Sinker Design Document

Date: 04/28/2017

Author: Garrett Engelder

This design document is divided into sections:

1. Introduction and Overview
2. Requirements
3. Use Cases
4. Domain Class and Class Diagram
5. Implementation Details
6. Testing Plan
7. Risks

# 1. Introduction and Overview

## Introduction

This design document has been created to describe the “Sinker” program, which is a simple game that can be played on a desktop computer. This program is a Java GUI-based game called "Sinker." It is a simple game where the player has to guess the location of ships in a 10x10 grid. The game starts with 10 ships hidden in the grid, and the player has 80 tries to guess their location.

The program uses the Java Swing library to create the graphical user interface, with the JFrame class as the main window. The grid is represented by a two-dimensional array of JButtons, and each button has an ActionListener that handles the player's input.

The game starts with the system randomly placing the ships on the grid using the placePieces() method. The method generates a random row and column position for each ship and ensures that the ship doesn't overlap with any other ship already placed on the grid using the overlapsPiece() method.

When a player clicks on a grid button, the ActionListener checks if the button corresponds to a ship location. If it does, an explosion image is set as the button's icon, indicating that the player has hit a ship. Otherwise, a water image is set as the button's icon, indicating that the player has missed. The number of tries left is updated, and the program checks if the player has sunk all the ships or used all their tries.

If the player sinks all the ships before running out of tries, a congratulatory message is displayed, and the player is given the option to play again or exit the game. If the player runs out

of tries, a message is displayed indicating that the game is over, and the player is given the option to try again or exit the game.

## Component Overview

The Sinker program is composed of several major components:

**Game Board**: This component is responsible for initializing the game board and placing the pieces. It also handles updating the game board after each move and checking for game completion.

**GUI**: This component provides a graphical user interface for the game board. It includes a 10x10 grid of buttons and displays the number of tries left and the status of each piece.

Grid Panel: This component holds the 100 grid buttons and arranges them in a 10x10 grid using GridLayout.

**Grid Buttons**: These components are the individual buttons that make up the game board. Each button represents a single cell on the grid.

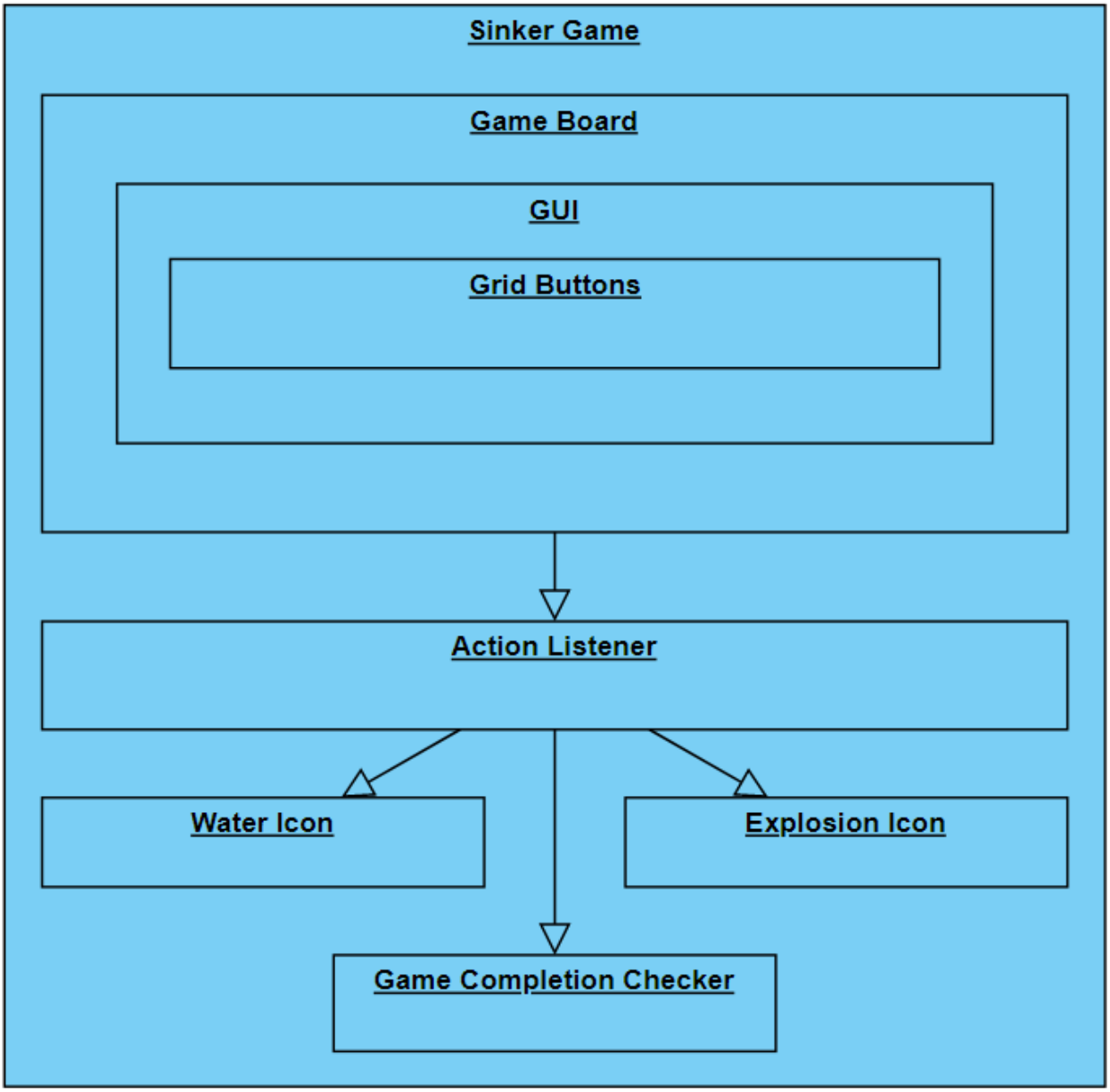
**Action Listener**: This component listens for button clicks and triggers the appropriate response. It checks whether the clicked button is a piece or not and updates the game board accordingly.

