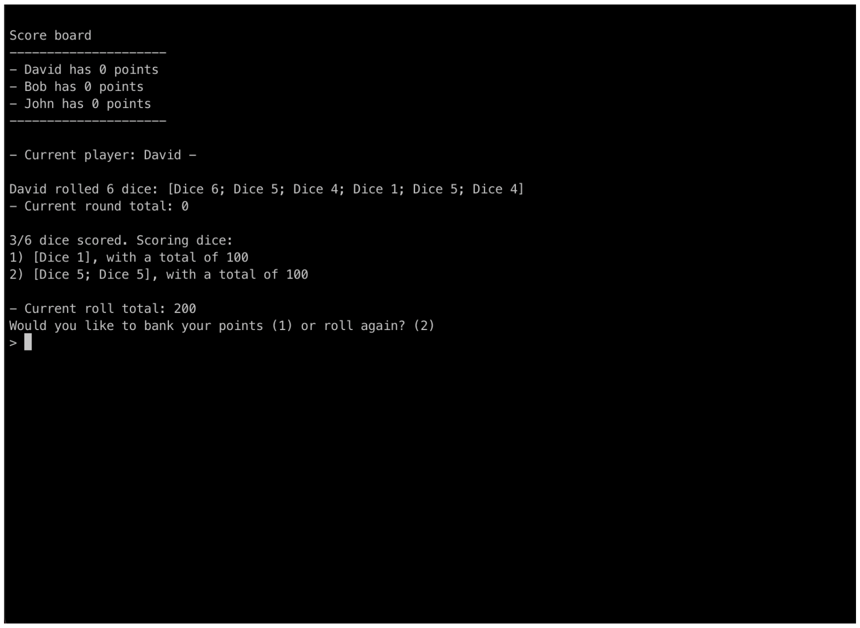
4.1 Manual Testing

4.1.1 Trial run 1 - valid input



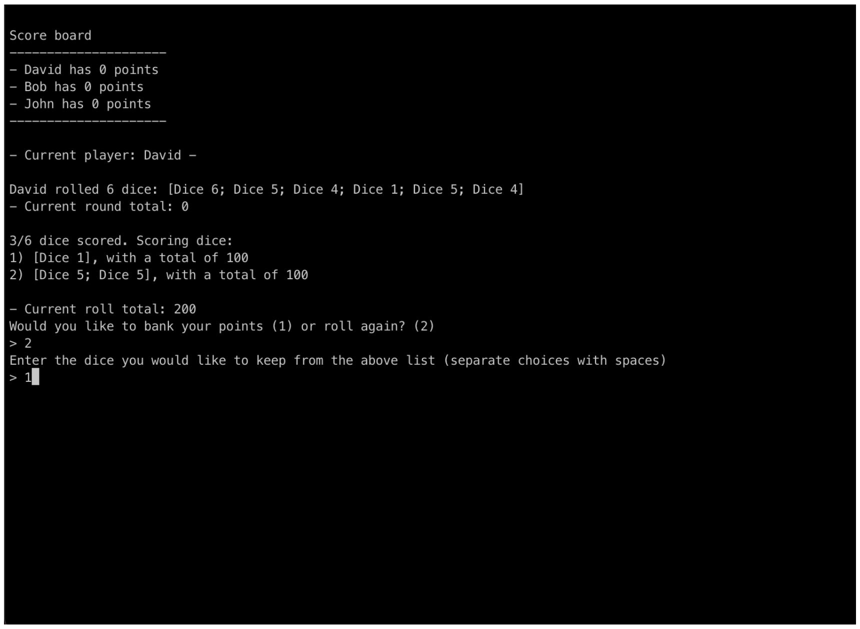
```
➜  FarkleApp git:(master) dotnet run
– Add players –
Enter player's name or type 'stop' to stop.
> David
> Bob
> John
> 
```

Figure 3: Entering players into the system



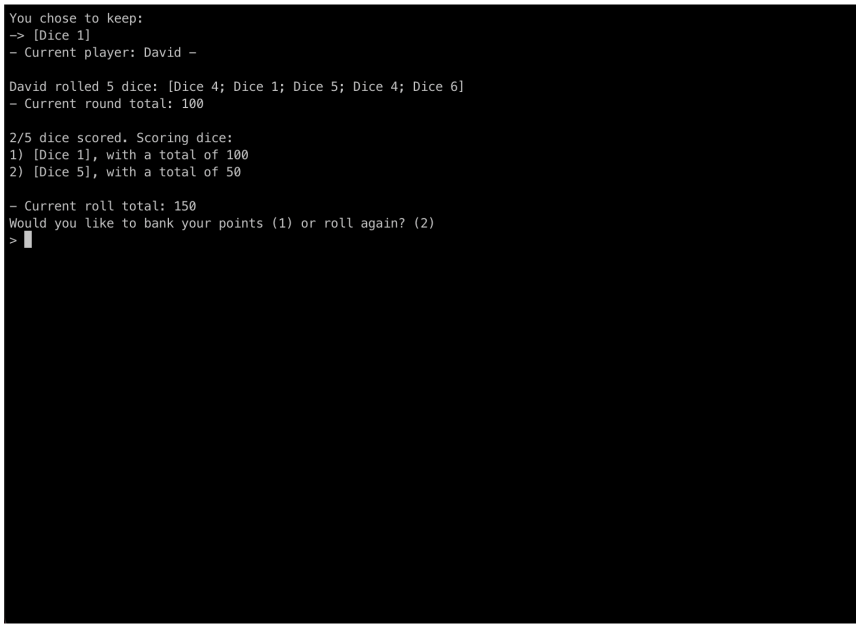
```
Score board
--------------------
- David has 0 points
- Bob has 0 points
- John has 0 points
--------------------

- Current player: David -

David rolled 6 dice: [Dice 6; Dice 5; Dice 4; Dice 1; Dice 5; Dice 4]
- Current round total: 0

3/6 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 5; Dice 5], with a total of 100

- Current roll total: 200
Would you like to bank your points (1) or roll again? (2)
> 
```

Figure 4: after ”stop” entered. First player’s turn



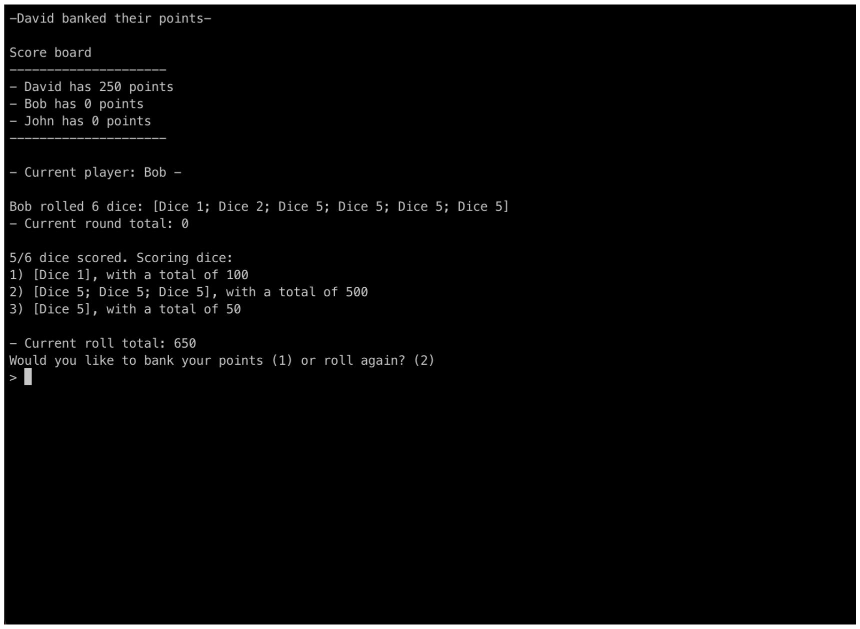
```
Score board
--------------------
- David has 0 points
- Bob has 0 points
- John has 0 points
--------------------

- Current player: David -

David rolled 6 dice: [Dice 6; Dice 5; Dice 4; Dice 1; Dice 5; Dice 4]
- Current round total: 0

3/6 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 5; Dice 5], with a total of 100

- Current roll total: 200
Would you like to bank your points (1) or roll again? (2)
> 2
Enter the dice you would like to keep from the above list (separate choices with spaces)
> 1
```

