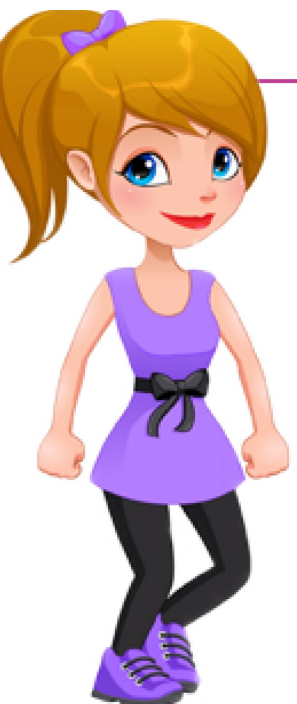
Table of Contents



INTRODUCTION

Code Quest aims to provide an engaging and challenging gaming experience, assessing players' responsiveness and decision-making skills within a 60-second timeframe. The objective is to create a dynamic and visually appealing game environment that serves as a refreshing break for software engineers and anyone seeking an enjoyable challenge. Leveraging the Pygame library and embracing Object-Oriented Programming (OOP) principles, Code Quest aspires to transcend traditional gaming boundaries and cater to a wide range of audiences.

This report is structured to provide a comprehensive overview of Code Quest, covering its background, specifications, design, implementation, testing, and evaluation. As we navigate through the project roadmap, each section of this report will shed light on the specific aspects, challenges, and achievements encountered during the development of Code Quest, offering a transparent and insightful perspective into the creative and technical processes that shape this interactive gaming experience.

BACKGROUND

In the realm of software engineering, we, as students, intimately understand the intricate challenges and occasional frustrations that accompany coding endeavours. Motivated by this first-hand awareness, our project, Code Quest, unfolds against the backdrop of software engineering. The game showcases falling objects symbolising familiar icons encountered in the daily tasks of engineers—ranging from rubber ducks and Python logos to tick symbols, bugs, errors, and warnings.

This 60-second break serves as a deliberate and refreshing pause for engineers, providing them respite from the intricacies of ongoing projects. The falling objects, symbolic of their professional realm, not only offer a delightful diversion but also maintain an engaging connection to the theme, infusing an enjoyable element into their workday.

However, the inclusive design of Code Quest extends beyond catering exclusively to engineers; it is meticulously crafted to be accessible and enjoyable for individuals of all backgrounds who appreciate a stimulating challenge.

Game Logic And Rules

1. Objective:

- Players must navigate through a series of levels each within a 60-second timeframe to capture as many items as possible to increase points, all while maintaining alive to progress onto the next level.
- Movement is controlled via keyboard events—either the arrow keys (left and right) or the 'a' and 'd' keys. With players having the option to pause the game via the space key.

2. Items

The game environment presents six distinct items, divided into two groups. In the first group, players encounter items designed to augment their points, providing a positive boost to the overall score. However, the second group introduces items with a dual impact—capturing them not only reduces points but also inflicts damage. If a player obtains 30 damage points, they will lose a life.

Figure 1: Positive Items.

Figure 2: Negative Items.

3. Scoring System:

Positive Items:

- Tick: +1 Point
- Python: +5 Points
- Duck: +10 Points

Negative Items:

- Warning: -1 Points, 1 Damage
- Error: -5 Points, 10 Damages
- Bug: -10 Points, 30 Damages

4. Three-Life System:

- Players start with three lives at the beginning of the game.
- 30 damage points are responsible for the loss of one life.
- Losing all lives will conclude the game.

Figure 2: Three-Life System.

5. Level Progression:

The game comprises of three levels, to move up a level the player must have at least one point remaining life. In order to win, the player must finish all 3 levels.

6. User Registration and Login System:

Code Quest offers players the ability to register new accounts and log in to track their gaming journey. This feature enhances user engagement by allowing players to save their game progress and statistics. By storing data in a database, the game ensures that players can seamlessly resume their adventures across different gaming sessions. This feature not only provides convenience but also adds a personalised touch to the gaming experience, making each player's journey unique.