**Explosion Icon**: This component displays an explosion icon on the game board when a piece is hit.

**Water Icon**: This component displays a water icon on the game board when a button that does not contain a piece is hit.

**Game Completion Checker**: This component checks if all the pieces have been sunk or if all the tries are used up, prompting the user with a dialog box to play again or exit the game.

Game Disposer: This component disposes of the old game and creates a new one if the user chooses to play again, or exits the game if not.



# 2. Requirements

**Game Mechanics Requirements**

- Allow the player to fire missiles at randomly placed enemy ships.
- The objective of the game is to sink all ships before running out of missiles.
- Display an explosion image when a missile hits a ship.
- Display a water image on a button when the user selects a button without a ship on it.
- Limit the number of missiles the player can fire.

**Graphical User Interface Requirements:**

- Provide a GUI that allows the player to interact with the game.
- Display a game board consisting of a grid of buttons.
- Display a missile image when the player fires a missile.
- Display a missile counter showing the number of missiles the player has left.

**Software Architecture Requirements:**

- Implement the game using an object-oriented design.
- Separate the game logic from the GUI code.
- Use event-driven programming to handle user input and update the game state.
- Use exception handling to catch and handle errors.

**Testing and Quality Assurance Requirements:**

- Conduct unit tests to verify the functionality of the program.
- Conduct integration tests to verify the interaction between components.
- Conduct acceptance tests to verify the program meets the requirements.
- Ensure the program is robust and error-free.

**Deployment Requirements:**

- Provide a standalone executable file for Windows operating systems.
- Ensure the program can be installed and uninstalled easily.
- Provide documentation on how to install and use the program.
- Ensure the program runs on standard desktop and laptop computers.

# 3. Use Case

User: This actor represents the user playing the game.

Game Completion Checker Service: This actor represents the service which determines whether or not the game has been completed.

Hit Checker Service: This actor represents the service which determines whether or not a fired missile from the user actor has either hit or missed a piece.

Tries Left Checker Service: This actor represents the service which keeps track on the amount of tries the user actor has left in the game before the game ends.

Initialize game: This use case involves the player starting the game.

Fire missile: This use case involves the player selecting a button on the game board to fire a missile.

