**TECHNICAL DESIGN DOCUMENT:**

**BACK TO SCHOOL 2: ELECTRIC BOOGALOO!**

****

****

THIS DOCUMENT DETAILS THE DESIGN AND IMPLEMENTATION OF THE GUI FOR THE QUIZ-BASED ADVENTURE GAME **BACK TO SCHOOL 2: ELECTRIC BOOGALOO!**

**AUTHOR/Lead Developer: NATHAN PAGE**

**DATE: 16/01/2021**

usv20umu@uea.ac.uk

+44 7513 310 398

University of East Anglia, Norwich

www.back2school-eb.com

**Introduction:**

The following report details the design, development, testing and evaluation of a graphical user-interface (GUI) system for a quiz-based adventure game. The program was developed in Python using the Tkinter package for GUI development. The game is designed to work on all systems with Python 3 installed. Several additional modules are used but all are found in Python’s standard library. These modules are utilised to maximise efficiency and reliability through the re-use of previously written and thoroughly tested code.

**What is the game/GUI?**

The user creates a custom character and must navigate around the game world and win subject-specific quizzes. When the user wins a quiz, they pick up a totem item. The user must collect all totems to win the game. If a question is answered incorrectly, the player will be ‘attacked’ by the quizmaster and lose health. If health drops to 0, the player dies, and the game is over. The player can pick up consumables and use these to increase health along the way.

**What is in the report?**

Specifically, the report contains analysis of our chosen agile methodology (scrum), requirements, UML (Unified Modelling Language) diagrams, evidence of testing, evaluation and analysis of the implementation and concludes with an honest self-reflection of the design and development process.

**Agile methodology (Scrum):**

All agile methodologies share several characteristics:

* Focus on iterative processes
* Able to respond to change
* Prioritising key functionality
* Constant testing and documentation
* Emphasis on communication among team members and between development team and customer.

The agile methodology chosen for this project was **Scrum**. Research was conducted into agile game development methodologies and scrum was chosen because it is a commonly used approach with numerous resources and examples available (pertaining specifically to game development projects).

The Scrum methodology involves creating a product backlog which itemizes and prioritizes the tasks to be completed - and completes these tasks in sprint cycles. Ordinarily, a scrum is conducted whereby progress is shared among the group and tasks are designated.

As this project is individual, the scrum development methodology needs to be tailored to enable effective and efficient individual work. The scrum process for this project will consist of a daily, individual progress report (instead of the usual scrum meeting) whereby progress is tracked, and the product backlog is updated (primarily through Trello and a product-backlog excel spreadsheet). Following each ‘scrum’, the work for the day (or another slightly longer time frame) is specified and work towards this objective(s) begins.

Step 1 is to identify the stakeholders in our project. From there, we can create a product backlog by producing user stories for each stakeholder.

**Stakeholders:**

A stakeholder is any party with an interest in the success of the project. For this project, we have identified three primary stakeholders to consider:

--------------------------------------------------------------------------------------------------------------------------------------

**Stakeholder 1: Future developer**

Internal stakeholders for this project are the development team (both current and future). In this instance, no other developers are involved but it is vital to consider the possibility of developers being involved in expanding, updating, fixing or modifying the product at some point in the future. Documentation is key here so that future developers can understand the codebase and development process and modify/update as necessary.

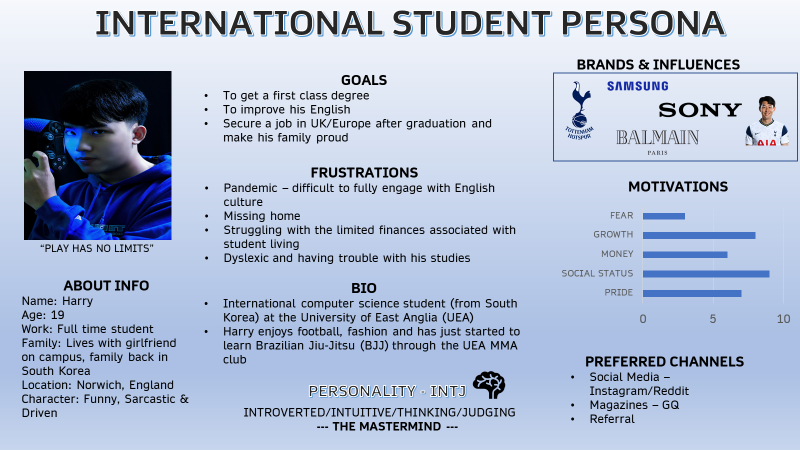
The **future developer** requires:

* thorough documentation in order to comprehend the purpose, scope and mechanics of the GUI system being produced.
* Easy to understand (and ideally modular) code.

To satisfy these requirements, this design document is created which details the development process. In addition, the code itself will contain plentiful yet concise comments to ensure the game mechanics are understandable. Functions and classes are paramount and are used where possible to separate functionality and make troubleshooting easier.

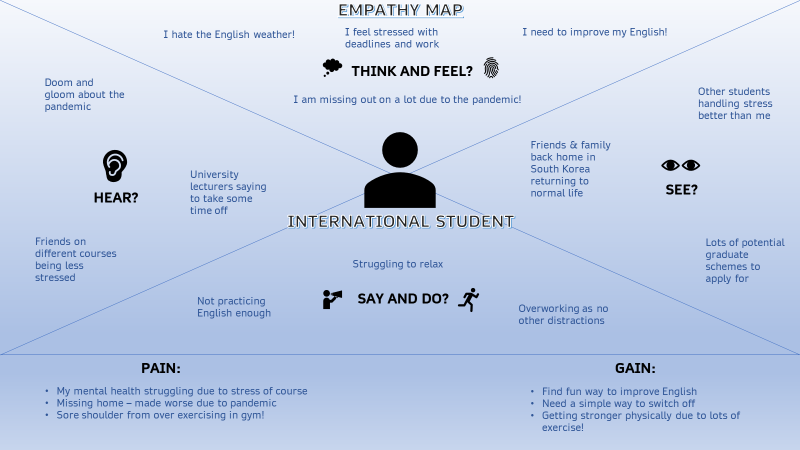
--------------------------------------------------------------------------------------------------------------------------------------

**Stakeholder 2: End user**

To ascertain the requirements of an **intended user**, we create a fictional persona. This allows us to determine our target audience and focus on producing a user experience that our target audience will enjoy. Our game is fun and educational and would suit someone looking to learn and relax. To this end, we create an international student persona (*Figure 1*). Harry is from South Korea and studying

*(Figure 1 – persona of international student gamer)*

computer science in Norwich. He is driven and career focused and would like to improve his English and succeed in his degree to help with his career prospects. It is important to note that Harry has Dyslexia. Our game should be usable and enjoyable for people of all abilities, and disabilities must be considered when designing our GUI.

Having created a persona, an empathy map (*Figure 2*) allows us to better understand Harrys perspective (what he sees, hears, thinks, feels, says, and does). This serves to create a game better suited to Harrys needs. We can note that Harry is feeling stressed and would like a way to take his mind of the stresses of study. To this end, perhaps a simple game which requires little concentration would be suitable.

*(Figure 2 – Empathy Map)*

--------------------------------------------------------------------------------------------------------------------------------------

**Stakeholder 3: Customer**

The project specification, received by the client, states several goals that needs to be realized in order to satisfy the **customer** requirements for the GUI:

1. The GUI must be able to relay and receive instructions to a user (as opposed to interaction via the command line).
2. Show current location (using text and images).
3. The user must be able to view their location history (and sort this data in either direction).
4. The user must be able to view character information (text and images).
5. The user must be able to view general game progress.
6. The user must be able to view an alphabetised inventory.

Taking these requirements into account, we can begin to consider and design some basic game functionality.

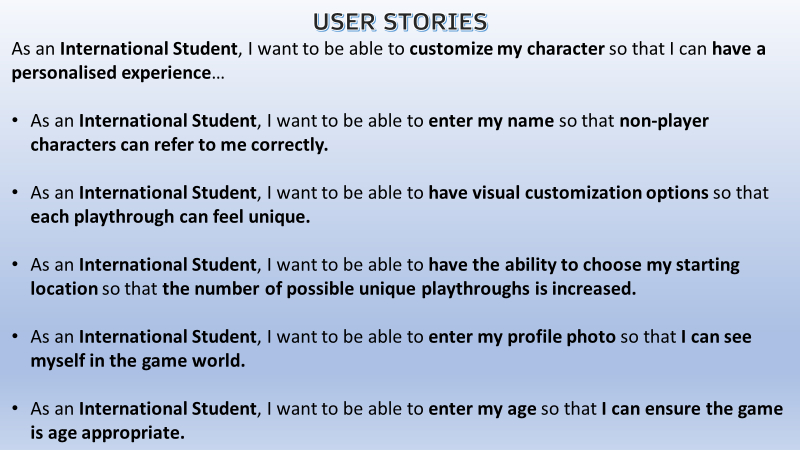
1. To relay information, text will be displayed using large Tkinter label widgets. Instructions will be received by the user using different input widgets based on what is most appropriate. For example, if only a few inputs are acceptable buttons and radio boxes can be used as appropriate. When a more open-ended question is asked of the user, entry widgets can be used to receive information.
2. To show player location, the area name will be displayed in addition to a photo which depicts said location. This will be implemented on the main screen (where navigation takes place).
3. The functionality to view location history will be implemented in a separate frame (accessed via the main screen). To change the order of the location history, a button will be used which alters directionality.