7. History:

Players are able to track and compare their score to their peers via a history menu.

SPECIFICATIONS AND DESIGN

Functional Requirements

We carefully considered the functional requirements for our MVP in order to guarantee a simple yet entertaining gameplay experience. In addition to being able to move left and right, the player also had the vital ability to crash into objects that were falling.

These falling objects were designed to fall steadily in anticipation of the level starting, stay on the game board for exactly three seconds after they fall, or explode if they pose a threat. We used the concepts of OOP to create an abstract Menu class that contained winning and losing conditions as well as a pause function. Then, we added a progression aspect by adding player movement animation and levelling up, which improved the game's aesthetic appeal.

We integrated statistics shown on the gaming board, dynamically reflecting the player's stats, and introduced registration and logging functionalities to raise gamification. This not only allowed us to store player stats, but it also elevated our game from a Minimum Viable Product to a Minimum Lovable Product, allowing users to compare their accomplishments, measure progress, and contribute to a more personalised and enjoyable gaming experience. Incorporating gameplay screenshots would clearly demonstrate our game's menus, statistics, and overall immersive features.

1. Responsive Navigation

Code quest allows the player to seamlessly control their character's movement using the "a" and "d" key, or left and right key arrows. This approach not only enhances the overall gaming experience but also offers a more intuitive and efficient alternative to traditional mouse navigation. The responsive navigation ensures that players can quickly and accurately manoeuvre their character, responding to the dynamic challenges presented in the game.

The implementation of this functionality is illustrated in the code snippet above (Figure X), where the move method dynamically updates the player's position (rect) in response to keyboard inputs. It enables animated leftward movement upon pressing the 'A' key or left arrow key and animated rightward movement upon pressing the 'D' key or right arrow key. Importantly, this implementation guarantees that the player remains within the boundaries of the game board.

Expanding on the navigation functionality, players are empowered with the capability to move left and right, allowing for collisions with falling items. This interaction triggers consequential outcomes, determining whether the player earns damage or positive points and influencing their progression within the game. This pivotal aspect of the game mechanics significantly contributes to the overall gameplay and is an incredible vital function.

This requirement and its interactions can be found in player.py, main.py, and falling_items_factory.py

2. Menus

The game is equipped with eight distinct menus, each meticulously crafted to serve a specific purpose, enhancing user interaction, and contributing to the overall gameplay experience. These menus include the Starting Menu, Game Over Menu, Winning Menu, Pause Menu, Registration Menu, Login Menu, History Menu, Credits Menu and Instructions Menu.

The Starting Menu acts as a comprehensive hub, offering a range of functionalities such as registration, login, access to game history, instructions for gameplay, and acknowledgement of credits. It provides users with the ability to sign up, track their progress, and gain a comprehensive understanding of the rules and logic governing the game.

The Game Over Menu dynamically surfaces when the player exhausts all available lives, presenting two pivotal options: "Play Again" and "Exit." This menu allows players to swiftly make decisions, providing a seamless transition for those eager to replay the game or conclude their gaming session. The Game Over Menu serves as a critical element in maintaining user engagement and providing a clear path for the next steps in the gaming journey.

Similarly, the Winning Menu makes its appearance on the screen upon the successful completion of all three game levels. mirroring the structure of the Game Over Menu, it offers the player the option to either play again or exit the game. The Winning Menu plays a crucial role in celebrating the player's achievements, fostering a sense of accomplishment, and providing a clear pathway for subsequent gameplay.

The Pause Menu emerges when the space key is triggered, allowing the game to be temporarily suspended. This feature adds flexibility to the gameplay, accommodating the players' schedules and preferences. Whether it's a quick break or a pause in the action, the Pause Menu contributes to a user-friendly experience, ensuring that the game aligns with the players' needs.

