Group Project 11
Design Specification

Author:	Bhagya Wanni Arachchige [bhw], Mateusz Mazur [mam148], Maciek Traczyk [mat78], Ash Bagnall [asb20], Dean Plank [dep22], James Green [jag77]
Config Ref:	DesignSpecGroup11
Date:	29th March 2022
Version:	1.0
Status:	Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

CONTENTS 2
1. INTRODUCTION 3
  1.1 Purpose of this Document 3
  1.2 Scope 3
  1.3 Objectives 3
2. DECOMPOSITION DESCRIPTION 3
  2.1 Programs in the system 3
  2.2 Significant classes 3
  2.3 Table mapping requirement onto classes 3
3. DEPENDENCY DESCRIPTION 4
4. INTERFACE DESCRIPTION 4
5. DETAILED DESIGN 10
  5.1 Sequence Diagrams 10
  5.2 Significant algorithms 12
  5.3 Significant data structures 12
REFERENCES 12
DOCUMENT HISTORY 13

1. INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to provide a clear definition of what the outcomes of the game should be[1].

1.2 Scope

Describes the important

1.3 Objectives

The objective of this document is to [1]:

- Identify the significant programs in the system
- Identify what classes are relevant to each requirement
- Specify the relationship and dependencies between modules
- Outline a specification of each class of the program

2. DECOMPOSITION DESCRIPTION

2.1 Programs in the system

The system is only composed of with a singular part:

- Desktop Application

2.1.1 Desktop Application

The desktop application is where the board game can be played. Users can select a player with a username and boat of their choosing. Users can interact with the Buccaneer game board's functions using this boat, for instance, they are allowed to move the boat to various Islands and ports located on the game board. Users can finish the game by collecting 20 points of treasure and successfully bringing it back to their home port.

2.1.2 Significant classes

Desktop application

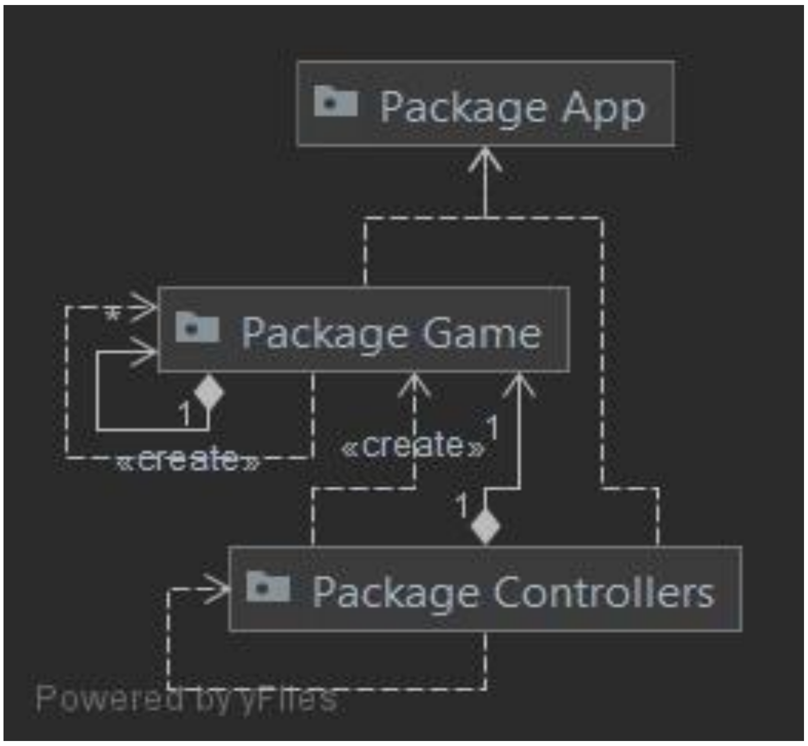
- Game: The class acts as the main body of the game and contains the game board and player objects.
- App: Oversees the shifting between screens/ different stages of the game.
- Player: The player class contains all details related to the player ship including their name, number, location, and the direction they are facing
- Attack: Handles all the attacking mechanics for the game. Performs the necessary calculations to find the winner and loser of each attack and determines what to do based on this result.
- Tile: Super Class to IslandTile, PortTile, PlayerTile and OceanTile. Contains the image for each tile of the board and assigns properties for each separate tile e.g., Can you attack the tile object?

