Group Project 07 - System Design Specification

Classes, relationships, data structures, implementation of the requirements

Author: Kamil Cupial, Mathew David Jones

Config. Ref.: SE.SYS-DESIGN
Date: 2017-04-04
Version: 1.0
Status: Release

Department of Computer Science,
Aberystwyth University,
Aberystwyth,
Ceredigion, SY23 3DB,
U.K.

CONTENTS

1 INTRODUCTION **2**
1.1 Purpose of This Document . . . 2
1.2 Scope . . . 2
1.3 Objectives . . . 2

2 DECOMPOSITION STRUCTURE **3**
2.1 Description of the Application class . . . 3
2.2 Description of The Classes (System) . . . 3
2.2.1 Interfaces . . . 3
2.2.2 Abstract Classes . . . 3
2.2.3 Instantiable Classes . . . 3
2.2.4 Exceptions . . . 5
2.3 Description of The Classes (User Interface) . . . 5
2.3.1 Instantiable Classes . . . 5
2.4 Mapping From Requirements to Classes . . . 6

3 CLASS AND DEPENDENCY REVIEW **7**
3.1 Model-View-Controller . . . 7
3.2 Package Description . . . 7
3.3 UML Class Diagram . . . 8

4 INTERFACE DESCRIPTION **9**
4.1 CardHolder . . . 9
4.2 MapObject . . . 10
4.3 TreasureHolder . . . 11

5 DETAILED DESIGN **12**
5.1 Typical Use Cases and Expected Program Behaviour . . . 12
5.1.1 Program behaviour/scenario . . . 12
5.1.2 Activity diagrams . . . 12
5.2 State Machine Description - Valid states, State-holders names . . . 12
5.2.1 Turns . . . 12
5.2.2 Other . . . 13
5.3 Logic Between States and Windows (moved from UI document[2]) . . . 13
5.4 Use-Case Scenarios Detailed Implementation . . . 13
5.4.1 Starting the game . . . 14
5.4.2 Game behaviour . . . 14
5.4.3 Popups . . . 14
5.4.4 Chance Cards . . . 14
5.4.5 Passing through players . . . 15
5.5 Other Notes . . . 15

REFERENCES **15**

DOCUMENT HISTORY **16**

1 INTRODUCTION

1.1 Purpose of This Document

The purpose of this document is to provide a detailed design of the "Buccaneer" game according to the requirements[1] set by the client in Java language. Any changes to the design must be documented and reviewed. Additionally - changes to the design should be kept to minimum and resolve as many flaws as possible at once, to avoid constant re-design and the necessity to rewrite the code.

1.2 Scope

The document describes classes and interfaces to be used in the system in detail, including (but not limited to) their variables and methods, relationships between them and overall functioning of the system as a whole.

Every project member involved in coding activities should read this document.

1.3 Objectives

The objective of this document is to provide team members with a complete system design that they should follow, enabling parallel and organised workflow in the project and making sure that the structure of the system is flawless and thought through (requires as few modifications as possible).

2 DECOMPOSITION STRUCTURE

All classes described below make up the architectural structure of the system that manages the data of the game. This part is linked to the User Interface part of the code by a GameEngine object being available to it, which provides functionality to manage the game as the UI events specify.

2.1 Description of the Application class

public class Application - should create a GameEngine object, a FXMLLoader object (for the main menu window) and pass the first one to the other's controller (which should be W_MainMenu), then show the latter's Stage (window), initiating the event-driven window system.

