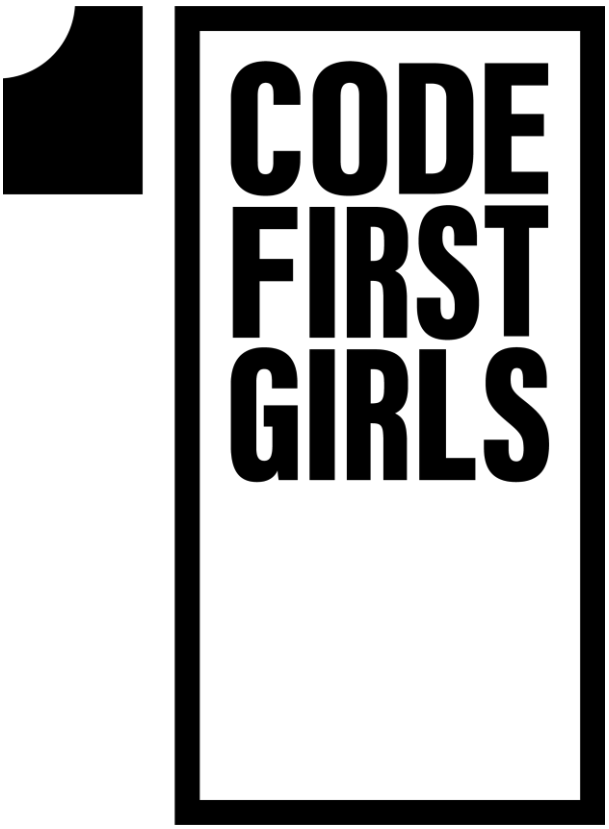
Code First Girls Nanodegree: **Group#3**



GAMES PROJECT DOCUMENT

Word Guesser

A game by:

Faizah Malik, Vanessa Sjoborg, Becky Sorsby, Zoe Scott, Halyna Maslak

CONTENTS

INTRODUCTION

AIMS & OBJECTIVES

The project has sought to fulfil the stated requirements in the Project Workshop PowerPoint slides that were shared with the cohort (see Appendix 1). It states that the project has a clear objective, based on an algorithm that we wrote. The final product is a game inspired by the traditional Hangman, played via an interactive interface (`turtle`) that allows a user to guess an unknown word picked at random based on various set of criteria, such as various difficulty levels and customised tasks. Hence, the objective of this project is to provide entertainment for end users.

OUTLINE

- **Introduction**

This section seeks to establish the aims and objectives of the report.

- **Background**

This section describes the details regarding the project.

- **Specifications and design**

This section discusses the requirements of the project and justifies the design choices and architecture of the project.

- **Implementation and execution**

This section presents the implementation approach of the project. Furthermore, it discusses the team members roles. Moreover, it introduces the specific libraries and tools that were used in the project. Lastly, it discusses the challenges associated with the implementation.

- **Testing and evaluation**

This section presents the chosen testing strategy that was implemented. In addition, it discusses and illustrates the different test types that were utilised during the development process. Lastly, it provides a reflective account of the challenges associated with testing and how the team solved these.

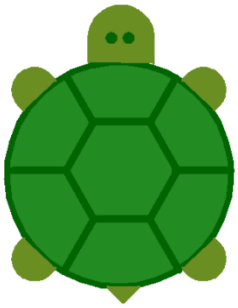
- **Conclusion**

This section summarises what has been covered in the report. Moreover, the team reflect upon the project journey and presents their learning outcomes.

BACKGROUND

GAME BACKGROUND

We have built a word guessing game designed for players who enjoy guessing words. The game design was inspired by the original hangman game. To make it more interesting, we added some additional features. By the same token, we are aiming to bring hangman into the 21st century – the final drawing will be that of a turtle, rather than the traditional drawing of a hangman, which was inspired by the module used to create a user interface and which we have affectionately named 'Donatello'.



The aim of the game is as follows; the user needs to guess a randomised secret word in as few guesses as possible, the game is over when the full turtle drawing is shown on screen. When the user guesses a letter correctly, the computer displays the place of the letter in the word. When the user fails an attempt, the game will begin drawing the outline of our turtle. There are three difficulty levels to choose from; beginner, medium, and hard levels which change the set number of attempts to guess the correct word. The beginner level allows the user nine attempts, the medium level allows the user eight attempts, and the hard level seven attempts. The Levels mode unfolds very much like the traditional hangman. In addition, the game also has a 'Campaign' mode which presents the user with 10 different tasks we carefully designed which the player must complete to win (see section below for description of tasks.)

