Design Document

Version 0.1 – 2022.11.4

Created 2022.11.4

Unsinkable - Battleship Concept Game

SECTION 1: PROJECT IDENTIFICATION

Our team's motivation to create the Unsinkable game is to gain a deeper understanding of the inner workings of JavaFX, and making the classic battleship game more accessible through the use of voice-overs announcing each move and by using high contrast UI. We picked Battleship specifically because it follows a grid-like structure similar to the Boggle Assignment and it was a project we could build entirely from scratch by ourselves.

SECTION 2: USER STORIES

Name	Id	Owner	Description	Implementation Details	Priority	Effort
Audio button	1.1	Brian	As a user, I want an option that allows me to turn the voiceover on or off	Create a button that switches voiceover on or off	2	3
Quitting option	1.2	Brian	As a user I want to be able to quit the game whenever I want or return to the main menu	Create a button that takes the user to the main menu	1	2
Save file	1.3	Brian	As a user, I want to a UI that allows me to save my progress so that I can continue where I left off when I return to the software	Create a UI so that users can save progress on their hard drive.	2	3

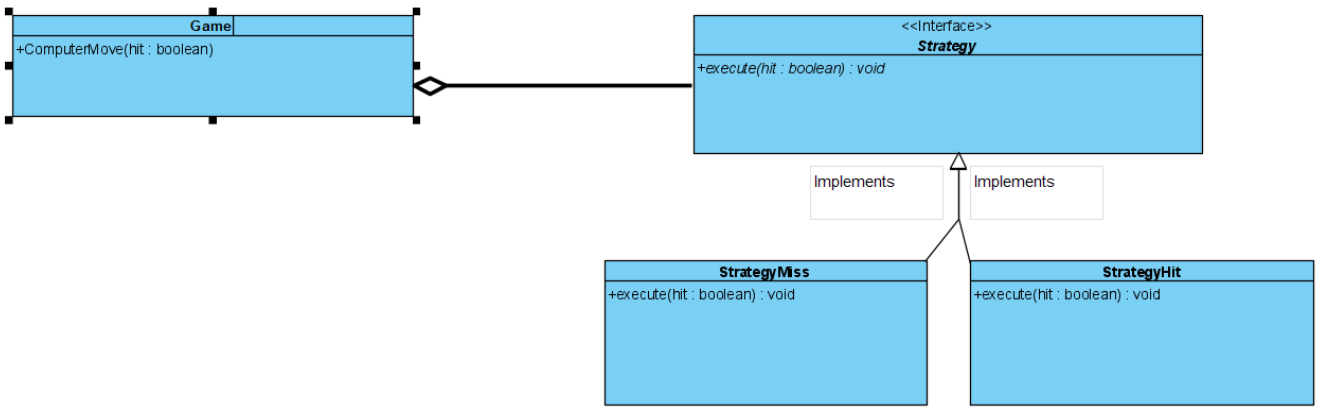
Voiceover	1.4	Aryaman	As a user, I want there to be a voiceover function to make the gameplay more accessible	A voiceover that announces each move	1	3
Summary	1.5	Aryaman	As a user, I want to be able to see my statistics at the end of the game	A screen that shows user statistics: - total hits -hit percentage	1	2
Recent move	1.6	Aryaman	As a user, I want to be able to keep track of my last shot to make it easier to plan where I shoot next	Make the latest shot a different color than the rest	1	1
High contrast colors	1.7	Itgel	As a user, I want a high contrast UI for accessibility purposes	Make the UI with high contrast colors	1	2
Fast Mode	1.8	Itgel	As a user, I want a game mode that allows me to play a faster version of the game	Create a game mode with a 7x7 board and fewer ships that allows faster gameplay	2	2
Computer v Player	1.9	Itgel	As a user, I want a game I can play alone on my own time.	The main mode of the game will be Computer v Player	1	3

SECTION 3: SOFTWARE DESIGN

Design Pattern #1: Strategy Pattern

Overview: This pattern will be used to implement a strategy for a computer move depending on if the previous move was a hit or a miss.

UML Diagram:



Implementation Details: The UML diagram outlines these main components:

- The *Strategy* interface, which includes one method: execute
- The *Game* class with a method *ComputerMove()* that choses a position to target on the board for the next move depending on if the *hit* value is true or false.
- Two classes, *StrategyMiss* and *StrategyHit* implement the Strategy interface.

