Tom McClelland – 100339002

Applications Programming: Game Assignment Part II

‘Bad-Guy Brawl’ GUI edition

Design Document

Introduction

This report will cover the design and build process of a multi-user domain (MUD) game which can be played through a graphical user interface (GUI). A terminal MUD game was made in Part I of the assignment, with Part II of the assignment aiming to create the GUI in order to enhance user experience (UX). A MUD game is usually text based with minimal graphics; everything is described using words (Mortensen, 2006). The MUD game that this report covers is called 'Bad-Guy Brawl' and was built completely using the Python programming language (with in-built tkinter library) and the Visual Studio (VS) Code integrated development environment (IDE).

Included within the report is an analysis of the design steps taken throughout the GUI development, including the chosen Agile methodology, player persona, empathy map, MoSCoW, user stories, UML diagrams and testing process. A reflection and conclusion section has also been included at the end of the report.

Agile Methodology

'Bad-Guy Brawl' is designed to entertain the player as they adventure, train and battle their way to becoming Champion. The GUI development began with a functioning terminal game that was developed in the first part of the assignment. This terminal game was developed using a hybrid of the 'code & fix' and 'big bang' models of software development. The reason that these models were used in the early stages was due to a lack of knowledge about agile methodologies and the steep learning curve that was being undertaken at the time. Using these simple models in the early stages allowed for learning of python, and game development, to take place simultaneously as the process was simple and light on resources.

The second part of the assignment was approached differently as at this point the knowledge of agile methodologies had been gained. At the onset of the second assignment, the scrum agile methodology was deemed the most suitable to the situation. With the scrum methodology, the entire software development life cycle (SDLC) is divided up into a series of iterations where each iteration is called a sprint. After each sprint, there is a working increment of the software that is being developed (Sharma et al., 2012). Figure 1 illustrates the scrum agile methodology.

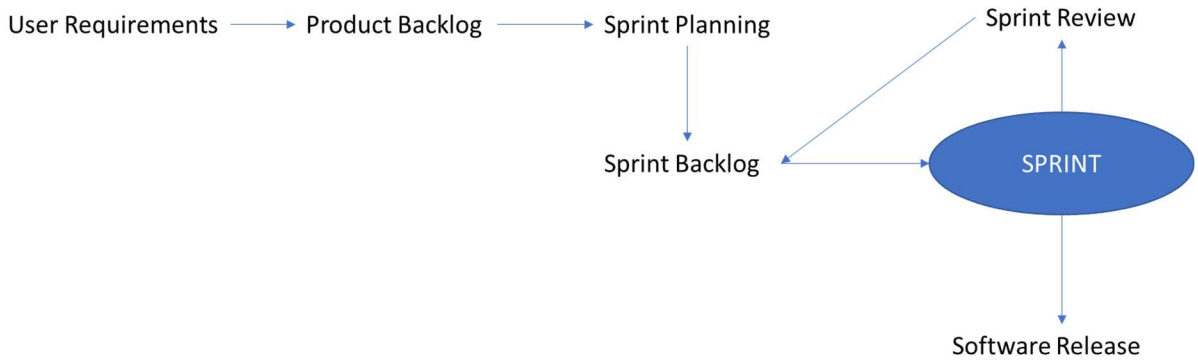

Figure 1. The Scrum Agile Methodology

By using the scrum methodology, this allowed the implementation process to begin earlier - each sprint would allow for the game to become slightly more advanced as more was learned about the built in tkinter library. The 1st sprint resulted in the development of an over-simplified version of the

game where adventuring did not involve puzzles, battles had no choice of moves and shopping wasn't an option at all. However, the 2nd sprint introduced all the features previously mentioned as, at this point, there was more confidence with using the tkinter library. The 3rd sprint allowed for 'finishing touches' to be made to the game, most of which were guided by the assignment brief. For example, the ability to sort collectables alphabetically when being viewed. Before any of these sprints could take place, it was essential to assess user requirements and create a product backlog. This was done by reading the assignment brief, creating player personas, empathy map, MosCoW analysis and user stories, all of which are shown below.

Assignment Brief

The following features were requested by the assignment brief:

- Allow user to relay and receive instructions via the GUI instead of command line.
- Allow user to request current and previous location information and display this with both text and images.
- Allow a user to request character information and display this with both text and images.
- Allow a user to request current collectables and display them in alphabetical order
- Allow a user to request general progress
- Optional Customisation/Additional Features

Player Personas

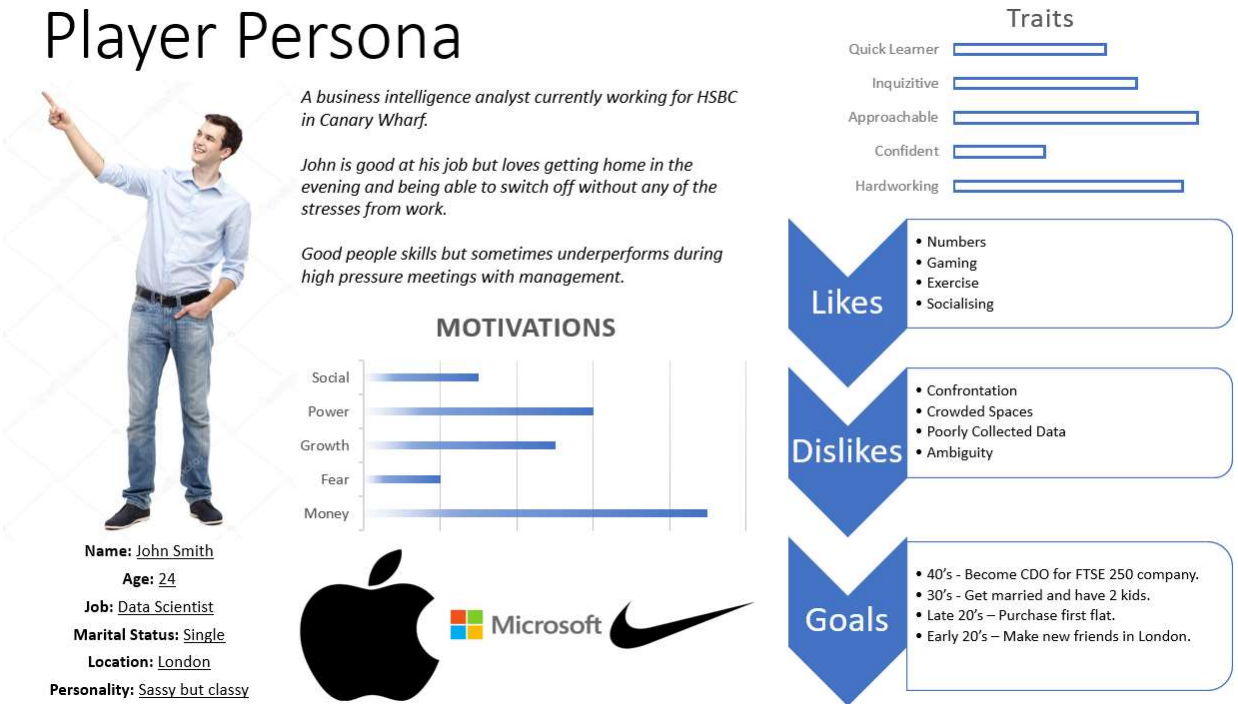

Figure 2. John Smith Player Persona

Part of the process of gathering user requirements involved creating a player persona, therefore giving developers an idea of the type of person the game would cater for, by providing a user background which can include a player's computer competency (Butt and Li, 2015).

Figure 3. Jane Turner Player Persona

Both player personas were considered when putting together the product backlog for 'Bad-Guy Brawl'. One consideration that could not be overlooked was Jane's hearing impairment, which causes her to struggle in lectures at University. The game will accommodate for this by being completely text based with zero audio requirement for the user. Neither persona considered a player with poor eyesight, but this will be accommodated for by having bold text that is large and easy to read.

