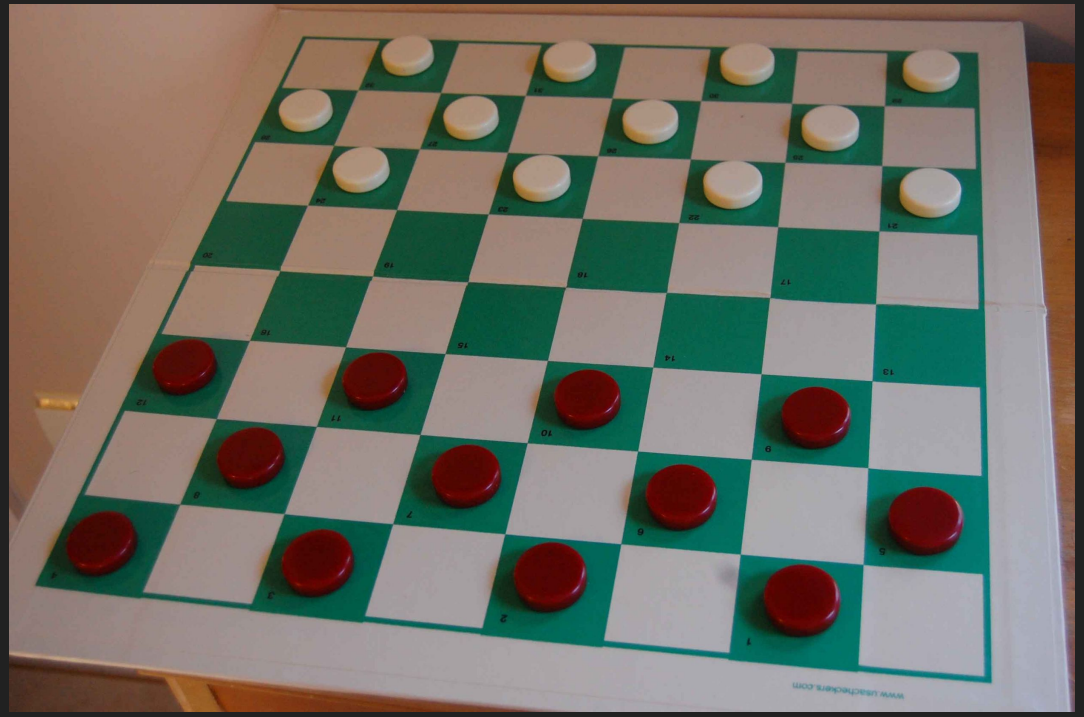
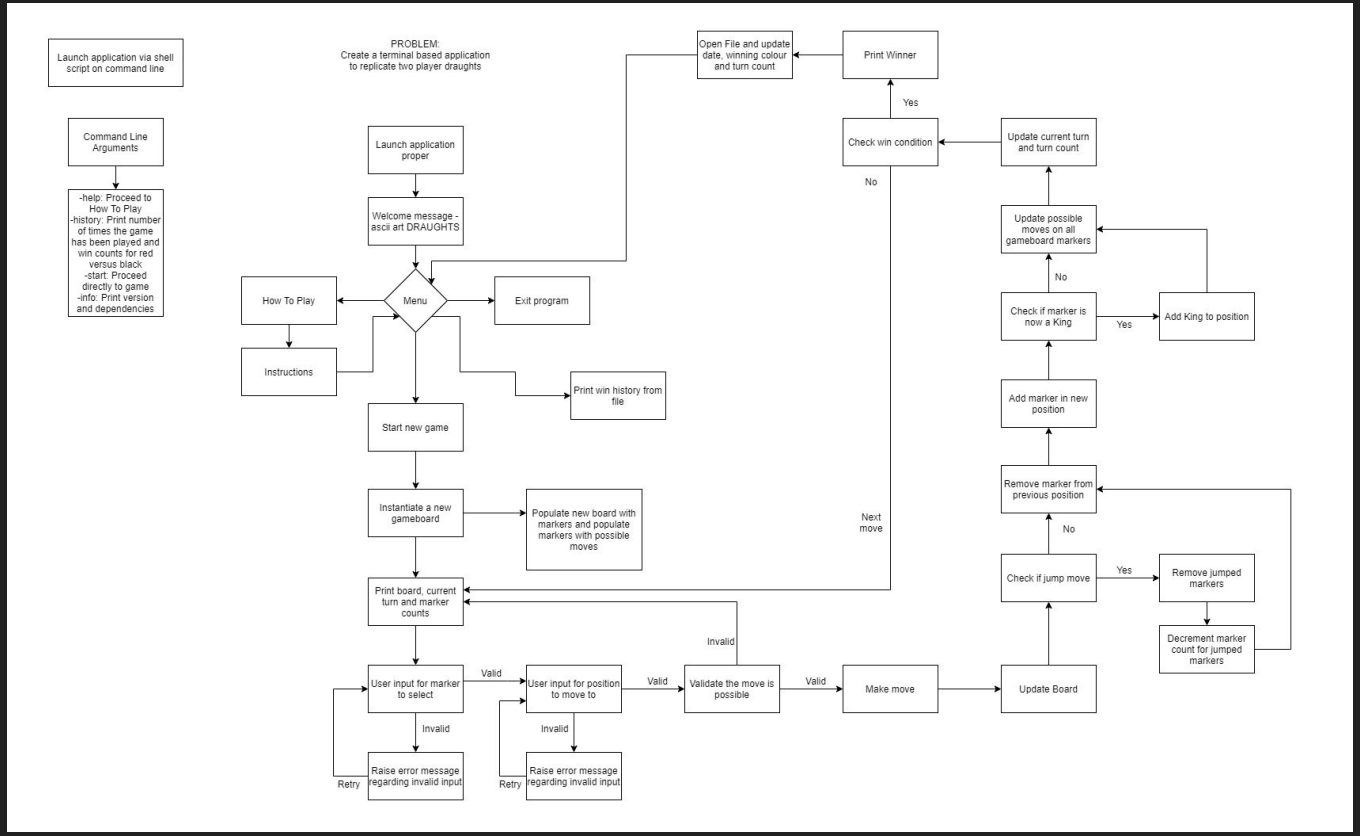
Draughts Terminal App



