GCSE Computer Science NEA

Task 3 – Card Game

Miles Watson

USERNUMBER

Contents

[Analysis 2](#_Toc536030829)

[Introduction 2](#_Toc536030830)

[Success Criteria 2](#_Toc536030831)

[Design 3](#_Toc536030832)

[Key Elements 3](#_Toc536030833)

[User Interface 3](#_Toc536030834)

[Game Algorithm 5](#_Toc536030835)

[Robustness and Security 6](#_Toc536030836)

[Decomposition 7](#_Toc536030837)

[Database 7](#_Toc536030838)

[Widget 9](#_Toc536030839)

[Summative Tests 11](#_Toc536030840)

[Development 12](#_Toc536030841)

[Process 12](#_Toc536030842)

[Final 12](#_Toc536030843)

[Testing 13](#_Toc536030844)

[Evaluation 14](#_Toc536030845)

[Achievements 14](#_Toc536030846)

[Improvements 14](#_Toc536030847)

[References 15](#_Toc536030848)

[Python Syntax 15](#_Toc536030849)

[Libraries 15](#_Toc536030850)

[appJar 15](#_Toc536030851)

[sqlite3 15](#_Toc536030852)

# Analysis

## Introduction

For my GCSE Computer Science non-examined assessment, I have been tasked to make a two-player card game. The program should authenticate both players, before generating a shuffled deck of thirty numbered cards (ten red, ten black, and ten yellow). Next, each player takes a card from the top of the deck, and the winner is decided using the following rules:

1. Red beats black
2. Black beats yellow
3. Yellow beats red
4. If both cards have the same colour, the card with the greatest number wins.

The winner then receives both cards. This process continues until there are no cards left in the deck, at which point the game is finished – the winner’s score (number of cards left) is recorded. The program should be able to show the top five players and their high scores.

## Success Criteria

The program will be deemed successful if:

1. It allows two players to create accounts and login
2. The cards are shuffled randomly and unpredictably
3. Players can play the game specified in the introduction
4. Players cannot cheat the game to gain an advantage
5. The game decides the winner according to the rules
6. The usernames, passwords, and number of cards are saved in an external file
7. The program is be able to display the top five players in a leader board

# Design

## Key Elements

### User Interface

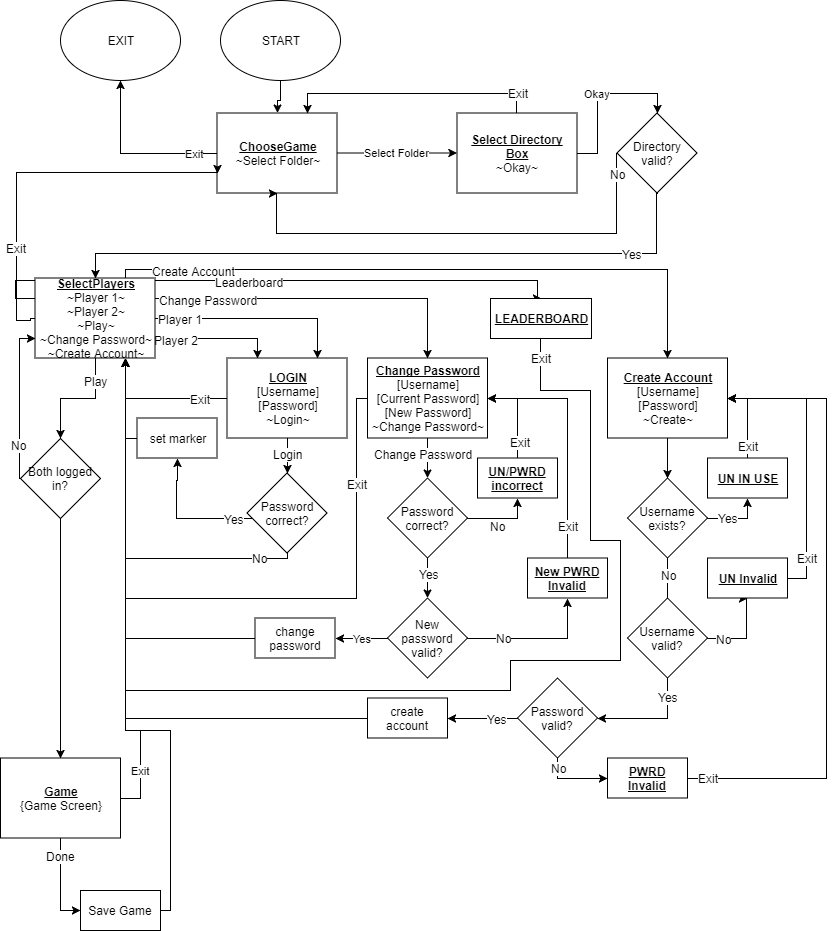
The main structure of the program is as follows:

1. Choose Game

* Allows the user to select a directory in which the game file will be saved in
* select folder button -> opens a choose directory box that will redirect to select players window if a valid directory is chosen
* Will stop the program on exit

1. Select Players
   * Allows both players to login, create accounts, or change password.
   * two player login buttons -> open a login window
   * change password button -> open a window that allows the user to change their stored password
   * create account button -> open a window that allows the user to create an account
   * play button -> open play window if both players are logged in
   * Leaderboard button -> open a window showing the top five players by total number of cards
   * Will open Choose Game window on exit
2. Play
   * The main game window, where both players can play against each other.