2.2 Description of The Classes (System)

2.2.1 Interfaces

- **public interface CardHolder<T extends Card>** - characterises any class that can hold objects of type Card/CrewCard/ChanceCard. Enforces methods in these classes
- **public interface TreasureHolder** - characterises any class that can hold Treasure objects. Enforces methods in those classes
- **public interface MapObject** - characterises any class that is placed on the board/grid. Provides functions regarding the positioning and is used by the Board/GameEngine to get/place a particular object based on the position on the grid. **Does not** include Ship objects, as their position is kept/retrieved separately to compare to corresponding objects

2.2.2 Abstract Classes

Used to provide functionality without the need of instantiating (static methods/variables) or common functionality for sub-classes

- **public abstract class Card** - base class for other Card classes (used by Pack class to restrict the types taken by the Pack to Cards only)
- **public abstract class IslandInteraction** - helps manage an interaction between a given Player/Ship and Islands
- **public abstract class Island impelements MapObject** - contains basic variables/methods common for every type of Island/Port (position and size-wise)
- **public abstract class PlayerInteraction** - helps manage interactions between Players
- **public abstract class PortInteraction** - helps manage interactions between a given Player and Port

2.2.3 Instantiable Classes

Cliff Creek and Mud Bay not included simply because they're just a set of coordinates on the board.

- **public class AnchorBay extends Port** - distinguishes the Anchor Bay port (made a separate class since it requires more functionality than Cliff Creek and Mud Bay)

- **public class Board** - provides functionality of the grid/board and manages Players (whose details can be requested from, such as crew strength etc.). Should be the second place, after GameEngine in terms of looking for methods to perform more complex actions (other classes can still be used to retrieve data)

- **public class ChanceCard implements Card** - provides functionality specific to chance cards

- **public class CrewCard extends Card** - provides functionality specific to crew cards

- **public enum Direction** - specifies directions a Ship can have for easier recognition in code (e.g. Direction.NE, Direction.N, Direction.SW)

- **public class FlatIsland extends Island implements CardHolder<CrewCard>, TreasureHolder** - provides functionality specific to flat island

- **public class GameEngine** - manages the game from the beginning to its end, repeats the process if another game is started. Manages Players' turns and creates/invokes objects based on the state of the game. The object of this class is what UI should create/get initially and work from there.

- **public class GameSetup** - provides functionality related with setting up the game for GameEngine. Used to create initial objects, initialise Packs, Player details etc.

- **public class Pack<T extends Card>** - contains a set of Card objects and has methods related to drawing/returning/shuffling them

- **public class PirateIsland extends Island implements CardHolder<CrewCard>** - provides functionality for pirate island

- **public class Player** - contains details of a player, like: Ship, ship's Position on the board, cards in hand, etc.

- **public class Port implements CardHolder<Card>, TreasureHolder, MapObject** - contains variables/methods common for every type of port

- **public class PlayerPort extends Port** - provides functionality specific only for Ports owned by Players

- **public class Position** - provides public integers x and y, which represent either a position on the board or size of an object (as in width and height). Provides an equals() method to compare if two Position objects are the same in terms of x & y values.

- **public class Ship implements TreasureHolder, MapObject** - provides functionality for ships. Ship's direction is of Direction class

- **public class Treasure** - represents a single object of a specific type of treasure (name + value)

- **public class TreasureIsland extends Island implements CardHolder<ChanceCard>, TreasureHolder** - provides functionality of the treasure island (managing treasure, accepting/rejecting treasure requests)

- **public enum TreasureType** - Contains possible treasure types (for cosmetic purposes): DIAMOND, RUBY, GOLD, RUM, PEARL

2.2.4 Exceptions

- **public class EmptyPackException extends Exception** - thrown when a request to remove a Card from an empty Pack is made
- **public class IllegalOperationException extends Exception** - thrown whenever an invalid operation is to be performed and no other exception is suitable
- **public class InvalidMoveException extends Exception** - thrown when invalid move is tried to be made (e.g. negative position values, going 'inside' an island)
- **public class InvalidDirectionException extends Exception** - thrown if a ship is tried to be turned into an invalid direction (e.g. facing border of the board)
- **public class InvalidPositionException** - thrown whenever an invalid position is requested/used at crucial points of the program (e.g. requesting an object at coordinates (-1,1))
- **public class InvalidStateException extends Exception** - thrown when invalid state is requested to be set for an unmatching String (e.g. a value State.paused is tried to be assigned to string "game" which shouldn't accept such)
- **public class NotInHomePortException extends Exception** - thrown when depositTreasure() is called on the port that isn't owned by the player specified or the player's ship is not in the port
- **public class NullStateException extends Exception** - thrown by the state machine if state requested doesn't exist (its value is null)
- **public class TooMuchTreasureException extends Exception** - thrown when trying to add more Treasure than the TreasureHolder can store
- **public class UnevenTradeException extends Exception** - thrown when the trade requested is uneven