SPECIFICATIONS & DESIGN

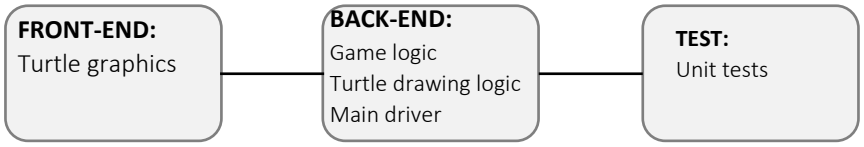
REQUIREMENTS

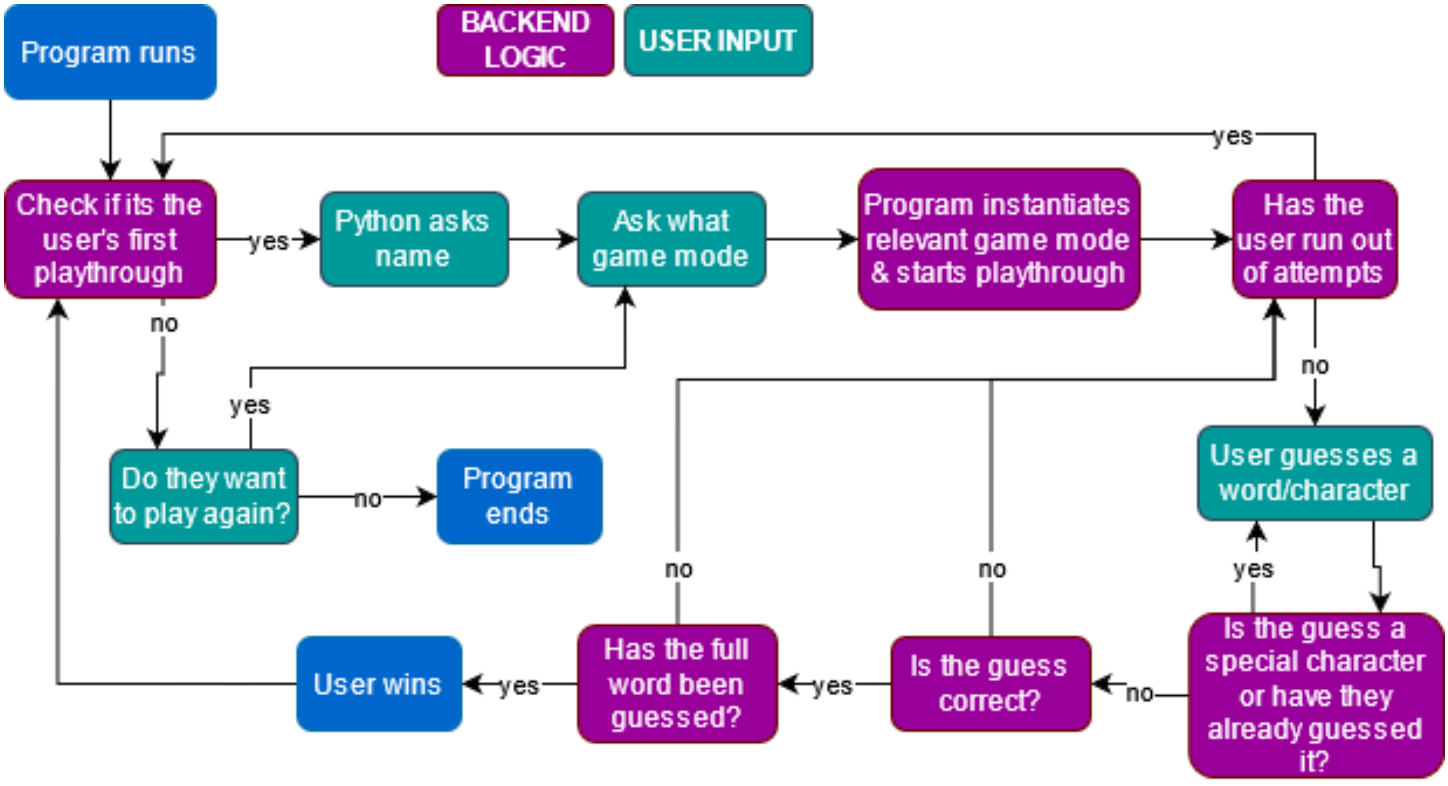
The requirements that the project has sought to address are available in Appendix 1.