All menus are inherited from the class Menu found in menu.py with any buttons created using the class Button from button.py. They're functionality is defined in main.py

Each of these menus serves as more than mere conveniences; they are vital components that enrich the overall gaming experience. The Registration and Login Menus not only provide access but also enhance security and personalization. The History Menu fosters a reflective experience, allowing players to assess their progress. The Starting Menu sets the tone for user engagement, offering a centralised space for various functionalities. The Game Over and Winning Menus guide players seamlessly through post-game decisions. The Pause Menu adds a layer of flexibility, adapting to the players' preferences. The Instructions Menu explains the gameplay mechanics, detailing the point-scoring system and potential damage sources, while the Credits Menu acknowledges and summarises the individuals behind the game's creation, fostering a personal connection between players and the development team. Together, these menus form an indispensable part of Code Quest, ensuring a seamless and interactive interface that elevates the gaming experience.

3. Items and Points

The inclusion of a comprehensive scoring system within Code Quest is absolutely essential, serving as a fundamental component crucial for player engagement and progression. This dynamic system goes beyond a mere tally of points, playing a pivotal role in motivating players and shaping their experience throughout the game.

4. Timer

Implementing a timer was a must; it not only sets a timeframe to complete the game but it defines level progression. And adds a dynamic element, making the game feel more complex and fun.

In the timer.py file, a Timer class is defined, inheriting from the abstract base class Stats and the abstract class ABC. This class manages the game timer and is responsible for updating and drawing the timer on the game board. The draw method utilises the pygame library to render the timer text.

In the main.py file, the Timer class is instantiated as part of the game loop in the run function. The timer's initial duration is set to timer_seconds, and its progression is monitored during the game loop. When the remaining time reaches zero, the game checks for a winner, level-ups the player and updates the display accordingly.

The Timer class is part of a broader system that includes other game elements such as player movement, falling items, and menus. The game's functionality is orchestrated within the loop, with conditional checks for winning, losing, and pausing the game. The timer's role is crucial in influencing the player's strategy, adding a time-based challenge to the overall gaming experience.

5. Levels

The incorporation of a level system in Code Quest stands as a paramount functional requirement, seamlessly intertwining with various game elements to elevate the overall gaming experience. Much like the responsive navigation, menus, scoring system, and timer, the implementation of levels is a cornerstone in shaping the dynamic and engaging nature of the game.

Catching falling items alone would make the game very monolithic and boring, in order to add a purpose a level with a history level is a must. The code required for this function is incorporated in almost all Python files, however, it's physical attributes can be found in level.py and it is run on main.py. More information on how each line of code works can be found in the Python document using doc strings.

6. Player

It was essential that the Code Quest game implemented a Player class with the following functionalities:

Player Movement:
- The player must be able to move left and right within the boundaries of the game board.
- Player movement must be responsive to keyboard input.

Player Animation:
- The player's sprite must animate smoothly when moving left or right.

Player Interactions:
- The player must be able to collide with falling items on the game board.
- Points-falling items increase the player's points while damage-falling items decrease points and life.

Game State Management:
- The system must manage the player's life, points, and level.
- The game should identify if the player wins or loses based on specific conditions.

Database Integration:
- The system must update the player's statistics in the database, including points, life, and level.

Player Reset:
- The player should be able to reset their position and statistics in the game.

These functional requirements are handled in the Player Class, defined in player.py. The Player class represents a player entity in the game. It includes attributes and methods to handle the player's movement, interactions with falling items, scoring, and game state.

Usage

An instance of the Player class is created within the game loop, where the initial position, game board instance, falling items group, and player name are passed as parameters into the class. The player name is passed in as a variable, which is saved from the user registration and login.

The game loop calls Player methods such as move, check_falling_item_collision, and others to handle player interactions and game logic.The game functionality can retrieve player statistics by using getter methods like get_lives, get_points, and get_level. These are then displayed on the gameboard through the functionality of other classes such as: Life, Level, Points; as the Player instance is passed into these classes, meaning the output of these player class methods is readily available within these other classes.