Figure 5: Choosing to roll again without selected dice



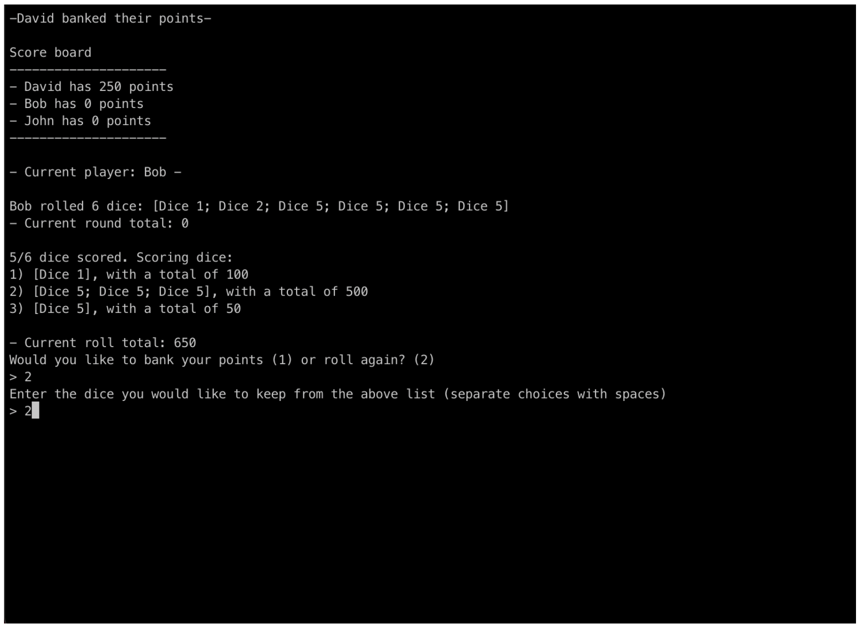
```
You chose to keep:
-> [Dice 1]
- Current player: David -

David rolled 5 dice: [Dice 4; Dice 1; Dice 5; Dice 4; Dice 6]
- Current round total: 100

2/5 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 5], with a total of 50

- Current roll total: 150
Would you like to bank your points (1) or roll again? (2)
> 
```

Figure 6: Output of choosing roll again without selected dice



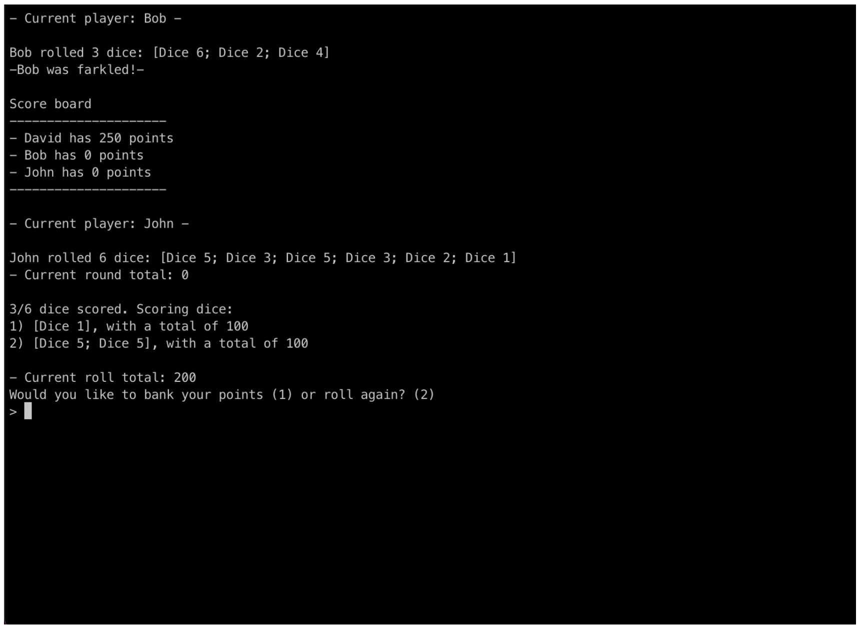
```
-David banked their points-

Score board
---------------------
- David has 250 points
- Bob has 0 points
- John has 0 points
---------------------

- Current player: Bob -

Bob rolled 6 dice: [Dice 1; Dice 2; Dice 5; Dice 5; Dice 5; Dice 5]
- Current round total: 0

5/6 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 5; Dice 5; Dice 5], with a total of 500
3) [Dice 5], with a total of 50

- Current roll total: 650
Would you like to bank your points (1) or roll again? (2)
> 
```

Figure 7: Output of choosing to bank points



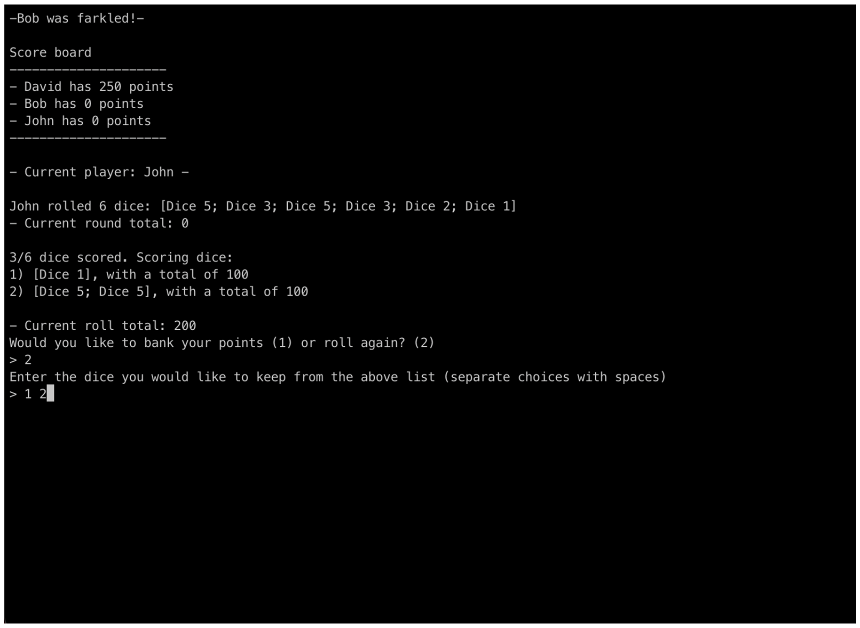
```
-David banked their points-

Score board
---------------------
- David has 250 points
- Bob has 0 points
- John has 0 points
---------------------

- Current player: Bob -

Bob rolled 6 dice: [Dice 1; Dice 2; Dice 5; Dice 5; Dice 5; Dice 5]
- Current round total: 0

5/6 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 5; Dice 5; Dice 5], with a total of 500
3) [Dice 5], with a total of 50

- Current roll total: 650
Would you like to bank your points (1) or roll again? (2)
> 2
Enter the dice you would like to keep from the above list (separate choices with spaces)
> 2
```