FEATURES

- A single-player game, with two modes and three difficulty levels, for speakers of the English language;
- A game that can be played over and over again, but can be exited early;
- Use of third-party libraries, e.g., Turtle, Random, NLTK;
- User friendly interface
- Classes and inheritance with subclasses to optimise code by increasing conciseness, reusability and readability while adhering to the D.R.Y. principle (don't repeat yourself);
- Multiple file scripts with specific categories/usages to adhere to best practice of structuring code

ARCHITECTURE





GAME DESIGN

The program chooses a random word from words list and displays a blank version of the word using an underscore for each letter, in the same format of a traditional Hangman game. There are two game modes that the user can select, “Levels” mode or “Campaign” mode. More information on these is detailed below.

The words list is constructed using the NLTK (Natural Language ToolKit) module which includes words from the Brown University Standard Corpus. The Brown Corpus is a collection of text samples consisting of 1 million categorised words. Although this is a huge library, it was decided to limit the words list to a smaller 5000 to reduce the program runtime. The unique words were chosen based on their frequency in the Brown corpus and the corresponding part of speech (noun, verb, adjective, adverb). The words are written to a json file.

5000 words from the Brown corpus that constitute the game word list are selected according to the following criteria:

- All words are lemmatized (only the dictionary form of the word is used, e.g. think instead of thought, school instead of schools, etc.)
- Stop words are eliminated. Therefore, the final list does not include articles, pronouns, prepositions, auxiliary verbs and other functional parts of speech or other frequent words that do not add much semantical meaning.
- The words from the Brown corpus are filtered to include only nouns, verbs, adjectives, and adverbs that consist of more than 3 letters.
- Punctuation and numbers are eliminated, the user will receive a message stating letters accepted only if they try to enter numbers or punctuation.
- The words are lowercased and sorted according to their frequency. 500 most common words that meet the above-mentioned criteria are selected and written to a json file.

In ‘Levels mode’, the game allows the user to select a difficulty, “Beginner”, “Medium”, or “Hard”, and play one round, i.e., one random word is selected, and the user tries guessing it, afterwards the game will ask the user if they want to play again. The number of attempts varies between difficulties as such:

- Beginner: 9
- Medium: 8
- Hard: 7

The steps of the turtle drawings were altered to match the number of attempts, so that when the attempts run out, the user will still see the entire turtle drawing. Another feature of this mode is allowing the user to add a custom list of words to guess from. This could be useful in a setting of multiple people who want to keep a specific theme or category of words rather than random ones. As a bonus, it also helped significantly with testing purposes as the word was already defined.

In “Campaign” mode, the player must guess words based on different criteria in the task. For example, nouns, verbs, adjectives or adverbs of a variable length, a palindrome or consisting of only unique characters, very common words, or those with low frequency in American English. Campaign mode is set to “Beginner” difficulty by default.

- **Tasks for the Campaign mode**
  - Task 1. "Let’s start! Try guess the most common noun:" → The most common noun in American English (time).
  - Task 2. "Guess an adjective this time: " → An adjective from 100 most frequent adjectives that consist of 7-9 letters (American, several, general, possible, national…)
  - Task 3. "Great work! How about an adverb? " → An adverb consisting of 6-7 letters where at least one letter appears at least once (however, always, perhaps, really, already...)

- Task 4. "Hmmm, a palindrome perhaps? " → A words that reads the same forward and backward with maximum 9 characters (radar, level, civic, refer, noon).
- Task 5. "This is a word with all unique letters:" → A noun, adjective, adverb or verb that consists of 7 unique characters (briefly, movable, heading, quality, destroy...)
- Task 6. "Now we got a verb with all unique characters: " → A verb with 7-9 unique characters (provide, consider, include, produce, explain...)
- Task 7. "You got this! Another adjective this time:" → A less frequent 9-letter adverb (corporate, startling, numerical, southeast, selective...)
- Task 8. "Is it a noun or is it a verb?! Who knows!" → A low-frequent noun or verb that consists of 7-9 characters (breakdown, defendant, counsel, payroll, explosion...)
- Task 9. "You're close to the end! Try an adverb:" → A 9-letter adverb (sometimes, therefore, certainly, generally, obviously...)
- Task 10. "Well done! One last task, guess a verb:" → A verbs that consists of 10 characters (understand, contribute, experience, accomplish, illustrate...)