The player's statistics are updated in the database within the game loop using the update_db method. These can then be viewed by the user when accessing the history menu. Within the game loop, the player's position is reset on level up using the reset_player method. Whilst the player's statistics are reset using the reset_player_stats method when the game is reset.

7. User Registration and Login

Registration

The Code Quest registration process is user-friendly and secure. Players input a desired username and password, subject to specific criteria (3-20 characters for the username, letters, numbers, and underscores only; password requirements include a minimum length of 8 characters, at least one uppercase letter, one lowercase letter, one digit, and one special character). After validation, the system securely hashes the password before storing user credentials in the database.

Login

Returning users access their accounts securely through the login functionality. The system prompts them to enter their registered username and password. It retrieves the hashed password associated with the username from the database and compares it to the entered password. A match grants access, while a mismatch prompts the user to retry. This process ensures a secure and efficient login experience.

8. Database Integration

Code Quest utilises a MySQL database and leverages the MySQL Connector to store essential user information and game statistics. The database schema consists of two tables: users and game_statistics.

1. Users

Columns:
- user_id (Primary Key): Unique identifier for each user.
- username: The chosen username of the player.
- password: The securely hashed password of the user.

After a successful registration, the system inserts a new record into the users table, including the user's unique ID, chosen username, and the securely hashed password.

2. game_statistics

Columns:
- statistics_id (Primary Key): Unique identifier for each set of game statistics
- user_id (Foreign Key): Links to the corresponding user in the users table.
- life: Represents the number of lives the player currently has.
- points: Records the player's score.
- level: Indicates the current level reached by the player.

Upon registration, an initial set of game statistics is created for the user, with default values for life, points, and level.

After gameplay, the system updates the game_statistics table to reflect the player's progress. This includes updating the number of lives, points earned, and the current level reached.

When a user logs in, the system retrieves the game statistics from the game_statistics table based on the user's ID and shows the 8 best scores based on the statistics saved.

If a user decides to play again after completing a game or chooses to replay Code Quest, the system updates the game_statistics table with the new game data, ensuring that the player's progress is accurately reflected.

Here are examples of data:

user_id	username	password
1	Thunder_007	6e2f6e…
2	BlazeRunner_24	cd61e8…
3	Vortex_5Gamer	839432…
4	Phoenix_FuryX	608631…
5	Quantum_Star_9	68104e…
6	StealthStriker_13	c48153…
7	Nitro_Nova2022	6e2f6e…
8	Riot_Raptor42	608631…
9	DynamoDragon_88	608631…
10	HavocHunter_55	839432…

Figure 3: User data.

statistics_id	user_id	points	life	level
1	1	100	80	3
2	2	150	75	2
3	3	200	60	4
4	4	50	90	1
5	5	300	70	5
6	6	120	85	2
7	7	180	50	3
8	8	90	15	1
9	9	250	40	4
10	10	160	55	3

Figure 4: User data.

Non-functional Requirements

1. Colours and Size:

As programmers, we deemed it paramount to recognize the diverse needs of players, especially those with ADHD and colour vision deficiencies. Code Quest is intentionally brief, featuring an unpredictable falling pattern that ensures each session is consistently fresh and challenging.There is a clear contrast between the background colour and items to avoid confusions and each icon and font has also been selected and given appropriate sizing to ensure readability is not a problem and the design is distinct and clear.

2. Sounds:

The implementation of sounds was crucial; it contributes to the atmosphere of the game and creates a more engaging and realistic experience for players, increasing satisfaction. In our game, each item when caught has a different sound, this is incredibly valuable as it confirms successful or unsuccessful action. Implemented through a decorator in the decorators directory under sound.py.

3. Instructions:

Code Quest's starting menu includes vital instructions, serving as a prerequisite for a smooth gaming experience.
Clear guidance on navigation, game mechanics, objectives, and controls fosters an enjoyable and immersive gaming environment.