Figure 8: Next players turn, choosing to roll again without selected dice



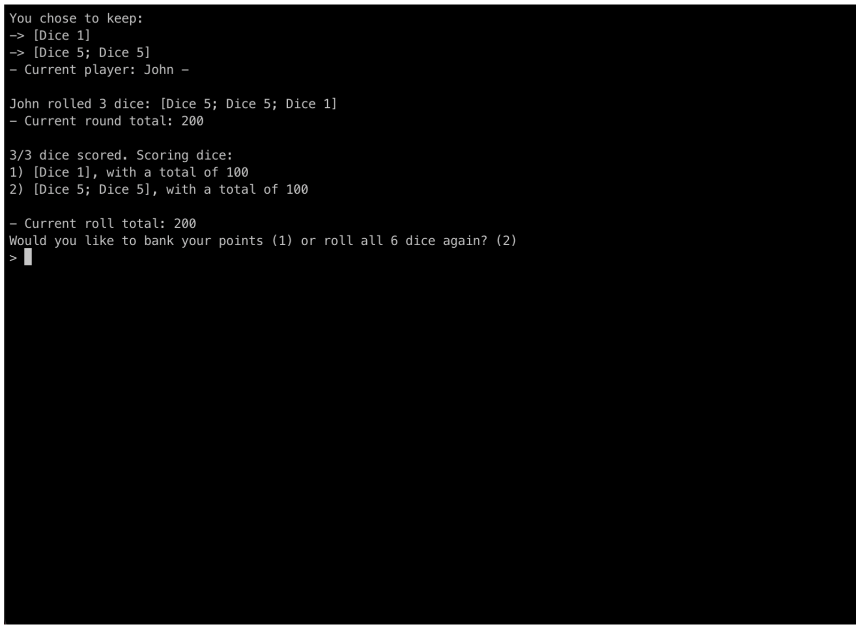
```
- Current player: Bob -

Bob rolled 3 dice: [Dice 6; Dice 2; Dice 4]
-Bob was farkled!-

Score board
---------------------
- David has 250 points
- Bob has 0 points
- John has 0 points
---------------------

- Current player: John -

John rolled 6 dice: [Dice 5; Dice 3; Dice 5; Dice 3; Dice 2; Dice 1]
- Current round total: 0

3/6 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 5; Dice 5], with a total of 100

- Current roll total: 200
Would you like to bank your points (1) or roll again? (2)
> 
```

Figure 9: Player failing to score any points (”farkled”)



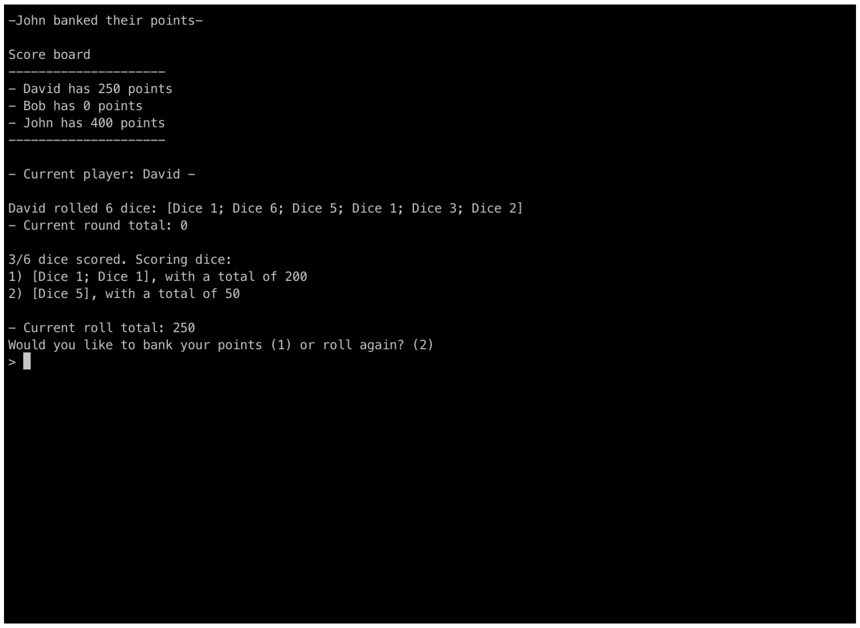
```
-Bob was farkled!-

Score board
---------------------
- David has 250 points
- Bob has 0 points
- John has 0 points
---------------------

- Current player: John -

John rolled 6 dice: [Dice 5; Dice 3; Dice 5; Dice 3; Dice 2; Dice 1]
- Current round total: 0

3/6 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 5; Dice 5], with a total of 100

- Current roll total: 200
Would you like to bank your points (1) or roll again? (2)
> 2
Enter the dice you would like to keep from the above list (separate choices with spaces)
> 1 2
```

Figure 10: Next player's turn, choosing to roll again without selected dice



```
You chose to keep:
-> [Dice 1]
-> [Dice 5; Dice 5]
- Current player: John -

John rolled 3 dice: [Dice 5; Dice 5; Dice 1]
- Current round total: 200

3/3 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 5; Dice 5], with a total of 100

- Current roll total: 200
Would you like to bank your points (1) or roll all 6 dice again? (2)
>
```

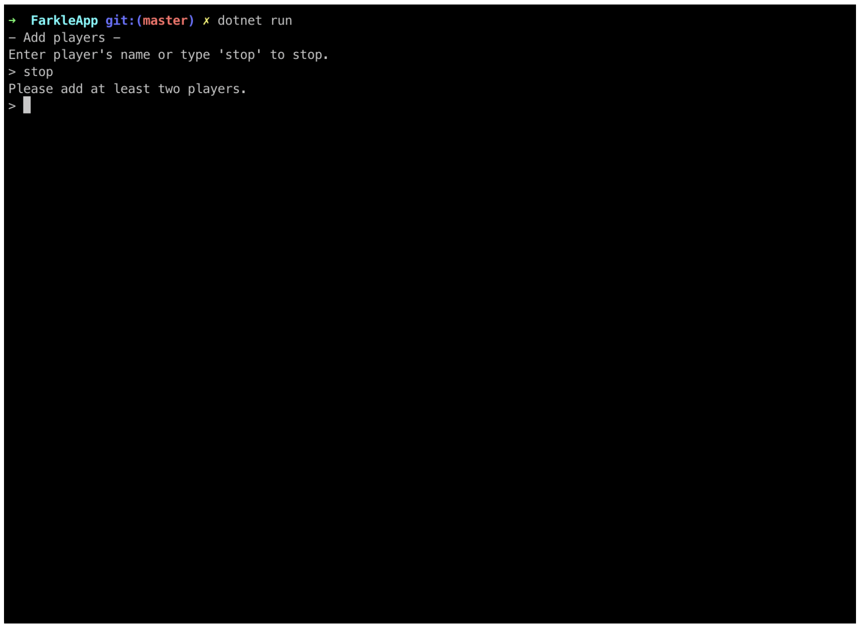
Figure 11: Output of choosing roll again without selected dice



```
-John banked their points-

Score board
---------------------
- David has 250 points
- Bob has 0 points
- John has 400 points
---------------------

- Current player: David -

David rolled 6 dice: [Dice 1; Dice 6; Dice 5; Dice 1; Dice 3; Dice 2]
- Current round total: 0

3/6 dice scored. Scoring dice:
1) [Dice 1; Dice 1], with a total of 200
2) [Dice 5], with a total of 50

- Current roll total: 250
Would you like to bank your points (1) or roll again? (2)
> 
```