4&5) Character information will be accessible via separate information/stats screen which will also show game progress (number of totems collected).

1. An inventory screen will be accessible from the main menu which shows alphabetized consumables and totems.

--------------------------------------------------------------------------------------------------------------------------------------

**User Stories**

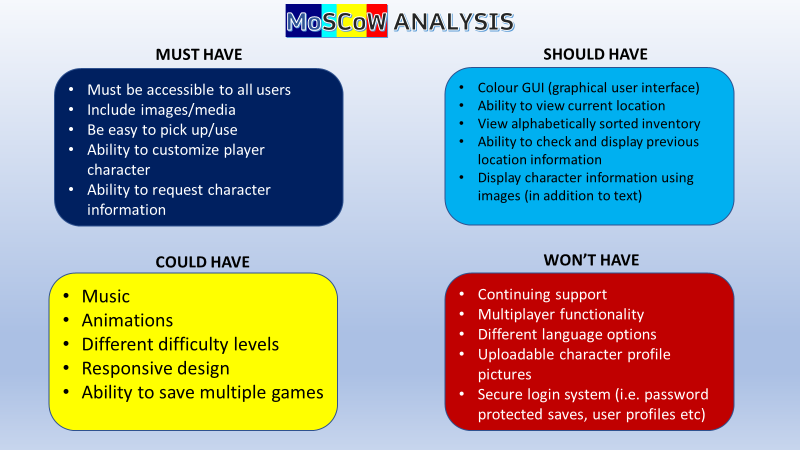
****The user story below (*Figure 3*) utilises our persona and empathy map to focus on the perspective of the international student user of the game. This user story described the process of character creation/customisation.

*(Figure 3 – User Stories)*

User stories are created for each stakeholder and when combined, create our product backlog from which we plan and implement our sprint cycles.

**MoSCoW Analysis**

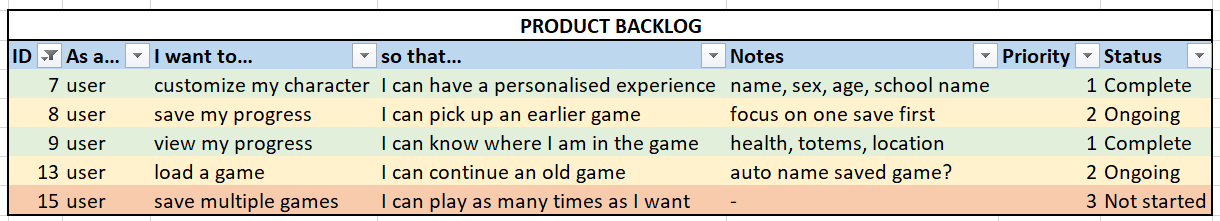
To prioritise our tasks, a MoSCoW analysis (*Figure 4*) is utilised. A MoSCoW analysis aims to itemise and prioritize features and should consider all stakeholders (in this case the development team, end-user and customer).

****

*(Figure 4 – MoSCoW Analysis)*

With priorities defined, our product backlog can be completed, and priorities assigned. Priority 1 is the highest priority i.e. the must-have features/functionality. Priority 2 refers to features the GUI should have. Priority 3 are features that could be implemented if time and resources allow.

“Won’t-have” features are not included in the product backlog as these will not be implemented under any circumstances.

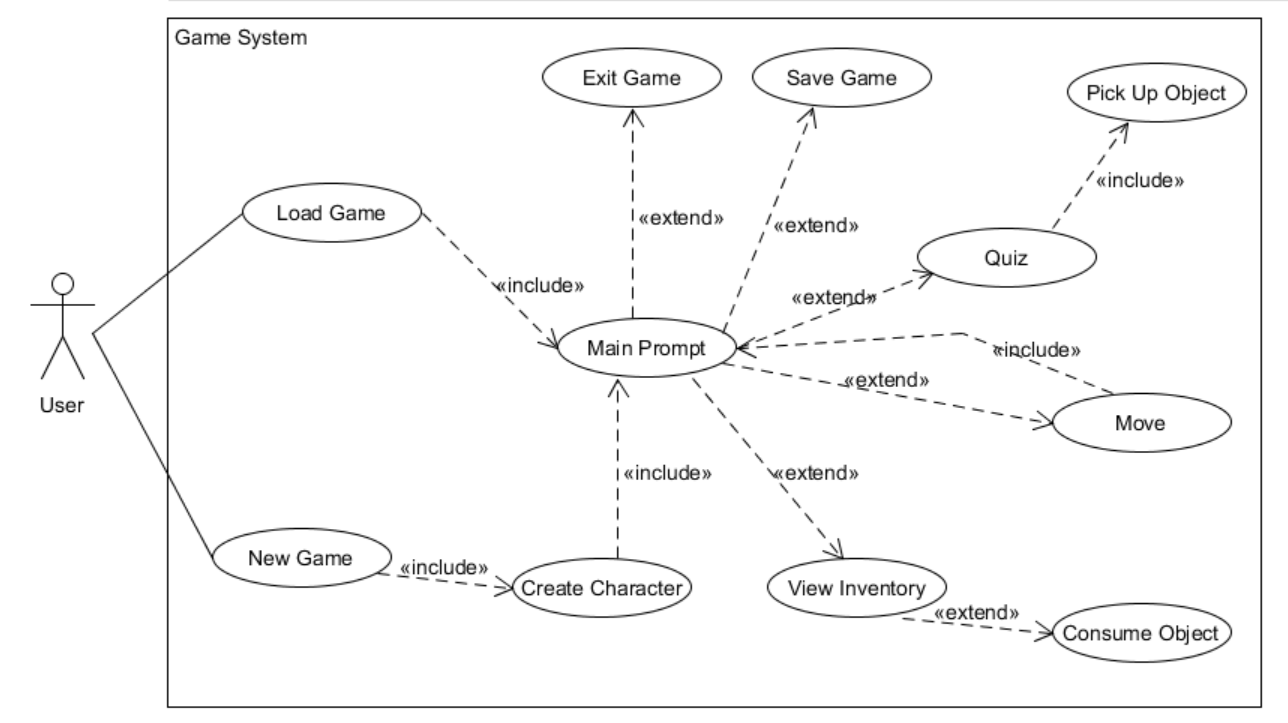
A snippet of the product backlog is shown below (*Figure 5*). Each primary item is broken down further (as shown above) and allow us to plan precisely what needs to be completed in order to satisfy the requirements of each stakeholder. It is crucial that communication is near constant with the stakeholders to ensure the project is moving in the correct direction. Additionally, requirements may change during the development process and the use of an agile methodology (and the iterative process it promotes) mean that changes can be made with minimal disruption.

*(Figure 5 – Snippet of Product Backlog)*

The product backlog is colour coded (by status) so should be intuitive to read and comprehend. Green rows are complete, yellow are ongoing and red rows have not yet started.

**UML (Unified Modelling Language) Diagrams**

With requirements defined and prioritised, we can begin to design the system. The use case diagram (*Figure 6*) is used to visualise the system and its components. Each component in this instance refers to a separate functionality.

**Use Case Diagram:**

(*Figure 6 – Use Case Diagram*)

Here, the user is our actor (the individual interacting with our game system). When a new game is started, the user creates a character and is sent to the main prompt screen which has a number of options which change based on the location of the player (this functionality is expanded upon further in the class diagram shown below).