2.3 Description of The Classes (User Interface)

2.3.1 Instantiable Classes

- **public class W_MainMenu** - This will be the controller for the main menu which will handle all the interactions between the buttons and loading of the FXML files
- **public class W_Setup** - This will be the pop-up window where the player's names are taken. The player's names are entered in four textboxes and when all four have been filled in you can then proceed to the next step of the main game screen. If the same name is entered more than once an error message will be shown
- **public class W_Credits** - This will be the display of members who produced the game. There will be a button in the bottom right of the screen displaying return to the menu, allowing the player to return to the main menu
- **public class W_Pause** - The controller for the pause screen will have interaction between buttons. I.e. return to the game, exit to the menu or quit the game
- **public class W_Game** – The game controller has multiple options on it, from being able to react to clicks on the grid to updating the view. It also invokes info screens by clicking and other pop-ups as the game progresses through play

- **public class W_Trade** - The pop-up window will be initiated and you will be able to select the items to be traded. The screen will darken and option to trade will become available. The treasures and cards available to trade will be displayed. If no tradeable items are available this field will be empty

- **public class W_MyCards** - This will display a small dialogue box with the available chance/crew cards of the player and any treasure in the port and on the ship. You are then able to hover over and confirm the selection

- **public class W_TakeTreasure** – This option will enable the player to select the treasure that they would like, from the ones available at treasure island providing its value is within the limit given

- **public class W_FightConfirm** - The screen transitions into a smaller pop-up enabling you to confirm the fight or abort the fight. At this point, the current fighting strength will be displayed. If the static player wishes to attack the fight will commence and the loser will have to sail away to maximum moves of their sailing strength

- **public class W_Scoreboard** - The scoreboard will be displayed with the user's information about all the players in the game session. The individual boxes will contain information regarding name, what port they belong to as well as all treasures in said port as well as the crew/chance cards that they hold

- **public class Popup_Simple** – Simple pop-up that will contain a confirm dialogue that will also be able to display an image. When the simple pop-up is displayed the current screen will darken leaving you with the confirm button

- **public class Popup_MultipleOptions** – This will accept a prepared list of buttons and adds them to the stage, so it can be displayed properly (each button will have their own action defined)

- **public class Popup_Customised** – Where multiple images and buttons are accepted. For instance, the treasure value at any one time on a ship and asking the player whether they would like to attack another ship or not

- **public class InteractionController** – This handles the whole interaction step by step. Then modifies data appropriately (according to the game requirement specification)