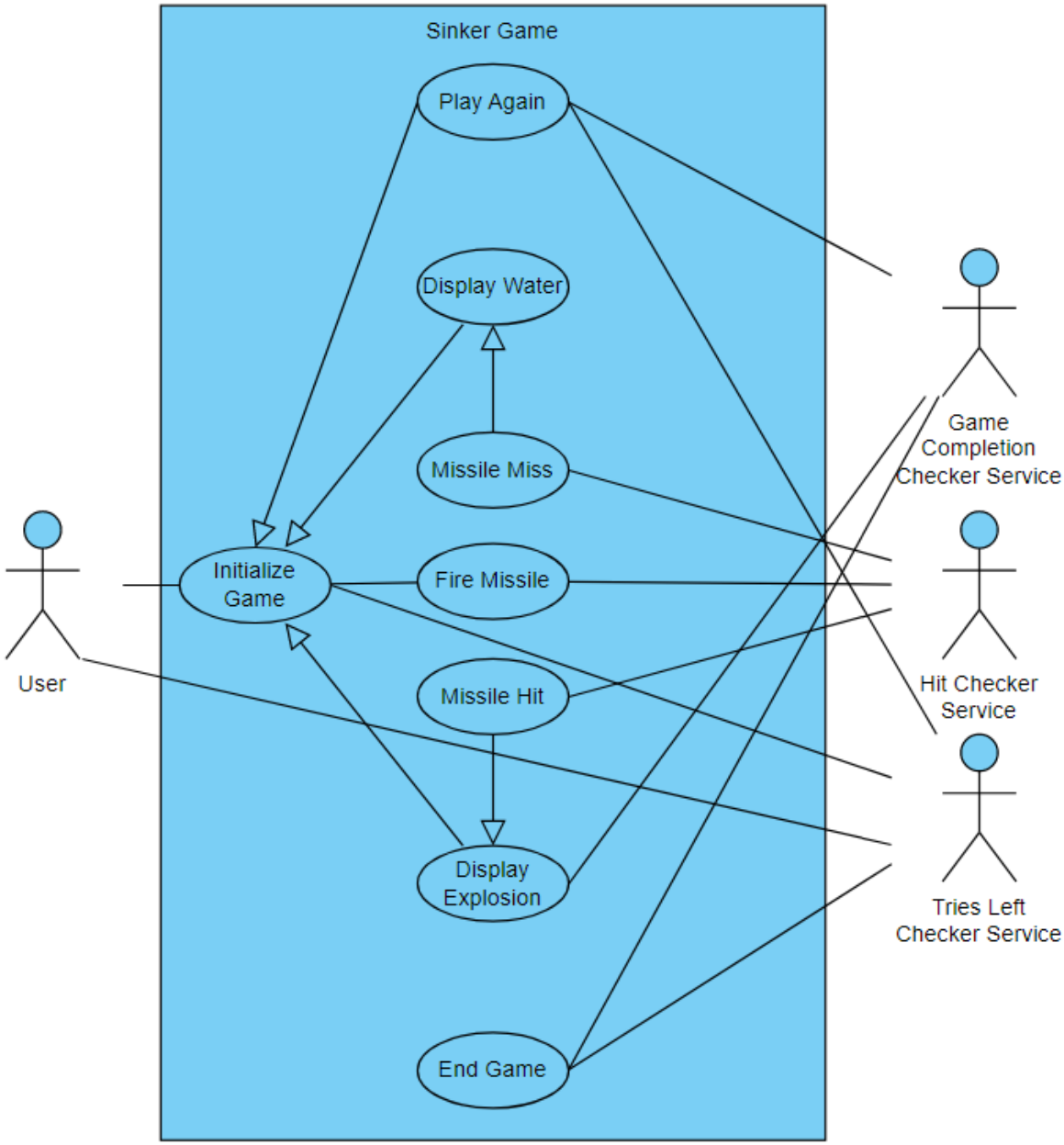
Missile hit: This use case involves the system determining the hit checker service if a piece was hit.

Missile miss: This use case involves the system determining the hit checker service if a piece was not hit.