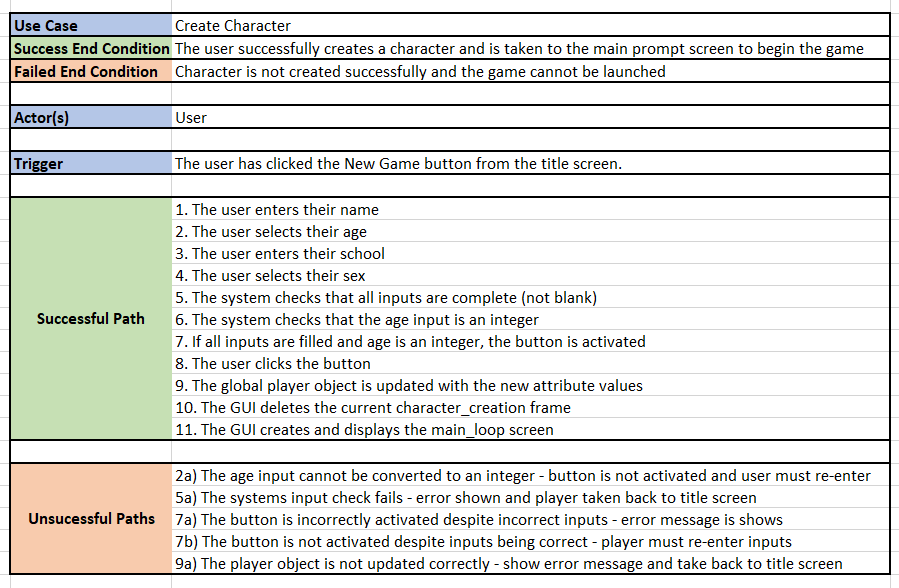
As the diagram shows, the user can quiz with a character (if one exists in their current location), move, view their inventory, save game and exit game. To keep the diagram legible and simplistic, several functions are not depicted – the user can also pick up consumable objects, view their route and check their status.

When a game is loaded, the user is put back into the main loop at their last location (skipping the character creation screen) and the game continues as they left off.

The <<include>> notation is used to denote functionality that is essential, <<extend>> denotes functionality that is optional.

For example, from the main prompt, the ‘move’ functionality is one of several options (<<extend>>). However, when the user selects to ‘move’, they are automatically taken back to the main prompt menu (<<include>>), albeit with different options (on where to move next, or who to quiz) as they are in a different location.

**Textual Use Case**

**The textual use case takes one component of the use case diagram above and breaks down its functionality into a series of individual steps. The character creation textual use case is shown below (*Figure 7*).

*(Figure 7 – Textual Use Case)*

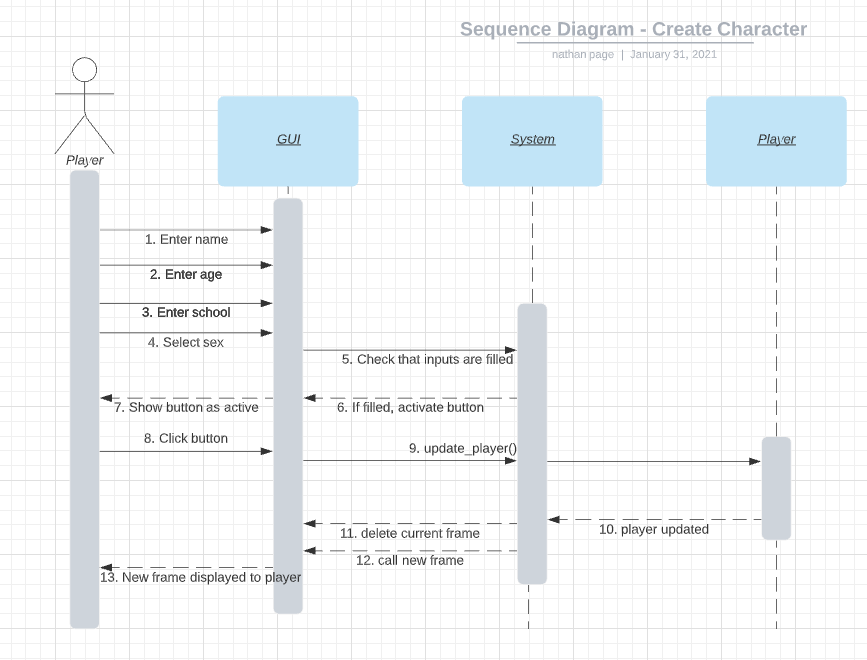
The character creation functionality can be broken down into 11 distinct steps shown above. At several of these steps, there is a possibility of an error occurring. These errors are accounted for and listed in the ‘unsuccessful paths’ section with their corresponding number.

The relevant actor here is the user who is aiming to customise and create a character with which to play the game. Several attributes can be customized here – the user must enter their name, their age (18-100), their school and their sex (male or female). To simplify this process, the user is provided with a set number of specific options where appropriate ( ‘male’ or ‘female’ radio buttons will be used to select sex, a spinbox will be used to display valid ages from which a user can spin to and select). This means that input validation is not necessary here as the user is only presented with 2 (both valid options). Input validation should be implemented on the open-ended inputs (name, school) to ensure only strings are entered. To ensure a valid option is selected and all inputs are complete, the system will check all 4 inputs each time a key is pressed and will only activate the ‘start game’ button when all inputs are complete and valid options are selected.

It is imperative we are aware of the unsuccessful paths a user can go down and account for these in our design. Where possible, try/except blocks are used to manage errors, display the error message in a user-friendly manner, prevent the program from crashing and provide a mechanism for the game to continue.

**Sequence Diagram**

Following on from the textual use case, we can better visualize the GUI mechanics by creating a sequence diagram. Figure 8 depicts the sequence diagram for the Character Creation use case described textually above.



(*Figure 8 – Sequence diagram for Character Creation*)

The player enters their name, age, school and sex using the various Tkinter input components (entry, spinbox, entry and radiobuttons). The system checks that all inputs are filled in, and when they are, the button .state attribute is switched from ‘disabled’ to ‘normal’.

The player can now click the ‘new game’ button and the global player object is updated with the new inputs entered by the user. Next, the current frame is deleted and the next screen is called – effectively moving the user to the next screen with customer character attributes.

The alternative paths here are left out for simplicity but an example of how alternative paths are incorporated into sequence diagrams is displayed below (Figure 9).

--------------------------------------------------------------------------------------------------------------------------------------

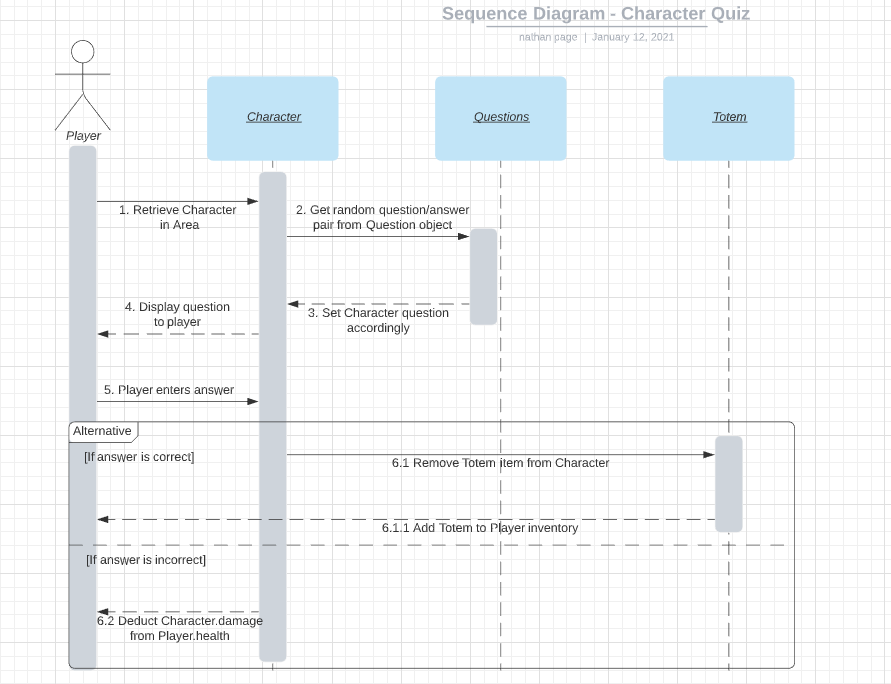
Figure 9 is an example of a sequence diagram for the quiz component of the GUI game. This component contains a couple of alternative paths.

Once the user has submitted their answer, there are 2 possible scenarios:

1. If the submitted answer is correct (matches the answer in the appropriate question dictionary entry), the Character drops their Totem item and this Totem is appended to the Player.totems list.
2. If the submitted answer is incorrect (does not match an answer in the appropriate question dictionary entry), the Character “strikes” the Player and the Character.damage amount (randomly assigned at the beginning of each game) is deducted from Player.health.

*Note – there is a health check here which ensures Player.health is above 0. If Player.health drops to 0 or below, the you\_died() screen will show and the game is over. This functionality is removed from the below sequence diagram for simplicity.*

To clarify, Character here refers to the quizmaster (a non-player Character object, as opposed to the **Player**s character)

****

*(Figure 9 – Sequence diagram for Quiz component)*

**Class Diagram**

The GUI section of the game itself does not contain any classes. However, the gameplay mechanics (derived from the original command line game) does utilise classes which interact to create the game world. These are defined below (Figure 10):

**Diagram

Description automatically generated**The class system described here will be used to create the navigable map and game world.

An instance of the **Player** class is used to store the details of the user and will be customised in the character creation screen. Name, age, school and sex are chosen by the user. Other attributes are set at initialisation and will change as the game plays out.