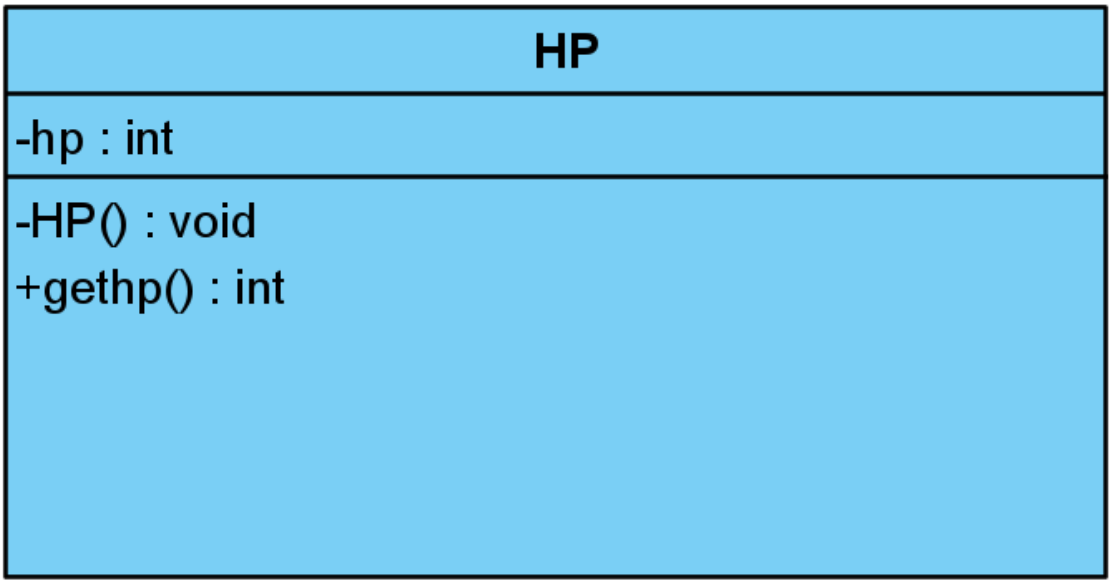
The execution depends on whether the previous move from the computer was a hit or a miss given by the value *hit*. If the previous move was a hit, then the positions adjacent to the hit will be a target of the next move in the class *StrategyHit*. If the move was a miss, then a random position on the board will be chosen as the position to target for the next move by the computer in the class *StrategyMiss*.

Design Pattern # 2: Singleton Pattern

Overview:

We are using this pattern to provide the starting health points of both the player and computer, which are initialized at the game's start. We have decided that having only one instance to store the health points while providing a public access point would make comprehension and accessibility smoother.

UML Diagram:



Implementation Details: The UMl shows three important sections where there is a private attribute, a getter method, and a private constructor that would initially store the health points as a constant.

- **Private static attribute:**

  This is a private attribute that stores the health point and is the only instance of the class. The initialized integer will depend on the number of ships. Since we have two modes where a game of 5 ships and a game of 3 ships are being played. The integer will be set to the total of all ships' hp.

```
  private static int hp;
```

- **Private constructor:**

  To prevent any inadvertent creation of this instance in the other classes.

```
private hp() {}
```

- **Getter method:**

  Will be used to provide public access to the only instance of this class.

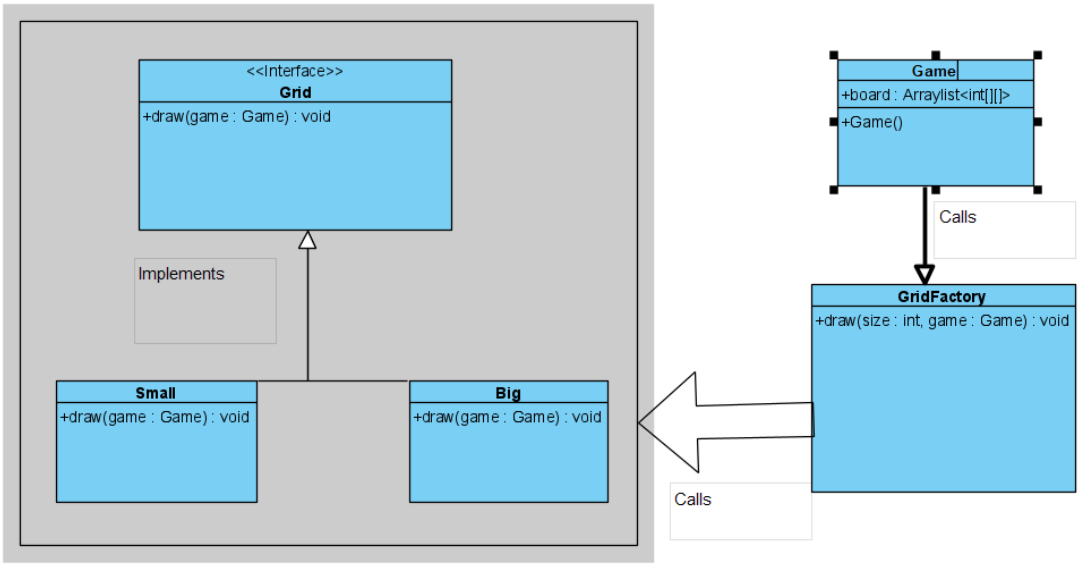
```
public static int gethp(){
     return hp;
}
```

This will be used mostly as a constant where this instance will be called about by methods wanting to update the health points during a game. Moreover, this would be initialized in the Game() constructor and stored in a separate variable for editing the health points.

Design Pattern #3: Factory Pattern

Overview: This pattern will be used to create a big(how big tho) grid or a small grid depending on the player's choice.

UML Diagram:



Implementation Details: The UML diagram outlines these main components:

- The *Grid* interface, which includes the method: draw.
- The main class which has the initializer to call grid factory
- The Grid Factory class which will call the Grid interface to draw a big or small grid depending on the mode the player chose.

The Game() constructor will initialize the grid the game will be played on depending on the Player's selected game mode. Depending on the size, the draw method of the Grid interface will be called on either Small or Big. And these functions will set the board attribute of the game.

SECTION 4: EXPECTED OVERALL PROJECT TIMELINE

Project: Unsinkable

Project timeline

PROCESS	11.9-11.22				11.23-11.30				12.1 - 12.7			
	11.9	11.11	11.16	11.22	11.23	11.25	11.27	11.30	12.1	12.3	12.5	12.7
Creating design document												
Coding back-end												
Mid-coding review												
Coding JavaFx and adding voiceover												
Testing, fixing												
Finalizing												