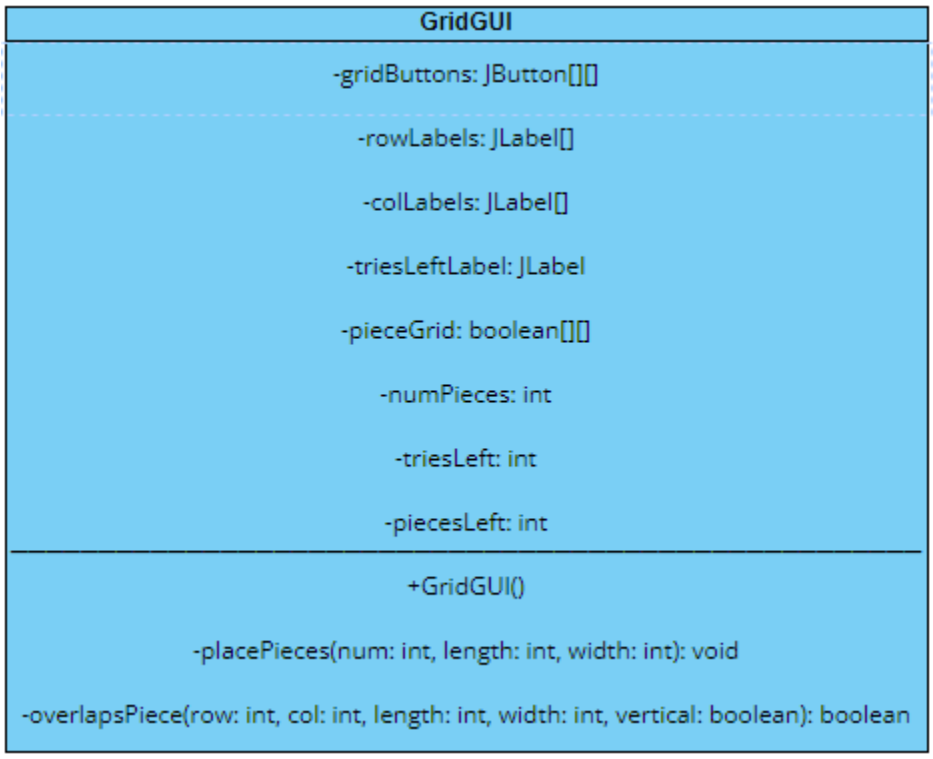
Display water image: This use case involves the system displaying the water image on a button when there is no ship on it.

Display explosion image: This use case involves the system displaying the explosion image on a button when a ship has been hit.

End game: This use case involves the system ending the game when all the ships have been sunk or the player runs out of missiles.


Sinker Game
Play Again
Display Water
Missile Miss
Initialize Game
Fire Missile
Missile Hit
Display Explosion
End Game
User
Game Completion Checker Service
Hit Checker Service
Tries Left Checker Service

# 4. Domain Class and Class Diagram




| Properties | Type | Description |
|---|---|---|
| gridButtons | JButton[][] | 2D array of JButtons representing the game board |
| rowLabels | JLabel[] | Array of JLabels representing the row labels |
| colLabels | JLabel[] | Array of JLabels representing the column labels |
| triesLeftLabel | JLabel | JLabel representing the number of tries left |
| pieceGrid | boolean[][] | 2D array of booleans representing which squares on the game board are occupied by pieces |
| numPieces | int | Number of pieces on the game board |
| triesLeft | int | Number of tries left for the player |
| piecesLeft | int | number of pieces left on the game board |

| Methods | Signature | Description |
|---|---|---|
| GridGUI() | GridGUI() | Constructor for GridGUI class |
| placePieces | (num: int, length: int, width: int): void | Places the specified number of pieces of the specified length and width onto the game board |
| overlapsPiece | (row: int, col: int, length: int, width: int, vertical: boolean): boolean | Determines whether a piece of the specified length and width, oriented vertically or horizontally, overlaps with any existing pieces on the game board |

# 5. Implementation Details

Sinker Board Service Notes:

This service maintains a 2D array of Buttons that represents the game board. The service provides functionality to both initialize the board and to determine if a button has a ship on it or not. The interface methods to the module provide access allowing for the checking of user button selections.

