Project Report

Munisha Boora, Rebecca Brunning, Ellie Redmayne, Shida Koka, and Xian-Ting Lee.

Introduction:

Our project involves creating a revision application that will allow a user to test their knowledge of the various topics taught throughout the CFG nanodegree. Users will be able to start up the application and select a category they want to revise from the various options presented to them. The game has been split up into four categories with more to come. The quiz categories included so far are: Python Classes, Python Decorators, SQL Queries, and SQL Tables.

We were inspired to build this project after seeing many individuals on the CFG nanodegree struggling to learn complex topics in a memorable and enjoyable manner. The application we have built aims to help improve the knowledge of CFG students whilst they're taking the course and also assist them in their revision for their assessments. We hope to reach our objective of increasing retention for CFG students through the creation of an application consisting of various revision-based quizzes. The application will allow those individuals who struggle with time keeping to revise the course content anytime and anywhere.

Our revision application works by presenting an individual with multiple quizzes to start with. These quizzes would all be split by the categories to which they relate. When our application is run, an individual would be presented with all possible quiz categories. Upon selecting a category, an individual would then be presented with all of the quiz questions relating to that particular category. As an individual selects a quiz and begins answering the quiz questions, their score would increase by 1 for each question answered correctly and by 0 for each question answered incorrectly. After completing a single quiz, the overall score for the quiz in question would be displayed on the screen.

Our project uses a MySQL driver, MySQL Connector, to access the MySQL database, after creating a connection to the database. The use of a database in our project to store all quiz questions and answers allows us to integrate our knowledge of SQL and databases into the project.

This report will go into the decisions that we made and why we made them to, ultimately, create all aspects of our application. We followed the agile framework and this report will highlight the ways in which this was implemented at each stage of the software development lifecycle, including specifications and design, implementation and execution, and testing and evaluation.

Background:

The background of our app stems from aiding students with their learning. As a team we agreed that building an app which helped CFG students learn was something we were all passionate about and interested in. We did some research and discovered a report at Stanford University that found ‘using games as an educational tool provides opportunities for deeper learning’ (Mackay, 2013). Furthermore, we learned that extensive research has revealed that tests may be an efficient way to boost long-term memory of learned content, in addition to serving as an assessment tool (Roediger and Karpicke, 2006a; Karpicke, 2012; Eisenkraemer et al., 2013).

From here we began the agile process and created a plan to integrate educational learning into a game, which would benefit both us and other CFG students, when revising the course’s content.

Our revision app follows a multiple-choice format. The user is first offered a range of possible topics to revise that have been taken from the CFG Software course. After selecting a topic, they are then presented with their chosen topic’s questions, alongside four possible answers that they must choose between. The questions are presented one-by-one, to prevent information overload for the user, and a score is calculated at the end of the game, so the user can monitor their progress.

If the user selects a correct answer their score increases by 1, however if they select an incorrect answer their score does not increase and they are moved onto the next question. Their total score is revealed to them at the end of the game. We hope that a user will be motivated to replay the quiz and keep improving on their score until they get 100%. After the user gets 100% in one topic, we hope they would feel driven to move onto the next topic to revise new content which follows the same format.

Although our application was designed specifically for CFGdegree students, and the included questions are based on the course’s curriculum, the application could be applied to different areas of education by simply updating the questions stored within the SQL database.

After completing our research and planning how our app would function, we then moved onto the next stage of the Software Development Lifecycle; specification and design.

Specifications and design:

Technical Requirements:

- The system must present a main menu with a selection of categories for the user to choose from.
- Once the category is selected, the system should retrieve related category questions from the database to run the quiz.
- The system would then close the main menu window and open a new window for the quiz game.

- The user will then be shown related questions, each with four options to choose from (as potential answers to the question presented).
- Each question must have a randomised order of options, from which the user will then have to select an option before moving on to the next question.
- Once the user completes the quiz, the user will then be presented a small message box displaying the number of correct answers, incorrect answers, and the overall percentage of their score.

Non-technical Requirements:

- Large text for visually impared
- Well documented code within the python file for developers
- Extensive unit testing
- The system should be able to support different screen sizes and operating systems.