Empathy Map

To take the player persona a step further, an empathy map is used to draw out user perspective to a greater degree, often carried out over the course of lengthy working sessions with story boarding and other techniques (Darrin and Davereux, 2017). However, due to time constraints, the empathy map in figure 4 was developed individually over the course of a single lab session. The empathy map includes how the player thinks & feels, what they say & do and what they hear & see. By understanding this, developers can better predict what the user wants from the software.

MoSCoW Analysis

The MoSCoW analysis was used to come up with features that the game **m**ust have, **s**hould have, **co**uld have and **w**on't have. This MoSCoW analysis naturally followed on from the development of the player persona and empathy map as it meant the user was better understood before any features were developed for them. The MoSCoW analysis also enables prioritisation of these features so that the development team understands which features are most important and so should have the most time dedicated to them (Kuhn, 2009).

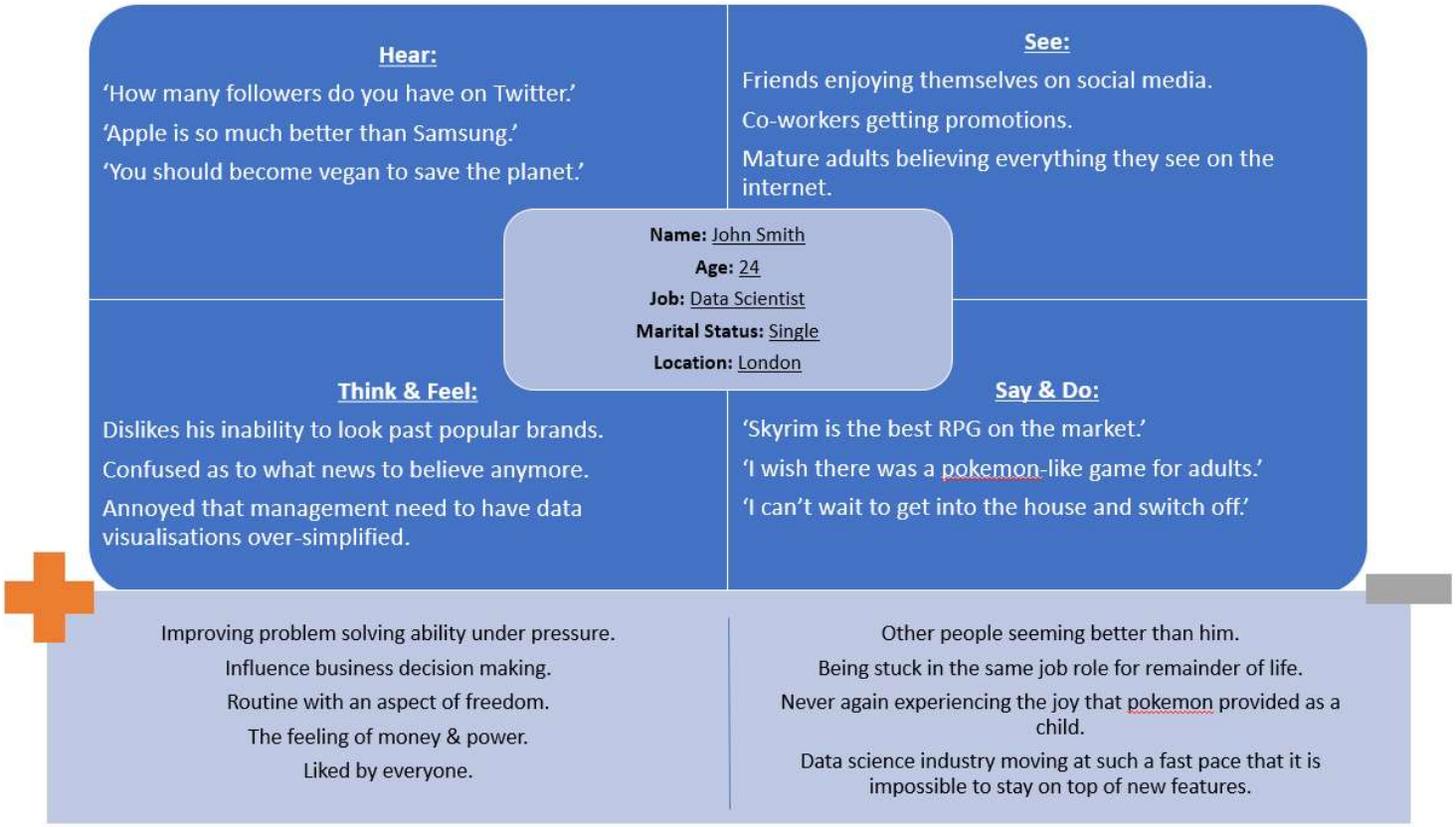

Figure 4. John Smith Empathy Map

Figure 5. MoSCoW Analysis

User Stories

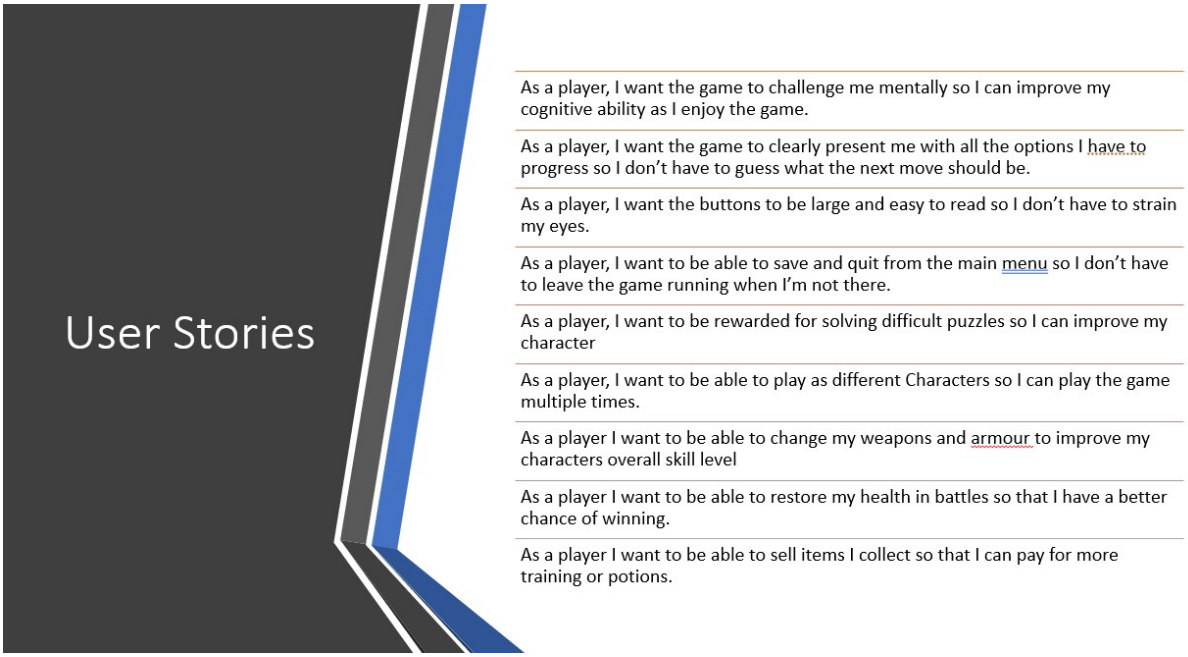

Figure 6. User Stories

A user story is a tool used in Agile development to capture a description of a software feature from an end-user perspective (Pallavi and Tahiliani, 2015). The user stories are useful for providing the development team with reasons as to why a user would want certain features, and help them to further empathise with the user. The user stories were created after the MoSCoW analysis had been carried out but alternatively they could have been created before. The player persona and empathy map were useful prerequisites for developing the user stories, due to building empathy towards the player from an early stage.