The **Area** class is used to create several distinct area instances which serve as ‘rooms’. The map is created by linking these rooms together by adding a ‘connections’ list – which contains other area instances. Navigation will work by iterating through player.location.connections and producing buttons for each connection. When a specific button is clicked, the player.location is set to this new area and the screen is ‘refreshed’. The new locations connections are listed, and the cycle perpetuates.

*(Figure 10 – Class Diagram)*

Within a given area, there are 0 or 1 **Character** objects. If an area contains a character (as defined in the Area.characters list), a button will be created to link to the quiz screen for this specific character. Each character instance has its own name, questions dictionary (with questions and answers as key value pairs), weapon and damage. Additionally, a character has an items list which contains **Totem** objects. The totems are dropped by the character if they are defeated in a quiz and appended to the Player.totems list. An area can also contain **Consumable** objects which can be picked up and used by the player to increase their health. Consumables contain a value attribute which dictates how much health is added.

**Testing**

Testing was conducted throughout the design and implementation phases of development and is broken down into 2 categories – validation and verification.

**Validation**

Validation checks that the software developed matches the requirements/specification set. Validation testing was undertaken three times. First, when the Use Case diagram was created to ensure all of the necessary game mechanics are included in the system. Second, when the initial GUI designs were created to ensure these designs allowed for our required functionality and met usability standards. Third, at completion to ensure our completed product met specifications.

At each point, we checked that the designs could provide the necessary functionality and that the GUI was accessible, usable, and intuitive. **Nielsen’s usability principles** were integral to our design and implemented to ensure maximum usability. Specifically:

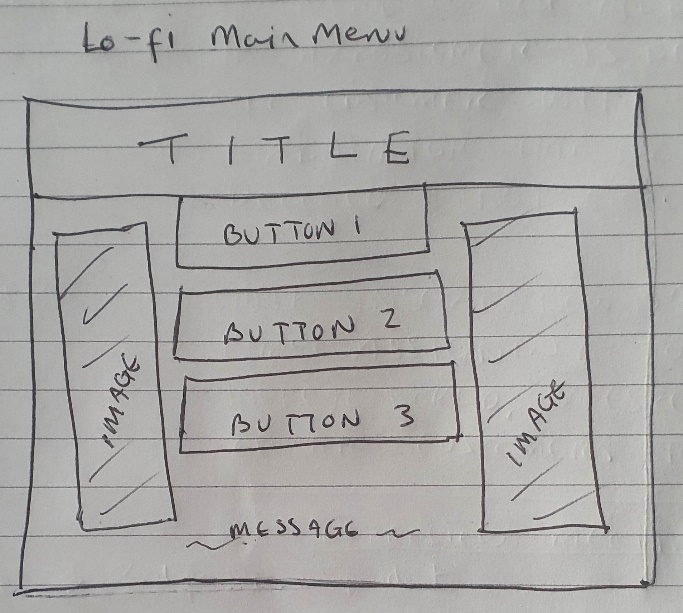
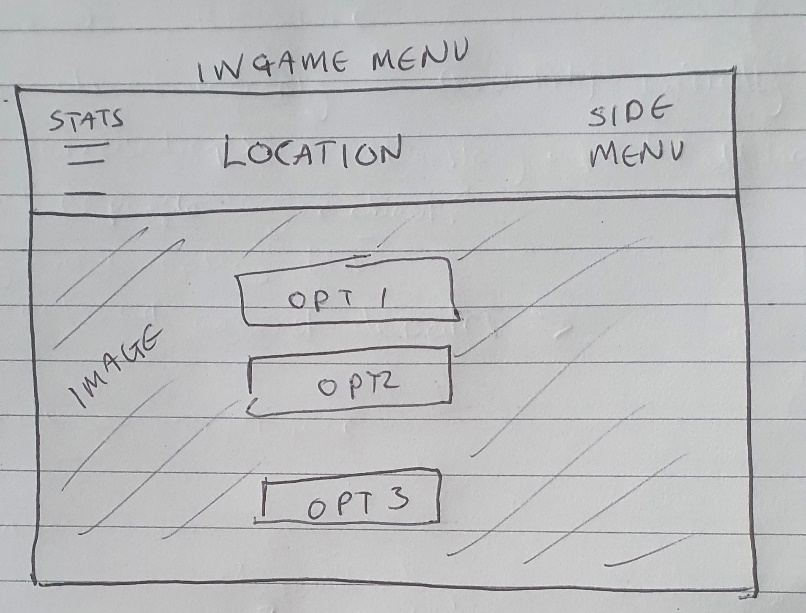
1. If there is significant loading times, we will let the user know that the game is loading (as opposed to crashed).
2. The game will feel intuitive and natural and fit the context.
3. When a user chooses to save or exit, we will double check that the user wished to confirm this action.
4. Layouts will remain consistent.
5. Errors are prevented at source – for example, we will ensure a user must enter a name while creating their character.
6. The GUI will be efficient and uncluttered.
7. If an error does occur, the user will be alerted.
8. An information/help page will be available when the game begins to ensure the user understands how to play the game.

When considering usability, it is also vital that the application is usable for the greatest number of individuals. When designing our GUI, the following factors are considered:

1. Demographics/culture
2. Age
3. Personality types
4. Disabilities

To ensure our application is usable for elderly people, visually impaired people, and individuals with dyslexia, we use large fonts and high contrast foreground/background differentiation. Additionally, we accommodate colour-blind users with careful consideration of colour schemes. Layouts will be uncluttered, simplistic and hopefully intuitive.

**Low-Fi prototypes:**



*(Figure 12 – Lo-fi for main prompt screen) menu)*

*(Figure 11 – Lo-fi for main menu) menu)*

Initial lo-fi design drafts are created to visualise how each screen will look. The lo-fi versions of the main menu (*Figure 11*) and main prompt screen (*Figure 12*) are shown above. Lo-fo designs are intentionally rough to encourage honest feedback and constructive criticism.

Upon completion of the lo-fi’s, user testing is conducted, and feedback is collected from 3 individuals. These individuals are chosen from different groups as defined in the usability section – with varying demographics, age, personality type and different abilities/disabilities.

**Tester 1 Richard – British male, Doctor, aged 60**

Feedback for main menu – “*space the buttons out further, one photo better than 2*.”

Feedback for in-game menu – “*too much wasted space*.”

----------------------------------------

**Tester 2 Eileen – British female, Retiree, age 86, disability – very poor eyesight**

Feedback for main menu – “*make buttons bigger (and font), can’t read the message at bottom*.”

Feedback for in-game menu – “*too plain, stats area too small to read*.”

----------------------------------------

**Tester 3 Desiree – American female, English Teacher, age 23, disability - dyslexic**

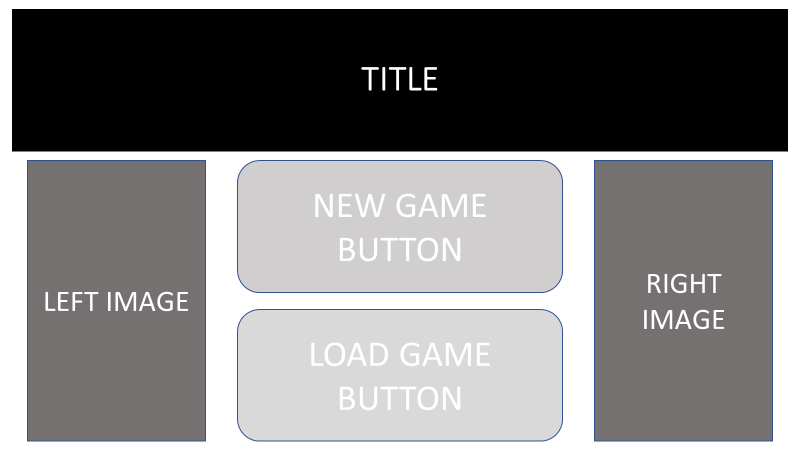
Feedback for main menu – “*make sure proportions are equal, bring button 1 down (add a gap from the title section).”*

Feedback for in-game menu – “*background image will make things messy and possibly hard to read, lots of wasted space in main area, make bigger buttons or make better use of space*.”

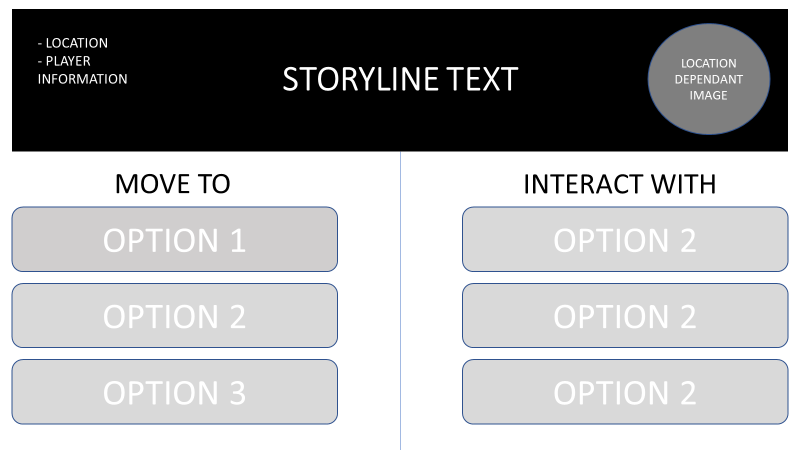
**Medium-Fi prototypes:**

The medium-fi prototypes implement general feedback from user testing to declutter the screen and increase readability by making buttons and fonts large.