2.4 Mapping From Requirements to Classes

Classes - regular font, words start uppercase

Interfaces/Abstract Classes - *emphasized font, words start uppercase*

Requirement	***Classes/interfaces providing requirement***
FR1 – Setup	W_Setup, Player, GameSetup
FR2 – Port Assignent	GameSetup, Port, PlayerPort
FR3 – Crew card management	GameEngine, GameSetup, Pack, *Card*, CrewCard
FR4 – Chance card management	GameSetup, GameEngine, Pack, *Card*, ChanceCard
FR5 – Treasure management	GameEngine, GameSetup, Treasure, *Island*, TreasureIsland, FlatIsland *Port*, PlayerPort, AnchorBay
FR6 – Player management	GameEngine, Player, *Card*, ChanceCard, *Port*, PlayerPort, *MapObject*, Ship
FR7 – Port management	Port, PlayerPort, Treasure
FR8 – Flat Island management	*Island*, FlatIsland, CrewCard, Treasure
FR9 – Board Display	W_Game, GameEngine, Board, *MapObject*
FR10 – Game setup	GameSetup
FR11 – Taking turns	W_Game, InteractionController, W_FightConfirm, GameEngine, PlayerInteraction, StateMachine, State
FR12 – Attacking rules	GameEngine, PlayerInteraction, Player
FR13 – Treasure Island	GameEngine, InteractionController, IslandInteraction, *Island*, TreasureIsland
FR14 – Flat Island	GameEngine, InteractionController, IslandInteraction, *Island*, FlatIsland
FR15 – Arriving at a port	GameEngine, InteractionController, PortInteraction, Port
FR16 – Anchor Bay	GameEngine, InteractionController, PortInteraction, *Port*, AnchorBay
FR17 – Detection of end of game	GameEngine
OTHER	Ship, Position, Direction, PirateIsland, *TreasureHolder*

3 CLASS AND DEPENDENCY REVIEW

3.1 Model-View-Controller

The design is following the Model-View-Controller pattern.
The model is the current state of the game stored inside a GameEngine object (state of the board, Ship, Player objects, etc.). This part of the software (and all the packages that start with .system) is independent to user interface and can be used by another UI implementation if need be. Controllers (usually the classes starting with W_) prepare the view based on that model, send commands according to user input, updating the view (JavaFX Stage objects) afterwards (which the user reacts to again).

3.2 Package Description

1. **(default package)/** - contains Application class that creates the Model part of the software (GameEngine) and passes it on to the View/Controller part (Main Menu controller).
2. **/ui/** - everything related to User Interface
   - **ui/menus/** - controllers handling FXML window template files and user interaction within menus and before the game starts. Also provides Popup classes that allow reuse for displaying information to the user or asking for user input.
   - **ui/game/** - controllers handling the windows and user input after the game starts. Handles parts of the game like: fighting, trading, moving. Contains InteractionController that takes care of Interaction within the game specifically
3. **/system/** - contains the crucial parts of the model with all the required functionality to enable the controllers to make the game proceed

- **system/resources/** - contains classes that represent entities that exist in the game (and supporting classes like Position or Direction). These are used to create objects that make up the game (Ships, Players, Islands, etc.)
- **system/helpers/** - contains supporting abstract classes that provide more complex methods, required to perform operations like fighting, trading and/or drawing and executing chance cards. This allows further separation of the part that is responsible for the game flow (e.g. transferring loot after the fight from one player to another, based on the winner) that has to follow functional requirements, and the controller part, that is supposed to only react to user's input and just inform the model of the decision (e.g. whether to actually perform the fight or not)
- **system/exceptions/** - custom exceptions that are thrown when the commands sent by the controllers or classes inside the model are invalid (e.g. trying to make a move happen that should not be possible from the Functional Requirements point of view)

3.3 UML Class Diagram

The following UML Diagram includes all the classes and methods required in the system for it to work properly and meet the specification's requirements. It should be followed and any changes to it should be represented in the code as well: UML Class diagram. The diagram can also be found under the name **SE.ClassUML.png** in this document's folder.
The UML Diagram above also represents the relationship between UI part of the system and the functional part of it, linked by the Application class.

4 INTERFACE DESCRIPTION

4.1 CardHolder

```
public interface CardHolder<T extends Card> {
        /**
         * Adds card to the pack
         */
        public void addCard(T card);

        /**
         * Returns and removes card from the top
         */
        public T draw()
        throws EmptyPackException;

        /**
         * Returns card from the top
         */
        public T peek();

        /**
         * Removes a specific card
         */
        public void removeCard(T card);

        /**
         * Returns Pack of Cards
         * @return pack of cards
         */
        public Pack<T> getPack();

        /**
         * Returns an ArrayList of cards
         */
        public ArrayList<T> getCards();

        /**
         * Returns number of cards
         */
        public int size();

        /**
         * Sorts CrewCards increasingly (doesn't do anything if ChanceCard holder)
         */
        public void sortInc();

        /**
         * Sorts CrewCards decreasingly (doesn't do anything if ChanceCard holder)
         */
        public void sortDec();

}
```