2.2 Table mapping requirement onto classes

Function Requirement	Classes that meet the requirement
FR1	CharacterScreenController

FR2	Game
FR3	CrewPack, CrewCard
FR4	ChanceCard
FR5	Treasure, Player
FR6	Player, Game, CrewHand
FR7	PortTile, Game
FR8	FlatIsland
FR9	GameScreenController, OceanTile, Player,Tile, PortTile, IslandTile
FR10	Game
FR11	NextPlayerScreen
FR12	Attack
FR13	IslandTile, TreasureIsland
FR14	IslandTile, FlatIsland
FR15	PortTile
FR16	PortTile, AnchorBay
FR17	EndGame

3. DEPENDENCY DESCRIPTION



4. INTERFACE DESCRIPTION

4.1.1 App

The **App** class contains the main methods that initially launch the program. It contains the methods that enable FXML to work.

```
public class App extends Application {

        // Is the main method of the project. Starts everything.
        public static main(String[] args) { ; }
```

```
            // Starts the main FXML stage
            public static start(Stage) { ; }
}
```

4.1.2 CrewCard

The **CrewCard** class holds all the data necessary regarding a single Crew card in the game. It holds both a value and a color.

```
public class CrewCard {

            // The constructor for the class. Creates a new instance with a card value (int) and
            // a card color (String).
            public CrewCard(int, String) { ; }

            // Returns the card value
            public int getValue() { return int; }

            // Returns the color of the card
            public String getColor() { return String; }
}
```

4.1.3 CrewHand

The **CrewHand** class holds an array of object CrewHand that acts as the player's current hand of crew hands within the game. Contains functionality to calculate the necessary values for combat and movement.

```
public class CrewHand {
            public CrewCard[] cards; // Contains all the cards for a CrewHand.
            public int totalCards; // Contains the amount of cards currently in the CrewHand.

            // The constructor for the class. Creates a new instance.
            public CrewHand() { ; }

            // Adds a card to the Hand.
            public void addCard(CrewCard) { ; }

            // Returns the total value of black color cards.
            public int getBlackValue() { return int; }

            // Returns the total value of red color cards.
            public int getRedValue() { return int; }

            // Returns the combat value of the player's hand. (red - black or black - red)
            public int getCombatValue() { return int; }

            // Returns the amount the player can move. The total of all CrewCards.
            public int getMoveAbility() { return int; }

            // Returns the amount of total cards in the Hand.
```

```
        public getTotalCards() { return int; }
}
```

4.1.4 CrewPack

The **CrewPack** class holds and keeps track of all CrewCards in the game and handles handing out cards to players and keeping them all in one place within the Game object. Acts as a card pack, hands out cards on the top of the stack.

```
public class CrewPack {
        public int totalSize; // Stores the total size of the pack of crew cards.

        // The constructor for the class. Creates a new instance.
        public CrewPack() { ; }

        // Returns a Card at a given index.
        public CrewCard getCard(int) { return CrewCard; }

        // Returns all cards in the Pack.
        public CrewCard[] getCards() { return Crewcard[]; }

        // Gives a player a card from the top of the deck.
        public CrewCard givePlayerCard(Player) { return CrewCard; }

        // Creates a new card pack.
        public void newCrewPack() { ; }

        // Shifts all cards after one is dealt from the top of the pack
        public void shift(CrewCard[]) { ; }

        // Shuffles all cards.
        public void shuffle() { ; }
}
```

4.1.5 Game

The **Game** class is the main backend class that handles the entire game. It holds the gameBoard and the player objects. It executes most of the game logic.

```
public class Game {
        public Player[] players; // Contains all the players (4)
        public int turn; // Contains the Player's index of who's turn it is (1-4)
        public Tile[][] gameBoard; // Contains all data for the actual board

        // The constructor for the class. Creates a new instance.
        public Game() { ; }

        // Gets the current player (who's turn it is)
        public Player getCurrentPlayer() { return Player; }

        // Gets a player
        public Player getPlayer() { return Player; }
        // Gets the current turn index (4)
        public int getTurn() { return int; }

        // Populates all the tiles with the relevant data
        public void populateTiles() { ; }
}
```