MVP DEFINITION

The Minimum Viable Product had to be clearly defined so that we could know when it was achieved. The reason for having an MVP is detailed later in the report. Below are our criteria for the MVP. The product should be able to:

- Select a random word from a words list
- Show the hidden words in a hidden format i.e. "_ _ _ _"
- Allow the user to guess a letter or word
- Process that guess and tell them whether they are wrong or right
- Replace the hidden word with the correctly guessed characters in the right positions. i.e. if they guess “L” from “hello” it would show "_ _ l l _"
- Ask the user to guess again until the whole word has been guessed or until the user runs out of attempts

IMPLEMENTATION & EXECUTION

DEVELOPMENT APPROACH

The workload was distributed based on our individual skills and strengths. In addition, a breakdown of the project requirements specified by CFG helped to guide our approach and structure of the project.

The project approach revolved around an Agile mindset. Agile was decided upon rather than the other popular software development framework, Waterfall, because it allowed us to keep the project plan more fluid and susceptible to change. Waterfall requires a solid plan on system design and features right at the start of the project which is not feasible as there was not much time to come up with a plan, and not enough Python experience to estimate a timeframe.

Adopting Agile allowed the team to have a more dynamic approach to development, where a Minimum Viable Product (MVP) is established, and features can be gradually extended along with our increasing knowledge of Python. These features are taken from the product backlog, decided by the team, and implemented on a rolling basis until sufficient time before the before the project deadline. After which, the team worked on finalising the project document and submitting it along with the code.

Our main project management technique and the one which stood out the most was Kanban. Although this is not the traditional Scrum framework which was taught during the course, it still adopts an agile mindset. It was decided that using entirely Scrum would not be feasible because of the extreme focus on communication such as having daily stand-ups which did not fit around the schedules of team members who had other commitments such as full-time jobs. Hence, elements of both Scrum and Kanban were used.

