REPORT – PHASE 3: TESTING

Unit Testing:

In the iterative development process of our game, we embarked on a crucial phase aimed at enhancing code cohesion and ensuring robustness through comprehensive unit testing. This report delineates the key features we identified for unit testing, the refactoring strategies employed to augment code modularity, and the comprehensive unit testing executed across various components of our game.

Feature Identification and Refactoring:

During the second phase of development, we observed a lack of cohesion among various classes, leading to code complexity and potential maintenance challenges. Consequently, we opted for refactoring to enhance code clarity and modularity.

In the Node class, functions were isolated for individual testing to assess their independent functionality. The refactoring process focused primarily on the Entity package. We identified that responsibilities such as collision box management, animation handling, and pathfinding were intertwined within this class, leading to decreased clarity and increased coupling between classes. To address this, we extracted these responsibilities into separate classes. Collision boxes, animations, and game images were assigned their own classes, allowing them to function autonomously and reducing inter-class dependencies. This modular approach not only improved code clarity but also facilitated easier maintenance and extensibility in the long term.

Comprehensive Unit Testing:

Following the refactoring process, comprehensive unit testing was conducted across various components of our game to ensure their individual functionality and independence from other entities.

Collision boxes, animations, and game images underwent rigorous unit testing to validate their standalone functionality and ensure they operated autonomously without reliance on other components.

Game state classes, excluding the play class, were subjected to thorough unit testing to validate their logical operations and rendering functionalities. The modular design ensured minimal interaction between different game states, facilitating effective unit testing.

UI components, including defeat, menu, pause, and win buttons, were meticulously tested for competent interaction on the screen, ensuring seamless user experiences.

Keyboard and mouse listeners were scrutinized through comprehensive testing to verify their responsiveness and accurate interpretation of user input.

Specific classes such as the Knight class had unit tests created for manipulating its score, reward count, and for checking its death state. Additionally, the Board class was also subjected to rigorous unit testing to validate its individual functionalities and adherence to expected behaviors. Both sets of tests were possible due to their relatively isolated operations.

In conclusion, our endeavor to enhance code cohesion through strategic refactoring and rigorous unit testing has significantly bolstered the robustness and maintainability of our game. By isolating and thoroughly testing individual features and components, we have mitigated potential risks and laid a solid foundation for future iterations and enhancements. This meticulous approach underscores our commitment to delivering a seamless and enjoyable gaming experience to our users.

Integration Testing:

Our pursuit of a seamless gaming experience led us to conduct integration testing, aiming to validate the interactions among different components within our game system. This report outlines the crucial interactions identified and the integration tests devised to ensure their effectiveness.

Key Interactions and Integration Testing:

Integration testing was pivotal in assessing the interplay between various modules of our game system. The Play class, serving as the cornerstone game state module, underwent rigorous testing to validate its interaction with essential components.

Testing the reset functionality of the Play class revealed its ability to wipe the board and recreate entities seamlessly, ensuring a consistent game state transition. Additionally, interactable entities, both player-controlled and non-player characters, underwent testing to validate their interactions with other game components.

Integration tests specifically focused on interactions between the Knight character and other entities, ensuring the manipulation of scores and game mechanics occurred seamlessly. Notably, the interaction between the Goblin and Knight classes underwent thorough testing to validate pathfinding and movement mechanics. These tests confirmed the Goblin's ability to navigate towards the player within the game environment effectively.

Furthermore, integration tests were devised to assess the interaction between the Knight class and the Play state class, responsible for managing the game state. These tests ensured that player inputs were appropriately interpreted, resulting in responsive and intuitive character movement within the game world.

The integration testing conducted on various interactions between game components has validated the seamless functionality and compatibility of our system. By ensuring effective communication and interaction between different modules, we have elevated the gaming

experience for our players. This meticulous approach underscores our commitment to delivering a polished and immersive gaming experience, setting a solid foundation for future enhancements and iterations.

Ensuring Quality of Test Cases:

To guarantee the reliability and effectiveness of our test cases, we implemented several meticulous measures aimed at upholding quality throughout the testing process. One key initiative involved the systematic removal of unused classes and methods, ensuring that our testing suite remained focused and uncluttered. This not only streamlined our testing efforts but also eliminated potential sources of confusion or ambiguity.

Furthermore, we conducted a thorough review of our test cases to identify any instances of redundancy or inefficiency. Through this process, we were able to consolidate overlapping test cases, reducing duplication and improving overall maintainability. This allowed us to achieve a more comprehensive coverage of critical functionalities while minimizing the risk of overlooking important test scenarios.