4.1.6 GameHandler

The **GameHandler** class handles all of the object persistence within our game. It saves and loads all player and game data.

```
public class GameHandler {
        public Player[] players; // Holds all the players (4)

        // The constructor for the class. Creates a new instance.
        public GameHandler() { ; }

        // Continues the game if possible, if not returns false
        public boolean ContinueGame() { return boolean; }

        // Gets all the players and returns them in an Array
        public Player[] getAllPlayers() { return Player[]; }

        // Gets all the player's data directly from .json files, returns an array of object Player.
        public Player[] getAllPlayersFromFile() { return Player[]; }

        // Gets a particular player from file using their playerNumber.
        public Player getPlayerFromFile(int) { return Player; }

        // Gets all the board's data from file and returns it.
        public Game loadBoard() { return Game; }

        // Creates a new game, resets files etc.
        public void newGame() { ; }

        // Saves all players directly to .json files, returns if successful.
        public boolean saveAllPlayers() { return boolean; }

        // Saves the entire board.
        public void saveBoard(Game) { ; }

        // Saves a particular player
        public boolean savePlayer(Player) { return boolean; }
}
```

4.1.7 GameScreenController

The **GameScreenController** handles loading FXML scenes for the main game.

```
public class GameScreenController  {
        // Initializes the class
        public void initialize() { ; }
}
```

4.1.8 NextPlayerScreen

The **NextPlayerScreen** handles switching FXML scenes. It contains no public methods.

```
public class NextPlayerScreen {
        // No public methods
}
```

4.1.9 Player

The **Player** class holds all the data about a player. It holds data regarding their hands of both crew and chance cards, their treasure, their position and other data.

```
public class Player {
        public int[] coordinate;
        public CrewHand crewHand;
        public String playerName;
        public int playerNumber;
        public Image shipImage;

        // The constructor for the class. Creates a new instance.
        public Player() { ; }

        // The constructor for the class. Creates a new instance with player name and number.
        public Player(String, int) () { ; }

        // Returns the player's current column coordinate (x).
        public int getColCoordinate() { return int; }

        // Returns the player's current coordinates.
        public int[] getCoordinate() { return int[]; }

        // Returns the player's ship image.
        public Image getIcon() { return Image; }

        // Returns the player's name.
        public String getPlayerName() { return String; }

        // Returns the player's number.
        public int getPlayerNumber() { return int; }

        // Returns the player's row coordinate (y).
        public int getRowCoordinate() { return int; }

        // Sets the player's column coordinate.
        public void setColCoordinate(int) { ; }

        // Sets the player's coordinate (x, y).
        public void setCoordinate(int, int) { ; }
```

```
            // Sets the player's ship image.
            public void setIcon(Image) { ; }

            // Sets the player's row coordinate (y).
            public setRowCoordinate(int) { ; }
}
```

4.1.10 PlayerTile

The **PlayerTile** implements **Tile** as it is much easier to implement game tiles this way. Since all tiles have the same functionality, it works out much nicer creating an interface.

This Tile indicates that there is currently a player in this current position.

```
public class PlayerTile implements Tile {

            // Returns the OceanTile icon.
            public Image getIcon() { return Image; }

            // Returns if this tile is attackable by a player.
            public boolean isAttackAble() { return boolean; }

            // Returns if this tile is an island.
            public boolean isIsland() { return boolean; }

            // Returns if a player can travel to this tile.
            public boolean isTraversable() { return boolean; }

            // Sets the tile's image.
            public void setIcon(Image) { ; }
}
```