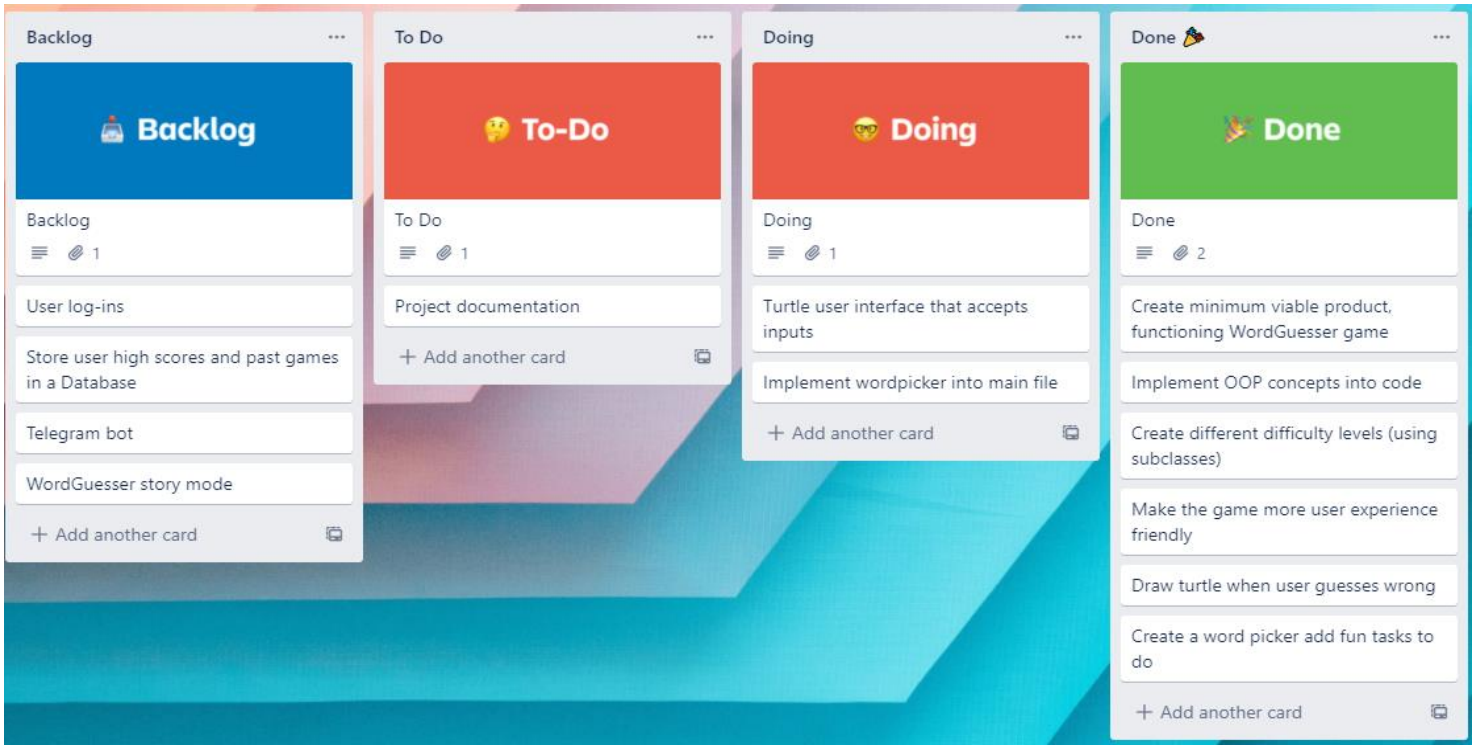


Figure 1: Screenshot of the team's Scrum/Kanban board during project development.

An example of adopting an Agile framework is engaging in peer code reviews This was very important to the team's success as it allowed every member to see the changes made by the other team members and keep up to speed with the latest code. We mainly reviewed code through pull requests on GitHub and organising regular meetings in Teams. Another example of an Agile practice would be the Kanban/Scrum board we used, shown in figure 1 above. This was helpful in visualising our tasks and workload.

TEAM ROLES

Faizah, Becky and Zoe were mainly responsible for creating a functional base game upon which we could later expand with more features. Tasks performed included, although not exclusively (please see GitHub for complete history of contributions):

- Created the *levels.py* file with one base class with the associated methods to inherit, and three subclasses to vary the difficulty from "Beginner", "Medium" and "Hard" with their associated attributes.
- Identified bugs and worked towards debugging the code – finding alternative solutions to implementation.
- Created *turtle_window.py* file after researching the Turtle module on how to create a user-friendly interface and allow user input through the Turtle module.
- Created *main.py* which includes one class, "PlayGame" with two subclasses "CampaignMode" and "LevelsMode".
- Responsible for creating the README.md file with all the relevant information, including modules used, project plan, how to play, file explanations, and an about section.

In addition, Halyna had experience with natural language processing and is familiar with the third-party module NLTK. Consequently, the team agreed that she could expand the base-game with additional features (please see GitHub for complete history of contributions).

- Responsible for creating a *words_list.json* file using the NLTK module which includes a large words list.
- Researched ways to reduce speed issues associated with having a large words list
- Created *word_picker.py* which was later implemented by the team for the Campaign mode of the game

Furthermore, Vanessa expressed that she was comfortable with writing unit tests for the program and oversaw the overall testing of the game (please see GitHub for complete history of contributions).

- Created unit tests for the programme
- Reformatted classes and methods to enhance readability and "testability" of code
- Identified bugs and debugged code
- Improved *levels.py* file by breaking up methods for ease of testing and to adhere to best practices (i.e. keep methods single responsibility)

All team members contributed with ideas to the written report to gather multiple perspectives resulting in a well-rounded project documentation with inputs from team members with different specialisations in the project.

TOOLS & MODULES

The following tools were used to create the project:

- ***Project deployment and testing***: The programme was written in the Python programming language and utilised various add-on packages.
  - `random` – This module implements pseudo-random number generators for various distributions. It was used in the *levels.py* file to generate a random word from a given list.
  - `turtle` – The turtle module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. It allows shapes and pictures to be drawn easily. This was used to provide a type of user interface for a more enjoyable experience.
  - `nltk` – NLTK is a Python library for symbolic and statistical natural language processing (NLP) to work with human language data. It was used to generate a list of words for the game.

  - `collections` – This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`. Counter from Collections was used in the *word_picker.py* file to count the number of instances of unique letters in a word.
  - `json`– JSON is a syntax for storing and exchanging data and the package can be used to work with JSON data. In this case, the word list was stored as JSON data.
  - `unittest` – It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. It was used to write unit tests for the different classes and methods.
- ***Team collaboration and project storage***:
  - Git – A version control system used to facilitate collaborative work.
  - GitHub – A cloud-based version control management system that uses Git to enable collaboration within a development team.
- ***Communication:***
  - Microsoft Teams – Used for scheduled meetings/debugging sessions and storing files (such as the project report) to allow the team to access and work on the documents concurrently.
  - Slack – Used for frequent chat communication and impromptu voice calls within the team and between the team and their designated instructor.
  - Zoom – Used during schedule CFG class sessions with designated time allotted to project work.