Stakeholders

McManus (2011) defines a stakeholder as anyone who is involved in, or affected by, a project or programme. From this definition, stakeholders of 'Bad-Guy Brawl' could be the development team whose sense of pride may depend on success. A possible stakeholder could also be the end-user whose happiness may depend on the game being bug free or even a funding organisation who have invested large amounts of cash and want to see positive returns over time. Whilst the end-user is a stakeholder in this sense, this is not to be confused with their role as a customer also.

Unified Modelling Language (UML)

Modelling simplifies the systems design, detailing important parts and reducing any confusion within the team. UML Diagrams can illustrate how the system will interact with people, organisations, or other systems. Microsoft Visio was trialled during the creation of the UML Diagrams for this assignment but it was eventually decided that Microsoft Powerpoint would be a simpler alternative that was better understood. The exception to this came during the creation of the UML Class Diagram, which was done on VS Code using the PlantUML extension, as shown below in figure 7.

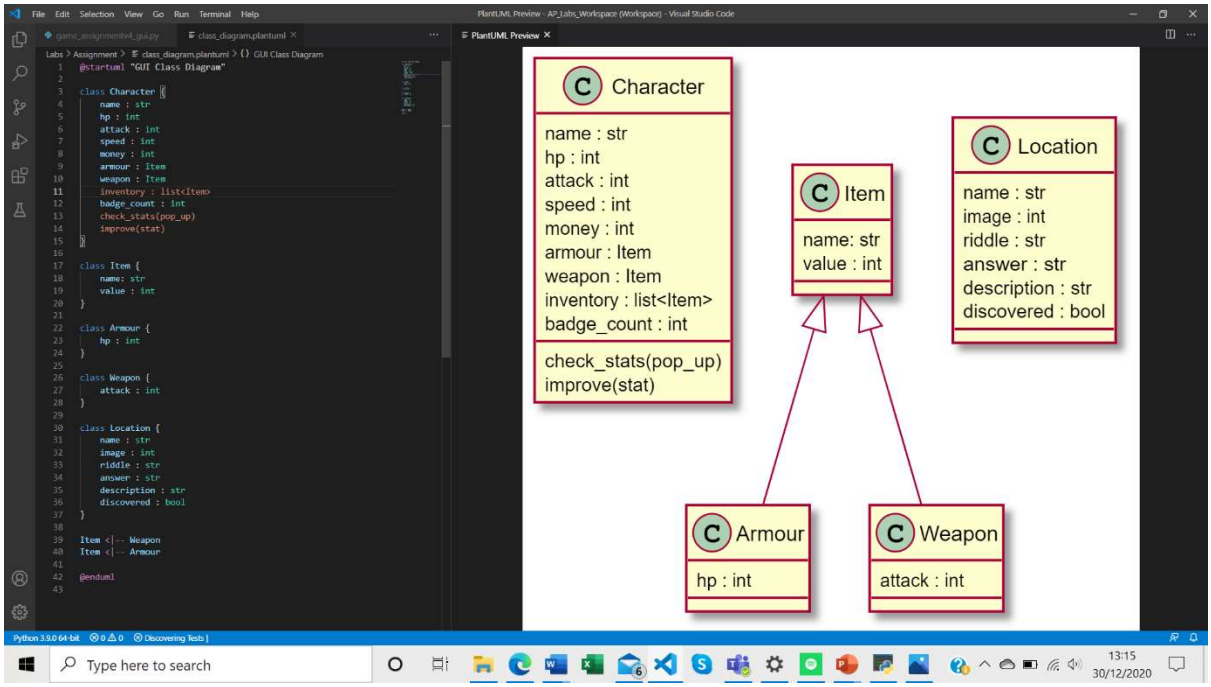

Figure 7. Class Diagram using PlantUML

As can be seen from figure 7, both the Armour and Weapon Classes inherit from the Item Class but this is the only relationship between classes. PlantUML was found to be quite easy to use to develop a class diagram and in future is likely to take preference over Microsoft Powerpoint, although for simple diagrams, Microsoft Powerpoint is often the best tool.

The next step of the modelling process involved creating a 'Use Cases' diagram alongside a 'textual use' case table. The use cases diagram consists of a boundary to show what is inside the system and what is not, an actor which illustrates the systems user, use cases which are the actions that can be taken within the system and relationships which show how different use cases and actors interact. Figure 8 shows the full use cases diagram for the Bad Guy Brawl Game and figure 9 shows the textual use case table for the 'Battle' action.

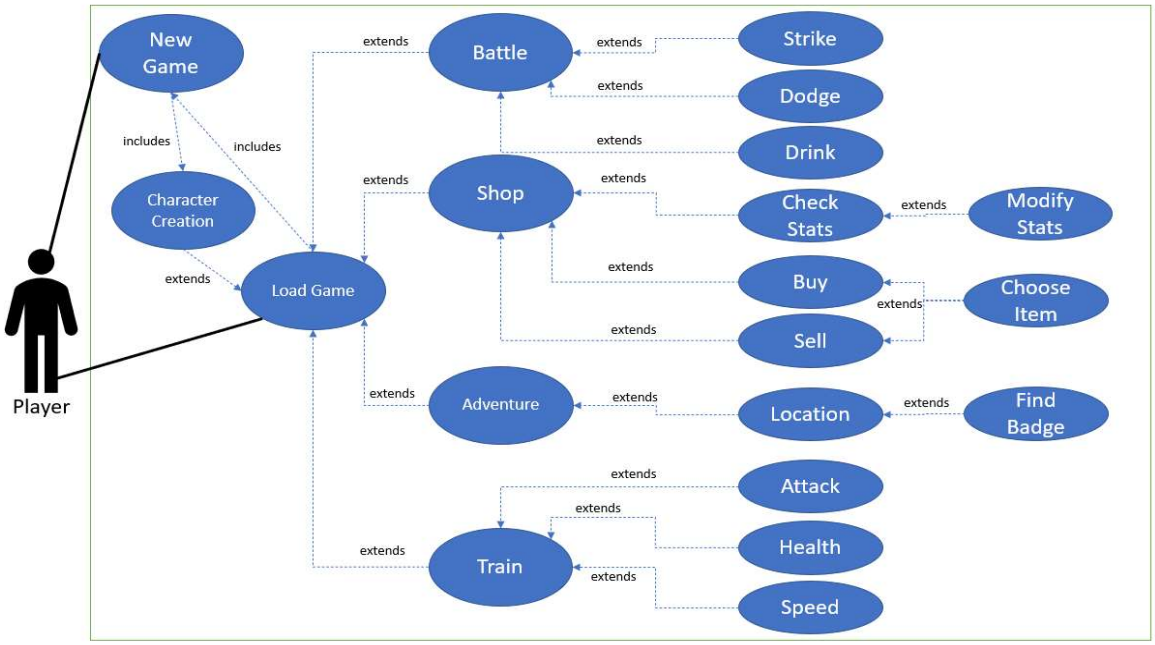

Figure 8. Use Cases Diagram for Bad Guy Brawl

Use Case Name:	Battle
Goal in Context:	Allow player to challenge the next foe in the Battle Hub.
Scope and Level	Must Have and High Level.
Pre-Condition(s)	Player must have started a new game, selected their character and found a badge while adventuring to unlock the next foe. They may also load a game where all this has been done already.
Success End Condition:	Player has a fight with the next foe in the Battle Hub.
Failed End Condition:	Player cannot fight with the next foe in the Battle Hub.
Actor(s)	Player.
Trigger:	Player clicks on 'Battle' button.
Normal/Successful Path Description:	1. Player clicks on battle() button 2. System back end cycles through foe_list to find the next undefeated foe (ranked in order of difficulty). 3. System back end checks that Player has found enough badges to challenge the Foe. 4. Selected foe is then loaded into the fight with player's character 5. Foe and Player stats appear so player can adjust fighting tactics. 6. Player can choose to strike(), dodge() or drink() their way to glory. 7. If the Player wins, their health is restored fully and they loot the Foe's items and gold. The win is recorded within the Character class and the defeated foe will not be available to fight again. 8. If the Player loses, their health is restored fully and so is the Foe's. The loss is recorded within the Character class and the Foe will be available to fight again. 9. Win/Loss Confirmation appears and Player is returned to main menu.
Alternative/Unsuccessful Paths:	1. A) Unknown error occurs when battle() button is pressed. Bug report request appears for player and they remain in main menu. 2. A) There are no Foes left to defeat. Confirmation that you are Champion appears and you stay in the main menu. 3. A) Not enough badges have been found. Confirmation of not enough badges appears and player advised to adventure. 4. X 5. X 6. A) There are no more health potions in your inventory. Confirmation of no potions appears and it remains your turn to strike() or dodge().