Our architectural design shows that the user will be able to access the quiz interface by running it via the main python file. In the main python, it runs the main menu window and the quiz game logic which are the middleware. To access the database, a data access layer handles all calls to the database.

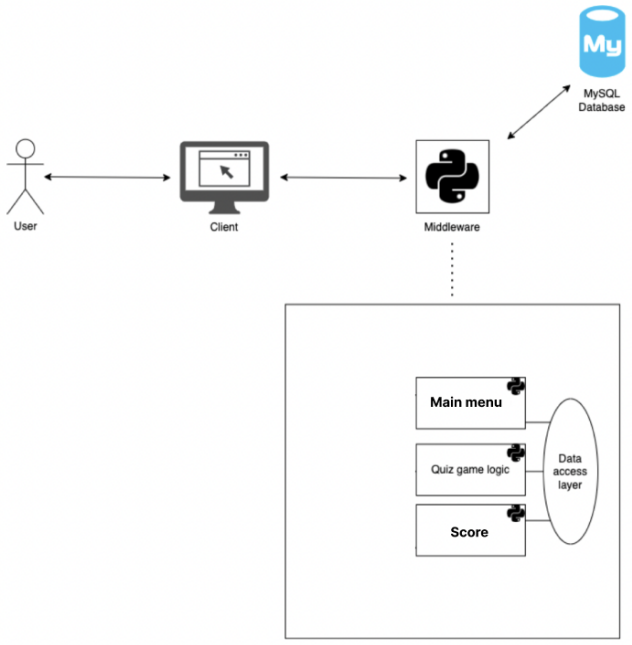

Figure 1: Architecture diagram of the system

The database was designed based on the requirements of the quiz discussed, in terms of the data to be displayed on the interface, specifically the questions and the answer options. The quiz categories table is used in the main menu, as it can retrieve questions related to the topic selected. Other tables, related to the participant, are where the user details and the quiz record containing the quiz_id and quiz_score are stored. These participant tables are for further development of the quiz game, in hope to use it for the leaderboard system.

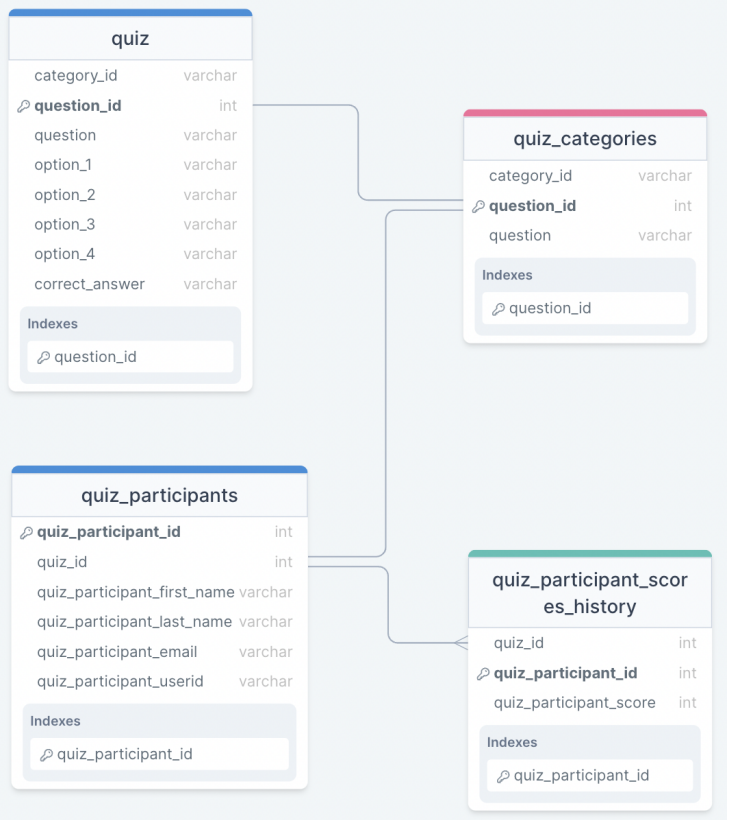

Figure 2: Database design

Implementation and execution:

Due to the constraints of this project, (using relatively new content and techniques, as well as working remotely), we utilised an agile methodology, adhering to the Scrum framework, to break up initially intimidating challenges into smaller incremental tasks that we could tackle as a team and work through systematically.