IMPLEMENTATION PROCESS

WORKFLOW

The code was managed by using the cloud-based version control, Git, and web hosting system, GitHub. This was used to enable the team to work on different elements of the code simultaneously and that any additions or updates to the code could be reviewed and approved by the other team members before they were merged into the main branch of the project. In addition, Slack was used to keep in regular contact, arrange meetings and discuss any developments with the project. Microsoft Teams was also useful in working on the on the project documentation files simultaneously and hosting regular team meetings.

As per the agile approach to the development process, we had regular meetings to discuss our progress. Some members of the team would meet on a one-to-one basis to work together on particular issues/bugs or code reviews and update the rest of the team on progress and resolutions later. Following our regular meetings, a team member would send an agreed "next steps" bullet point list so that everyone was clear on what the assigned tasks were and had a greater sense of responsibility to get them done, we would also agree the date of the next meeting. Hence, we engaged in an Agile approach to our work.

ACHIEVEMENTS

As a result of our efforts to split the work based on our individual strengths, as well as effective time management among the team members – we were able to produce an MVP at the end of week three. Based upon this the team was able to incrementally add more features.

Furthermore, the project demanded us to think creatively and engage in research-oriented problem-solving to address certain needs of the project. E.g., additional understanding of the add-on library Turtle to request user input within the window and the use of `setUp` and `tearDown` class methods as part of the `unittest` framework. Hence, an additional achievement is that we increased our knowledge in Python whilst working on the project.

CHALLENGES

One challenge we initially encountered was the idea of building the game inside a Telegram bot, initially discussed at the start of the project, but upon research the intricacies of implementing our code within the bot and realising that some important features i.e., Turtle Graphics, would not be possible in conjunction with the bot, it was decided to reconsider this idea upon achieving the MVP. We also considered implementing the `tkinter` module which would allow us to make the user interface and user experience much more attractive, however this was also not carried forward due to time constraints.

The next biggest challenge was to begin building the game and implement the logic of a word guessing game. This was first originally achieved by Becky who created the logic of the game, then the code was expanded on by Faizah who improved logic and organised everything into a class, then Zoe who created the subclasses i.e., different difficulty levels and tidied up the code. Finally, Vanessa split all the methods up, so that each of them adhere to the single responsibility solid principle. This was the evolution of the *levels.py* file which required a lot of team input and attention as it does contain the main logic of the guessing process which is central to the game.

Creating sound and (as far as we know) bugproof logic took a considerable amount of effort as no one in the team has had much experience designing games. Once we reached the Minimum Viable Product criteria that we defined at the start of the project, we realised the importance of prioritization, given the limited timeframe and considering that the team has not been learning Python for very long. As mentioned above, it was decided that there would not be enough time to implement a Telegram bot or make use of the Tkinter module.

It was therefore decided that refactoring and reviewing the code was more useful, as well structured, readable code took a higher priority than adding technically demanding features which could break our code and take a lot of time and effort to put together given the team's current abilities.

An additional implementation challenge involved integrating the tasks as an additional mode in our game. This part of the code had been developed in parallel with the Levels class, sub-classes and the main script, and therefore was not created with the rest of the code in mind. Hence, although the WordPicker class was well-written, it soon became evident that the goal of integrating a class with consecutive tasks was a lot more complex than previously anticipated. This was primarily due the main script being optimised to run the Level classes in isolation. This realisation left us considering whether to completely abandon the Campaign mode as the deadline was rapidly approaching or spend additional time to refactor the code to implement the additional feature. It was a difficult decision, but the team agreed that part of the charm of the game was the additional Campaign mode. Hence, by splitting up the remaining tasks among the team members, the decision was made to refactor the code to incorporate the WordPicker class.

Another challenge we ran into was building appropriate unit tests. Initially, there were a couple of functions within our parent Level class that could not be tested as it contained multiple functions. In addition, some functions were not returning a value which caused an issue when testing that the real output was equal to the expected output.

One more challenge we faced involved increasing the game speed. Originally, every time the game was played, the game iterated through 1M words in the Brown corpus in the NLTK library and created the required 5,000-word list. It took 8-10 seconds to do so. Therefore, it was decided to write some chosen 5,000 words into a json file once, and later read from this file when the game is played. With this approach, the retrieval of the needed words to be guessed takes only 0.008 seconds. Halyna was responsible for finding the solution and implementing it.