	B) Player drinks potion but already has full health. Potion is removed from inventory and health doesn't increase. C) strike() or dodge() fails and player receives damage from foe. 7. X 8. X 9. X

Figure 9. Textual Use Case for 'Battle' action

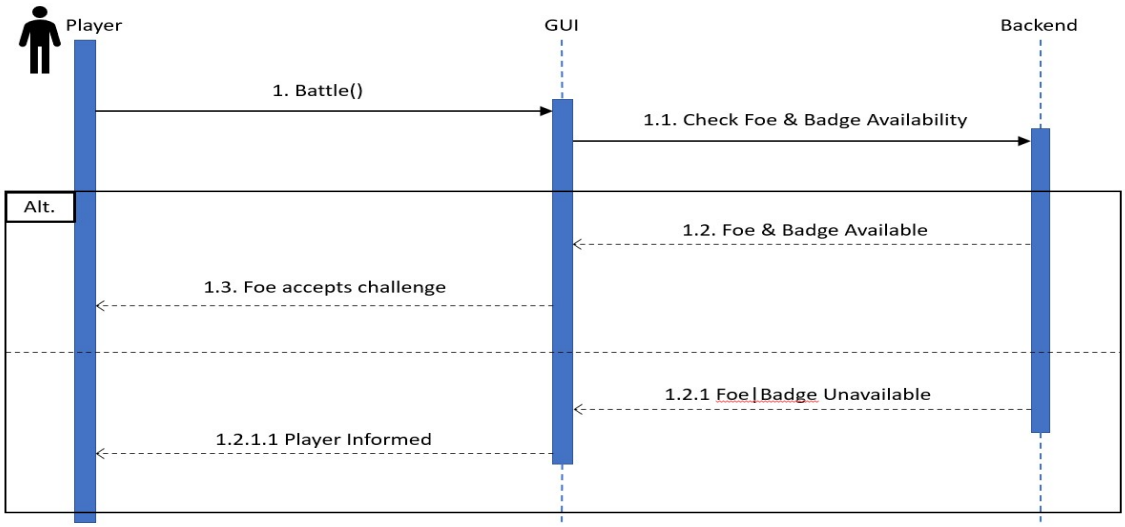

Figure 10. Sequence Diagram for 'Battle' action

The final UML Diagram that was created during the assignment was a sequence diagram, again, for the 'Battle' action. The sequence diagram shows the process that the system goes through once the 'Battle' action is taken within the game. The system will first check if the player has sufficient badges to get into the battle, and secondly it will check if any foes remain unbeaten. Should both these prerequisites be met, the player will enter the battle where further options on how to approach the fight will be presented (strike, dodge, drink potion). Should either of these prerequisites not be met, the player will be informed of the reason that they cannot access the battle.

GUI Design

How the game was going to look aesthetically was another important part of the design process. By creating lo-fi and medium-fi drawings, this allowed for multiple iterations of the GUI to be created with minimal effort. Feedback on the lo-fi and medium-fi drawings were taken into consideration during the coding of the game. Figure 11 shows the initial lo-fi idea at the top which made it through to user testing before issues were discovered with the buttons moving that will be discussed later in the report. The lower lo-fi drawing is what the game was based on once the user feedback had been considered and they agreed that it would be more suitable.

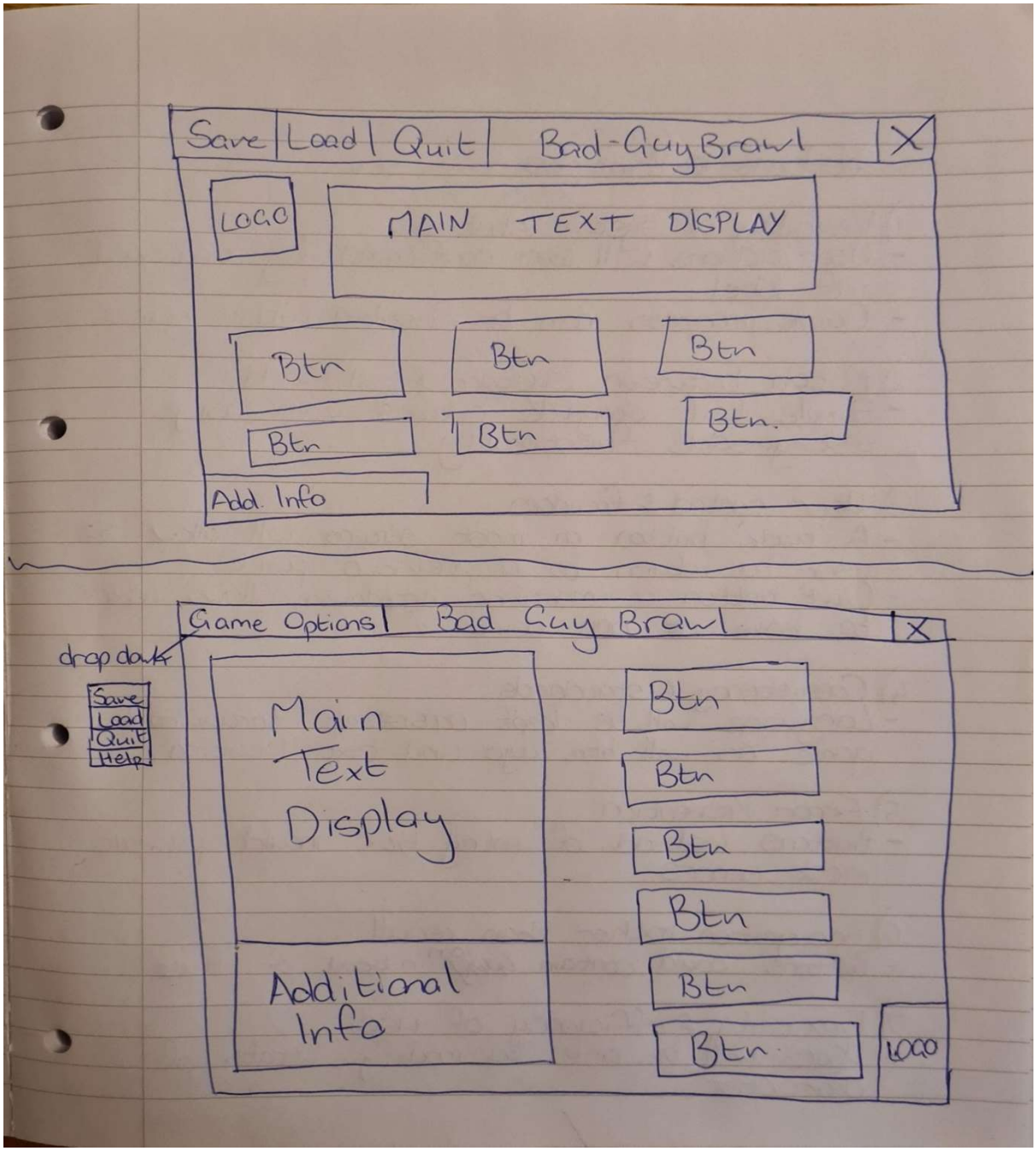

Figure 11. Lo-Fi Drawings for GUI

As these lo-fi drawings were being made, Nilsen's 10 Heuristics were constantly being considered as this would allow for a greater user experience (UX). They were adhered to as follows:

1. Visibility of system status.
    - User actions are confirmed by additional info label.
    - Game progress can be checked within the game.
2. Match between system and real world.
    - Simple text alongside buttons with easily distinguishable functionality.
3. User Control & Freedom
    - A 'back' button on most screens allows the user to return to a previous point.