Features

1. A game menu featuring TTY-prompt navigation and ASCII art
2. Ability to play a 2-person game of draughts complete with King markers and multiple jumps
3. Leaderboard for total wins and colour wins, handled by interacting with external files
4. Command line arguments to print current version, help commands and bypass menu straight to a game and instructions

Flowchart


Launch application via shell script on command line
Command Line Arguments
-help: Proceed to How To Play
-history: Print number of times the game has been played and win counts for red versus black
-start: Proceed directly to game
-info: Print version and dependencies
PROBLEM:
Create a terminal based application to replicate two player draughts
Launch application proper
Welcome message - ascii art DRAUGHTS
Menu
How To Play
Instructions
Exit program
Print win history from file
Start new game
Instantiate a new gameboard
Populate new board with markers and populate markers with possible moves
Print board, current turn and marker counts
User input for marker to select
Invalid
Retry
Raise error message regarding invalid input
Valid
User input for position to move to
Invalid
Retry
Raise error message regarding invalid input
Valid
Validate the move is possible
Invalid
Valid
Make move
Update Board
Check if jump move
Yes
Remove jumped markers
Decrement marker count for jumped markers
No
Remove marker from previous position
Add marker in new position
Check if marker is now a King
Yes
Add King to position
No
Update possible moves on all gameboard markers
Update current turn and turn count
Check win condition
Yes
Print Winner
Open File and update date, winning colour and turn count
No
Next move

Menu Screen

- ASCII art header
- TTY-progressbar gem to simulate menu load effect
- TTY-prompt gem for improved menu navigation

Menu Implementation

- Each feature is organised within a self contained method
- Once the method functionality is completed, prompt to loop back to main menu
- Catch all error handling as this is towards the bottom of the call stack

```ruby
# Make a selection at main menu
def main_menu
    prompt = TTY::Prompt.new(
        active_color: :red,
        symbols: {marker: "➲"},
        quiet: true
    )

    puts ""
    menu_selection = prompt.select("Select an option:") do |menu|
        menu.choice "❶ Start a new game", 1
        menu.choice "❷ How to play", 2
        menu.choice "❸ Display win counts", 3
        menu.choice "❹ Exit program", 4
    end

    case menu_selection
    when 1
        start_game
    when 2
        instructions
    when 3
        check_win_history
    when 4
        puts "\nThanks for playing\n\nGoodbye!"
        exit
    else
        raise InvalidMenu
    end

    rescue InvalidMenu
        puts "Invalid menu input. Please try again!"
        retry
    rescue
        puts "An unexpected error has occured. The program will now exit."
end
```

How To Play

- Simple method to print how to play instructions to the terminal screen to inform the user of the rules of the game