Design and Architecture

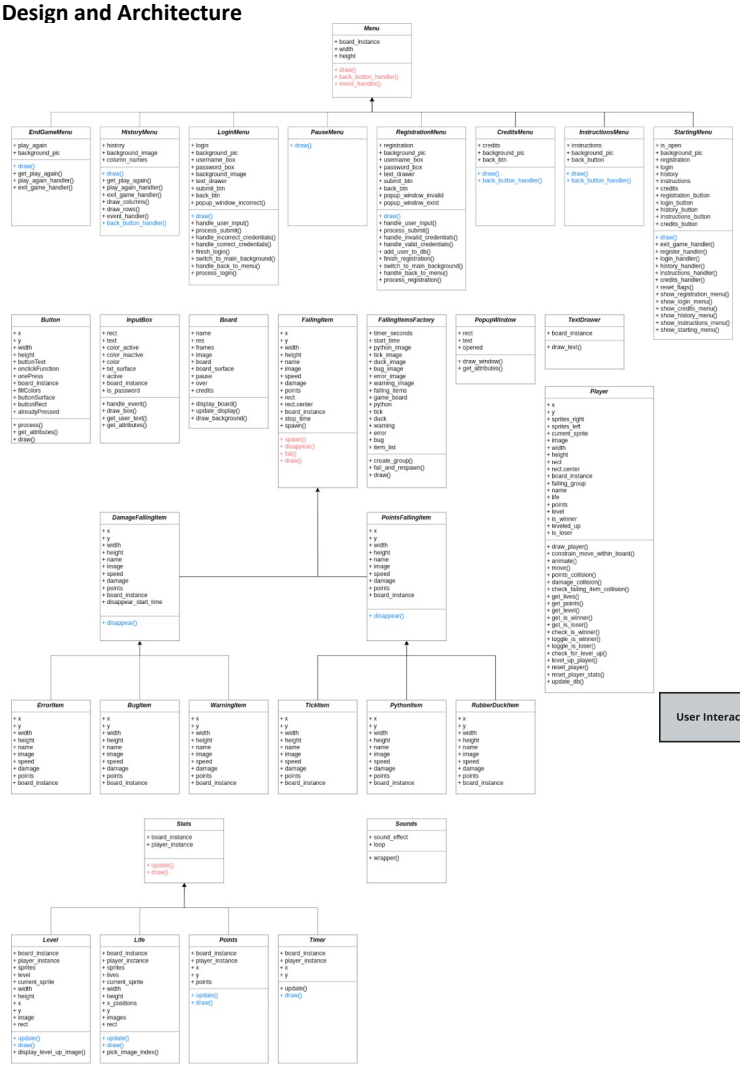


Figure 5: UML Diagram.

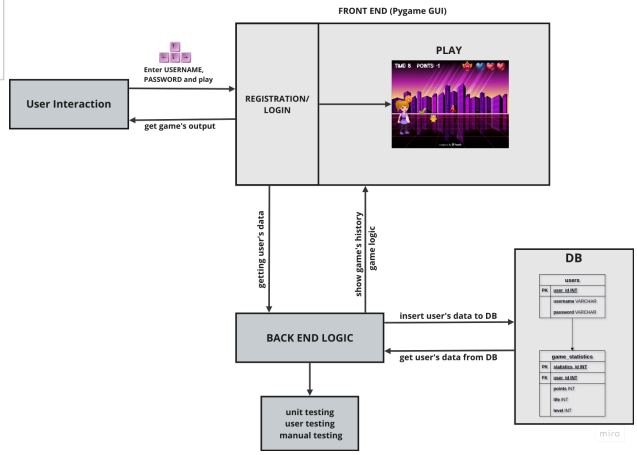


Figure 7: Architecture Diagram.