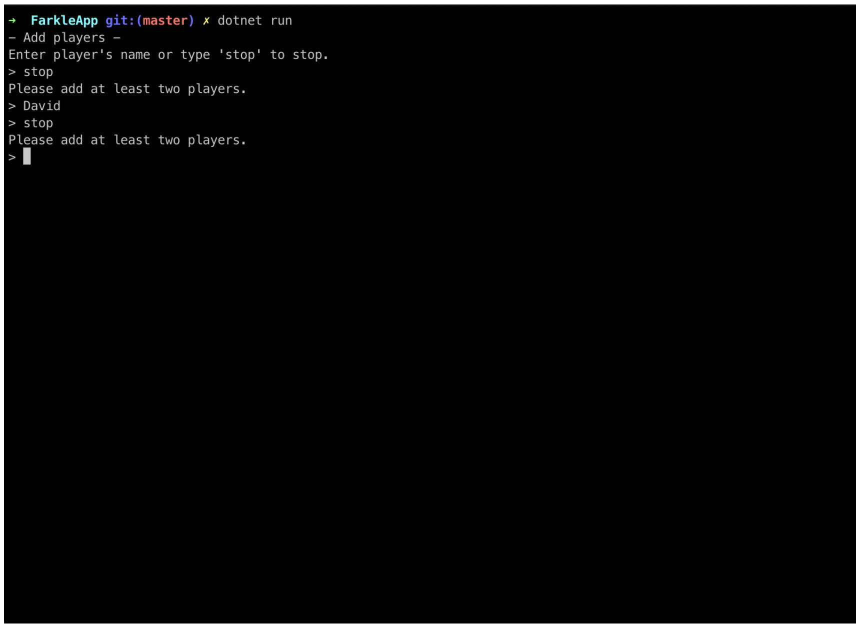
Figure 12: Output of choosing to bank points

4.1.2 Trial run 2 - invalid input



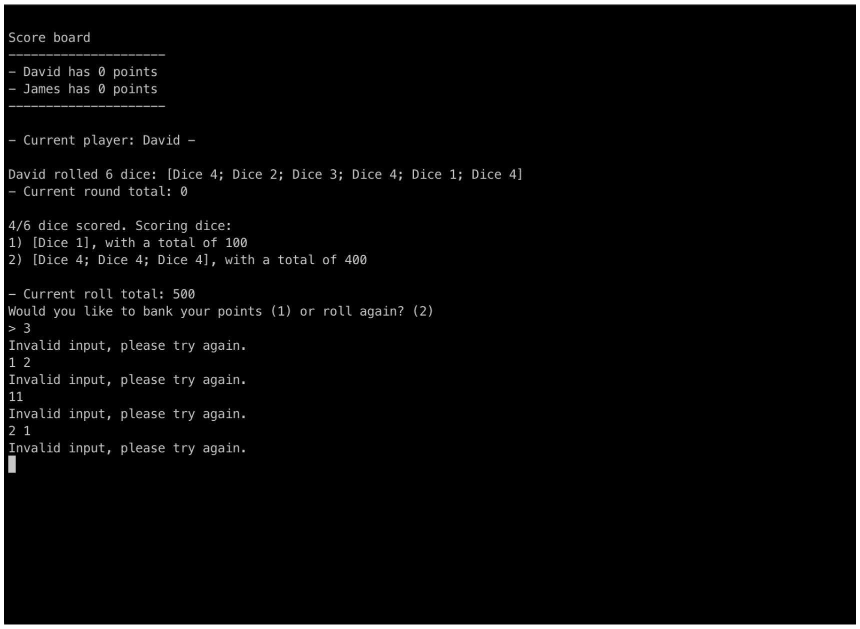
```
➜  FarkleApp git:(master) ✗ dotnet run
- Add players -
Enter player's name or type 'stop' to stop.
> stop
Please add at least two players.
> 
```

Figure 13: Attempting to add no players



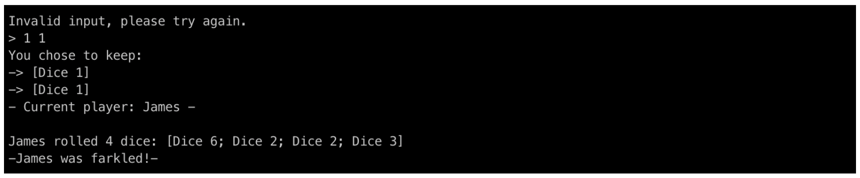
```
➜  FarkleApp git:(master) ✗ dotnet run
- Add players -
Enter player's name or type 'stop' to stop.
> stop
Please add at least two players.
> David
> stop
Please add at least two players.
> 
```

Figure 14: Attempting to add 1 player



```

Score board
---------------------
- David has 0 points
- James has 0 points
---------------------

- Current player: David -

David rolled 6 dice: [Dice 4; Dice 2; Dice 3; Dice 4; Dice 1; Dice 4]
- Current round total: 0

4/6 dice scored. Scoring dice:
1) [Dice 1], with a total of 100
2) [Dice 4; Dice 4; Dice 4], with a total of 400

- Current roll total: 500
Would you like to bank your points (1) or roll again? (2)
> 3
Invalid input, please try again.
1 2
Invalid input, please try again.
11
Invalid input, please try again.
2 1
Invalid input, please try again.
```

Figure 15: Entering invalid choices

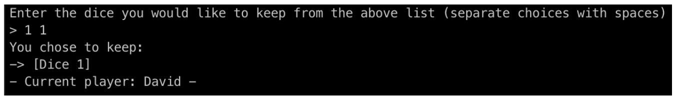


```
Invalid input, please try again.
> 1 1
You chose to keep:
-> [Dice 1]
-> [Dice 1]
- Current player: James -

James rolled 4 dice: [Dice 6; Dice 2; Dice 2; Dice 3]
-James was farkled!-
```

Figure 16: Entering invalid roll again choice, error needs to be fixed

This final test highlighted an error that can be caused by repeated the same valid roll again option. To solve this issue I converted the list that contains the roll again choices the user entered “parsedChoices” to set when they are being used to get the desired scores. This solves the error as a set cannot contain duplicate values.

```
for choice in Set(parsedChoices) do
    chosenScores <- chosenScores @ [scoreList.[choice - 1]]
```



```
Enter the dice you would like to keep from the above list (separate choices with spaces)
> 1 1
You chose to keep:
-> [Dice 1]
- Current player: David -
```

Figure 17: Error fixed

4.2 Automatic Testing

To perform the bulk of the testing, I will be using the unit testing library XUnit and the property-based testing library FsCheck. Unit testing relies on providing examples of valid data that the test must compare some actual result from calling a function against. This enables quick and easy testing that “serve as anchor points to ensure that your development efforts are proceeding as desired” (Steinhauser, 2018).

Property-based testing on the other hand allows you to define specific attributes for the tested system that must always be true, regardless of any test input data provided.

As stated by Scott Wlaschin, property-based testing “checks that this something (a “property”) is true for all cases, or at least a large random subset” (2014)

4.2.1 Adding XUnit and FsCheck

To add both of these libraries to the project, I moved to the root of the project directory and used:

```
dotnet new xunit -lang "F#" -o Farkle.Testing
```

To add XUnit to the project. From there I moved to the newly created "Farkle.Testing" directory and executed the both the below commands to add a reference of the main application and library to the testing project.

```
dotnet add reference ../FarkleApp/FarkleApp.fsproj
dotnet add reference ../Library/Library.fsproj
```

I then moved to the root of the project directory to add a reference of the testing project to the solution file.

```
dotnet sln add ./src/Farkle.Tests/Farkle.Tests.fsproj
```

Finally, I moved to the testing project and added FsCheck.

```
dotnet add package FsCheck --version 2.15.3
dotnet add package FsCheck.Xunit --version 2.15.3
```

4.2.2 Automated Testing Goals

In order to effectively test the application, I will add automatic tests that ensure the dice scoring component of the application works correctly for as many different variations of potential ***DiceList***s as possible.