Another challenge was – with the advice of instructors – that our main needed to be completely refactored due to not adhering to single use methods, not being divided into classes, and the biggest issue, being very

hard to test because once one function was called, other functions were also initiated as the game ran in a loop. This advice was received a few days before the deadline, and we were questioning whether we had time to restructure our main file or not. Faizah promptly reformatted the code within a few hours by organising the game modes into 2 subclasses, rewriting, and writing new methods to adhere to D.R.Y, K.I.S.S, giving methods single uses and putting the actual game loop outside of the methods, instead of being intertwined with them. This reformatting made the code look much cleaner, easier to understand, run, debug and ultimately test.

TESTING & EVALUATION

TESTING STRATEGY

Based on the content that was covered in the course, we decided to pursue a partial Test-Driven Development approach to testing our programme. The reasoning behind this decision was to facilitate the writing of the code as we believed it would give us a sense of direction of what may be required of the code. However, during the later stages of the project it became clear that we required additional classes and functions that we had not initially anticipated. Hence, parts of the test cases were written in parallel with the code.

TESTING IMPLEMENTATION

In terms of testing, the team undertook a variety of different forms of testing such as Quality Assurance, Quality Control and functional testing to ensure a pleasant user experience. **QA** was performed by following best practices within the industry, e.g., the files were created based upon the OOP principles. Also, the team continuously engaged in peer code reviews, e.g., a team member would push their latest modifications to their GitHub branch and open a pull request. This allowed the rest of the team to review their code and comment on any changes. In addition, the team implemented an agile approach to developing the programme. E.g., the we held frequent team meetings where we discussed our progress and next steps, as well as engaged in debugging session together. The team sought to adhere to best practices as it would improve readability and therefore assist in the marking process of the project - but also to enable the team members to cooperate and debug parts of the code that another member had written.

QC was performed systematically during the development of the programme with the aim of detecting bugs. E.g., there appeared to be a bug that would prevent the user from exiting the programme, even though they answered “no” when asked if they wanted to play again. To ensure that this bug had indeed been resolved, a team member tested the game several times. Hence, manual testing was performed in additional to automated testing.

In addition, several types of functional tests were implemented during the creation of the programme. As described in the aforementioned sub-section (testing strategy), **unit testing** was an integral part of the project to ensure that each part of the code was executed and provided the expected output. E.g., the test file *test_turtle_window.py* was created to ensure that each function within the class drew the expected part of the turtle. Additionally, **smoke testing** was performed consistently throughout the development process to ensure that the basic functionalities of a new build was working. E.g., before the class TurtleWindow from the *turtle_window.py* file was imported to the *levels.py* file, an instantiation was created to ensure that the object could perform the methods as expected, i.e., to test whether the turtle window appeared.

Furthermore, **regression testing** occurred on multiple occasions throughout the development. E.g., it became clear that the Level class (available in the *levels.py* file) required modifications as some of the functions did not adhere to the single-responsibility principle. Subsequently, the functions were broken down further. To ensure that the modifications had not affected other parts of the programme, the team members heavily tested the amended file through QC. Moreover, the team implemented **integration testing** to ensure that separate parts of the programme worked as expected when merged. This was tested when Donatello (an object of the class TurtleWindow) was imported into levels and its methods were implemented within the existing code.

Once the individual modules had been created, the programme was subjected to rigorous **system testing** to establish that it met the objectives of the programme (see Aims and Objectives and Appendix 1). E.g., ensuring that the “end product is a game played via an interactive interface that allows a user to guess an unknown word picked at random based on various set of criteria, such as various difficulty levels and customised tasks”.

Lastly, the team members asked family and friends to play the game to make sure it could handle required tasks in real-world scenarios. Hence, **user acceptance testing (UAT)** was performed as well.

SYSTEM LIMITATIONS & TESTING CHALLENGES

The implementation of the different types of tests offered us some additional understanding of the limitations of our programme. E.g., a **non-functional test** occurred impromptu during the development process. It became clear that importing a list of words by using the NLTK library severely slowed down the game. Hence, we realised that *speed* was causing a system limitation. This 'hiccup' (which was also described in section Challenges) required us to adjust our approach to how we retrieved and stored the words. The team discussed whether to potential create a generator rather than a list to retrieve the words. However, ultimately an alternative solution in the form of a json file was implemented to store the words once rather than creating a list every time the game was run. This option was preferred by most team members since it would reduce a step in the programme, as we would have the filtered list of words accessible after having run the *create_json_wordfile.py* just once, rather than iterating over words and filtering them out simultaneously.