The UML diagram for our project visually represents the classes, their attributes, and methods. Each class is depicted with its specific attributes and methods, illustrating the structure of our software system. Inheritance relationships are indicated by arrows, showcasing the hierarchy and connections between parent and child classes. Abstract methods, denoted in red text, signify essential functionalities outlined in abstract classes. Blue text indicates the implementation of these abstract methods in the respective child classes, highlighting the concrete realisation of these abstract functionalities in the system.

Implementation and Execution

Development Approach and Team Member Roles:

To nurture a sense of unity and coherence within the team, each team member was assigned specific roles and weekly tasks. Kseniia, in her role as Project Manager, oversaw all project coordination and checked the requirements, and played a key role in the technical implementation. She set up the database with MySQL, creating a robust foundation for data management, and added a layer of security by implementing the registration and login functionalities. Kseniia's contributions extended to crafting essential classes like Button, InputBox, and TextDrawer. Collaboratively, Sadaf, Kseniia, and Sandra managed documentation. Sadaf, in addition to documentation duties, brought creative aspects by suggesting the falling items game concept, implementing the DamageFallingItems class, and working on the timer and game over menu. Klaudia led the testing efforts, creating foundational classes like Board and falling item abstractions, contributing to various game features like endgame, pause, and winning menus as well as the stats bar. Kerri, as the Scrum Master, facilitated communication and made substantial technical contributions with the Player class, player collision methods, scoring system, and history menu. Natalia collaborated on the main menu development with Kseniia and Kerri and implemented game mechanics with the PointsFallingItems class and FallingItemsFactory class. Sandra focused on documentation, presentation and play_collision.

A preparatory meeting preceded each CFG learning session, Monday to Thursday, between 18:00 – 18:30. During these sessions, the team discussed upcoming tasks, progress updates, and addressed any questions or challenges team members might have been facing. This daily stand-up was set up in response to our reflections on our previous group project, which highlighted that our communication needed improvement. By actively participating in these discussions, team members could connect and collaborate, ensuring a smoother and more effective working dynamic, and improving our project outcome.

Task management was streamlined through the use of Jira, where tasks were posted and tracked. This system allowed team members to easily identify backlog tasks, monitor their assignments, and stay informed about the progress of each team member, promoting transparency and accountability within the team.

Upon task completion, members initiated a pull request on GitHub. The submitted work underwent immediate review by team members, and constructive criticism was given. This process not only exposed team members to each other's tasks, fostering a valuable learning experience through reviewing works but also encouraged collaborative efforts in producing tasks that met the standard set by CFG.

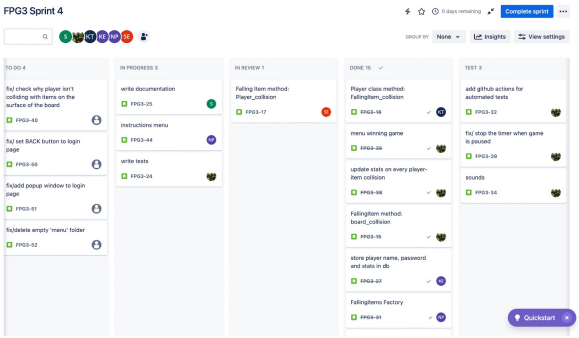


Figure 7: Jira Board.