The 'scoreRoll" function should be considered correct if:

1. The maximum number of sets for a given roll are never exceeded by scoreRoll
2. The correct ***SetCombination*** is identified, given a ***DiceList*** containing exactly one set and 3 random dice
3. The correct ***RemainderCombination*** is identified, given a ***DiceList*** containing exactly one remainder and 5 random dice
4. The correct ***RemainderCombination*** and ***SetCombination*** are identified given a ***DiceList*** containing exactly remainders and sets

4.2.3 Test 1

```
[<Fact>]
let ``test that scoreRoll never produces more than (n - (n % 3)) / 3 sets`` () =
    let property num =
        (rollDice (if num > 0 then num else 0))
        |> ``A DiceList should never have more than (n - (n % 3)) / 3 sets``
    Check.Quick property
```

Where the property is defined as:

```
[<Property>]
let ``A DiceList should never have more than (n - (n % 3)) / 3 sets``
    (roll:DiceList) =
    let scoreResults = scoreRoll roll
    let mutable numOfSets = 0
    for score in scoreResults do
        match score with
        | SetCombination _ -> numOfSets <- numOfSets + 1
        | _ -> ()

     let n = List.length roll

    numOfSets <= ((n - (n % 3)) / 3)
```

To run tests, the "dotnet test" command can be used from the solution's root directory. Once that command is executed, any functions with the "Fact" attribute will be executed.

As mentioned previously, the formulae below can be used to determine how many sets a given dice roll has for a given dice.

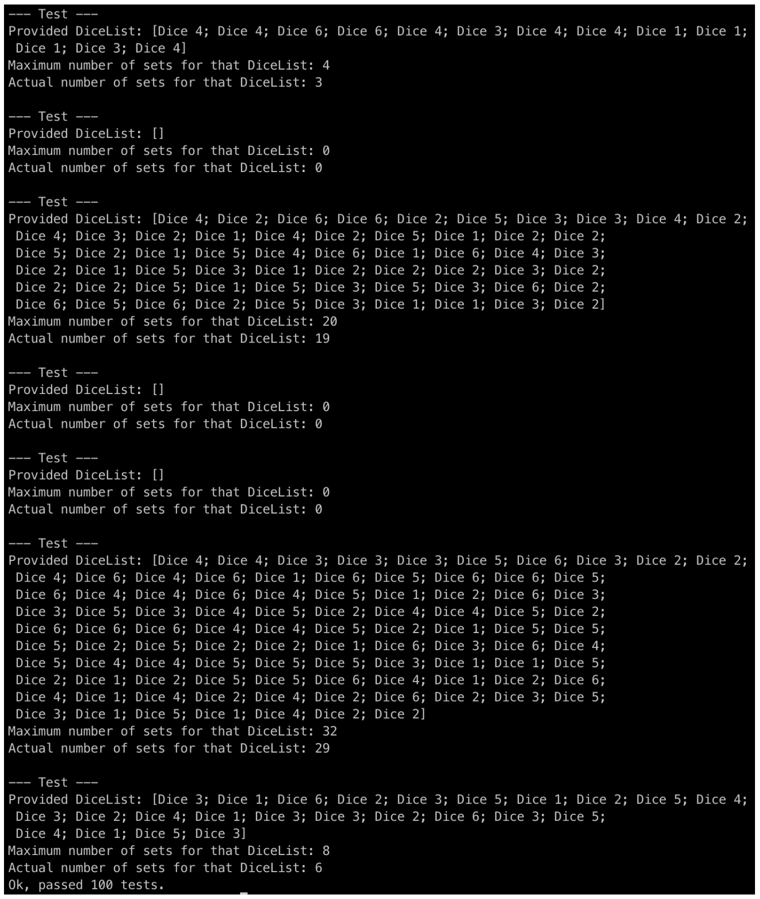
$$numOfSets = \frac{numOfOccurrences - (numOfOccurrences \bmod 3)}{3}$$

If the "scoreRoll" function is working as intended, it should never return more ***SetCombination***s than the integer division of the length of the list by 3 (or $\frac{n - (n \bmod 3)}{3}$ sets).

The "test that scoreRoll never produces more than (n - (n % 3)) / 3 sets" test ensures that for after passing a ***DiceList*** of length n (where n is greater than or equal to 0) to the "scoreRoll" function, this maximum number of sets is never exceeded.

It does this by using FsCheck's "Check.Quick" function. This will take the property defined previously and test it against a large amount of generated test data.

To ensure the test is working correctly, I made the property output the data it was generating and ran that specific test by manually calling it within the source code and using the "dotnet run" command within the solution's "Farkle.Tests" directory. The output of running this command is shown below:



```
--- Test ---
Provided DiceList: [Dice 4; Dice 4; Dice 6; Dice 6; Dice 4; Dice 3; Dice 4; Dice 4; Dice 1; Dice 1;
 Dice 1; Dice 3; Dice 4]
Maximum number of sets for that DiceList: 4
Actual number of sets for that DiceList: 3

--- Test ---
Provided DiceList: []
Maximum number of sets for that DiceList: 0
Actual number of sets for that DiceList: 0

--- Test ---
Provided DiceList: [Dice 4; Dice 2; Dice 6; Dice 6; Dice 2; Dice 5; Dice 3; Dice 3; Dice 4; Dice 2;
 Dice 4; Dice 3; Dice 2; Dice 1; Dice 4; Dice 2; Dice 5; Dice 1; Dice 2; Dice 2;
 Dice 5; Dice 2; Dice 1; Dice 5; Dice 4; Dice 6; Dice 1; Dice 6; Dice 4; Dice 3;
 Dice 2; Dice 1; Dice 5; Dice 3; Dice 1; Dice 2; Dice 2; Dice 2; Dice 3; Dice 2;
 Dice 2; Dice 2; Dice 5; Dice 1; Dice 5; Dice 3; Dice 5; Dice 3; Dice 6; Dice 2;
 Dice 6; Dice 5; Dice 6; Dice 2; Dice 5; Dice 3; Dice 1; Dice 1; Dice 3; Dice 2]
Maximum number of sets for that DiceList: 20
Actual number of sets for that DiceList: 19

--- Test ---
Provided DiceList: []
Maximum number of sets for that DiceList: 0
Actual number of sets for that DiceList: 0

--- Test ---
Provided DiceList: []
Maximum number of sets for that DiceList: 0
Actual number of sets for that DiceList: 0

--- Test ---
Provided DiceList: [Dice 4; Dice 4; Dice 3; Dice 3; Dice 3; Dice 5; Dice 6; Dice 3; Dice 2; Dice 2;
 Dice 4; Dice 6; Dice 4; Dice 6; Dice 1; Dice 6; Dice 5; Dice 6; Dice 6; Dice 5;
 Dice 6; Dice 4; Dice 4; Dice 6; Dice 4; Dice 5; Dice 1; Dice 2; Dice 6; Dice 3;
 Dice 3; Dice 5; Dice 3; Dice 4; Dice 5; Dice 2; Dice 4; Dice 4; Dice 5; Dice 2;
 Dice 6; Dice 6; Dice 6; Dice 4; Dice 4; Dice 5; Dice 2; Dice 1; Dice 5; Dice 5;
 Dice 5; Dice 2; Dice 5; Dice 2; Dice 2; Dice 1; Dice 6; Dice 3; Dice 6; Dice 4;
 Dice 5; Dice 4; Dice 4; Dice 5; Dice 5; Dice 5; Dice 3; Dice 1; Dice 1; Dice 5;
 Dice 2; Dice 1; Dice 2; Dice 5; Dice 5; Dice 6; Dice 4; Dice 1; Dice 2; Dice 6;
 Dice 4; Dice 1; Dice 4; Dice 2; Dice 4; Dice 2; Dice 6; Dice 2; Dice 3; Dice 5;
 Dice 3; Dice 1; Dice 5; Dice 1; Dice 4; Dice 2; Dice 2]
Maximum number of sets for that DiceList: 32
Actual number of sets for that DiceList: 29

--- Test ---
Provided DiceList: [Dice 3; Dice 1; Dice 6; Dice 2; Dice 3; Dice 5; Dice 1; Dice 2; Dice 5; Dice 4;
 Dice 3; Dice 2; Dice 4; Dice 1; Dice 3; Dice 3; Dice 2; Dice 6; Dice 3; Dice 5;
 Dice 4; Dice 1; Dice 5; Dice 3]
Maximum number of sets for that DiceList: 8
Actual number of sets for that DiceList: 6
Ok, passed 100 tests.
```