- ‘Quit’ button on menubar dropdown allows user to leave at any point.

4. Consistency and standards.
    - Language will be kept consistent throughout the game as will button, picture and text location.
5. Error prevention.
    - Buttons instead of entry box should prevent most errors. Thorough testing will be required to confirm.
6. Recognition rather than recall.
    - Some buttons contain an icon alongside the text.
7. Flexibility and efficiency of use.
    - Player will be able to modify stats of character by changing loadout.
8. Aesthetic and minimalist design.
    - Simple layout without any hidden features.
    - Clear buttons and text.
9. Help users recognize, diagnose, and recover from errors.
    - Help button on the menubar that is always accessible.
10. Help and documentation.
    - Help button on the menubar that is always accessible.

Testing

As mentioned previously, the scrum agile methodology was followed for the creation of the GUI with 3 sprints taking place. At the end of each sprint, the code was thoroughly tested to check for bugs. By testing at the end of each sprint and not just at the end of all 3 sprints, this prevents bugs from manifesting and becoming more expensive (and time consuming) to fix. The 10 best practice principles to prevent bugs were followed, as is mentioned in the video presentation. While there were multiple bugs found during each phase of testing, I will report on only 1 that was found at each phase. Testing after sprint 1, 2 and 3 was done on an entire system basis with the game being played multiple times taking different actions each time and checking for any irregularities. Before the system testing after sprint 3, some unit testing was carried out also. Unit testing should have been carried out at each step but was not due to a lack of knowledge on how to do it. Additionally, after the system testing in sprint 2 (prior to sprint 3) some black box user testing was carried out to get some feedback on the usability of the GUI. This was done using 3 different users with varying computer competency levels. While 1 of the test users understands the python programming language, they were asked not to delve into the code to look for mistakes. Throughout the testing process everything was documented in an agile test plan spreadsheet where tests and results were logged along with the solutions to the bugs. A section of this test plan can be seen in figure 20.

Sprint 1 System Testing

One of the bugs found during the system testing after sprint 1 was that when the player tried to load a game without first having a saved game, 2 inactive buttons appeared on the screen, which may have frustrated the user if they tried to use them. The problem is shown below in figures 12 and 13.


```
except FileNotFoundError:
    label["text"] = "No save game found. Please start a new game"
    btn_reset(4)
```

Figure 12. Sprint 1 Bug (Code)

This was occurring as 4 was being used as the argument for the 'btn_reset()' function instead of 2. The 'btn_reset()' function was created so each screen would have the correct number of buttons appear by using the pack_forget() method on all of the buttons followed by the pack() method on the number of buttons required. There was also a reconfiguration of button width and image content within the btn_reset() function but this will be explained later in the report.

```
except FileNotFoundError:
    label["text"] = "No save game found. Please start a new game"
    btn_reset(2)
```

Figure 13. Sprint 1 Fix (Code)

Sprint 2 System Testing

One of the bugs found during the testing after sprint 2 was the back button within the adventuring sequence didn't actually do anything. This meant that if you decided to adventure but didn't have enough money, or had already discovered everywhere, there was no way of returning to the main menu unless you were to quit and load again. The fix to this bug was quite simple as all that had been forgotten was to assign a command to the button and not just text.

```
def adventure(my_character = my_character, Location = Location, location_list = location_list, pop_up = pop_up, root = root, btn_list = btn_list):
    pop_up["text"] = f"There are {Location.undiscovered_locations} undiscovered locations remaining."
    label["text"] = "Where would you like to explore?"
    btn_reset(10)
    btn_list[0]["text"] = "The Jungle"
    btn_list[0]["command"] = lambda:search(location_list[0])
    btn_list[1]["text"] = "The Desert"
    btn_list[1]["command"] = lambda:search(location_list[1])
    btn_list[2]["text"] = "The Swamp"
    btn_list[2]["command"] = lambda:search(location_list[2])
    btn_list[3]["text"] = "The Abandoned City"
    btn_list[3]["command"] = lambda:search(location_list[3])
    btn_list[4]["text"] = "The Lost Temple"
    btn_list[4]["command"] = lambda:search(location_list[4])
    btn_list[5]["text"] = "The Bull Ring"
    btn_list[5]["command"] = lambda:search(location_list[5])
    btn_list[6]["text"] = "The Great Plains"
    btn_list[6]["command"] = lambda:search(location_list[6])
    btn_list[7]["text"] = "The Mountain"
    btn_list[7]["command"] = lambda:search(location_list[7])
    btn_list[8]["text"] = "The Volcano"
    btn_list[8]["command"] = lambda:search(location_list[8])
    btn_list[9]["text"] = "Return to Battle Hub"
    btn_list[9]["command"] = lambda:load()
```

Figure 14. Sprint 2 Bug Fix (Code)

Sprint 2 User Testing

Feedback from the user testing after sprint 2 included:

- Text is small and difficult to read at times.
    - Solved by using the 'bold' argument for the font parameter (on all labels, buttons, combo and entry boxes).
- Riddles should be presented on a separated line to make them stand out.
    - Solved by introducing a couple of line breaks before the riddle.
- The undiscovered locations progress report does not change from 9.
    - Solved by adding a single line of code (Location.undiscovered_locations -= 1) as shown in figure 15.

```
def search(location, btn_list = btn_list, root = root, pop_up = pop_up, my_character = my_character):
    if location.discovered == False:
        entry = tk.Entry(root, width = 40)
        entry.pack()
        pop_up["text"] = f"You have decided to explore The {location.name} in search of another badge.\n{location.description}"
        label["text"] = f"You walk into a dark room and a door slams behind you. To revel the badge and escape you must solve this puzzle:\n{location.riddle}"
        Location.undiscovered_locations -= 1
        btn_reset(1)
        btn_list[0]["text"] = "Submit"
        btn_list[0]["command"] = lambda:get_entry(entry, location.answer, location)
    elif location.discovered == True:
        pop_up["text"] = f"You have already discovered The {location.name}.\nPlease choose another area to explore"
```

Figure 15. Undiscovered Location Fix (Code)

- Player would like to be able to chose which items they equip and unequip.
    - Solved by adding a completely new feature to the game where the player could check their stats and then chose what they wanted to equip from their inventory.
- The movement of buttons depending on text length in pop_up and label frustrated the player at times.
    - Solved by changing the layout so buttons were always on the right and text on the left instead of having text on top and buttons below. This meant that there was less mouse movement required and buttons were in a consistent location.

Sprint 3 Unit Testing

The unit testing was carried out on the Character class, with the first test class ensuring that initialisation occurred as expected and the second test class ensuring that the improve() method worked as expected. Each test class consisted of 4 tests, 3 of which passed first time and 1 failed. The TestInit class failed as the final test was set up incorrectly. Instead of trying to confirm that the characters money was 15, the test was trying to confirm that it was 10. The money was 10 in the early stages of the game but during testing it was discovered that if a player spent all 10 gold on training, they were unable to adventure and therefore unable to progress in the game which would be an easy mistake to make in the early stages of the game.