We designated a scrum master to direct sprint planning sessions, manage the Trello board, and remove any developing roadblocks. We compared advantages and disadvantages of project ideas on a Miro board. Then, we developed the project's architectural design, separated it into its essential parts, (the backend, database, and user interface), and then decided on a minimum viable product (MVP). Our plan was to finish each component separately in the first sprint, combine them all in the second sprint, and then to style, debug, and test during the third sprint. We wanted to spend the fourth and final sprint resolving any remaining issues and making sure our code adheres to the SOLID principles. We managed to stay on course and accomplish our objectives.

We created individual tickets on our Trello board for each task in our product backlog. We moved the relevant tickets from our product backlog into the first sprint backlog at our sprint planning meeting. We also assigned responsibilities and spoke about potential difficulties

during this meeting. As teamwork is useful for problem solving, pair programming was used for several components. This procedure was repeated throughout each sprint. Due to remote working and personal obligations, daily stand-ups were not practicable; nonetheless, we continuously used Slack and WhatsApp to connect, and any inquiries were shared with the team to prevent wasting more time on a single task through knowledge sharing.

After each sprint, we held a sprint review/retrospective meeting to communicate progress and make sure everyone on the team was aware of how each feature worked. We also had a chance to address any issues we encountered and what we learned from the previous sprint, including what went well and what didn't, in addition to what we could do better. We used Git and Github for version control, and whenever we merged different sprint branches back into the main branch, we did so collectively to ensure that everyone could address any disputes.

Our project was organised using the MoSCoW prioritisation. Even though not all aspects of the project were completed as originally anticipated, we succeeded in producing a useful productivity revision quiz app by focusing on the "must have" components when creating our MVP. We intended to use Python and SQL to demonstrate our newly acquired skills while also making plans for potential deployment.

Tools:	**Libraries and Packages:**
● Miro - brainstorming the initial idea ● Slack & Whatsapp - communication ● Zoom - meetings ● Trello - product backlog ● Pycharm - IDE ● Postman ● MySQLWorkbench - Database	● Random ● SQL Connector Python ● Tkinter

Our implementation consists of the following parts:

SQL database
A SQL database is used to store the following; question ID numbers, topics for revision, quiz questions, four options for choosing between, and correct answers. We also included tables for users' names and high scores.

Topic selection menu
Upon running the application, the user is presented with a choice of the current range of revision topics stored within the SQL database. Our decision to present topics as they appear in the database, as opposed to using static 'topic buttons', should improve the application's adaptability and scalability, as the presented menu will automatically reflect any changes in the database without the need to manually update the buttons. The user's choice of topic will be stored and used to filter the revision questions presented to them.

Multiple-choice quiz
The user is presented with a series of questions and four possible answers relating to their chosen topic for revision. The four possible answers are shuffled each time the game is

played, so the user is not able to rely on memorising a ‘path’ through the game, but must read the content. If they attempt to move through the quiz without selecting an answer, an exception will be raised and shown to the user as a pop-up warning box. After completing the game, their score is calculated and shown to them on a pop-up window.

Implementation challenges:

- Tkinter gui is presented differently on different operating systems, i.e. buttons are not displayed the same on macOS and Windows.
- Updating features that have dependencies with other classes or functions. For example, if a function makes a call to the data access layer.

Testing and evaluation:

We used unit testing to check that our functions behaved as we would expect by checking what they returned. We also allocated testing of particular features to members of the team responsible for writing the code for those features. We used the unittest module that is included in the Python standard library to help us write and run tests for our Python code. The functions as well as the methods of the classes in the following files were tested: invalid_selection_error.py, data.py, and quiz_fucntions.py.

The Question class within the question_model.py file was not tested using the unittest module as the class only contains an __init__ method. The topic_question_bank function within the main.py file was also not tested as an object at a different memory location would be returned each time the function is tested. As a different output was returned each time we ran the tests for the topic_question_bank function, we decided to not test the topic_question_bank function.