Tools and Libraries

Game functionality

- **Pygame**: Pygame simplifies graphics management, sprite handling, and user input events, providing an intuitive interface for various game development tasks. Its flexibility and compatibility with Python make it an ideal choice for creating an engaging and visually appealing game.
- **Mysql.connector:** Used to establish a reliable connection with MySQL databases, ensuring smooth database operations for storing and retrieving crucial game-related data, including user and game statistics.
- **Datetime & Time:** Used for effectively managing time-based components, specifically used for tracking creation timestamps of in-game items and controlling their lifecycle.The 'time' feature was smoothly incorporated into our game development workflow, allowing for precise administration of time-related operations as well as serving as a critical tool for establishing timers within the game.
- **Random:** Enhances gameplay by introducing unpredictability through random falling items. The 'random' component diversifies player experiences, adding distinct and interesting scenarios.
- **Sys:** Incorporation for accurate and controlled system-level functions. Enables graceful exits from the game, optimising resource and process management for a smooth termination process and improved user experience.
- **abc (Abstract Base Classes):** Our team was able to construct abstract classes as key blueprints, establishing consistent structure and interfaces for linked game items. Enhances code consistency, lowers error risks, and facilitates a well-organised hierarchy, improving game maintainability. The abc function created the core inheritance in our project.
- **Logging:** Handles exceptions and generates log messages for debugging purposes. Records relevant information about errors or unexpected events during game execution.
- **Re(Regular Expressions):** The re (Regular Expressions) component was useful in the development of our game because it was used to evaluate and guarantee that passwords met specified requirements or forms within the game.
- **Hashlib:** Utilised for hashing input passwords, enhancing security for storage and verification in the game.

For tests

- **Unittest:** for implementing unit tests within our game
- **Coverage:** The coverage library is utilised to measure the percentage of code executed during testing. Code coverage measurements help identify untested code parts, ensuring a comprehensive testing strategy and enhancing the overall software quality.
- **Numpy:** Used to compare 3D matrices for visual validation of rendered images with expected images in the game.
- **io and sys:** Employed to capture print statements as string buffers (StringIO) during testing. Useful for functions with output, like those with a sounds decorator. Captured output is redirected for easy comparison with expected outputs, with standard outputs restored post-testing.
- **Os:** Used for managing file paths, ensuring the test runner looks for imported files from the root directory, ensuring consistency in referencing across modules and test cases.

Other:

- **Jira:** for project management, enabling efficient tracking of tasks, issues, and project progress.
- **Slack:** used daily for quick communication, sharing thoughts, and exchanging screenshots.
- **Diagrams.net - draw.io:** for creating UML diagrams of our project
- **Google doc:** employed for collaborative documentation and note-taking
- **Git:** In our project, Git is the linchpin for version control and collaborative development. We strategically use feature, fix, and test branches to ensure concise changes, making it easier to manage, review, and integrate into the main codebase. This branching strategy, coupled with thorough code reviews, establishes a feedback loop that fosters continuous learning and improvement for us as developers.

Implementation Process

In our initial team meeting, we collectively brainstormed whether to embark on an API-based project or opt for a Pygame implementation. Faced with a divided group, we resolved the dilemma through a vote, and the majority consensus leaned towards Pygame. Subsequently, we delved into researching various Pygame possibilities, ultimately settling on a game concept where players engage by catching falling items through responsive movement.

The following week marked the commencement of our coding journey, laying the foundation with essential components such as the falling item class and player class. Over the subsequent weeks, we iteratively built upon this foundation, continually refining and expanding our code until we successfully created the envisioned product.

Because we have used an agile method of working together, discussions regarding the navigation of our project were made incredibly easy. The agile approach allowed us to adapt and respond to changing requirements seamlessly. Regular meetings, short development cycles, and continuous feedback loops facilitated efficient communication within the team. This iterative process enabled us to incorporate valuable insights from each team member, ensuring that the evolving product met both our internal standards and the dynamic needs of the project. As a result, our use of agile methodologies not only streamlined decision-making but also fostered a collaborative environment that contributed to the successful development of our Pygame-based game

Implementation Challenges

We encountered minimal challenges throughout our project, with the primary hurdle being the busy schedule of one team member outside of the course, leading to delayed responses to our messages. This occasionally created confusion within the group regarding her progress. Despite this, she exhibited remarkable diligence and promptly caught up with the workload. Apart from this, our team collaborated intensely, leveraging direct messages, Slack, and Zoom to successfully accomplish the tasks outlined in the Jira sprint.