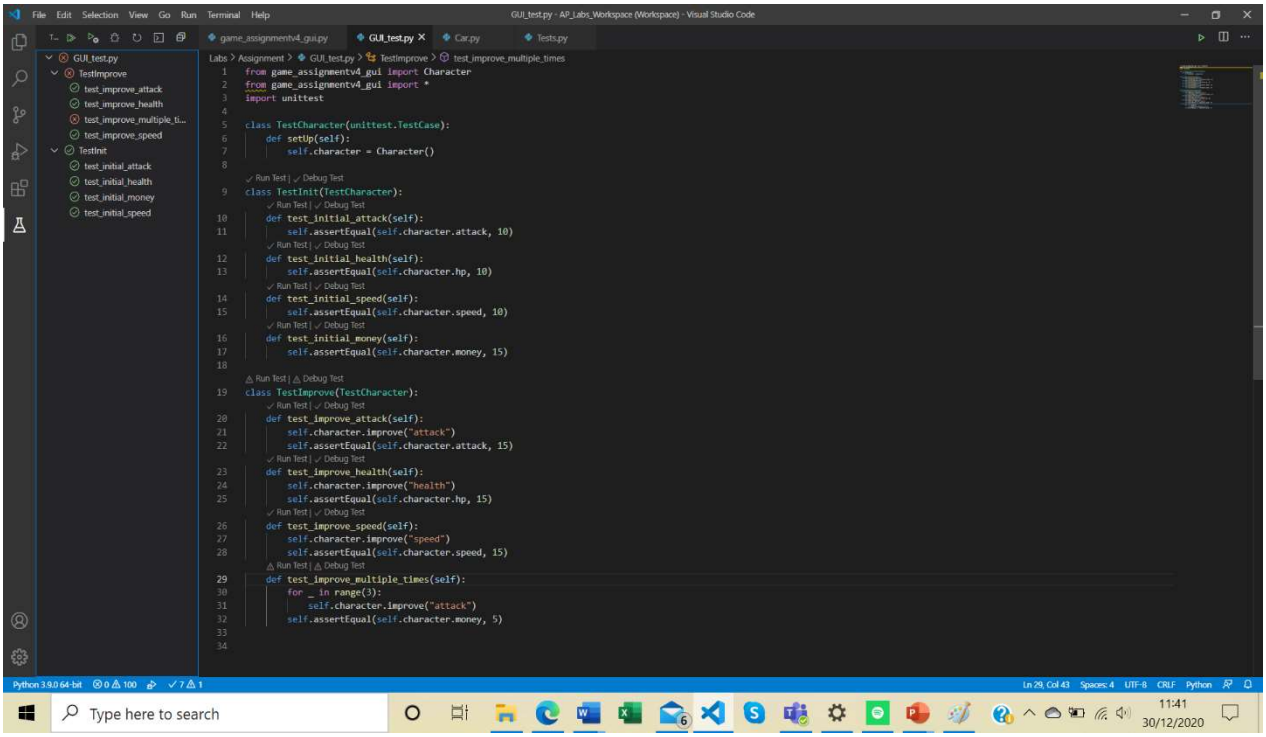

Figure 16. TestImprove Fail

```
def improve(self, stat):
    if self.money >= 10:
        self.money -= 10
        if stat == "attack":
            self.attack += 5
        if stat == "health":
            self.hp += 5
        if stat == "speed":
            self.speed += 5
```

Figure 17. TestImprove Fix (Code)

The TestImprove class failed as the 'if self.money >= 10:' line of code was not present within the games code, so the player was able to spend money that they didn't have which would have resulted in the same issue as before where the player was left unable to adventure.

Sprint 3 System Testing

One of the bugs found during the final systems test was when the player tried to sort their inventory alphabetically, either in the shop or when trying to change their equipment, the current selection always went to the second item in the list instead of the first.

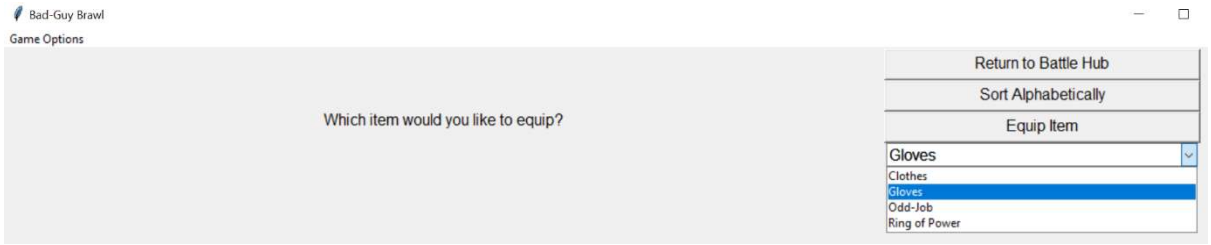

Figure 18. Alphabetically Sorting Problem

```
def alphabetise(listt):
    listt.sort()
    combobox["values"] = listt
    combobox.current(0)
```

Figure 19. Alphabetically Sorting Solution

The solution to this bug was simple and could be put down to inexperience with python. The problem was with the 'combobox.current(0)' which previously had 1 as the argument.

Created By		Tom McClelland						
Project Name		Application Programming Assignment						
Sr.No.	Module	Sub-module	Pre-Requisite	Steps to be followed	Expected Result	Actual Result	Comments	Status [Pass / Fail]
1	GUI	newgame()	newgame() function must be written.	1. Start Game 2. Click 'New Game'	GUI takes player to character creation.	GUI takes player to character creation.	As expected.	Pass
2	GUI	load()	load() function must be written and ensure there isn't a savegame.txt file in folder.	1. Start Game 2. Click 'Load Game'	GUI indicates that no save game is available and remains at the start menu.	GUI indicates that no save game is available and remains at the start menu. However, 2 extra, blank, non functional buttons appear on the screen.	btn_reset() function had incorrent btns_required parameter. Change btn_reset(4) to btn_reset(2).	Fail
3	GUI	load()	load() function must be written and ensure there isn't a savegame.txt file in folder.	1. Start Game 2. Click 'Load Game'	GUI indicates that no save game is available and remains at the start menu.	GUI indicates that no save game is available and remains at the start menu.	As expected.	Pass
4	GUI	load()	load() function must be written and ensure there is a savegame.txt file in folder.	1. Start Game 2. Click 'Load Game'	GUI takes player to main menu. All stats, inventory items and locations discovered are the same as when saved.	GUI takes player to main menu. Can check undiscovered locations and inventory items through adventuring and shopping respectively but no way of checking stats are the same.	Add a check_stats() method to the Charcacter class which can be accessed through shopping.	Fail
5	GUI	load()	load() function must be written and ensure there is a savegame.txt file in folder.	1. Start Game 2. Click 'Load Game'	GUI takes player to main menu. All stats, inventory items and locations discovered are the same as when saved.	GUI takes player to main menu. All stats, inventory items and locations discovered are the same as when saved.	As expected.	Pass

Figure 20. Agile Test Plan Spreadsheet

Design & Implementation

Implementing the game within a GUI required a majority of the code used in the first part of the assignment, to be completely restructured and changed. Only 10-15% of the original code remained. By changing the backend code, this allowed the front end to be kept simple. Initially, the game was designed to have the buttons below the text, but this frustrated users (as mentioned previously), so it was necessary to create a left and a right canvas on the root display. The left canvas contained the text and images, whilst the right canvas was for buttons only. All objects required for the GUI are shown below in figure 21. Buttons were packed and unpacked as required by each screen and this was controlled by the custom btn_reset() function in figure 22. Figures 24 and 25 illustrate how buttons and information were presented on the GUI.

```
72  #GUI ROOT OBJECTS
73  root = tk.Tk()
74  root.title("Bad-Guy Brawl")
75  root.geometry("1920x1080")
76  menubar = tk.Menu(root)
77  options = tk.Menu(menubar, tearoff = 0)
78  menubar.add_cascade(label = "Game Options", menu = options)
79  options.add_command(label = "Save", command = lambda:save("y"))
80  options.add_command(label = "Help", command = lambda:help())
81  options.add_command(label = "Quit", command = root.destroy)
82  root.config(menu = menubar)
83
84  if True:···
157
158 #GUI OBJECTS
159 canvas_left = tk.Canvas(root)
160 canvas_left.grid(row=0,column=1)
161 canvas_right = tk.Canvas(root)
162 canvas_right.grid(row=0,column=2)
163
164 btn_0 = tk.Button(canvas_right, text = "NEW GAME", command = lambda:newgame(), width = 35, font = "bold", anchor = "n")
165 btn_1 = tk.Button(canvas_right, text = "LOAD GAME", command = lambda:load(conf = "y"), width = 35, font = "bold", anchor = "n")
166 btn_2 = tk.Button(canvas_right, width = 35, font = "bold", anchor = "n")
167 btn_3 = tk.Button(canvas_right, width = 35, font = "bold", anchor = "n")
168 btn_4 = tk.Button(canvas_right, width = 35, font = "bold", anchor = "n")
169 btn_5 = tk.Button(canvas_right, width = 35, font = "bold", anchor = "n")
170 btn_6 = tk.Button(canvas_right, width = 35, font = "bold", anchor = "n")
171 btn_7 = tk.Button(canvas_right, width = 35, font = "bold", anchor = "n")
172 btn_8 = tk.Button(canvas_right, width = 35, font = "bold", anchor = "n")
173 btn_9 = tk.Button(canvas_right, width = 35, font = "bold", anchor = "n")
174
175 pop_up = tk.Label(canvas_left, anchor = "n", width = 100, font = "bold")
176 pop_up.pack()
177
178 label = tk.Label(canvas_left, anchor = "n", width = 100, font = "bold")
179 label.pack()
180
181 combobox = ttk.Combobox(canvas_right, width = 33, font = "bold")
182
183 entry = tk.Entry(canvas_right, width = 35, font = "bold")
184
185 btn_list = [btn_0, btn_1, btn_2, btn_3, btn_4, btn_5, btn_6, btn_7, btn_8, btn_9]
186 btn_0.pack()
187 btn_1.pack()
```