4.2 MapObject

```
public interface MapObject {

        /**
     * Sets a position of an object on the board
         * @param pos position of that object
         */
    public void setPos(Position pos);

    /**
     * Sets a position of an object on the board
     * @param x coordinate
     * @param y coordinate
     */
    public void setPos(int x, int y);

    /**
     * Gets a position of an object on the board
     * @return position of an object
     */
    public Position getPos();

    /**
     * Gets a size of an object on the board
     * @return size of an object
     */
    public Position getSize();

    /**
     * Sets a size of an object on the board
     * @param size of an object
     */
    public void setSize(Position size);

    /**
     * Sets a size of an object on the board
     * @param x horizontal
     * @param y vertical
     */
    public void setSize(int x, int y);

}
```

4.3 TreasureHolder

```
public interface TreasureHolder {

        /**
         * Sets a maximum number of Treasure objects a class can hold
         * @param count the limit of objects
         */
        public void setMaxTreasureCount(int count);

        /**
         *
         * @return maximum number of treasure objects a class can store
         */
        public int getMaxTreasureCount();

        /**
         * Adds treasure to the container class
         * @param treasure      Treasure object to add
         */
        public void addTreasure(Treasure treasure)
        throws TooMuchTreasureException;

        /**
         * Removes a Treasure object from the container
         * @param treasure      Treasure item to remove
         */
        public void removeTreasure(Treasure treasure);

        /**
         * Removes first treasure from the container
         (combine with sortTreasureInc/Dec)
         * @return a Treasure object
         */
        public Treasure removeFirstTreasure();

        /**
         * Removes last treasure from the container
         (combine with sortTreasureInc/Dec)
         * @return a Treasure object
         */
        public Treasure removeLastTreasure();

        /**
         * Gets current number of Treasure objects stored
         * @return number of treasure stored
         */
        public int getTreasureNum();

        /**
         * Returns all the Treasure objects stored
         * @return ArrayList of Treasure stored
         */
        public ArrayList<Treasure> getTreasureList();
```

```
        /**
         * Sorts treasure pieces according to their value, increasingly
         */
        public void sortTreasureInc();

        /**
         * Sorts treasure pieces according to their value, decreasingly
         */
        public void sortTreasureDec();
}
```

5 DETAILED DESIGN

5.1 Typical Use Cases and Expected Program Behaviour

5.1.1 Program behaviour/scenario

The following set of flowcharts covers the functional requirements in the requirement specification and how the program should behave in each one: FR visualisation chart

5.1.2 Activity diagrams

Below diagrams should provide general insight on the order of calls and the implementation of the Model-View-Controller pattern.

1. Game Setup, choosing names, starting the main game window
2. Main game window - updateView() method
3. Main game window - moving / pressing tiles on the grid
4. Main game window - Fighting sequence + Popup_Customised template
5. Main game window - Interaction Controller - (ports/islands)
6. Main game window - Interaction Controller - Chance Cards

5.2 State Machine Description - Valid states, State-holders names

This section explains states and their valid values in the state machine, with explanation for each of them. The states are presented below in a *Keystring - value1/value2/...* schema. All states are acquired from the GameEngine through *State getState(stateName : String)* method.
This section is mostly for UI use, but not restricted to it. Any states agreed upon and used by the program must be kept up to date in related functions

5.2.1 Turns

- turn - t01/t02/t03/t04 depending on the current player's turn. The 'turn' value represents the correct ordering of turns, excluding the short interruptions e.g. while fighting.
- action - 'prepare' if the player switch is in progress and the system is awaiting next player's confirmation of taking over, 'move' if waiting for the player to select where to move to, 'attmove' if the player is retreating after a fight rather than moving on their own, 'turn' if current player is turning, 'attturn' if the turning player is the one who just lost a fight

5.2.2 Other

- attloser - t01/t02/t03/t04. Used to store the losing player after fighting sequences. Should be checked if the action's state is 'attmove'

5.3 Logic Between States and Windows (moved from UI document[2])

The following is based on and referred to the User Interface specification document's[2] use cases.