*It is important to note that the medium-fi versions are in grey-scale and font/background contrast will be improved in the final versions.*

The title menu (*Figure 13*) is altered to introduce a small gap between the title bar and the first button, and the screen is made symmetrical to improve aesthetics. In addition, the smaller message at the bottom is removed, as is button 3. The changes are made to facilitate the larger font/buttons which improve readability for older users with poor eyesight and reduce the number of options to make the game play process more intuitive.

*(Figure 13 – med-fi prototype of main menu)*



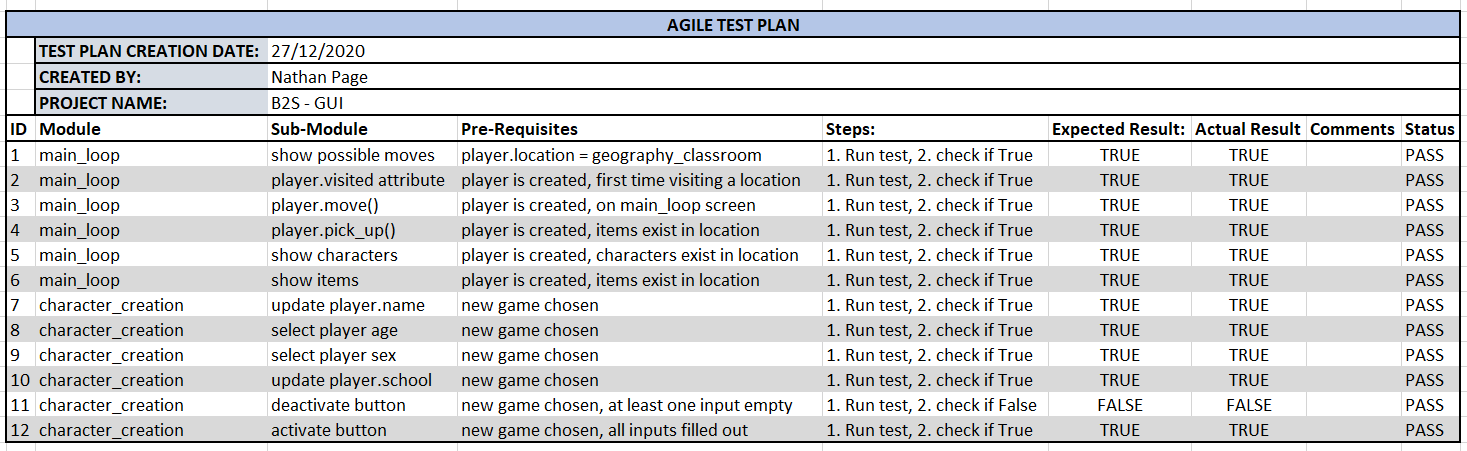
*(Figure 14 – med-fi prototype of main prompt)*

In the updated medium-fi main prompt screen (*Figure 14*), player status (health etc) is always shown but is kept on the periphery to focus the user on their options. To make better use of the space, the buttons are divided into 2 columns (with differing associated actions). To improve readability, the background image is removed, and a smaller image is placed into the top right-hand side of the title bar.

**Verification**

Verification testing is done throughout and aims to verify that the system is working as designed. First, the original game code was tested (classes, objects). Once it was established that all classes and objects are working as desired, each screen was created and tested separately. This testing was initially informal and testing protocols were uncovered by attempting to break the code. For example, we would enter String variable for ‘age’ in the character creation screen – determine that this causes issues – and write a test to ensure that these issues are remedied.

**Test Plan**

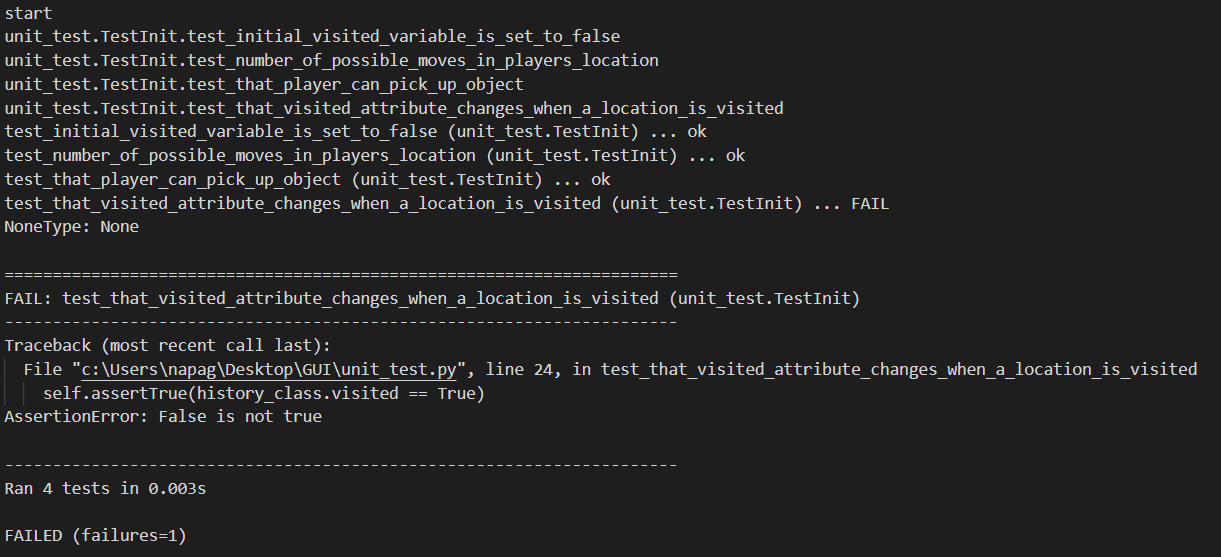
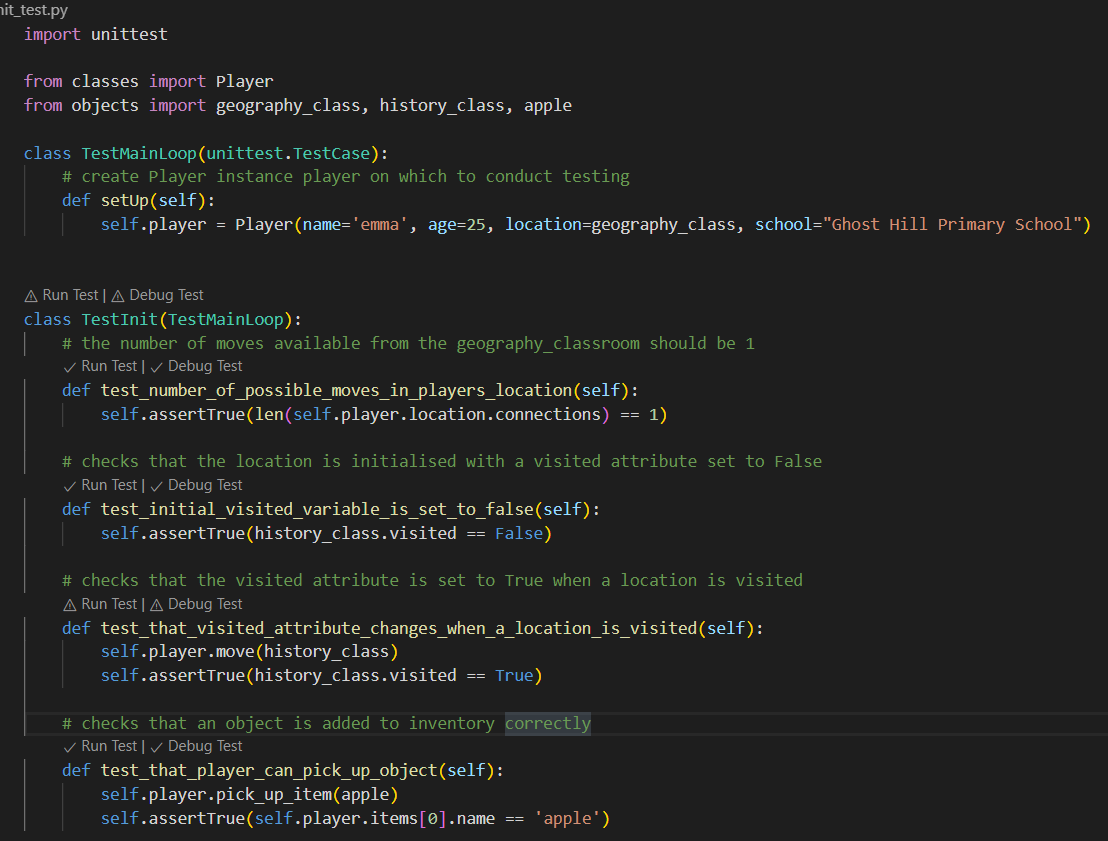
****A throughout testing plan (*Figure* 15) was created and implemented to ensure testing standards were maintained throughout the development phase. Below shows some of the testing procedures and results for the main\_loop and character\_creation functions.