Testing the methods of the class within the quiz_functions.py file:
From the quiz_functions.py file, we tested the remaining_q, next_q and get_score methods of the QuizLogic class. We created three classes, TestRemainingQ, TestNextQ and TestGetScore to test the output of the remaining_q, next_q and get_score methods respectively. Within the methods of the TestRemainingQ class, we used the assertTrue and assertFalse functions. assertTrue() is used to compare a test value with True and assertFalse() is used to compare a test value with False. If the test value is true then assertTrue() will return true else return false. If the test value is False then assertFalse() will return true else return false. Within the method of the TestNextQ class and the TestGetScore class, we used the assertEqual function to determine whether the two variables passed to the function were equal, as expected. We determined whether the two variables passed to the assertEqual function were equal by setting the result we expected equal to the variable expected and setting the result variable equal to the function being tested, with some inputs. We then compared the variable expected with the variable result in the assertEqual function.

Testing the functions within the data.py file:
From the data.py file, we tested the get_data and get_options functions. We created one class, TestGetData to test the output of the get_data function and another called TestGetOptions to test the output of the get_options function. Within the methods of the TestGetData class and the TestGetOptions class, we used the assertEqual function to

determine whether the two variables passed to the function were equal, as expected. We determined whether the two variables passed to the assertEqual function were equal by setting the result we expected equal to the variable expected and setting the variable result equal to the function being tested, with some inputs. We then compared the variable expected with the variable result in the assertEqual function. The strategy was followed in each method of each class within the data_tests.py file.

Testing the valid_option_selected function within the invalid_selection_error.py file:
From the invalid_selection_error.py file, we tested the valid_option_selected function. This was to determine whether an exception is raised when the valid_option_selected function is called with 0 passed to it as an argument.

Functional and user testing
We utilised manual functional testing such as smoke testing whenever new code was added to our main branch. We did this throughout the project to ensure that individual changes made did not impact the functionality of the overall software. As we have developed a relatively small scale project, we used more informal user testing by inviting our friends and family to play the revision quiz. We were able to gain constructive feedback and make changes to the application according to their experiences.

System limitations:
Incomplete features which fell under the 'could have' and 'would have' titles were:

- There is no countdown timer being displayed during the quiz to let a user know how much time they have remaining whilst taking the present quiz.
- Currently, there is no option to input a user's name, surname and email address so that they can keep track of their progress.
- Alongside the implementation of a user registration and login option, we envisioned creating a leaderboard so that students could compete against their peers.
- Due to the time constraints for the project, there are no accessibility controls, such as larger font sizes, screen readers, or alternative languages.

Conclusion:

We were able to meet our goal, which was to design and implement a revision application that is easy to navigate for CFG students, and offers an individual a selection of quizzes to choose from. We successfully coded a game containing four quizzes that utilised a variety of tools and methods that are functional, and could demonstrate an understanding of more specialised engineering concepts, such OOP.

A key strength of our team's approach was our flexibility to challenges, and commitment to collaboration. Team members with additional skill sets, such as front-end development, were able to advise those with limited experience. Such skill sets allowed us to implement a project with a working frontend, using the tkinter external library. By using an agile approach, we were able to implement additional features, such as a wider range of quiz choices and

improve the user experience too. Allocating different tasks to each team member helped the code to be developed in parallel to ultimately produce a MVP quickly.

Future improvements

A key rationale for our decision to design a quiz game was so that we could make learning more enjoyable for CFG nanodegree students. Therefore, we could improve our project by including an extensive game history display, which shows historical data analytics and detailed personalised user insights. An example of this, could be a visual peak times played per day for the individual user. In addition, to improve user experience, we could build a timer user interface component so that users can keep track of how much time they have left for a certain quiz.

The current revision game does not give users an option to compete with their peers or to see their score in a leaderboard. The project could be extended to accept user input to further personalise the quiz. Presently, we are also aware of an error code that appears on macbooks, ‘interrupted by signal 11: SIGSEGV’. The issue is due to an attempt by our program to write or read outside its allocated memory. This issue will be resolved in the next sprint.