1. **W_Setup** - Clicking the *Start Game* button should begin the game and proceed to the Main Game Screen W_Game while setting states: "turn" to State.t01, "action" to State.prepare.
2. **W_Game** - Based on the **"action"** state, the game window can display/behave in a different way ([U06]):
   - **[U06a] State.prepare** - this state is set at the start of the game and after each player's turn. It means that the current player's turn is beginning and they should press a button to begin making decisions and view their status. At this stage it should be impossible to see individual player's cards, as this state is used to let players 'switch' on the seat in-between turns, without disclosing sensitive information. Button "More info" should not be available at this point of the game. Clicking the ready/begin turn button should change the "action" state to State.move and follow the [U06b] sequence.
   - **[U06b] State.move** - this state is set after the player accepted that it's their turn. Player is able to move. Rotating the ship happens by clicking the player's ship (selecting 0 steps to take) and selecting the direction to turn to (see [U09] Turning the Ship). The board should represent available squares to move to, by highlighting them. Pressing one of the squares should move the player's ship to that position.
   - **[U06c] State.attmove** - this state is set after a player has lost a fight and is retreating. The behaviour should be similar to above, but the player being moved is chosen based on "attloser" field and the move cannot happen on special tiles (e.g. port).
   - **State.turn/State.attturn** - this state should happen after every move/when sequence [U09] is executed. If the state is State.turn, choose the person to display the moving buttons for and to change ship's direction based on "turn" state. Otherwise do the same based on "attloser" state.
   - **Fighting rules** - the "attloser" variable state should be set to State.t01/2/3/4 depending on the losing player's number, the "action" variable should be set to State.attmove after a fight.
3. **FightResult popup** - the "attloser" variable state should be set to State.t01/2/3/4 depending on the losing player's number, the "action" variable should be set to State.attmove.

5.4 Use-Case Scenarios Detailed Implementation

This section will cover all scenarios present in the UI Specification, explaining their implementation and logic. All windows are controlled by controller classes (all class names that start with *W_*) which aren't instantiated by the programmer directly, but through the FXML file loader, a method provided by JavaFX (see dev/sceneBuilderDemo on SVN). An instance of a particular controller can be then acquired through the loader instance, and set up using "SetUpController" method. This will map buttons to corresponding functions prepared in the window controller. After that the Stage (window) should be shown with appropriate modality (controls if the calling window can be interacted with before the modal window disappears, or if it's disabled for that duration) and with appropriate showing method (that decides whether the current executing method should keep executing lines of code or stop executing until the new window is closed).

5.4.1 Starting the game

The Application class should create an instance of GameEngine and pass it to the W_MainMenu controller and show it afterwards. From now on everything will be event-based and StateMachine (present in the GameEngine passed) dependent. The main menu window buttons should open different windows depending on what's clicked (credits, rules, start game). In case of simple credits/rules windows, the corresponding controller should get used to set the scene up and then the window should be shown (with menu still visible, but not interactive, in the background). In case of clicking "Start game" button, the main menu window should be hidden and the W_Setup controlled window should be shown. The inputs should not have any particular code assigned to them, but the button confirming has to call (multiple if more convenient) method(s) that check the validity of the names, before executing code responsible for setting the game up. After it's done, the W_Game window should be shown, and the setup window hidden (use one main method to control the execution of other helper methods that get executed on button press). Going back should hide the window and display the main menu (the caller) window.