*(Figure 15: Test Plan Document)*

Testing was broken down into modules or functions (eg. main\_loop) and further into sub-modules (eg. moving the player to another location within main\_loop). Expected results are compared to actual results and necessary adjustments to the code base are made when a test fails (as described below in *Fig. 15*).

Unit testing was used throughout development to ensure each part of a function or class work as desired. By testing each functionality thoroughly, we can continue to build our software and limit the potential for a critical error toward the end of the development lifecycle where an error is much more difficult and expensive to pinpoint and/or fix.

Several unit tests are shown here which determine that the main game loop menu mechanics are working as expected.

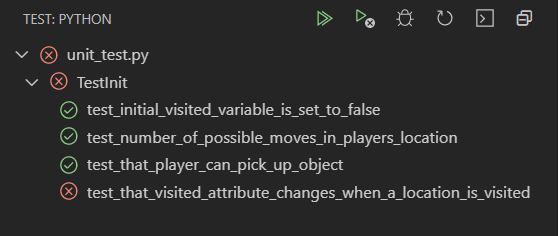
******Unit Testing**

Unit test 1 checks that the number of possible moves is correct from the players current location.

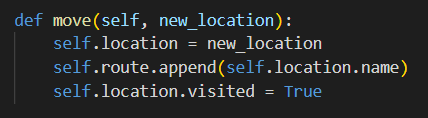
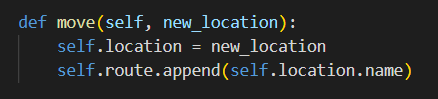
Unit test 2 checks that when an Area instance is initialized, the ‘visited’ attributed is set to False (as the player has not yet been there).

Unit test 3 checks that when a player has moved to a new location (using the Player method ‘move’, that the ‘visited’ attribute of the new location is now set to True.

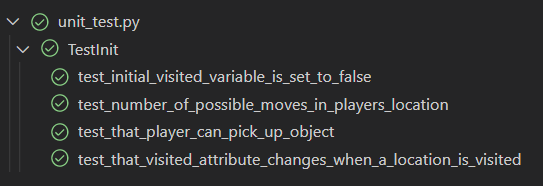
Unit test 4 checks that a player can pick up an object and add it to their inventory.

**Unit tests 1, 2 and 3 all behave as expected and these tests are passed. However, test 4 fails so we must troubleshoot and ascertain why the test has failed. In this instance, we can see that the AssertionError explains that the attribute value is False (when we want it to be true). When we check the Player.move() method, we note that there is no code to trigger the change of the ‘visited’ attribute.

*(Figure 15: Testing Code & Output)*

******

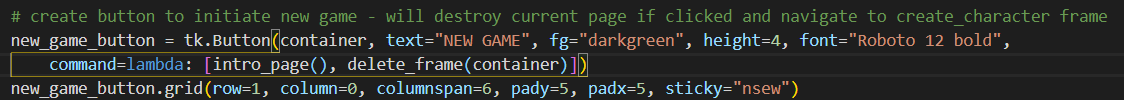
*(Figure 16: Code alterations implemented as a result of unit testing)*

****

We can see that because of the code alterations, our code is working as intended and all unit tests are now passing.

**Evaluation & Analysis**

**Implementation**

The GUI works by first creating a root window. Each ‘screen’ (title, create character, quiz etc) is a separate function that is added to the root window when it is called. Initially, the title\_page function is called – showing the main menu frame on the root window. When a user wishes to move to another screen, 2 custom lambda functions are attached to the command attribute of the relevant button – 1) destroys the current frame, 2) calls the function for the next ‘screen’. Figure 17 shows the button which moves from the title\_page to the intro\_page. First the current container (title\_page frame) is deleted, then the intro\_page to show the next page.

*(Figure 17 – example of a button showing the GUI mechanic)*

This mechanism works well for the dynamic elements of the game. For example, when the user wishes to move in the main\_loop (from one room to another), it is simple to implement. A third lambda function is added to a ‘move’ button which when clicked, changes the players location. The screen must be refreshed to show this new location so the main\_loop container is deleted and then the called once more – effectively refreshing the page.

Different Tkinter layout methods are used depending on what is to be shown. For example, where the number of items is unknown, .pack() is used in conjunction with for-loops to add a degree of flexibility to the layout. Where more precise placement is required, .place() is used. Where a complex consistent layout is needed, .grid() is used to arrange multiple components.

**Limitations/ Trade-offs**

Several limitations exist: this is primarily due to the time constraints involved with getting the project completed on time, with priority given to features deemed ‘must-haves’ in the MoSCoW analysis.

1. There is a slight delay between certain screens. I hypothesize that this is related to opening, loading, and displaying png image files.
2. Saved games cannot be easily deleted/overwritten by the user.
3. There is no user-friendly message to show when an object is picked up and/or consumed.
4. Menus are created dynamically and due to this, space is not always utilised in the most efficient manner.
5. Saved games cannot be named – they are given a default name based on the users’ name. This can lead to issues if the same player (or even 2 different players with the same name), save a game. In this instance, the new save will overwrite the old without any warning.
6. In the character creation section, theoretically a user can provide a nonsensical value for ‘name’ and/or ‘school’. For example, a user could provide an integer as their name. This won’t cause any errors but ideally input here would be checked to ensure a string is provided.
7. The buttons in the main\_loop, inventory and load\_game screens are created dynamically using for loops. If an excessive number of items or saved games are present, only the first few will be shown.
8. Currently, the answers are open-ended and entry based. It would be good to add additional question types (multiple choice etc.)
9. Due to the time constraints involved, certain screens are incomplete (the screens displayed when you win, or the game is over). The space here should be better utilised.
10. There are only a few objects created - consumable items in particular – so certain screens look sparse and space again is underutilised. The focus was on ensuring the game mechanics work and the time necessary to ‘fill in’ the details of the game (including the storyline) was not available.

**Potential Future Enhancements**

There is a lot of scope for future enhancements. Initial enhancements would be focused on addressing the limitations stated above. Numbers 1 to 7 correspond to limitations 1 to 7 above. The remaining numbers pertain to the introduction of entirely new features that would enhance the game/GUI but we did not have the time, resources or expertise to implement. Some are taken from the MoSCoW analysis.

1. A hypothesized solution to decreasing loading times (stemming from images) is to open all images upon the game loading. Switching between already open and loaded images is faster. This may cause memory issues however as the game increases in size and complexity.
2. The game is currently unable to facilitate a user-friendly method of deleting saved games. The current method is to delete the .bin file in the file directory but this is not accessible to the average user and the game should be designed so that the player can delete and/or overwrite saves from the ‘load game’ screen. This could be achieved by adding a series of ‘delete’ buttons with lambda commands which link to a delete\_game(game) function.
3. Currently, no message is displayed to denote an object has been picked up or consumed. To facilitate this, a label could be created (or updated) when an object is clicked which displays a message to state what has occurred. For example, when the user picks up an apple, a label shows at the bottom of the screen – “Jack picked up the apple. The apple is worth 15 health!”.
4. To better manage the space of menus and maintain the dynamic way menus are created, it would be possible to simply set the game map symmetrically so that all locations have x number of connections, items, and characters. This would mean that all menus would contain the same number of buttons and could be designed to make efficient use of the space. It would still be possible to edit the game easily and introduce variations to the map but switching connections/items/characters from one location to another (ensuring that the number of each remains consistent in each location).
5. To customise the name of saved games, the user could be asked to enter for an additional input in the ‘save game’ pop-up box. This would alter the name of the .bin file and would be accessible via the user designated name on the ‘load game’ screen. An additional feature could be added to check current saves and warn the user if they are about to overwrite an older save game by using a previously used name.
6. To ensure only sensible string inputs are entered for name and school, an additional check could be made to only activate the ‘start game’ button when inputs are strings. During the design phase, the intention was to introduce this type of input validation, but it proved too time consuming to implement.
7. To ensure all saved games are shown in the ‘load game’ screen, a vertical scrollbar could be added.
8. Upload player images – during the character creation screen, the user could add an image to depict themselves on certain screens (player status, for example). Currently the image shown on the player status screen is determined by the sex the player selects (with only generic male/female images rendered).
9. Currently the player cannot equip or inspect Totem items – this functionality would add another dimension to gameplay.
10. It would increase playability if different difficulty levels could be added – question difficulty can be adjusted (with different question dictionaries dependant on the difficulty selected) and player health and/or character damage adjusted.