Leaderboard

- Prints the players with the most wins
- Prints the colour with the most wins
- Handled with external files that are appended to following a win condition
- Leaderboard is populated by parsing these files and counting the number of times a winner's name appears before sorting, formatting and printing to screen

```ruby
# Check win history
def check_win_history
    win_counts = File.open("./game_history/win_counts.txt", "r")
    win_counts.readlines.each_with_index do |line, index|
        if index == 2
            split_line = line.strip.split(' ')
            puts "#{split_line[0].colorize(:blue)}: #{split_line[1].to_i}"
        elsif index == 3
            split_line = line.strip.split(' ')
            puts "#{split_line[0].colorize(:red)}: #{split_line[1].to_i}"
        else
            puts line
        end
    end
    win_counts.close

    puts ""
    puts "L E A D E R B O A R D"
    puts ""

    leaders = {}
    win_history = File.open("./game_history/win_history.txt", "r")
    win_history. readlines.each_with_index do |line, index|
        if index < 2 then next end
        winner = line.split(' ')[4]
        if leaders[winner]
            leaders[winner] += 1
        else
            leaders[winner] = 1
        end
    end
    sorted_leaders = leaders.sort_by {|k, v| -v}
    sorted_leaders.each {|name_count| puts "#{name_count[0]}: #{name_count[1]}"}
    win_history.close


    puts "\nEnter y when ready to return to menu"
    selection = ""
    while selection != "y"
        selection = gets.chomp.downcase
    end
    welcome
end
```

Handling command line arguments

- Options to print command line arguments to terminal -h or --help
- Print application version -v or --version
- Print how to play instructions -i or --info
- Go straight to win history `wins`
- Go straight to a new game `start`

```ruby
def command_line_info
    puts "\nCommand line arguments:\n\n"
    puts "This program will accept a single command line argument on launch. Arguments can be passed to draughts_app.rb directly or to draughts.sh\n"
    puts "Example: draughts.sh --help\n\n"
    puts "-h or --help      Display all command line arguments"
    puts "-i or --info      Display instructions on how to play"
    puts "start             Skip menu and immediately start a new game"
    puts "wins              Print win counts"
    puts ""
end

def handle_flags
    case ARGV[0]
    when "-h"
        command_line_info
        exit
    when "--help"
        command_line_info
        exit
    when "-i"
        ARGV.clear
        instructions
    when "--info"
        ARGV.clear
        instructions
    when "start"
        ARGV.clear
        start_game
    when "wins"
        ARGV.clear
        check_win_history
    when "-v"
        puts "Draughts 1.0.1 © Rhys Morris 2020. Ruby Version: #{RUBY_VERSION}"
        exit
    when "--version"
        puts "Draughts 1.0.1 © Rhys Morris 2020. Ruby Version: #{RUBY_VERSION}"
        exit
    end
end

if ARGV.length != 0
    handle_flags
end
```

Start Game

- Take players names
- Create a new instance of Gameboard class and calls the make_move method to handle internal game logic
- When a win condition is met in the game logic, the game_live flag becomes false and the loop is exited

```ruby
# Create a new game and allow gameboard logic to control flow
def start_game

    puts "Who is playing as red?"
    player_one = gets.chomp
    puts "\nWho is playing as blue?"
    player_two = gets.chomp
    new_game = Gameboard.new(player_one, player_two)

    while new_game.game_live
        new_game.make_move
    end

    # Prompt to return to main menu
    puts "\nEnter y when ready to return to menu"
    selection = ""
    while selection != "y"
        selection = gets.chomp.downcase
    end
    welcome
end
```

Live Demo Menu

Game Logic

- Handled via classes:
  - Gameboard class to control flow and game state
  - Marker classes to handle individual markers on the board and their internal state
  - Subclasses - RedMarker, BlueMarker, KingMarker

Why Classes?

- Classes made it easier to group game logic in a single place, and abstract the inner workings of the game away from the user.
- Game state is handled on the Gameboard class and accesses itself internally - by encapsulating game state in this manner it cannot be altered by the user in any way other than through the methods that are provided
- Conceptually it makes sense for each new game to be a new Gameboard instance