Additionally, we placed a strong emphasis on the quality of our assertions within test cases. Each assertion was scrutinized to ensure that it accurately verified the expected behavior of the system under test. Where necessary, assertions were strengthened to cover edge cases and potential failure scenarios, providing robust validation of system functionality.

By implementing these measures, we were able to enhance the overall quality and effectiveness of our test suite, ensuring that it provided thorough validation of our game system. This proactive approach to quality assurance not only minimized the risk of defects slipping through undetected but also instilled confidence in the reliability of our software.

Line and Branch Coverage Analysis:

Our commitment to quality assurance yielded tangible results in terms of line and branch coverage. Initially, our testing efforts revealed poor line and branch coverage, hovering below 30%. This underscored the need for comprehensive testing to identify gaps in our test suite. However, through rigorous refinement and strategic refactoring, significant progress was achieved. Subsequently, 90%-line coverage and a 71% branch coverage indicates substantial improvement. This enhanced coverage not only mitigated potential risks and vulnerabilities but also bolstered the reliability and resilience of our game system. By systematically addressing areas of low coverage and continuously monitoring our testing efforts, we have established a solid foundation for delivering a polished and immersive gaming experience to our users. Additionally, it's worth noting that the branch coverage was lower, but this was expected due to certain branches being difficult to replicate through unit or integration testing alone. For example, it's hard to achieve 100% branch coverage for the play class because it involves hard-to-replicate scenarios such as a knight being near a goblin, a goblin touching the knight's collision box, etc.

Features Not Tested:

While our testing efforts aimed to achieve comprehensive coverage of critical functionalities, there were certain features and code segments that posed challenges for test coverage. One notable challenge was the replication of complex integration scenarios, such as the interaction between the Knight and Goblin entities when positioned opposite each other on a tile. This scenario required intricate setup involving both entities moving simultaneously, which proved difficult to replicate in a controlled testing environment. Similarly, certain features, such as player movement requiring keyboard input, were challenging to test in isolation due to their dependence on external factors like user input. As a result, testing focused primarily on the core game logic, with less emphasis on rendering aspects for most entities and play states. Additionally, while getters and setters were not explicitly tested, their functionality would be implicitly validated through the testing of components that interact with them. Given their ubiquitous nature and minimal logic, testing these accessors was deemed unnecessary, as any issues would likely manifest as exceptions during the execution of other test cases. Despite these challenges, our testing approach prioritized thorough validation of interactions between critical components, ensuring the reliability and functionality of our game system.

Lessons Learnt:

Writing and running our tests provided valuable insights and lessons that have informed our development process. One of the key takeaways is the importance of adhering to the principles of high cohesion and low coupling in software design. We learned that designing modular and cohesive components facilitates easier testing and maintenance, while reducing dependencies between modules minimizes the risk of unintended side effects and improves code comprehensibility.

Furthermore, we recognized the necessity of making changes to production code to improve testability. Refactoring code to extract functionality into separate classes or methods not only enhances testability but also promotes code reuse and scalability. By embracing iterative development practices and embracing changes to production code, we were able to iteratively refine our tests and production code, leading to a more robust and reliable game system.

Moreover, writing and running tests highlighted the importance of thorough validation of critical functionalities. It underscored the need for comprehensive testing strategies that cover both individual components and their interactions. This approach allowed us to identify and address potential issues early in the development process, leading to a more stable and predictable system.

Overall, the experience of writing and running tests reinforced the importance of a systematic and disciplined approach to software development. By embracing principles of high cohesion, low coupling, and iterative refinement, we were able to build a test suite that not only validates the functionality of our game system but also enhances its maintainability and scalability in the long run.

Reveal and Fix Bugs:

Writing and running our tests indeed revealed several bugs and opportunities for improving the quality of our code. One notable issue we uncovered was the concurrent access of trees by different components, which posed a potential risk for future scalability. By detecting this issue early through testing, we were able to address it proactively by refactoring our code to utilize a concurrent deque data structure. This solution not only resolved the immediate bug but also ensured that our code could accommodate future extensions without encountering similar concurrency issues.

Moreover, the process of writing and running tests prompted us to scrutinize our codebase more closely, leading to various opportunities for improvement through refactoring. By refactoring our code, we were able to eliminate redundancy, improve readability, and enhance overall maintainability. This proactive approach to code improvement not only addressed existing issues but also laid a foundation for continued development and evolution of our game system.

In summary, the experience of testing our code not only helped us identify and fix bugs but also spurred continuous improvement in the quality of our codebase. By addressing issues early and adopting a proactive stance towards code quality, we were able to build a more robust and resilient game system that is better equipped to handle future challenges and enhancements.