**Code re-use**

The initial version of the game was designed to be used through the command line with strictly text inputs and outputs. Certain effects were added (slow typing to imitate speech, for example) to add dynamism and greater interactivity to the game. These features did not feel necessary - as the GUI added greater dynamism and interactive on its own – so were removed. Therefore, the way the story was told needed to be reimagined.

The classes and objects were mostly reusable. To simplify the game and adapt it, initially all but essential attributes and methods were removed. These were then gradually re-added/re-implemented to ensure that each method worked appropriately and consistently.

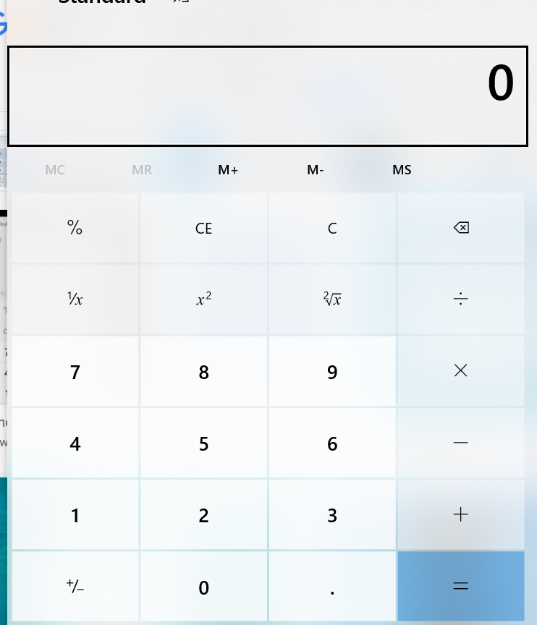
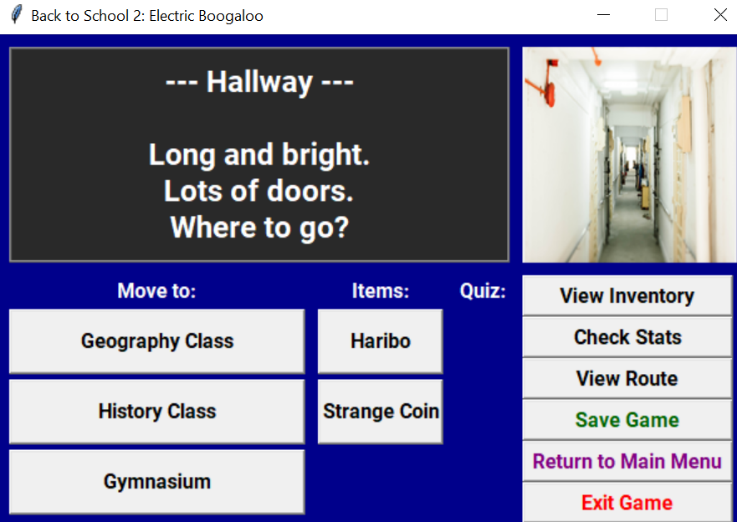
However, the various ‘screens’ (title, main\_loop, inventory etc.) needed to be completely rewritten to optimise for a GUI. Buttons were added where previously the user was asked for open-ended input. These buttons allowed for a simpler code-base where less input validation was necessary.

The command-line game was designed to be completely dynamic, so menus are created based on what objects are present in the location the player is currently situated. This is not an issue in the command line as when an additional object is present, it is simple to display this object on the next line. For the GUI however, adding an additional button can be problematic and throw off the aesthetics/symmetry of the screen. Therefore, while the objects and classes were mostly re-used, the number of objects present had to be reduced slightly so that the screen would not ‘overflow’. It is possible to add a vertical scrollbar and this would allow for greater flexibility, but it was decided that these are unpleasant to use (particularly on a main gameplay screen) and could contribute to lessen user satisfaction with the GUI.

**What other systems you looked at to get inspiration**

The focus of the GUI was to be as intuitive as possible – and Apple and its operating systems have always felt intuitive and aesthetically pleasing. Taking this into consideration, we tried to implement a system that felt natural and obvious to navigate. This involved using clear colour schemes (for example green text to indicate something positive), large buttons and uncluttered screens.

Additionally, we wanted the game to have a sort of retro feel so some of the older, pre-installed Windows games like Solitaire and Minesweeper were analysed to recreate their simplistic yet functional aesthetic. Perhaps more abstractly, the Windows 10 calculator (Figure 18) also served as an inspiration in the general layout it allowed (with buttons in a grid system along the bottom and a display spanning across the top).



*(Figure 19 – GUI main game play menu)*

The layout similarities are evident on this menu in particular.

*(Figure 18 – Windows 10 calculator)*

**Self-Reflection:**

**What did we learn?**

This software engineering project was illuminating. First, thorough planning and comprehensive documentation is crucial to creating a good product within a given timeline. Using the scrum methodology allowed me to better understand the requirements by considering all stakeholders, listing their requirements, and prioritising these. Once this product backlog was created, a good deal of uncertainty and anxiety is removed/reduced, and the task feels much more manageable.

In terms of technical knowledge, I learned to use the Tkinter library and deepened my understanding of python programming. Additionally, testing is not something I had properly utilised before and I found it to be enormously useful – both in terms of finding errors and in terms of designing code which is deliberately modular and reusable.

**Problems encountered?**

Most problems arose because I was trying to retrofit a GUI into a game that was initially designed without this consideration. For example, the command line version of the game (CW1) was designed to be dynamic. The game loop screen is dynamically created based on the map, which means that each area menu is different. This is easy to display on the command line and just adds another line of text to the screen. However, when transitioning to a GUI, these dynamic menus mean that the number/layout of buttons for each room is different and thus, screen space is not utilised consistently and/or symmetrically.

An early version of the game was created using a class-based system where all frames (the various game screens) were created upon launch. However, because a specific screen was created before the Player object was initialised, the screen did not include updated Player information. The solution to this was to move to a function-based game mechanic and create and destroy each frame as and when necessary. This fixed the issue but led to greater loading times between screens.

**What would we change?**

There is a lot that I’d change if time allowed and I could re-do the project. In the design phase, I’d like to implement some in-person talk-aloud testing so I could better understand usability limitations – this was not doable due to the pandemic as I couldn’t meet up with others!