Gameboard Class

- Passed players names as arguments
  - If no names passed provides default names
- Initialize method handles creation of state and population of the Gameboard with markers
- Class variables were used to store cells within arrays based on their row position. This structure made iterating over specific cells a lot easier for marker moves

```ruby
## Marker positions and rows
@@row1 = [:a1, :b1, :c1, :d1, :e1, :f1, :g1, :h1]
@@row2 = [:a2, :b2, :c2, :d2, :e2, :f2, :g2, :h2]
@@row3 = [:a3, :b3, :c3, :d3, :e3, :f3, :g3, :h3]
@@row4 = [:a4, :b4, :c4, :d4, :e4, :f4, :g4, :h4]
@@row5 = [:a5, :b5, :c5, :d5, :e5, :f5, :g5, :h5]
@@row6 = [:a6, :b6, :c6, :d6, :e6, :f6, :g6, :h6]
@@row7 = [:a7, :b7, :c7, :d7, :e7, :f7, :g7, :h7]
@@row8 = [:a8, :b8, :c8, :d8, :e8, :f8, :g8, :h8]
@@rows = [@@row1, @@row2, @@row3, @@row4, @@row5, @@row6, @@row7, @@row8]
@@cells = []

def populate_cell_array
    @@rows.each do |row|
        row.each do |cell|
            @@cells << cell
        end
    end
end

def initialize(player_one="Player 1", player_two="Player 2")
    @current_board = {}
    @red_markers = 12
    @blue_markers = 12
    @current_turn = "red"
    @game_live = true
    @winner = nil
    @player_one = player_one
    @player_two = player_two

    # Populate cell array
    self.populate_cell_array

    # Populate new board
    self.populate_new_board

    # Update possible starting moves
    self.update_possible_marker_moves
end
```

Gameboard Class

- Gameboard positions stored as a Hash - each position can be occupied by a Marker class or nil
- The gameboard is updated after each move

```ruby
def populate_new_board
    # Populate blue markers
    @current_board[:a1] = BlueMarker.new
    @current_board[:c1] = BlueMarker.new
    @current_board[:e1] = BlueMarker.new
    @current_board[:g1] = BlueMarker.new
    @current_board[:b2] = BlueMarker.new
    @current_board[:d2] = BlueMarker.new
    @current_board[:f2] = BlueMarker.new
    @current_board[:h2] = BlueMarker.new
    @current_board[:a3] = BlueMarker.new
    @current_board[:c3] = BlueMarker.new
    @current_board[:e3] = BlueMarker.new
    @current_board[:g3] = BlueMarker.new

    # Populate red markers
    @current_board[:b6] = RedMarker.new
    @current_board[:d6] = RedMarker.new
    @current_board[:f6] = RedMarker.new
    @current_board[:h6] = RedMarker.new
    @current_board[:a7] = RedMarker.new
    @current_board[:c7] = RedMarker.new
    @current_board[:e7] = RedMarker.new
    @current_board[:g7] = RedMarker.new
    @current_board[:b8] = RedMarker.new
    @current_board[:d8] = RedMarker.new
    @current_board[:f8] = RedMarker.new
    @current_board[:h8] = RedMarker.new

    # Populate empty spots
    @current_board[:b4] = nil
    @current_board[:d4] = nil
    @current_board[:f4] = nil
    @current_board[:h4] = nil
    @current_board[:a5] = nil
    @current_board[:c5] = nil
    @current_board[:e5] = nil
    @current_board[:g5] = nil
end
```

Gameboard Class

- I created a number of helper methods on the gameboard class to handle simple jobs
- Tried to consider the single-responsibility principle with method creation
- Examples:
  - check_win
  - handle_game_over
  - print_winner
  - print_current_turn
  - update_turn

```ruby
# Print current turn
def print_turn
    puts "\nIt is #{@current_turn.capitalize}'s turn!"
end

def print_marker_counts
    puts "\nThe current marker counts are:\nBlue: #{@blue_markers}\nRed: #{@red_markers}"
end

# Update color turn
def update_turn
    @current_turn = @current_turn == "red" ? "blue" : "red"
end

# Decrement marker count
def decrement_marker_count(color)
    if color == "red"
        @red_markers -= 1
    else
        @blue_markers -= 1
    end
end
```

Gameboard Class

- These helper method formed the basis for more complicated methods such as make_move and update_board which would call simpler methods to achieve their function
- I tried to name my methods in a way which made the code very easy to read for someone who is trying to follow the logic

```ruby
# Make a move
def make_move

    # Clear screen
    system "clear"

    # Print current turn
    self.print_board
    self.print_marker_counts
    self.print_turn

    # Check whether edge case no valid moves possible
    if self.no_valid_moves_possible(@current_turn)
        puts "\nNo valid moves are possible for #{@current_turn.capitalize}!"
        puts "Switching turns in 3 seconds"
        sleep(3)
        self.update_turn
        return
    end

    # Loop move selection until valid
    while true
        marker_to_move = self.select_marker
        position_to_move = self.select_move_position

        # Valid move? - Update board and turn
        if check_valid_move(marker_to_move, position_to_move)
            self.update_board(marker_to_move, position_to_move)
            self.update_turn

            # Check if game has been won
            if self.check_win
                self.print_winner
                sleep(3)
                self.handle_game_over
            end
            return

        # Handle invalid move selection
        else
            puts "\nInvalid move selection! Please try again!"
            self.print_board
        end
    end
end
```

