Brian Hert
Ataya, Ali
CSC 133
22nd, February 2024

Assignment 2 Report

In the initial iteration of the SubHunter game, the code was organized within a single class responsible for managing both logic and rendering. In a bid to incorporate Object-Oriented Programming (OOP) principles, we undertook a refactoring process, segmenting the codebase into multiple classes each with distinct responsibilities. This restructuring aimed to introduce abstraction, encapsulation, and class associations for enhanced modularity and reusability. The code was partitioned into several classes including SubHunter, SubHunterGame, GameBoard, Submarine, and Player, with each class encapsulating relevant data and behaviors. Subsequently, abstraction was implemented by defining clear interfaces for these classes while concealing their internal implementations.

The Player class within the SubHunter game encapsulates the player's actions and shooting behavior. With a focus on abstraction, this class offers essential methods for executing shots, retrieving the shot count, and resetting the count at the commencement of a new game. The takeShot method manages shot increments, targets specific screen coordinates, and updates the shotsTaken count while verifying hits through the GameBoard, facilitated by the SubHunterGame. Leveraging the getGameBoard method from SubHunterGame, the Player class maintains loose coupling with the game state, facilitating flexibility and streamlined maintenance. Encapsulation is employed by privatizing the shotsTaken field and providing getter and setter methods to prevent unauthorized access and manipulation. The resetShotsTaken method serves to reset the shot count.

The Submarine class in the SubHunter game embodies a submarine object positioned within the game grid. It encapsulates details regarding the submarine's location, comprising both horizontal and vertical coordinates. This class leverages the Point class to store position data and utilizes grid width and height parameters for precise placement. The Submarine class features a method named placeRandomly, which randomly assigns horizontal and vertical positions within the specified grid dimensions, ensuring a unique placement for the submarine at the onset of each new game. Moreover, the class provides getter methods, getVerticalPosition and getSubHorizontalPosition, enabling access to the current vertical and horizontal positions of the submarine, respectively.

The GameBoard class in the SubHunter game embodies the game state, encompassing submarine placement, hit detection, and monitoring the distance between player shots and the submarine. It encapsulates the submarine object, tracks the distance from the submarine, and records whether a hit has occurred. Upon instantiation, the class initializes a Submarine object, establishing a relationship between the GameBoard and the submarine. The placeSubmarineRandomly method is responsible for randomly situating the submarine within the game grid. The isHit method permits external components to ascertain if a hit on the submarine has happened, returning a boolean value indicating the hit status. The getDistanceFromSub method retrieves the present distance between a player's shot and the submarine, furnishing vital gameplay data. The checkHit method accepts a Point object representing a shot, compares it with the submarine's position, and adjusts the hit status accordingly. In the absence of a hit, it calculates the distance from the shot to the submarine using the Pythagorean theorem and updates the distanceFromSub field.

The SubHunter class serves as the core component of the SubHunter game. It extends the Android Activity class and manages user interactions, drawing operations, and transitions within the game state. The gameplay revolves around detecting submarines within a grid through shot placements initiated by user input.

As we seek out code smells, we found numerous areas requiring attention. The original code was structured with all game logic and drawing tightly bundled in a single class, which violated the Single Responsibility Principle. This lack of modularity made the code hard to grasp and maintain. Altering one section could inadvertently affect unrelated parts, leading to potential bugs. Furthermore, unrelated methods added to low cohesion, making the codebase difficult to modify.

Throughout the implementation and refactoring phases, we encountered numerous challenges. Dividing the code into multiple classes necessitated careful planning, as determining which classes to separate and which to retain proved difficult. One class we initially attempted to extract was the drawing class. While we believed that isolating drawing functionalities, such as canvas and paint, into a separate class was beneficial, it ultimately resulted in code breakages. Consequently, we preserved the code that didn't require refactoring. Additionally, ensuring the accuracy of interactions among classes and components posed a challenge. Through unit testing and integration testing, we verified each class's behavior and their interactions.

In summary, integrating OOP principles into the SubHunter game has significantly enhanced its design, maintainability, and readability. Through code refactoring to incorporate these principles, the codebase gained modularity, cohesion, and improved readability, thereby simplifying maintenance tasks. This approach also paves the way for seamless expansion, whether by adding new features or addressing bugs.