Given that PyGame was a new experience for us, team members regularly shared valuable resources to support each other, fostering a supportive environment where everyone felt comfortable seeking assistance. While navigating the intricacies of PyGame, we faced some specific challenges, such as creating a factory for falling items and managing state updates that could impact the game loop, including back buttons. The dynamic nature of the game presented a unique set of obstacles, especially concerning state management. Delving into the implementation of newly absorbed Object-Oriented Programming (OOP) principles provided an excellent learning opportunity, contributing to the overall growth and success of the project.

Future Development

We are incredibly proud of the game we have created, however, given if we had more time, we would incorporate an on-screen menu, allowing users to access instructions at their convenience whilst playing the game. Additionally, we aimed to introduce more levels, each featuring altered behaviours of falling items to intensify the game's challenge.

Moreover, our aspirations extended to a climactic final level. Upon successfully completing the initial three levels, players would face a unique coding challenge with an AI version of Kenny acting as the formidable final boss. This innovative addition would not only test their gaming skills but also engage them in a coding duel, adding a distinctive and intellectually stimulating element to the game's conclusion.

To enrich the player experience and offer real-time insights, game statistics like lives, points, and level will dynamically update in the database as users play. This ensures accurate tracking of achievements and enables personalised gaming experiences based on individual progress.

Finally, we would also add an option to re-enter previously saved games if the user wishes to.

Testing and Evaluation

Introduction

This document provides an overview of the testing procedures, strategies, and tools used in the testing phase of the project. The project employs a comprehensive testing suite to ensure code quality, functionality, and maintainability.
Strategy: we are aiming for at least 95% coverage in total. Unittests for models cover every aspect of creating them, from initialization through animation, reaction to changing state in a game, to drawing and updating. Db tests are testing if the correct queries are made, and proper exceptions are thrown and catched.Due to time constraints we sadly resigned from creating a testing db and mocked it instead.Also, manual tests ensured that everything works together smoothly, and constant user's feedback on gameplay, gave us the possibility to expand our MVP to the point where the game is scalable and easily maintainable.

Test Suites

1. **Unittests:** The project has over 160 unittests written using the unittest module in Python. These tests cover various aspects of the codebase, including unittests for individual functions, and classes.
2. **GitHub Actions Integration:** Tests are automatically executed on every push event for the remote repository using GitHub Actions. This setup ensures that developers receive instant feedback regarding the alignment of their changes with the expected outcomes. The GitHub Actions workflow runs the test suite, ensuring that the changes meet the project's testing standards before merging.
3. **Manual Testing:** Manual testing was performed throughout the iterative creation of the project. It involved exploratory testing, and verifying the project's functionality against specified requirements.
4. **User testing:** We introduced user testing throughout the whole process of making a game, starting from UI to the final points of actual gameplay. Due to user feedback, we quickly implemented sounds and added a history menu so the game stays attractive and makes a whole, ready-to-ship, product.

Test Execution

1. **Test Runner:** Locally, the test suite is executed using the test_runner script, which triggers the execution of all test cases within the tests directory. This facilitates running the complete test suite in a streamlined manner.And there is a test coverage report while running test_runner, so we get constant feedback on tests coverage and quality.
2. **Continuous Integration:** The project uses continuous integration practices to automatically run tests whenever new code is pushed. This ensures that the codebase remains stable and functional at all times.

Conclusion

To sum up, our project not only effectively fulfilled the set objectives outlined by both Code First Girls and our internal standards but also resulted in a fully operational and functional game crafted through an agile teamwork approach utilising Pygame. Throughout the development process, we adeptly navigated and overcame challenges, delving into previously unexplored aspects of Python. This exploration allowed us to demonstrate the skills honed during our tenure at Code First Girls and significantly broaden our understanding of the language.

Despite our desire to further enhance our game, we take great pride in what we have accomplished. The project stands as a testament to our collective capabilities and the valuable knowledge gained, reflecting the success of our efforts in both skill application and knowledge expansion.