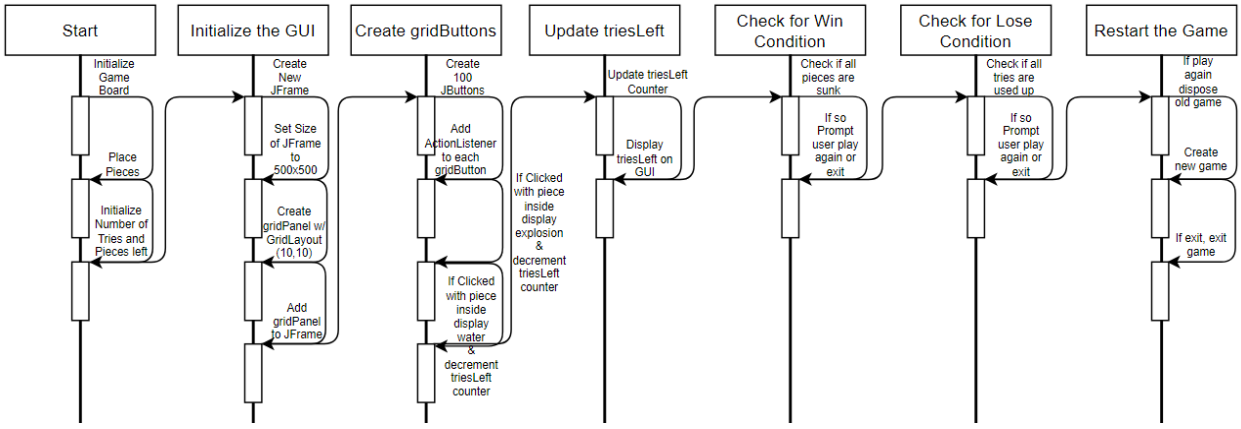
Game Controller Service Notes:

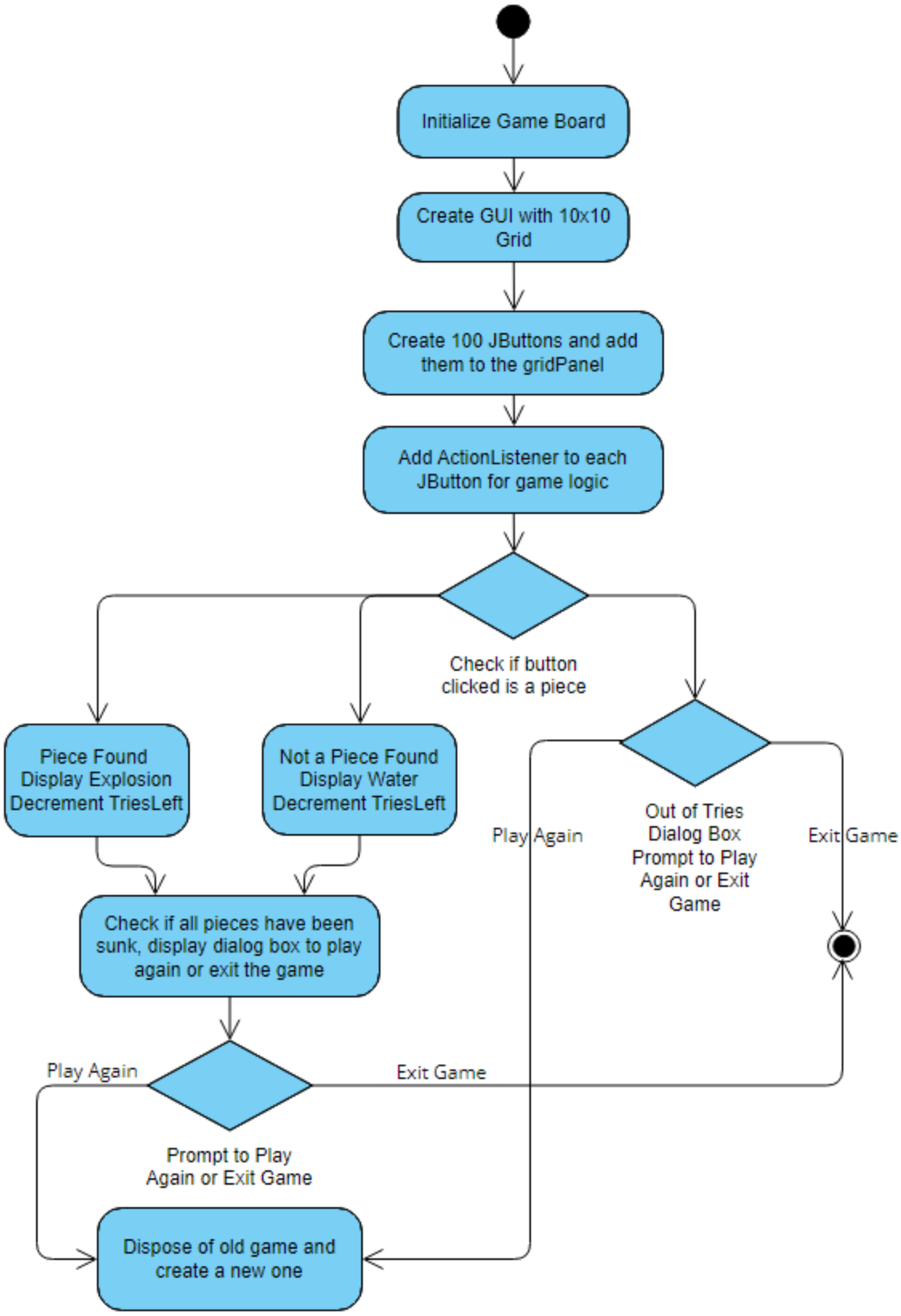
The system requirements for the Game Controller service detail many operations, most of which are related to the transfer of information to and from the other services. The interface to this module was written with this in mind. The GUI component within this service allows a user many read/write capabilities into the other services. The GUI also relays information back to the user as per the requirements section. The GUI Engine is designed to work in conjunction with the GUI. This is the baseline for a Model-View-Controller pattern. The mediator is designed to know about both the GUI Engine and the Game Controller Service, and help facilitate 'conversations' between them. So the GUI Engine and the Game Controller Service both know the mediator, and the GUI engine could be swapped out for another "Colleague" if it follows the same rules.