Moreover, the unit tests made us realise that a lot of our functions performed several tasks rather than one which we believed reduced readability and made it more difficult to debug the code. Therefore, as a direct response we split up the functions further. The most difficult part about testing was writing the code to be tested. In essence, with writing the tests came the realisation that testing can be a straightforward process – depending on how well the code is written. Hence, at some points in the development process it became evident that although the code worked and ran the programme, it was not necessarily well-written. Therefore, to deal with this challenge the team re-wrote parts of the code several times to enhance readability, e.g., writing functions that perform one task and ensure that functions had a return statement against which it could be tested. Subsequently, our understanding of the importance of clear functions and classes became further ingrained. In addition, we understood that writing good code is an incremental process which requires practice and time.

CONCLUSION

In conclusion, this report has sought to present the objectives of the project. It provides additional background information regarding the game. Furthermore, it discusses the implementation and execution of the project. The chosen development approach was presented and justified. In addition, the team member roles were discussed alongside the utilised tools and modules. Moreover, the report provides a reflection on the team's achievements and challenges that were a direct result of the implementation process. Lastly, the testing and evaluation section introduced the chosen testing strategy followed by a discussion concerning the testing implementation with relevant examples. The report gave a reflected account of the challenges that the team faced when testing and how we overcame them.

The major learning outcomes that this project has presented are plentiful. Essentially, it has allowed us to apply the skills that we have been taught during the Nanodegree, whilst also expanding our knowledge in Python. Furthermore, we learned the importance of prioritisation which can sometimes make or break a project with a limited timeframe. It has allowed us to enhance our time-management skills along the way which is very crucial to the success of projects. In addition, we were required to continuously engage in creative problem-solving when it came to matters such as debugging. This process was facilitated through good communication and collaboration among the team members, which was another important aspect to ensure that the project would be finalised in time for the designated deadline. Lastly, the project enabled us to get some practical experience in coding. It became clear to us that writing good code is an incremental process and to become a better coder one must be willing to constantly push oneself slightly out of one's comfort-zone.

APPENDIX 1 – Nanodegree Final Project Summary

Final Project Requirements:

1. Project to be delivered in groups of 3-5 people
2. Project has a clear objective:
    a. Solves a clearly defined problem
    b. Satisfies a need/demand for user(s)
    c. Automates a process
    d. Project is a simple algorithm game (e.g. battleships, sudoku, etc.)
3. Project is built and completed within given time frame
    e. You cannot 'reuse' any past projects, it has to be a new one
    f. Consider 'mocking' some components of your project by using team's judgement, scale, and complexity of your deliverable (e.g. Front end user input as a .py file, DB as a spreadsheet file)

Project includes...

4. Back-end Python code with clear, organised project structure, including:
    g. Logically grouped script modules
    h. An utils file (if applicable)
    i. Test files (unit tests for each function and class)
    j. 'Main' script (with 'run' function that would execute your project)
5. Use of existing API - any API from internet (not applicable for game projects)
6. Use of key OOP principles
7. Libraries such as itertools, collections, etc.

It is recommended that the project should include…

8. Use of Python, API, DB
9. Your own API endpoints
10. Decorators
11. Recursive functions
12. Regression testing (i.e. does implementing new features break the project?)

Code should demonstrate...

13. **Effective class/method/variable names** - names chosen should effectively convey the purpose and meaning of the named entity
14. **Effective top-down decomposition of algorithms** - code duplication should be avoided by factoring out common code into separate routines. Individual routines should be highly cohesive. Each routine should perform a single task/a small number of highly related tasks; routines that perform multiple tasks should call different subroutines for each task. Routines should be short in most cases (mostly <20 lines, almost always <50 lines, very rarely >100 lines.
15. **Code layout should be readable and consistent** - this means things like placement of comments, code indentation, wrapping of long lines, layout of parameter lists, etc.
16. **Effective source tree directory structure** - the source code for the project should be effectively organised into subdirectories

17. **Effective file organisation** - code should be effectively organised into multiple files; lumping all code into one or two files is unacceptable.
18. **Correct exception handling**
19. **Good unit test cases** - these will be evaluated according to the requirements in the project specification
20. All source code of the project:
    - Python files
    - DB files
    - External .txt files
    - Testing suite
    - README file with clear instructions on code execution
21. PowerPoint slides (2-3 mins max) with key points for presentation