Figure 21. GUI Objects

```
195  def btn_reset(btns_required):
196      for btn in btn_list:
197          btn.pack()
198          btn["image"] = ""
199          btn["width"] = 35
200
201      if btns_required == 0:
202          for btn in btn_list:
203              btn.pack_forget()
204
205  >   if btns_required == 1: ···
215
216  >   elif btns_required == 2: ···
225
226  >   elif btns_required == 3: ···
234
235  >   elif btns_required == 4: ···
242
243  >   elif btns_required == 5: ···
249
250  >   elif btns_required == 6: ···
255
256
257      elif btns_required == 7:
258          btn_list[7].pack_forget()
259          btn_list[8].pack_forget()
260          btn_list[9].pack_forget()
261
262
263      elif btns_required == 8:
264          btn_list[8].pack_forget()
265          btn_list[9].pack_forget()
266
267
268      elif btns_required == 9:
269          btn_list[9].pack_forget()
```

Figure 22. btn_reset() function

It was necessary to reconfigure the image and width of the button, due to the introduction of images to the button during the battle sequence. These images were added to allow users to use recognition instead of having to recall. This btn_reset() function was used at the beginning of every other function that was created. This meant that each time a function was called by pressing a button, the GUI would have the correct number of buttons on the canvas, therefore it was just a matter of reconfiguring the command and text to fit the function. As well as reconfiguring the button text and commands, it was necessary to reconfigure the text of the 2 labels on the left canvas. One label was assigned the name pop_up to avoid confusion, even though it didn't technically 'pop up'.

A good example of this style being implemented is found in the shop() function, which can be seen in figure 23. Line 517 was found to be necessary so the combobox did not remain in place, should the user decide to go back to the shop after changing their loadout or selling items. Line 518 and 520 reconfigure the text within both of the labels and the btn_reset() function ensures there are only 4 buttons on the screen. Line 521 to 528 show how these 4 remaining buttons were reconfigured.

```
516  def shop():
517      combobox.pack_forget()
518      pop_up["text"] = ""
519      btn_reset(4)
520      label["text"] = "Welcome to the shop. Would you like to buy, sell or go back to The Battle Hub?\n "
521      btn_list[0]["text"] = "Buy"
522      btn_list[0]["command"] = lambda:buy()
523      btn_list[1]["text"] = "Sell"
524      btn_list[1]["command"] = lambda:sell()
525      btn_list[2]["text"] = "Check Stats"
526      btn_list[2]["command"] = lambda:check_stats()
527      btn_list[3]["text"] = "Back"
528      btn_list[3]["command"] = lambda:load()
```

Figure 23. shop() function

This made it very easy to reconfigure the information and options on the screen each time a function was called by pressing a button, however is also the reason only 10-15% of the previous code from Assignment I could be used. Instead, code had to be developed that received its command from the buttons instead of from the terminal.

One of the most problematic functions was found to be load(). The first problem with this was an error found in the code for the terminal game, meaning if the game was saved after defeating Magneto, then the game crashed. This was due to Magneto being equipped with an item that was prefixed with the name 'Magneto'. When the load() function was called and the .txt file was read, the line containing 'Magneto's Helmet' was confused by the system as a character instead of an item. This problem was solved by renaming 'Magneto's Helmet' as 'Metal Helmet'. A second problem was that sometimes when the load function was called, the user would require confirmation that the game has loaded but on other occasions this confirmation wouldn't be required. This problem was solved by introducing a parameter called 'conf' which would require an argument of 'y' if the loading confirmation was required, and was defaulted to 'n' if not required. Figure 25 shows how the confirmation pops up when the load function is called at game start up, with figure 24 showing how the confirmation does not pop up after adventuring.

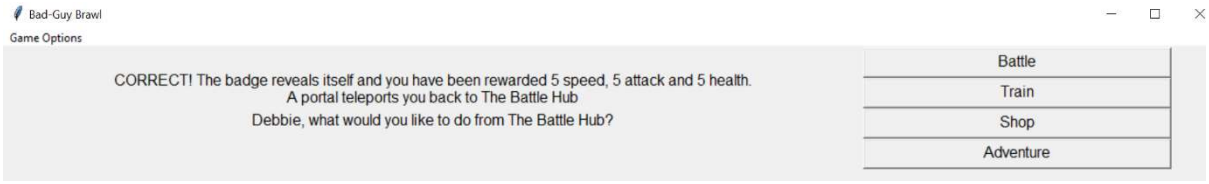

Figure 24. Loading Confirmation

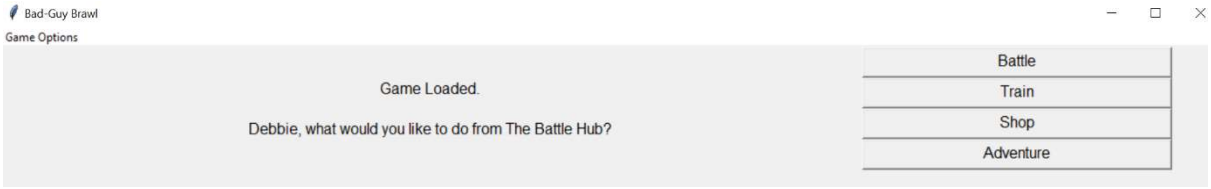

Figure 25. Loading after Adventuring

Implementing some sort of recognition rather than recall within the game was found to be very difficult but was eventually achieved by introducing a picture to the buttons during the battle sequence, as seen from figure 26.

