**Battleship Game Report**

**1. Introduction:**

The goal of this coursework is to implement a text-based version of the classic game Battleship. In this game, two players take turns to guess the coordinates of each other's ships on a grid. The application provides an interactive environment for players to place their ships and engage in strategic battles.

**Running the Program:**

To run the program, ensure you have Python installed on your system. Then, download the provided code file (**battleship.py**) along with the **grid\_size.txt** and **winner.txt** files. Place them in the same directory. Finally, execute the command **python battleship.py** in your terminal or command prompt.

**Using the Program:**

* Upon running the program, it prompts players to input coordinates to target their opponent's ships.
* Players take turns guessing coordinates until one player sinks all of the opponent's ships.
* The game announces the winner and records it in the **winner.txt** file.

**2. Body/Analysis:**

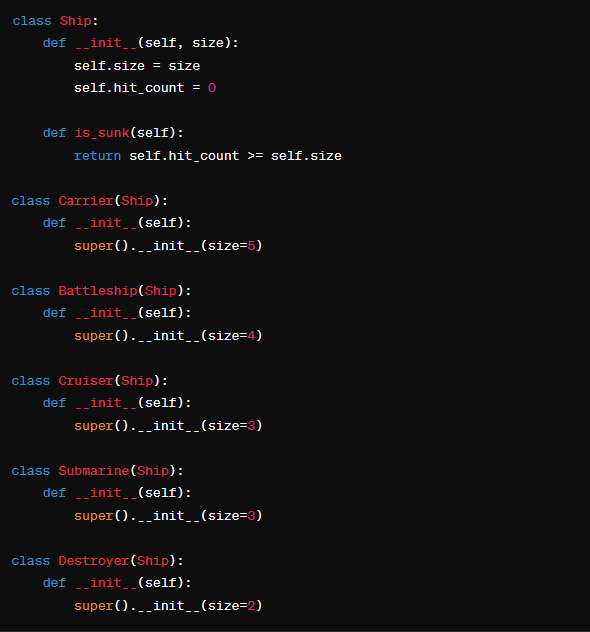
**Ship Classes:**

* The program defines ship classes representing different types of ships, each with its own size.
* The **Ship** class includes methods to track hits and check if the ship is sunk.
* Subclasses like **Carrier**, **Battleship**, etc., inherit from **Ship** and specify their sizes.

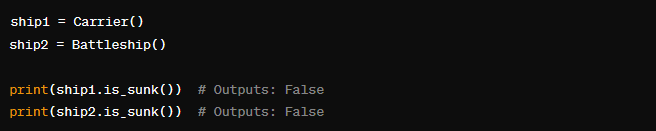
**BattleshipGame Class:**

* This class manages the game state and logic.
* It reads the grid size from a file, initializes player grids and ships, and keeps track of hits.
* Methods like **place\_ship** randomly place ships on the grid.
* **take\_turn** allows players to input coordinates to attack opponent ships.
* The **play\_game** method orchestrates the gameplay by alternating turns until a player wins.

**Polymorphism:**

****

* The **Ship** class defines a common interface with a method **is\_sunk()**.
* Each specific ship class (**Carrier**, **Battleship**, etc.) inherits from the **Ship** class and provides its own implementation of the **is\_sunk()** method.
* Despite having different implementations of **is\_sunk()**, instances of different ship classes can be treated uniformly using the common interface provided by the **Ship** class. For example, you can call the **is\_sunk()** method on a **Carrier** object, and it will behave as expected.

****

Both **ship1** and **ship2** are treated as objects of the **Ship** base class, even though they are instances of different derived classes (**Carrier** and **Battleship**). This demonstrates polymorphic behavior, where objects of different classes can be treated uniformly.

**Abstraction:**

**A computer screen shot of a black screen with white text

Description automatically generated**

* The **Ship** class encapsulates the concept of a ship in the game.
* The internal representation of the ship's state (such as **size** and **hit\_count**) is encapsulated within the class.
* The **is\_sunk()** method provides an abstract interface for determining whether the ship is sunk, hiding the details of how this determination is made.

**Inheritance:**

**A screen shot of a computer program

Description automatically generated**

* The **Ship** class serves as the base class, defining common characteristics and behaviors shared by all types of ships.
* Each specific type of ship (e.g., **Carrier**, **Battleship**, etc.) is implemented as a derived class that inherits from the **Ship** class.
* The derived classes (**Carrier**, **Battleship**, etc.) inherit the properties and methods of the base class (**Ship**). They can also extend or override these inherited properties and methods as needed.
* For example, the **Carrier** class inherits the **size** and **hit\_count** properties, as well as the **is\_sunk()** method, from the **Ship** class. This allows instances of the **Carrier** class to use these properties and methods without having to redefine them.

A black background with white text

Description automatically generated

This code creates an instance of the **Carrier** class and demonstrates how it inherits properties and methods from its base class, **Ship**.

**Encapsulation:**

**A computer screen with white text

Description automatically generated**

* The **Ship** class encapsulates the concept of a ship in the game.
* The attributes **size** and **hit\_count** are encapsulated within the class. They are accessible using dot notation (**ship.size**, **ship.hit\_count**) from outside the class.
* The method **is\_sunk()** encapsulates the logic for determining whether the ship is sunk. It operates on the internal state of the ship object (**self.hit\_count** and **self.size**) and provides a public interface for checking the ship's status.
* Encapsulation helps to achieve data hiding, abstraction, and modularity in the codebase. By hiding the internal implementation details of the class, encapsulation provides a clear separation of concerns and facilitates better code organization and maintenance.
* In the provided code, encapsulation is used extensively across various classes to manage the game state, player grids, ships, and game logic. Each class encapsulates its own set of attributes and methods, providing a modular and well-organized structure to the code.

**3. Results and Summary:**

* **Results:** The program successfully implements the Battleship game in a text-based format, allowing players to engage in strategic battles.
* **Challenges Faced:** Implementing the ship placement algorithm and managing game state transitions posed challenges, but were overcome with careful design.
* **Summary:** This coursework achieved the goal of creating an interactive Battleship game. Players can enjoy the classic gameplay experience in a simple text interface.
* **Future Prospects:** The program could be enhanced with features like graphical visualization, multiplayer support over networks, and more advanced AI opponents for single-player mode. Additionally, improvements in user interface and input validation can enhance user experience.