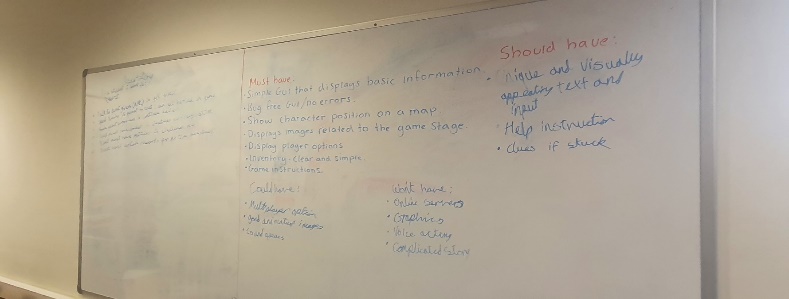
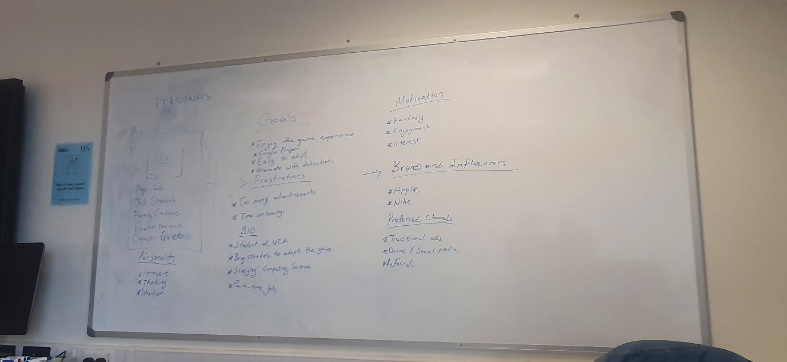
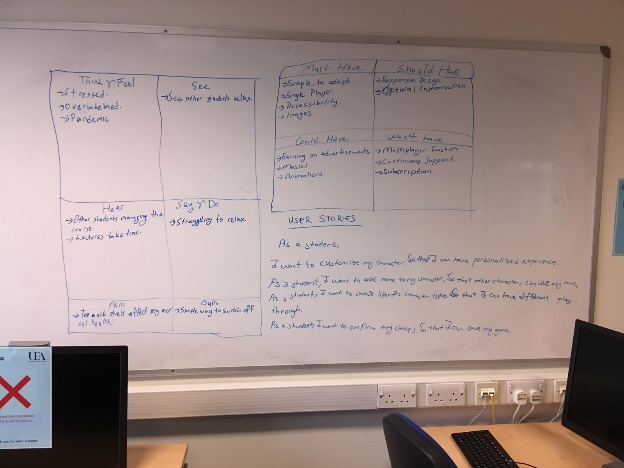
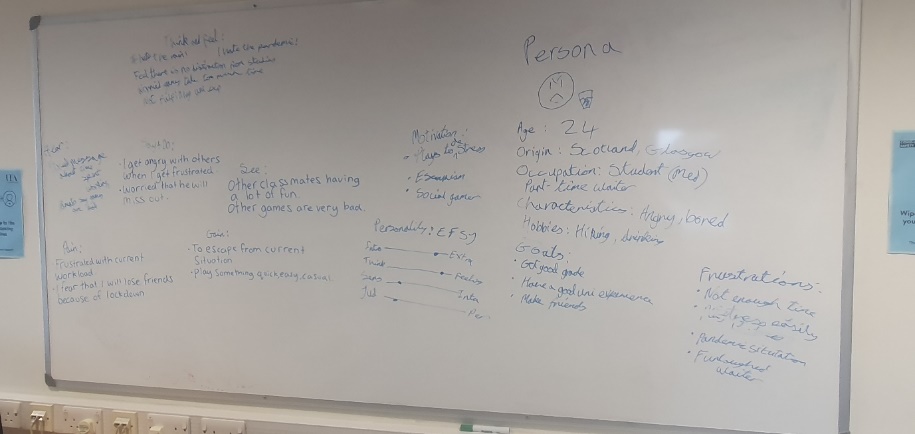
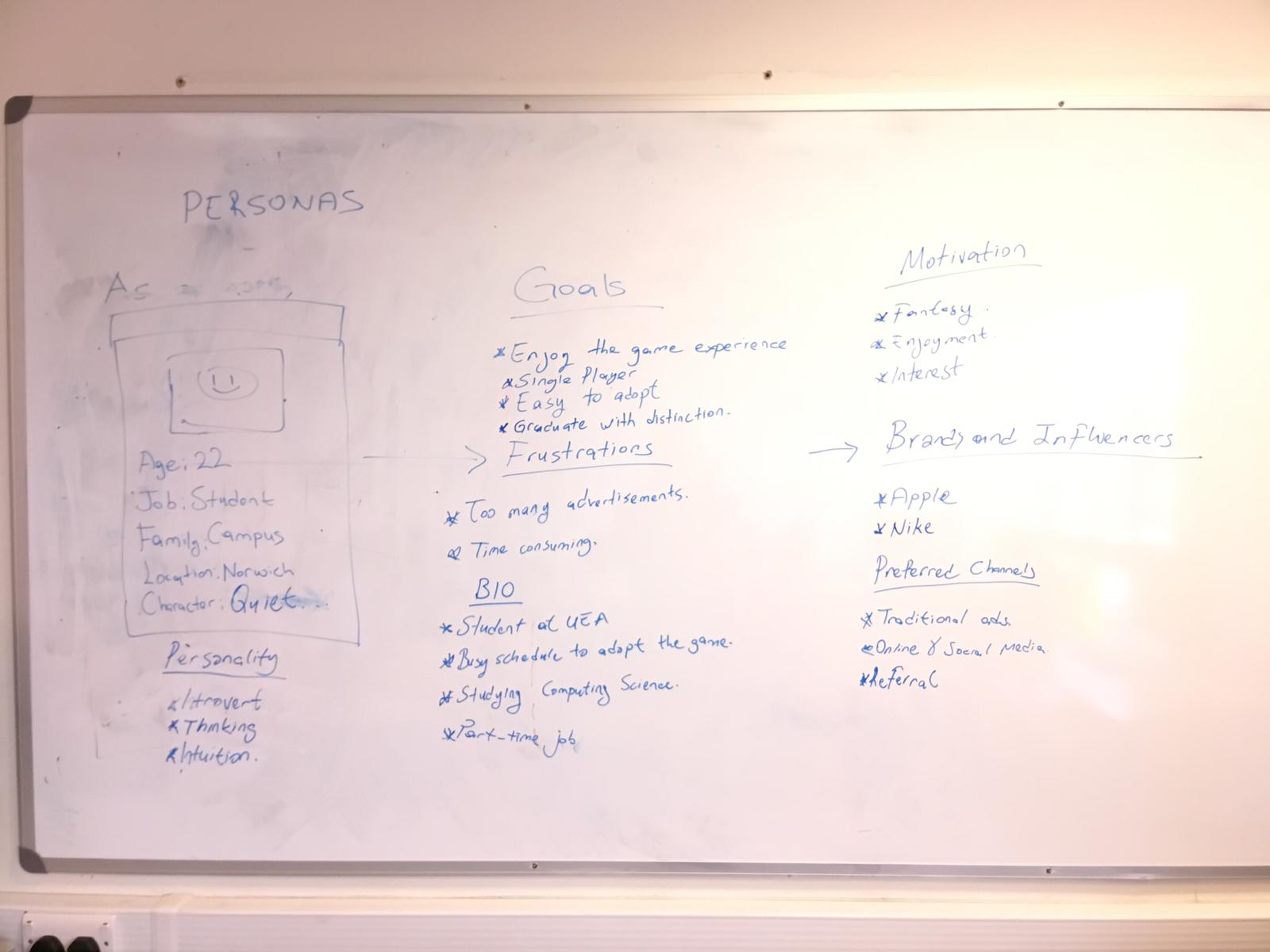
In the implementation phase, the method of moving from frame to frame is crude - destroying and recreating each frame on each click event. This method slows things down particularly where images need to be loaded. A more efficient game mechanic would work by creating each frame at the beginning (on launch) and then updating each frame as and when necessary. This would remove loading time between frames.

Finally, I now recognise the value of implementing a strict, agile methodology from start to end. This project involved retrofitting a command line game (which is poorly designed with no structured methodology) and building a GUI on top. I feel as though my GUI system is compromised by the poor design of the original command line game and given the time, I would completely redesign the original game using an agile methodology and create an improved foundation upon which to build the GUI (or ideally design the game and GUI concurrently in one combined project).

**Bibliography**

1. Tkinter documentation - https://docs.python.org/3/library/tkinter.html - last accessed on 14/01/21.
2. Tkinter tutorial - https://realpython.com/python-gui-tkinter/ - last accessed on 28/12/20.

**Appendices**

Photos from Week 9 lab, group session – Persona’s, Empathy Map, User Stories, MoSCoW Analysis

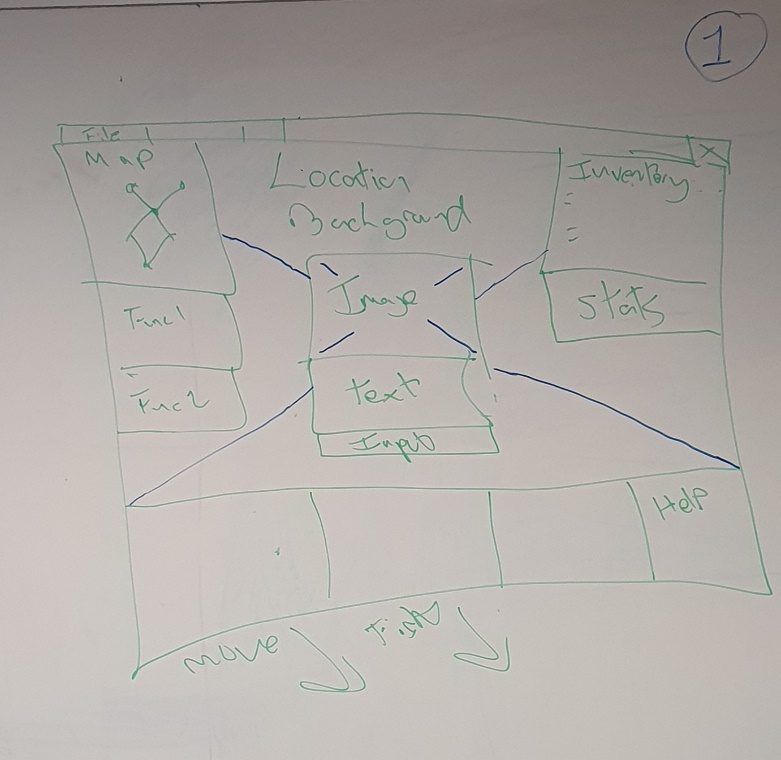
Diagram, engineering drawing, schematic

Description automatically generatedGraphical user interface

Description automatically generatedGraphical user interface

Description automatically generatedA picture containing text, whiteboard

Description automatically generatedA picture containing text, whiteboard

Description automatically generatedPhotos/images from Week 10 lab, group brainstorming – lo-fi/medium-fi designs