4.1.11 Tile

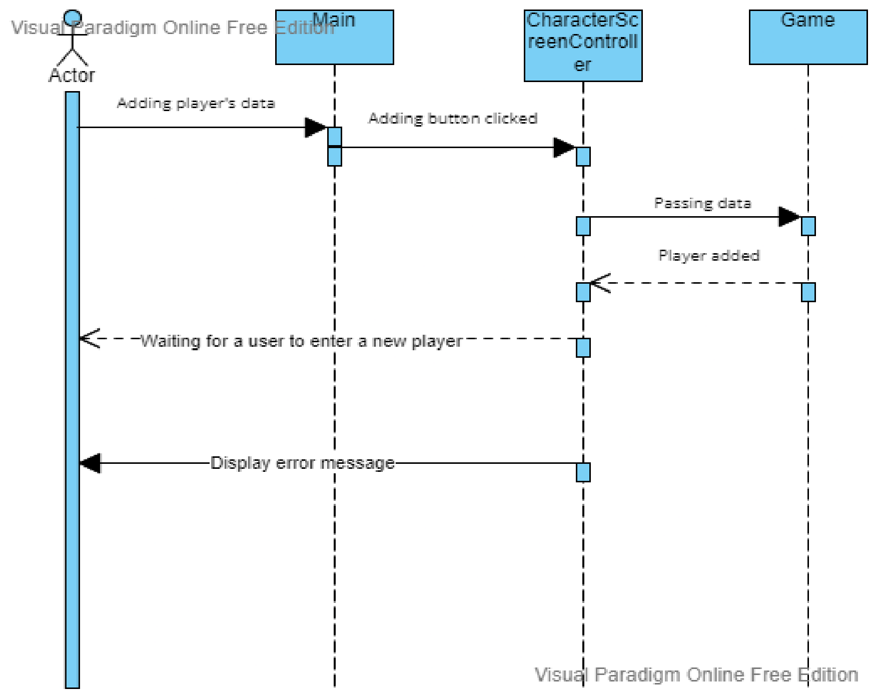
The **Tile** interface is set up to make using Tiles in the game much easier. It is an interface which PlayerTile and OceanTile implement. Sets the required details regarding each tile, such as its image and ability to attack.

```
public interface Tile {

            // Sets a tile icon.
            public void setIcon(Image icon);

            // Get's a tile icon.
    public Image getIcon();

    // Defines if a tile is attackable by a player.
    public boolean isAttackAble();

    // Defines if a tile is traversable by a player.
    public boolean isTraversable();

    // Defines if the tile is part of an island.
    public boolean isIsland();
}
```

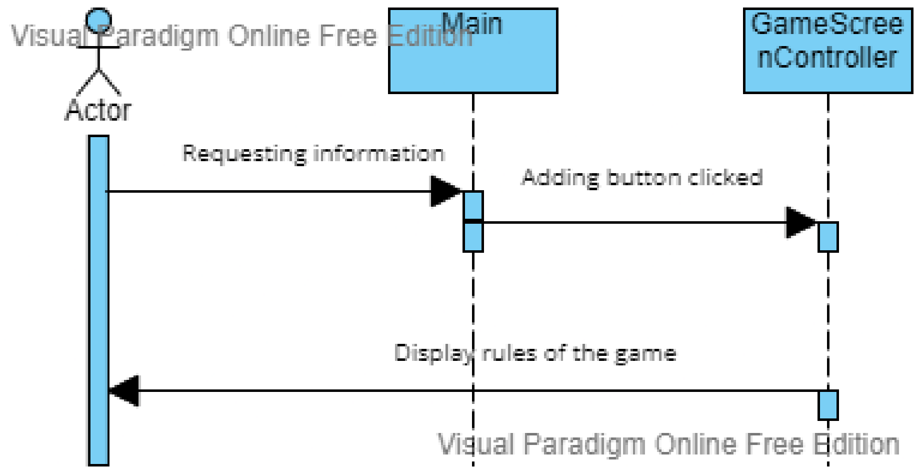
5. DETAILED DESIGN

5.1 Sequence Diagrams

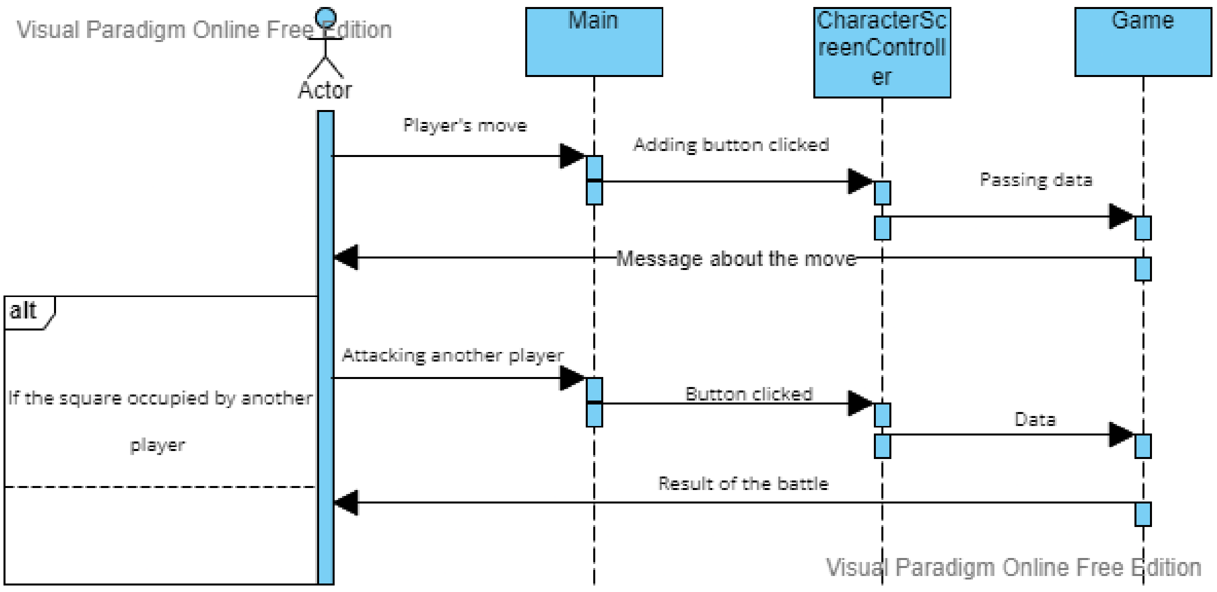
5.1.1 Use Case 1.1:


Visual Paradigm Online Free Edition
Actor
Main
CharacterScreenController
Game
Adding player's data
Adding button clicked
Passing data
Player added
Waiting for a user to enter a new player
Display error message
Visual Paradigm Online Free Edition

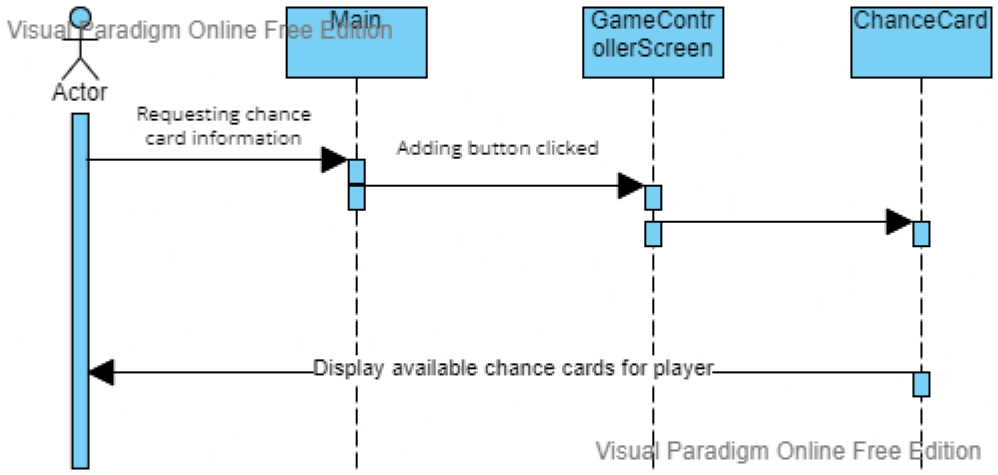
5.1.2 Use Case 1.2:


Visual Paradigm Online Free Edition
Actor
Main
GameScreenController
Requesting information
Adding button clicked
Display rules of the game
Visual Paradigm Online Free Edition

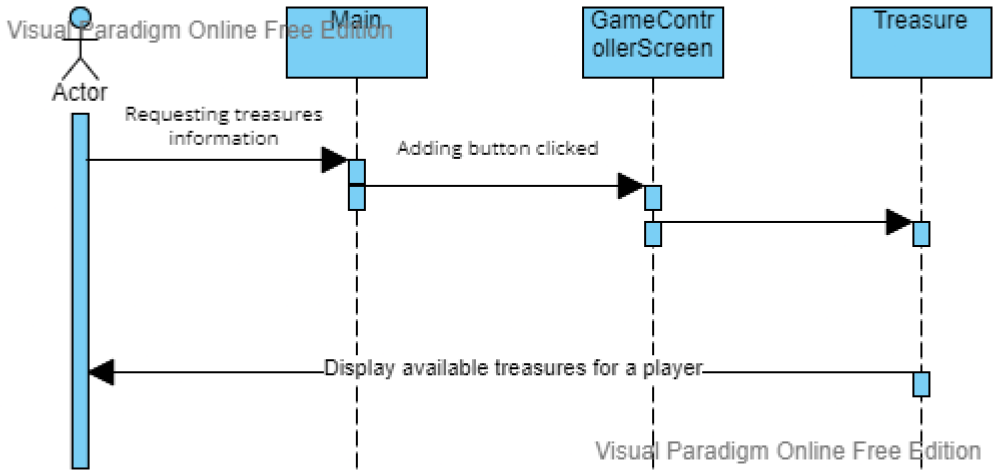
5.1.3 Use Case 1.3:



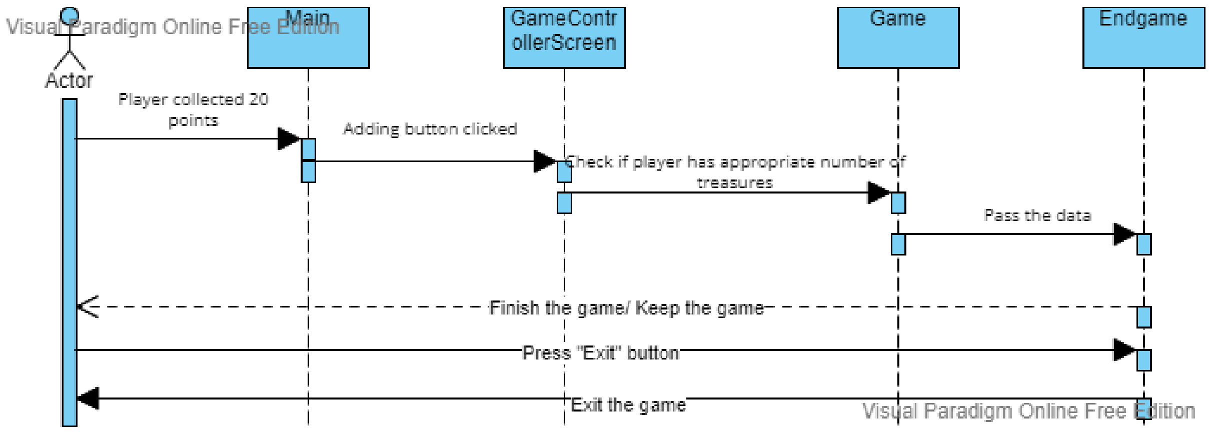
5.1.4 Use Case 1.5:


Visual Paradigm Online Free Edition
Actor
Main
GameControllerScreen
ChanceCard
Requesting chance card information
Adding button clicked
Display available chance cards for player
Visual Paradigm Online Free Edition

5.1.5 Use Case 1.6:


Visual Paradigm Online Free Edition
Actor
Main
GameControllerScreen
Treasure
Requesting treasures information
Adding button clicked
Display available treasures for a player
Visual Paradigm Online Free Edition

5.1.6 Use Case 1.8:


Visual Paradigm Online Free Edition
Actor
Main
GameControllerScreen
Game
Endgame
Player collected 20 points
Adding button clicked
Check if player has appropriate number of treasures
Pass the data
Finish the game/ Keep the game
Press "Exit" button
Exit the game
Visual Paradigm Online Free Edition

5.2 Significant algorithms

During our meetings with the group, we discussed the difficult parts of the code. We concluded that one of the more difficult tasks will be the implementation of chance cards. Since each card will contain an event, this will cause the game to react differently each time. This will make the game flow different every time. This will require separate 'event handlers' for each card.

Pseudocode:

```
If player arrives at Treasure Island, then
    Pick chance card from pile
        If card (14) selected
            Use appropriate ‘game handler’ to proceed
            (Example: Receive treasures for a total of six points)
            Implement the consequences of the chance card
            (add six points to player’s account)
    Move card to bottom of the pile
Next player’s move
```

5.3 Significant data structures

All data that the game uses is stored in JSON files in the GameHandler class, which is responsible for converting Java objects to JSON and JSON to Java. These objects are automatically converted to JSON format using a Google powered package called 'gson'. This is ideal for Java's persistence as we don't need to do any further coding beyond the package implementation. This means that it is done automatically and without unnecessary interruptions when handling complex JSON files such as "gameconfig.json" that contain all of the game data. The file "gameconfig.json" stores the entire game board and the data contained in each board tile. This includes what a tile is, each player on the tile, and islands. Moreover, this file also includes saving the players themselves. All players and the data associated with each player are stored in this file and include data such as player name, number, treasure, crew cards, chance cards and everything else that is an integral part of the player which is perfect for this type of game.

REFERENCES

[1] QA Document SE.QA.05 – Design Specification Standards

DOCUMENT HISTORY

Version	*Issue No.*	*Date*	*Changes made to document*	*Changed by*
1.0	N/A	29/03/22	N/A - original version	BHW