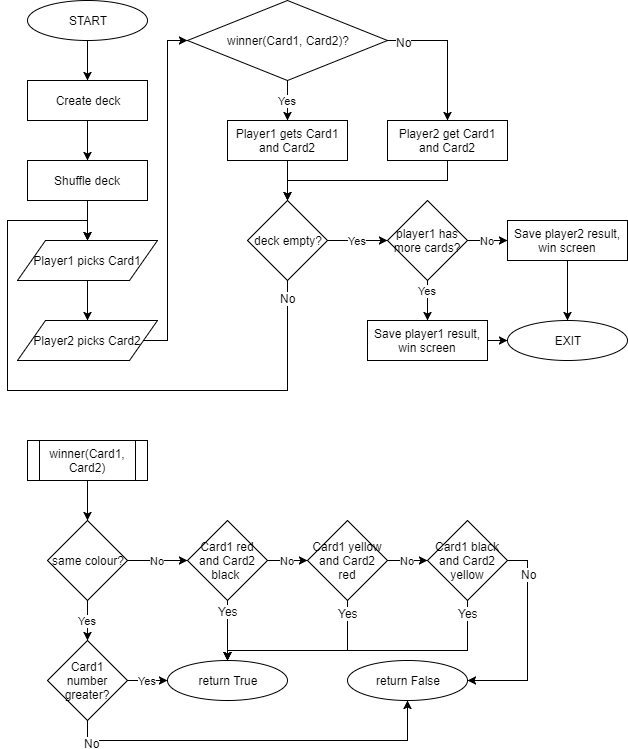
On the next page is a flowchart that shows the general structure for the program’s user interface, including the necessary decisions that the program needs to make.



### Game Algorithm

The main structure of the game is as follow:

1. Player 1 picks a card from the deck and places it on the table
2. Player 2 picks a card from the deck and places it on the table
3. The winner is decided according to the following rules:
   1. If both cards have different colours, the winner is determined by the rule: red beats black, black beats yellow, yellow beats red
   2. If both cards have the same colour, the winner is the card with the greatest number
4. The winning player obtains both cards
5. Repeat until the deck is empty
6. The winner is the player with the greatest number of cards

This flowchart outlines the basic algorithm the game should follow.

### Robustness and Security

It is important that any program should be reliable and resistant to crashing, whilst protecting against attacks. To achieve this, I will use a mixture of input validation and sanitisation to protect inputs, alongside basic password protection.

To make sure that all usernames are readably unique (they appear different to the human eye, i.e. not just different invisible characters) I will not allow users to create an account if their username contains non-alphanumeric characters. In order to achieve this, I will use python’s “string.isalnum()” function to check if every character is either a lower case letter, upper case letter, or a number. In addition, I will only allow usernames that are between 3 and 20 characters long.

Poor password management is a major cause of data theft and software insecurities. To combat this, I will implement four protection methods.

1. Passwords must be at least eight characters long. This improves security by setting a minimum number of bits of entropy.
2. The password field in the GUI will be a secret entry – all characters entered will be displayed as “\*”. This protects against shouldering: a common technique used to obtain passwords.
3. I will clear the password field from the GUI every time the window with the password field is closed, or if a user fails to enter the correct login details. This will prevent an attacker from reopening a login window and copying the password.
4. I will store a hashed version of the password in a server-less SQLite database. To protect against attackers looking at the plaintext passwords, I will use a cryptographically secure hashing algorithm, SHA-256. This can be done in python using the “hashlib.sha256(bytes)” function, followed by the “.digest()” function.
5. Before hashing, I will combine the password with a random 256-bit salt, to prevent the use of frequency analysis or rainbow tables. The hashing algorithm I will be using is python’s “hashlib.sha256()”, and I will store it as a “sha256.digest()”.

## Decomposition

In order to break down the code, I have split functionality between different classes, each responsible for different aspects of the code.

### Database

The database class is responsible for saving usernames, passwords, and cards owned in the external file.