Update Board Logic

```ruby
def update_board(moved_marker, position_moved_to)

    # Is this a jump move?
    if moved_marker.jump_moves.include? position_moved_to

        #Handle deletion of jumped markers
        moved_marker.jump_moves[position_moved_to].each do |opposite_marker|
            puts opposite_marker
            # puts "Attempting to delete #{@current_board[opposite_marker]}"           # DEBUGGING
            @current_board[opposite_marker] = nil

            # Decrement marker count
            if @current_turn == "red" then self.decrement_marker_count("blue") end
            if @current_turn == "blue" then self.decrement_marker_count("red") end
        end
    end

    # Delete marker from previous position
    @@cells.each do |cell|
        if @current_board[cell] == moved_marker then @current_board[cell] = nil end
    end

    # Add marker to new position
    if @current_turn == "red"
        if moved_marker.king
            @current_board[position_moved_to] = KingMarker.new("red")
        else
            @current_board[position_moved_to] = RedMarker.new
        end
    else
        if moved_marker.king
            @current_board[position_moved_to] = KingMarker.new("blue")
        else
            @current_board[position_moved_to] = BlueMarker.new
        end
    end

    # Check if any markers need to be converted to Kings
    red_king_row = @@rows[0]
    blue_king_row = @@rows[7]

    red_king_row.each do |cell|
        if !current_board[cell]
            next
        elsif @current_board[cell].color == "red"
            @current_board[cell] = KingMarker.new("red")
        end
    end

    blue_king_row.each do |cell|
        if !current_board[cell]
            next
        elsif @current_board[cell].color == "blue"
            @current_board[cell] = KingMarker.new("blue")
        end
    end

    # Update possible moves
    self.update_possible_marker_moves
end
```

Marker Classes

- I used the concept of inheritance to create specific types of game markers
- Each marker would be instantiated with a colour, whether it was a king, an empty array for storing valid moves and an empty hash for storing jump moves

```ruby
class Marker
    attr_reader :color, :valid_moves, :jump_moves, :king

    def initialize(color)
        @color = color
        @valid_moves = []
        @jump_moves = {}
        @king = false
    end

end
```

```ruby
require_relative('./marker.rb')

class BlueMarker < Marker
    def initialize
        super("blue")
    end
```

```ruby
require_relative './marker'

class KingMarker < Marker
    def initialize(color)
        super(color)
        self.flag_king
    end

    def flag_king
        @king = true
    end
```

Marker Classes

- Move validation was handled by storing valid moves on each marker as an array, and jump moves as a hash where the key was the position to move to and the value was an array of marker positions that would be jumped should the move be made
- Each RedMarker, BlueMarker and KingMarker have update_valid_moves methods that look around the marker to populate @valid_moves and @jump_moves
- After each move, the current board is iterated over and update_valid_moves called on each marker to update their internal state
- I handled additional jump moves by recursively calling a check_additional_jump method each time a valid jump move was found. Implementing this feature successfully was the biggest challenge I encountered during the project, especially for kings

BlueMarker Example:

```ruby
def update_valid_moves(board, current_state, current_position)

    # Reset moves
    @valid_moves = []
    @jump_moves = {}

    # Check current row and index position in row
    current_row = current_position[1].to_i - 1
    cell_index = board[current_row].find_index current_position

    # Store next row
    next_row = board[current_row + 1]

    # Store diagonal move positions
    diagonal_right_cell_index = cell_index - 1
    diagonal_left_cell_index = cell_index + 1

    # Get diagonal right cell
    if diagonal_right_cell_index >= 0 then diagonal_right_cell = next_row[diagonal_right_cell_index] end

    # If empty push to valid moves
    if diagonal_right_cell && !current_state[diagonal_right_cell]
        @valid_moves << diagonal_right_cell

    # If contains opposite marker - check if can be jumped
    elsif diagonal_right_cell && current_state[diagonal_right_cell].color == "red"
        jump_cell_index = diagonal_right_cell_index - 1
        jump_row_index = current_row + 2
        unless jump_cell_index < 0 || jump_row_index > 7
            jump_row = board[jump_row_index]
            jump_cell = jump_row[jump_cell_index]
        end
        if jump_cell && !current_state[jump_cell]
            @valid_moves << jump_cell
            @jump_moves[jump_cell] = [diagonal_right_cell]
            self.check_additional_jump(board, current_state, jump_cell, [diagonal_right_cell])
        end
    end
```