Figure 18: Test 1 output

This shows that even with ***DiceList***s of vastly different sizes, the maximum number of sets is never exceeded and offers a good amount of proof that that part of the "scoreRoll" function is working correctly.

4.2.4 Test 2

The next test ensures that for each number a dice could take (1 - 6), the correct ***SetCombination*** is returned by “scoreRoll” when it is provided with a ***DiceList*** that contains only one set (being 3 of the current dice to be tested).

```
[<Fact>]
let ``test for correct SetCombination`` () =
    for i in [1..6] do
        let expectedTotal = if i = 1 then SetTotal 300 else SetTotal (i * 100)
        let expectedDice = [Dice i; Dice i; Dice i]
        let expectedSet = SetCombination(expectedDice, expectedTotal)

        let generatedRolls = generateRollWithOneSetForDice (Dice i)

        for roll in generatedRolls do
            let actualScores = scoreRoll (roll |> List.ofArray)
            let actualSet = List.find (fun score ->
                    match score with
                    | SetCombination (_) -> true
                    | _ -> false) actualScores
            Assert.Equal(expectedSet, actualSet)
```

To generate the test data, I wrote the following function and put it inside the “Helpers” file/module.

```
let generateRollWithOneSetForDice (diceForSet:Dice) =
    let rand = Random()
    let mutable valid = false
    let mutable randomDice = []

    while not valid do
        randomDice <- [
            Dice (rand.Next (1, 7))
            Dice (rand.Next (1, 7))
            Dice (rand.Next (1, 7))
        ]
        if countOccurrences diceForSet randomDice <= 3 then valid <- true

    Gen.shuffle [
        diceForSet
        diceForSet
        diceForSet
        randomDice.[0]
        randomDice.[1]
        randomDice.[2]
    ] |> Gen.sample 0 6
```

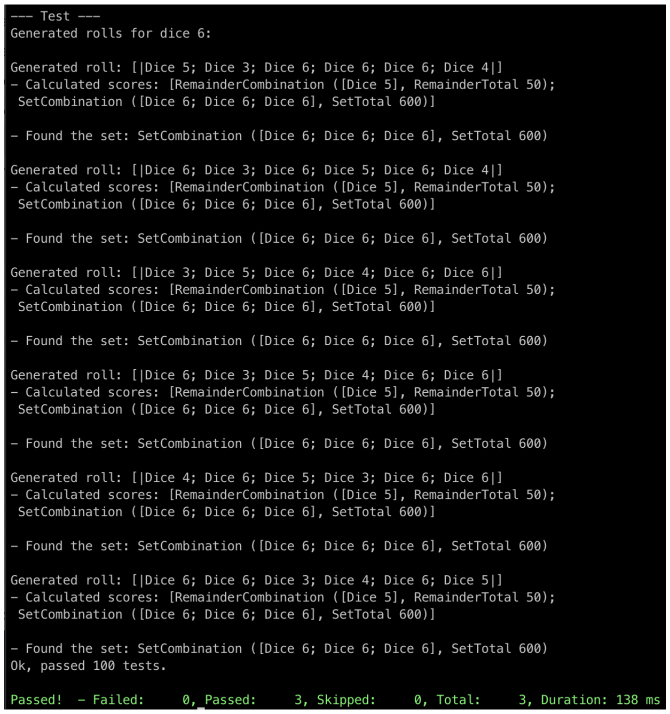
"generateRollWithOneSetForDice" creates 6 ***DiceList***s with exactly 1 set of the specified ***Dice*** and 3 other random ***Dice*** each. This is then passed to the "scoreRoll" function within the "test for correct SetCombination" test for each possible dice number (1 - 6). For each dice number, it is tested that the ***ScoreResults*** returned from "scoreRoll" include a ***SetCombination*** of the correct dice (as above):

```
for roll in generatedRolls do
            let actualScores = scoreRoll (roll |> List.ofArray)
            let actualSet = List.find (fun score ->
                    match score with
                    | SetCombination (_) -> true
                    | _ -> false) actualScores
            Assert.Equal(expectedSet, actualSet)
```

To ensure the test is working correctly, I modified the test to include some output of the generated data, as shown below:

```
[<Fact>]
let ``test for correct SetCombination`` () =
    for i in [1..6] do
        let expectedTotal = if i = 1 then SetTotal 300 else SetTotal (i * 100)
        let expectedDice = [Dice i; Dice i; Dice i]
        let expectedSet = SetCombination(expectedDice, expectedTotal)

        let generatedRolls = generateRollWithOneSetForDice (Dice i)

        printfn "\n--- Test ---"
        printfn $"Generated rolls for dice %d{i}:"

        for roll in generatedRolls do

            printfn $"\nGenerated roll: %A{roll}"

            let actualScores = scoreRoll (roll |> List.ofArray)

            let actualSet = List.find (fun score ->
                    match score with
                    | SetCombination (expectedDice, expectedTotal) -> true
                    | _ -> false) actualScores

            printfn $"- Calculated scores: %A{actualScores}"
            printfn $"\n- Found the set: %A{actualSet}"

            Assert.Equal(expectedSet, actualSet)
```

The output of running the tests is shown below along with the fact that all current tests are currently passing.



```
--- Test ---
Generated rolls for dice 6:

Generated roll: [|Dice 5; Dice 3; Dice 6; Dice 6; Dice 6; Dice 4|]
- Calculated scores: [RemainderCombination ([Dice 5], RemainderTotal 50);
 SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)]

- Found the set: SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)

Generated roll: [|Dice 6; Dice 3; Dice 6; Dice 5; Dice 6; Dice 4|]
- Calculated scores: [RemainderCombination ([Dice 5], RemainderTotal 50);
 SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)]

- Found the set: SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)

Generated roll: [|Dice 3; Dice 5; Dice 6; Dice 4; Dice 6; Dice 6|]
- Calculated scores: [RemainderCombination ([Dice 5], RemainderTotal 50);
 SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)]

- Found the set: SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)

Generated roll: [|Dice 6; Dice 3; Dice 5; Dice 4; Dice 6; Dice 6|]
- Calculated scores: [RemainderCombination ([Dice 5], RemainderTotal 50);
 SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)]

- Found the set: SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)

Generated roll: [|Dice 4; Dice 6; Dice 5; Dice 3; Dice 6; Dice 6|]
- Calculated scores: [RemainderCombination ([Dice 5], RemainderTotal 50);
 SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)]

- Found the set: SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)

Generated roll: [|Dice 6; Dice 6; Dice 3; Dice 4; Dice 6; Dice 5|]
- Calculated scores: [RemainderCombination ([Dice 5], RemainderTotal 50);
 SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)]

- Found the set: SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)
Ok, passed 100 tests.

Passed!  - Failed:     0, Passed:     3, Skipped:     0, Total:     3, Duration: 138 ms
```

Figure 19: Test 2 output

4.2.5 Test 3

This test will ensure that the correct ***RemainderCombination*** is identified by "scoreRoll" from a list of generated ***DiceList***s with 5 random dice, apart from exactly one dice that forms a ***RemainderCombination***.