## Flow of Control

1. Initialize the game by setting up the game board, placing the pieces and initializing the number of tries and pieces left.
2. Create a GUI with 10x10 grid and 10 pieces of varying size to be sunk.
3. Create a JPanel gridPanel with GridLayout(10,10) to hold the gridButtons.

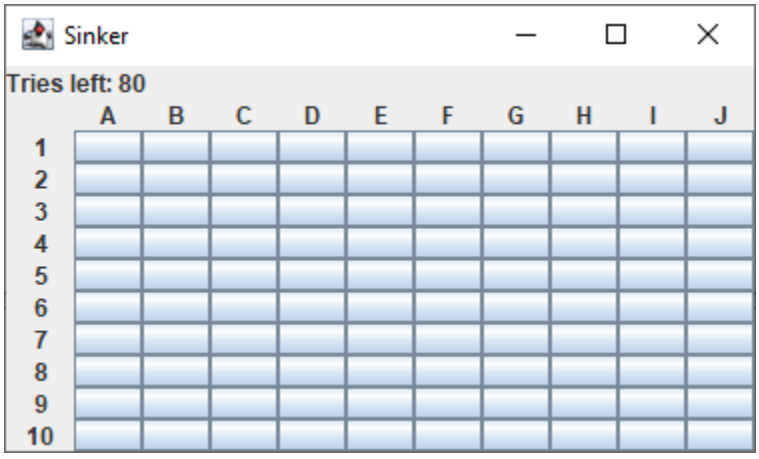
4. Create 100 JButtons gridButtons and add them to the gridPanel.
5. Add an ActionListener to each JButton that checks if the button clicked is a piece. If it is, an explosion icon is displayed, and the triesLeft counter is decremented. If it isn't, a water icon is displayed, , and the triesLeft counter is decremented
6. Update the triesLeft counter and display it on the GUI.
7. Check if all the pieces have been sunk. If they have, prompt the user with a dialog box to play again or exit the game.
8. If all the tries are used up, prompt the user with a dialog box to play again or exit the game.
9. If the user chooses to play again, dispose of the old game and create a new one. If not, exit the game.




Initialize Game Board
Create GUI with 10x10 Grid
Create 100 JButtons and add them to the gridPanel
Add ActionListener to each JButton for game logic
Check if button clicked is a piece
Piece Found
Display Explosion
Decrement TriesLeft
Not a Piece Found
Display Water
Decrement TriesLeft
Out of Tries
Dialog Box
Prompt to Play Again or Exit Game
Play Again
Exit Game
Check if all pieces have been sunk, display dialog box to play again or exit the game
Play Again
Exit Game
Prompt to Play Again or Exit Game
Dispose of old game and create a new one