Gameboard Call:

```ruby
# Iterate over all cells to find where markers are positioned
# Call update_valid_moves to populate possible moves on each marker
def update_possible_marker_moves
    # Pass in duplicate of current_state to prevent overwriting
    current_state = @current_board.dup
    @@cells.each_with_index do |cell, index|
        if @current_board[cell]
            @current_board[cell].update_valid_moves(@@rows, current_state, cell)
        end
    end
end
```

Handling Wins

- Wins are handled internally on the game instance. Once a win condition has occurred (0 markers of either colour) helper methods are called to open win_history.txt and win_counts.txt and update them with the winner's name and colour. I also used a Ruby gem to add a nicely formatted date for the win as well.

```ruby
# Write to colour win counts
def update_win_counts
    f = File.open("./game_history/win_counts.txt", "r")
    blue_wins = ""
    red_wins = ""
    f.readlines.each_with_index do |line, index|
        if index == 2
            blue_wins = line
        elsif index == 3
            red_wins = line
        end
    end
    f.close
    blue_win_count = blue_wins.split(' ')[1].to_i
    red_win_count = red_wins.split(' ')[1].to_i
    f = File.open("./game_history/win_counts.txt", "w")
    f.write("Win Counts")
    f.write("\n\n")
    if @winner == "blue"
        f.write("Blue: #{blue_win_count + 1}\n")
    else
        f.write(blue_wins)
    end
    if @winner == "red"
        f.write("Red: #{red_win_count + 1}\n")
    else
        f.write(red_wins)
    end
    f.close
end

# Write to win history
def update_win_history
    f = File.open("./game_history/win_history.txt", "a")
    time = Time.new
    date = Date.parse(time.to_s)
    player_winner = @winner == "red" ? player_one : player_two
    f.write("\n#{date.strftime('%a %d %b %Y')} #{player_winner} playing as #{winner}")
    f.close
end
```

Live Demo Game

Challenges

- Testing!
- Nature of the program meant that unit testing with Rspec could only take me so far
- Lots of manual testing was required to assess whether changes in the game state were occuring correctly
- I need to learn an integration testing framework!

```ruby
# Tests for Marker class
describe 'Marker' do

    it "should have a color property of red when red passed as instantiation argument" do
        red_marker = Marker.new("red")
        expect(red_marker.color).to eq("red")
    end

    it "should have a color property of blue when blue passed as instantiation argument" do
        blue_marker = Marker.new("blue")
        expect(blue_marker.color).to eq("blue")
    end

    it "should be instantiated with an empty valid_moves array" do
        new_marker = Marker.new("red")
        expect(new_marker.valid_moves).to eq([])
        expect(new_marker.valid_moves).to be_an_instance_of Array
    end

    it "should be instantiated with an empty jump_moves object" do
        new_marker = Marker.new("red")
        expect(new_marker.jump_moves).to eq({})
        expect(new_marker.jump_moves).to be_an_instance_of Hash
    end

    it "should have a readable property king that is instantiated to false" do
        new_marker = Marker.new("red")
        expect(new_marker.king).to be false
    end

end
```

Challenges

- Issues with game logic not considered initially. Often found as bugs during testing.
- Examples:
  - What to do if one player cannot make any moves.
  - How to handle when jumps to a certain position are possible in multiple ways - solved by only storing the longest jump if a jump already present

Challenges

- Accessibility is a concern. If I were to spend more time on this project I'd look at implementing a better UI, in particular, for printing of the gameboard each turn. It's not the easiest to look at immediately and know which marker you wish to move and to where.
- My major goal for this project was logic control and to challenge my understanding of the core concepts in Ruby to date, in that regard I'm very happy with what I was able to build.

Favourite Parts

- Correctly implementing multiple jumps for the first time using basic recursion
- Finding a bug, but also knowing why it was occurring and how to fix it. I’m starting to see myself make progress as a developer.
- Seeing the leaderboard print correctly inside the application
- Writing in Ruby - it’s a fun language to develop with