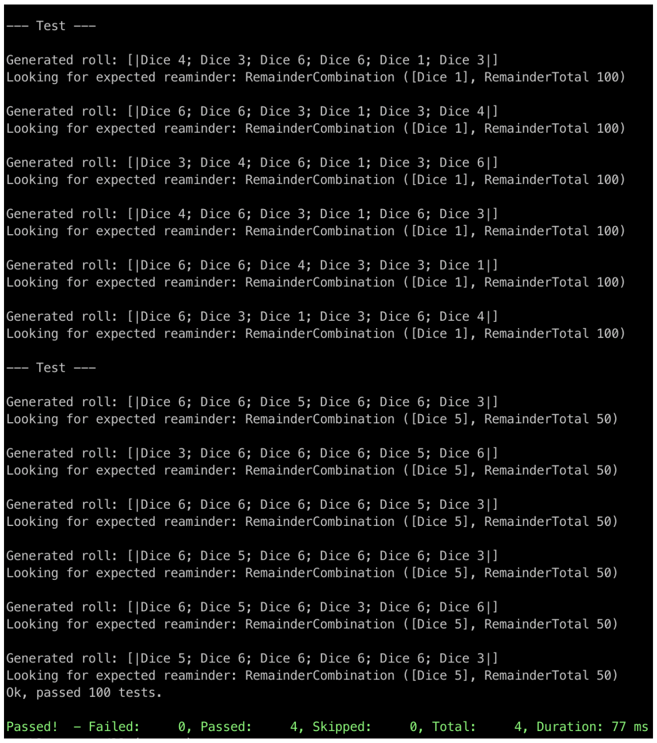
```
[<Fact>]
let ``test for correct RemainderCombination`` () =
    let remainderPossibilities = [Dice 1; Dice 5]
    for expectedDice in remainderPossibilities do

        let expectedTotal =
            if expectedDice = Dice 1 then RemainderTotal 100 else RemainderTotal 50
        let expectedRemainder = RemainderCombination([expectedDice], expectedTotal)

        let generatedRolls = generateRollWithOneRemainderForDice expectedDice

        for roll in generatedRolls do
            let actualScores = scoreRoll (roll |> List.ofArray)
            let actualRemainder =
                    List.find (fun score ->
                        match score with
                        | RemainderCombination ([expectedDice], expectedTotal) -> true
                        | _ -> false) actualScores

            Assert.Equal(expectedRemainder, actualRemainder)
```

As the only dice that score ***RemainderCombination***s are 1's and 5's, this test declares a ***RemainderCombination*** for each of these and generates the test data using the current dice and the function "generateRollWithOneRemainderForDice":

```
let generateRollWithOneRemainderForDice (diceForRemainder:Dice) =
    let rand = Random()
    let mutable valid = false
    let mutable randomDice = []

    let diceToExclude = if diceForRemainder = Dice 1 then Dice 5 else Dice 1

    while not valid do
        randomDice <- [
            Dice (rand.Next (1, 7))
            Dice (rand.Next (1, 7))
            Dice (rand.Next (1, 7))
            Dice (rand.Next (1, 7))
            Dice (rand.Next (1, 7))
        ]
        if countOccurrences diceForRemainder randomDice = 0
            && countOccurrences diceToExclude randomDice = 0 then
            valid <- true

    Gen.shuffle [
        diceForRemainder;
        randomDice.[0];
        randomDice.[1];
        randomDice.[2];
        randomDice.[3];
        randomDice.[4]
    ] |> Gen.sample 0 6
```

This will ensure the generated ***DiceList*** doesn't contain the other dice that could form a ***RemainderCombination***, that it only contains one of the desired dice and that the rest of the dice are random.

After adding some print statements, the output of running the test is shown below:



```
--- Test ---

Generated roll: [|Dice 4; Dice 3; Dice 6; Dice 6; Dice 1; Dice 3|]
Looking for expected reaminder: RemainderCombination ([Dice 1], RemainderTotal 100)

Generated roll: [|Dice 6; Dice 6; Dice 3; Dice 1; Dice 3; Dice 4|]
Looking for expected reaminder: RemainderCombination ([Dice 1], RemainderTotal 100)

Generated roll: [|Dice 3; Dice 4; Dice 6; Dice 1; Dice 3; Dice 6|]
Looking for expected reaminder: RemainderCombination ([Dice 1], RemainderTotal 100)

Generated roll: [|Dice 4; Dice 6; Dice 3; Dice 1; Dice 6; Dice 3|]
Looking for expected reaminder: RemainderCombination ([Dice 1], RemainderTotal 100)

Generated roll: [|Dice 6; Dice 6; Dice 4; Dice 3; Dice 3; Dice 1|]
Looking for expected reaminder: RemainderCombination ([Dice 1], RemainderTotal 100)

Generated roll: [|Dice 6; Dice 3; Dice 1; Dice 3; Dice 6; Dice 4|]
Looking for expected reaminder: RemainderCombination ([Dice 1], RemainderTotal 100)

--- Test ---

Generated roll: [|Dice 6; Dice 6; Dice 5; Dice 6; Dice 6; Dice 3|]
Looking for expected reaminder: RemainderCombination ([Dice 5], RemainderTotal 50)

Generated roll: [|Dice 3; Dice 6; Dice 6; Dice 6; Dice 5; Dice 6|]
Looking for expected reaminder: RemainderCombination ([Dice 5], RemainderTotal 50)

Generated roll: [|Dice 6; Dice 6; Dice 6; Dice 6; Dice 5; Dice 3|]
Looking for expected reaminder: RemainderCombination ([Dice 5], RemainderTotal 50)

Generated roll: [|Dice 6; Dice 5; Dice 6; Dice 6; Dice 6; Dice 3|]
Looking for expected reaminder: RemainderCombination ([Dice 5], RemainderTotal 50)

Generated roll: [|Dice 6; Dice 5; Dice 6; Dice 3; Dice 6; Dice 6|]
Looking for expected reaminder: RemainderCombination ([Dice 5], RemainderTotal 50)

Generated roll: [|Dice 5; Dice 6; Dice 6; Dice 6; Dice 6; Dice 3|]
Looking for expected reaminder: RemainderCombination ([Dice 5], RemainderTotal 50)
Ok, passed 100 tests.

Passed!  - Failed:     0, Passed:     4, Skipped:     0, Total:     4, Duration: 77 ms
```

Figure 20: Test 3 output

4.2.6 Test 4

The final test will test use some pre-defined examples of ***DiceList***s, including a mixture of ***RemainderCombination***s and ***SetCombination***s. Along with the previous tests, this will allow me to quickly determine with a high level of confidence that the “scoreRoll” function will work correctly for any given ***DiceList***.