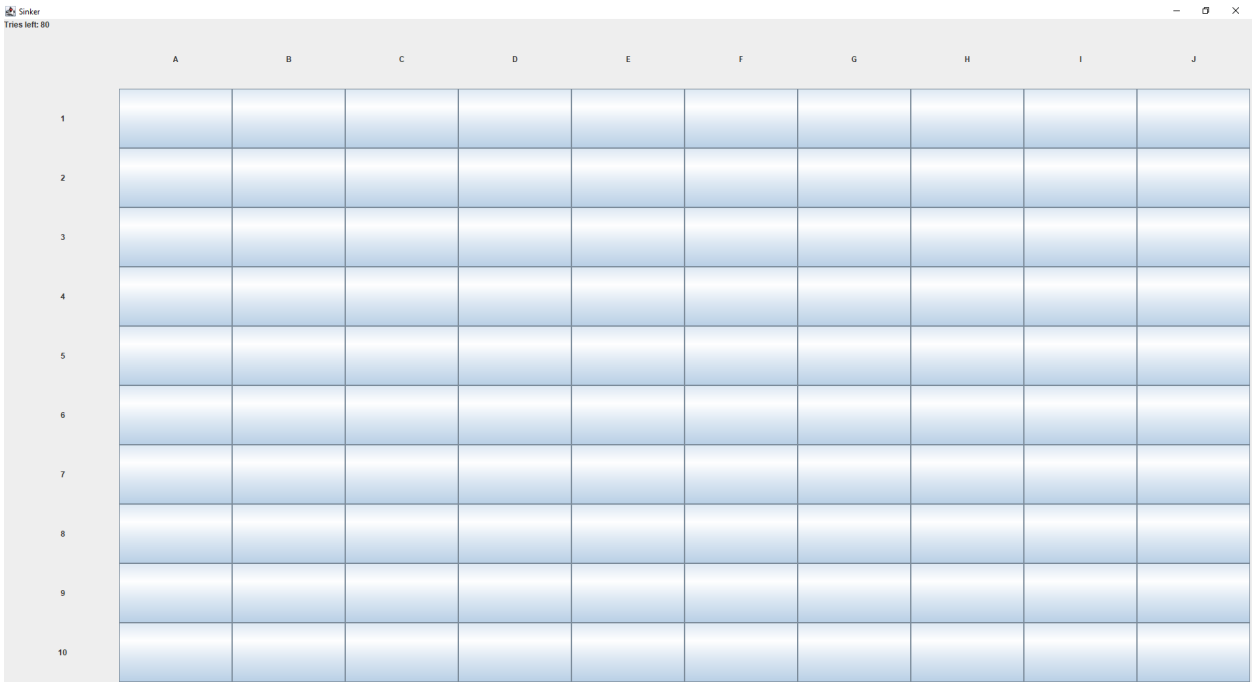
# GUI Designs

Sinker GUI On Launch:

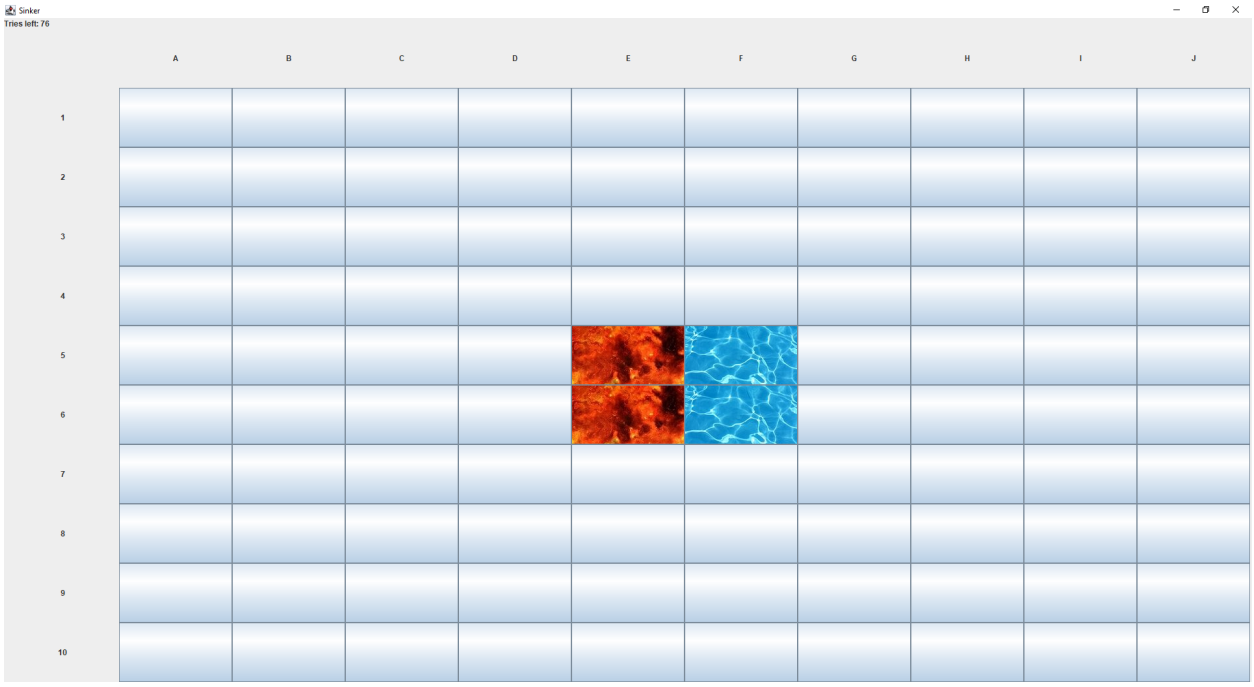



Sinker GUI Four Middle Buttons Post Click:

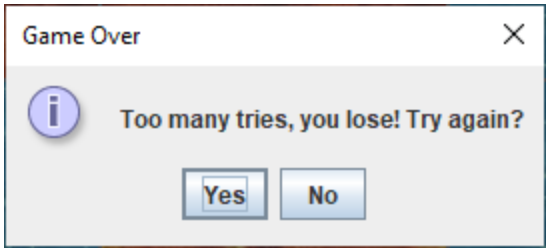

Sinker GUI Fullscreen:



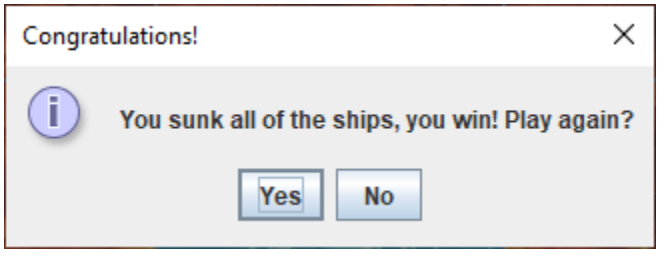

Sinker GUI Four Middle Buttons Post Click Fullscreen:




Game Over, You Lose Prompt:




Game Over, You Win Prompt:



# 6. Testing Plan

A. Unit Testing

- Game Board: Test that the game board is properly initialized, and the pieces are placed correctly.
- GUI: Test that the GUI is properly displayed, and the buttons and icons are functioning as expected.
- ActionListener: Test that the ActionListener is properly checking if the button clicked is a piece, and displaying the correct icon.
- Tries Counter: Test that the triesLeft counter is properly updated and displayed on the GUI.
- Win/Loss Condition: Test that the game ends correctly when all the pieces are sunk or all the tries are used up.

B. GUI Testing

- Layout: Test that the GUI layout is properly displayed, and all the buttons and icons are in the correct position.
- Functionality: Test that all the GUI buttons are functioning correctly, and the game can be played using the GUI.
- Responsiveness: Test that the GUI is responsive and user-friendly, and there are no delays or glitches in the game.

C. System Testing

- Performance: Test that the game is performing well, and there are no lags or delays during the gameplay.

- Compatibility: Test that the game is compatible with different operating systems and devices.

D. User Acceptance Testing

- End-User Testing: Test that the end-users can easily understand and play the game.
- Feedback Collection: Collect feedback from end-users, and make necessary improvements based on the feedback.

# 7. Risks

Compatibility Risk: The Sinker program may not be compatible with all types of operating systems or devices. This could lead to compatibility issues when the program is installed on a device that is not compatible, leading to a suboptimal user experience.

Performance Risk: The performance of the game may be impacted if there are multiple sessions of the game being played on the same system simultaneously. This could lead to lag or slow response times, leading to a negative user experience.

Security Risk: The game may have vulnerabilities that could be exploited by hackers or malicious actors.

User Error Risk: The program currently has limited fail-safe measures to account for user error.