#### Attributes

|  |  |
| --- | --- |
| Identifier | Description |
| connection | Represents the connection with the database stored in the external file. |
| cursor | A handle used to communicate with the database. |

#### Methods

|  |  |  |
| --- | --- | --- |
| Identifier | Description | Pseudocode |
| \_\_init\_\_ | Instance constructor, connects to the database and creates a “users” table (if one does not exist already). | FUNC \_\_init\_\_( directory ):  self.connection = connectToDB(directory + “/users.dbf”)  self.cursor = self.connection.get\_cursor()    self.cursor.run(“SELECT name FROM master-table  WHERE type=”table” AND name=”users”)  IF self.cursor.noItemsExist:  self.cursor.execute(“  CREATE TABLE USERS(  id INTEGER PRIMARY KEY,  username VARCHAR(16),  password CHAR(32),  cards INTEGER  )  “)  self.connection.save\_database() |
| createUser | Adds a user to the database, given a username, password hash, and a salt. | FUNC createUser(username, password, salt):  self.cursor.execute(“  INSERT INTO users(username, password, salt, cards) VALUES(username, password, salt)  “)  self.connection.save\_database() |
| changePassword | Changes the password of a user. | FUNC changePassword(username, password):  self.cursor.execute(“  UPDATE users SET password=password WHERE username=username  “)  self.connection.save\_database() |
| deleteUser | Deletes a user from the database. | FUNC deleteUser(username):  self.cursor.execute(“  DELETE FROM users WHERE username=username  “) |
| userExists | Returns True if the user exists in the database. | FUNC userExists(username):  self.cursor.execute(“  SELECT id FROM users WHERE username=username  “)  RETURN self.cursor.itemExists |
| getCards | Gets the number of cards owned by a given user. | FUNC getCards(username):  self.cursor.execute(“  SELECT cards FROM users WHERE username=username  “)  answer = self.cursor.top\_result()  RETURN answer |
| getAuthInfo | Returns the hashed password and salt of a given user. | FUNC getAuthInfo(username):  self.cursor.execute(“  SELECT password, salt FROM users WHERE username=username  “)  password, salt = self.cursor.top\_result()  return (password, salt) |
| getTopFive | Returns the top five users, sorted by most cards to least cards. | FUNC getTopFive():  self.cursor.execute(“  SELECT username, cards FROM users  ORDER BY cards DESC  LIMIT 5  “)  return self.cursor.all\_results() |
| addCards | Increases the number of cards held by a given user. | FUNC addCards(username, cards):  self.cursor.execute(“  UPDATE users SET cards=cards+cards  WHERE username=username  “)  self.cursor.save\_database() |
| close | Closes the connection to the database. | FUNC close():  self.connection.close() |

### Widget

The Widget class is a template from which each of the other widget types inherit from. It represents an item shown in the GUI, e.g. a button, an entry, or a table.

#### Attributes

|  |  |
| --- | --- |
| Identifier | Description |
| type | Stores the type of widget (e.g. entry, button, label) |
| window | A hex encoded SHA-256 hash of the window name, serves as an identifier. |
| name | The unique name of the widget. |
| id | The combination of the window hash and the unique name of the widget. |
| label | The display text of the widget. |
| row | Which row to display the widget. |
| col | Which column to display the widget. |
| special | Stores any special properties of the widget. |
| fg | The colour of the foreground. |
| bg | The colour of the background. |

#### Methods

|  |  |  |
| --- | --- | --- |
| Identifier | Description | Pseudocode |
| \_\_init\_\_ | Instance constructor, sets all fields entered. | FUNC \_\_init\_\_( widgetType, name, label=””, row=0, col=0, colspan=1, special=None, fg=None, bg=None ):  self.type = widgetType  self.window = sha256(“”).hexdigest()  self.name = name  self.id = self.window + “.” + self.name  self.label = label  self.row = row  self.col = col  self.special = special  self.fg = fg  self.bg = bg |
| setWindow | Sets the window attribute of the instance, and adds the widget to the given window. | FUNC setWindow(window):  self.window = sha256(window.name).hexdigest()  self.id = self.window + “.” + self.name  IF self.type == “button”:  app.addButton(self)  ELSE IF self.type == “label”:  app.addLabel(self)  ELSE IF self.type == “entry”:  IF self.special == “secret”:  app.addSecretEntry(self)  ELSE:  app.addEntry(self)  ELSE IF self.type == “table”:  app.addTable(self) |
| setLabel | Changes the visible text of the widget. | FUNC setLabel(text):  self.label = text  IF self.type == “button”:  app.setButton(self, self.label)  ELIF self.type == “label”:  app.setLabel(self, self.label) |
| changeBackground | Changes the background colour of the widget. | FUNC changeBackground(colour):  self.bg = colour  IF self.type == “button”:  app.setButtonBg(self, self.bg)  ELSE IF self.type == “label”:  app.setLabelBg(self, self.bg) |
| changeForeground | Returns True if the user exists in the database. | FUNC changeForeground(colour):  self.bg = colour  IF self.type == “button”:  app.setButtonFg(self, self.bg)  ELSE IF self.type == “label”:  app.setLabelFg(self, self.bg) |

## Summative Tests

# Development

## Process

## Final

# Testing

# Evaluation

## Achievements

## Improvements

# References

## Python Syntax

## Libraries

### appJar

* appJar.info

### sqlite3