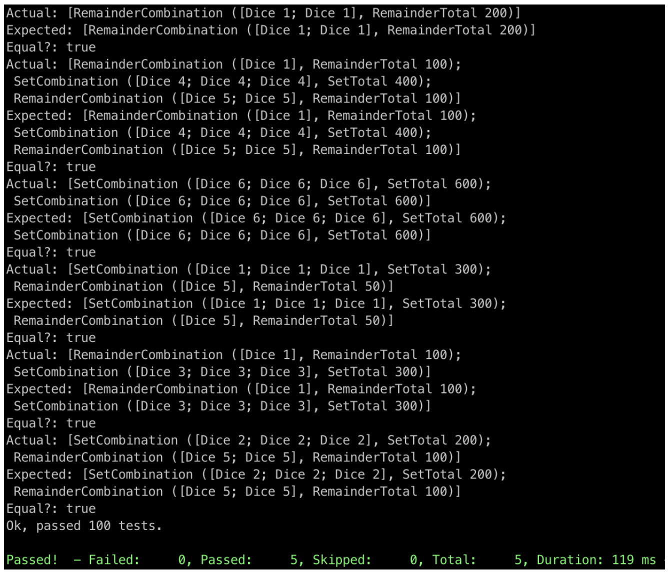
```
[<Fact>]
let ``test for correct RemainderCombination and SetCombination`` () =
    let testRolls = [
        [Dice 1; Dice 2; Dice 1; Dice 3; Dice 3; Dice 4]
        [Dice 5; Dice 1; Dice 5; Dice 4; Dice 4; Dice 4]
        [Dice 6; Dice 6; Dice 6; Dice 6; Dice 6; Dice 6]
        [Dice 5; Dice 4; Dice 1; Dice 2; Dice 1; Dice 1]
        [Dice 3; Dice 1; Dice 3; Dice 3; Dice 3; Dice 3]
        [Dice 2; Dice 2; Dice 2; Dice 6; Dice 5; Dice 5]
    ]

    let expectedResults = [
        [
            RemainderCombination([Dice 1; Dice 1], RemainderTotal 200)
        ];
        [
            RemainderCombination([Dice 1], RemainderTotal 100)
            SetCombination([Dice 4; Dice 4; Dice 4], SetTotal 400)
            RemainderCombination([Dice 5; Dice 5], RemainderTotal 100)
        ];
        [
            SetCombination([Dice 6; Dice 6; Dice 6], SetTotal 600);
            SetCombination([Dice 6; Dice 6; Dice 6], SetTotal 600)
        ]
        [
            SetCombination([Dice 1; Dice 1; Dice 1], SetTotal 300)
            RemainderCombination([Dice 5], RemainderTotal 50)
        ]
        [
            RemainderCombination([Dice 1], RemainderTotal 100)
            SetCombination([Dice 3; Dice 3; Dice 3], SetTotal 300)
        ]
        [
            SetCombination([Dice 2; Dice 2; Dice 2], SetTotal 200);
            RemainderCombination([Dice 5; Dice 5], RemainderTotal 100)
        ]
    ]
```

```
for i in [0..(testRolls.Length - 1)] do
    Assert.True(expectedResults.[i] = (scoreRoll testRolls.[i]))
```

I then added some print statements to the test:

```
for i in [0..(testRolls.Length - 1)] do
    printfn "Actual: %A" (scoreRoll testRolls.[i])
    printfn "Expected: %A" expectedResults.[i]
    printfn "Equal?: %b" ( (scoreRoll testRolls.[i]) = expectedResults.[i])
    Assert.True(expectedResults.[i] = (scoreRoll testRolls.[i]))%
```

The resulting output is shown below:



```
Actual: [RemainderCombination ([Dice 1; Dice 1], RemainderTotal 200)]
Expected: [RemainderCombination ([Dice 1; Dice 1], RemainderTotal 200)]
Equal?: true
Actual: [RemainderCombination ([Dice 1], RemainderTotal 100);
 SetCombination ([Dice 4; Dice 4; Dice 4], SetTotal 400);
 RemainderCombination ([Dice 5; Dice 5], RemainderTotal 100)]
Expected: [RemainderCombination ([Dice 1], RemainderTotal 100);
 SetCombination ([Dice 4; Dice 4; Dice 4], SetTotal 400);
 RemainderCombination ([Dice 5; Dice 5], RemainderTotal 100)]
Equal?: true
Actual: [SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600);
 SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)]
Expected: [SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600);
 SetCombination ([Dice 6; Dice 6; Dice 6], SetTotal 600)]
Equal?: true
Actual: [SetCombination ([Dice 1; Dice 1; Dice 1], SetTotal 300);
 RemainderCombination ([Dice 5], RemainderTotal 50)]
Expected: [SetCombination ([Dice 1; Dice 1; Dice 1], SetTotal 300);
 RemainderCombination ([Dice 5], RemainderTotal 50)]
Equal?: true
Actual: [RemainderCombination ([Dice 1], RemainderTotal 100);
 SetCombination ([Dice 3; Dice 3; Dice 3], SetTotal 300)]
Expected: [RemainderCombination ([Dice 1], RemainderTotal 100);
 SetCombination ([Dice 3; Dice 3; Dice 3], SetTotal 300)]
Equal?: true
Actual: [SetCombination ([Dice 2; Dice 2; Dice 2], SetTotal 200);
 RemainderCombination ([Dice 5; Dice 5], RemainderTotal 100)]
Expected: [SetCombination ([Dice 2; Dice 2; Dice 2], SetTotal 200);
 RemainderCombination ([Dice 5; Dice 5], RemainderTotal 100)]
Equal?: true
Ok, passed 100 tests.

Passed!  - Failed:     0, Passed:     5, Skipped:     0, Total:     5, Duration: 119 ms
```

Figure 21: Test 4 output

4.3 Reflections

Although I believe the unit tests I wrote test the system enough to be sure the game works as expected, I think writing tests before starting the development of the game would have been a better way of testing the system. That way I could have used them throughout development and after each major change to the game to ensure that the I hadn't inadvertently created bugs during development.

5 Project Reflection

To begin with I found using F# to be quite difficult and struggled to come to grips with the language. I believe this is because I am used to Object-Oriented languages such as Java and Python and found the paradigm shift to a functional language quite counter-intuitive. After completing the lab exercises and progressing through the assignment however I now feel that I have a much greater understanding of the language and the paradigm as a whole. This will also undoubtedly become a valuable skill in the future if a certain task suites the benefits of using a f# or a functional language.

In reflection of my efforts to complete the assignment, I believe taking part in the check-in sessions to gain feedback on my work would have likely resulted in a better result, and is something I will aim to take advantage of if there are opportunities for it for future assignments.

Discussing the project with friends from within the module helped with the planning stages of development. We made use of online meetings to share ideas and our individual progress towards the project.

As a whole I feel that the project was successful and have learnt a lot of valuable skills from it.

6 References

Gibbs, G. (1988). *Learning by doing: A guide to teaching and learning methods.* London: Further Education Unit.

Play-OnlineDiceGames (N.D.). *Farkle* [Computer game]. Available at https://libguides.ioe.ac.uk/c.php?g=482485&p=3299769 (Accessed: 30 March 2021).

Steinhauser, J. (2018) *Intro to Property-Based Testing.* Available at: https://dev.to/jdsteinhauser/intro-to-property-based-testing-2cj8 [Accessed 28 May 2021].

Werner, N. (2017) *Writing User Stories With Gherkin.* Available at: https://medium.com/@nic/writing-user-stories-with-gherkin-dda63461b1d2 [Accessed 12 April 2021].

Wlaschin, S. (2014) *Using F for testing.* Available at: https://fsharpforfunandprofit.com/posts/low-risk-ways-to-use-fsharp-at-work-3/#test-fscheck [Accessed 28 May 2021].