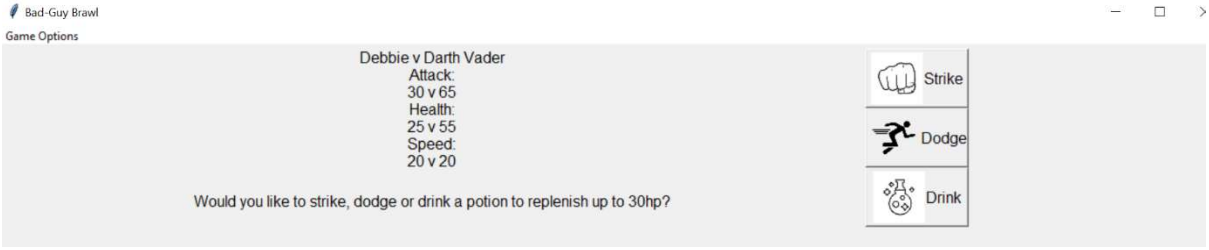

Figure 26. Button pictures during battle

```
87          drink_image = tk.PhotoImage(file="drink.png")
88          drink_image = drink_image.subsample(3, 3)
89          dodge_image = tk.PhotoImage(file="dodge.png")
90          dodge_image = dodge_image.subsample(3, 3)
91          strike_image = tk.PhotoImage(file="strike.png")
92          strike_image = strike_image.subsample(3, 3)
```

Figure 27. Image object creation

This was done by first creating an image object for each of the images, and then taking a subsample of this image as seen in figure 27. Once these objects were created, the buttons were then configured with both an image and text instead of just text. By doing this, the width of the button automatically adjusted, so it was necessary to reconfigure the button width when the fight() function was called. This width reconfiguration was then undone once the btn_reset() function was called which was previously shown in figure 22. Figure 28 shows how the code was laid out for the battle sequence.

```
477  def fight(foe): #contains strike and counter functions
478      btn_reset(3)
479      heal_limit = int(my_character.hp)
480      heal_limit_foe = int(foe.hp)
481      pop_up["text"] = f"{my_character.name} v {foe.name}\nAttack:\n{my_character.attack + my_character.weapon.attack} v {foe.attack +
482      btn_reset(3)
483      label["text"] = "Would you like to strike, dodge or drink a potion to replenish up to 30hp?\n "
484      btn_list[0]["image"] = strike_image
485      btn_list[0]["width"] = 120
486      btn_list[0]["compound"] = "left"
487      btn_list[0]["text"] = "  Strike"
488      btn_list[0]["command"] = lambda:strike(foe, heal_limit, heal_limit_foe)
489      btn_list[1]["image"] = dodge_image
490      btn_list[1]["width"] = 120
491      btn_list[1]["compound"] = "left"
492      btn_list[1]["text"] = "  Dodge"
493      btn_list[1]["command"] = lambda:dodge(foe, heal_limit, heal_limit_foe)
494      btn_list[2]["image"] = drink_image
495      btn_list[2]["width"] = 120
496      btn_list[2]["compound"] = "left"
497      btn_list[2]["text"] = "  Drink"
498      btn_list[2]["command"] = lambda:drink(heal_limit)
499
500  def battle():
501      battle_access = my_character.badge_count + Character.foes_remaining
502      btn_reset(0)
503      if battle_access >= 8:
504          for foe in foe_list:
505              if foe.hp > 0:
506                  fight(foe)
507                  break
508              elif Character.foes_remaining == 0:
509                  pop_up["text"] = "There is no one left to fight!! You are the Bad-Guy Brawl CHAMPION!!!."
510                  load(conf = "x")
511      else:
512          pop_up["text"] = "You must collect more badges through adventuring to reach your next foe."
513          load(conf = "x")
```

Figure 28. Battle Sequence

Limitations and Future Enhancements

The game was partially inspired by the Pokémon franchise where players earn badges through defeating gym leaders, which in turn allows them to fight the next gym leader. In Bad-Guy Brawl, these badges were earned through adventuring to different locations and completing riddles. In future game versions, this adventuring sequence could be improved so that the player must move a graphical character through a map to find the locations instead of simply fast travelling to them. Another Pokémon inspired enhancement could be the introduction of character types (fire, water, wind etc) and different moves that you can train your character to use via the training function. When starting a new game, the only choice that the player gets is what to call their character. In future versions the player could choose what type of character they want to play and perhaps how their stat points are distributed. These stats could be used in conjunction with the imported random library, in a similar fashion to how it currently is, but this would require huge amounts of effort from statisticians to ensure that whatever random number was generated did not make the game too easy or too difficult. While the random library was used in ‘Bad-Guy’ Brawl to add an element of chance to the battle sequence, there is still a lot of potential to improve its functionality by sitting down and crunching the numbers.

The saving/loading feature of the game was one of the most challenging features to develop yet the most rewarding once completed. One of the factors that contributed to it being a feasible save/load system was the small and simple scale of the game which may not work should the game be massively scaled up and if it included more variables. Therefore, it would need to be simplified in future versions.

Another improvement that should be considered for future versions is a graphical character that appears during adventuring, training and battles. This would not change the functionality of the game, but it would enhance UX.

None of the suggested improvements were included in the game due to time constraints and programming ability at the time of development.

Conclusion

Throughout the development of the game, there was a continual improvement of programming knowledge. Knowledge of theory and terminologies were gained through lectures and 'googling', while practical experience was gained by completing all the weekly labs (despite missing the odd QR code) and building the game. Work on the terminal game began within the same week as the assignment was released which was a mistake in hindsight, although, a lot of lessons were learnt in the first few weeks. This over-enthusiastic approach resulted in hours being spent on game features that could be done in a matter of minutes using techniques that were taught during lectures and labs in the following weeks. An example of this was the creation of multiple characters in the first week using massive dictionaries to store all the character information instead of just creating the 'Character' class. The reason that the game development was started almost immediately was due to the time constraints on its development and the fear that spending too much time on planning could result in not enough time being left to implement the design. Learning from mistakes is an important part of being a human-being so it is likely that in any future software development projects, much more time will be spent on planning, especially now that knowledge of agile methodologies and the software development life-cycle has been gained.

Something that would need to be improved are the front-end aesthetics which can be found lacking in places. This was due to inexperience and a lack of confidence with the tkinter library. Before embarking on future projects, gaining more experience and confidence with tkinter is essential. While the front end was slightly underwhelming, some of the back-end solutions that were developed brought a real sense of achievement and resulted in a growing confidence with overall programming ability (particularly the save/load solution). Rarely were any of these solutions conceived instantly which is something that is important to keep in mind when developing software. It is important not to be afraid of walking away from the computer screen whether it is for an hour or an entire day. Many problems were solved at the second or third attempt due to the different perspective that can be seen after taking a break.

Finding and fixing bugs was a constant battle throughout the game development. This may be slightly less of a battle in future developments if the 10 best practices for bug reduction are followed. While 5 of these were deemed as common sense and followed without knowing about them, there were 5 best practices that weren't implemented. They are as follows:

- Create testable code
    - This wasn't done as the testing process wasn't learnt until the end of the project.
- Don't use comments as a band-aid
    - Comments were used for anything and everything but for no particular reason.
- Take your time

    - Deadlines meant that time was of the essence and little time could be spent on the initial planning process.
- Implement coding standards
    - Some coding standards were not followed due to lack of knowledge, but as experience is gained, this knowledge should build.
- Use rubber ducky testing
    - No rubber ducky could be procured.

Lastly, this was an extremely challenging, yet rewarding experience. Any future developments will benefit massively from what has been learnt from this assignment and from the module as a whole although there is likely still many lessons to learn.

References

Butt, M. A. and Li, S (2015) 'UML-based requirement modeling of Web online synchronous collaborative public participatory GIS', *Applied Geomatics,* 7(1), pp. 203-242.

Darrin, M. A. G. and Devereux, W. S. "The Agile Manifesto, design thinking and systems engineering," *2017 Annual IEEE International Systems Conference (SysCon)*, Montreal, QC, 2017, pp. 1-5, doi: 10.1109/SYSCON.2017.7934765.

Kuhn, J. (2009) 'Decrypting the MoSCoW Analysis ', *,* 5(44), pp. 1-2 [Online]. Available at: *http://www.itsmsolutions.com/newsletters/DITYvol5iss44.pdf* (Accessed: 29th December 2020).

McManus, J. (2011) *Managing Stakeholders in Software Development Projects*, 2nd edn., New York: Routledge.

Mortensen, T. E. (2006) 'WoW is the New MUD: Social Gaming from Text to Video', *Games and Culture,* 1(4), pp. 397-413.

Pallavi, P. and Tahiliani, S. (2015) 'AgileUAT: A Framework for User Acceptance Testing based on User Stories and Acceptance Criteria', *International Journal of Computer Applications,* 120(10), pp. 16-21.

Sharma, S., Sarkar, D. and Gupta, D. (2012) 'Agile Processes and Methodologies: A Conceptual Study ', *International Journal on Computer Science and Engineering (IJCSE),* 4(5), pp. 892-898.