5.4.2 Game behaviour

Screen should be updated whenever the state of game is changed and/or underlying data is changed (the UpdateView() method).
Clicking on the board should make the game act according to the state of "turn" and "action" (sometimes also "attloser") which is explained above. If any windows need showing (e.g. trade/attacking confirmation) related data has to be passed to the controller. In trading situation it would mean passing the GameEngine (trading window would then check which turn's turn it is, see what port is that player's ship at and display items for both sides of trade accordingly). Buttons should react based on the GameEngine data provided and data available in the current window (e.g. in trading the confirmation button would call the method that checks values for both sides and change the data + states accordingly or display an error popup). The code for buttons should be universal - once written, a method in the controller should be able to handle all situations it might need to accordingly, no matter which player is currently in control. Stopping the execution of code, waiting for a window to finish and preventing user from clicking outside the window where required is crucial (modality + show method).

5.4.3 Popups

Popup classes are available. These do not use FXML files and need to be written manually, but offer more flexibility and can be used dynamically. Any UI Node objects that need to be displayed should be passed to the constructor of a Popup class and then getStage() method allows the programmer to get a prepared window, ready for display. It allows reuse of simple windows, where minor changes are required (e.g. a chance card that lets user select another player at TreasureIsland would display equal number of buttons to number of other players at the coast of the TreasureIsland, which can differ from 0-3, each having their own, personalised action depending on user's choice). Buttons and their actions should be prepared beforehand and then passed while creating the Popup object. To see when simple popups can be used, refer to the UI Specification's[2] use-cases.

5.4.4 Chance Cards

Depending on the number of the card, a particular static method with required parameters has to be called inside IslandInteraction abstract class (or other classes as needed). This will be handled by the InteractionController class, which will select appropriate code based on the drawn card's number. The InteractionController should distinguish any actions UI-side of the software might need to perform (ask for user's choice, display data), but data modification should be performed by those static methods.

5.4.5 Passing through players

W_FightConfirm's goal is to ask the static player if they wish to attack the passing player. The passing player can be determined by the window itself (from "turn"'s current state in the GameEngine), but the result of the move (if no action taken) should be stored in the controller, to move the initial ship to the previously selected destination if needed.

5.5 Other Notes

- Use of helper methods/variables is allowed as long as the expected functionality and the result format is met and should be made private. If a new class or a public method is required, it should be added in the design specification.
- If any change in the design is required for the system to work properly and efficiently, all team members should be notified about said change.

REFERENCES

[1] *Software Engineering Group Projects* Buccaneer Online Board Game Requirements Specification C. J. Price, SE.QA.RS - CS22120, 1.1 Release.

[2] *Software Engineering Group Projects, Group 07* User Interface Specification, K. Cupial, J. Wojciechowska, 1.1 Release.

DOCUMENT HISTORY

Version	CCF No.	Date	Changes made to Document	Changed by
0.1	N/A	2017-02-05	Initial creation	KAC12
0.11	N/A	2017-02-05	Added 5.1.1 Program behaviour/scenario	MAJ56
0.2	N/A	2017-02-16	Minor changes, changed state to 'for review'	KAC12
0.3	N/A	2017-02-18	Added TooMuchTreasureException, EmptyPackException	KAC12
0.4	N/A	2017-02-24	Removed 'no UI design' part	KAC12
0.45	N/A	2017-02-28	Added QA Comments	HAA14
0.5	N/A	2017-03-04	Updated states according to UI spec part	KAC12
0.6	N/A	2017-03-10	Added CardHolder interface, updated classes descriptions with the new interface, added state descriptions for W_Setup, W_Game	KAC12
0.7	N/A	2017-03-11	Moved states according from UI spec part (finished)	KAC12
0.8	N/A	2017-03-12	Changed Port class from abstract to instantiable	KAC12
0.81	N/A	2017-03-18	Added enum TreasureType	KAC12
0.82	N/A	2017-03-18	Added InvalidStateException	KAC12
0.83	N/A	2017-03-25	Added sequence diagrams 1, 2	KAC12
0.84	N/A	2017-03-26	Added sequence diagrams 3, 4	KAC12
0.85	N/A	2017-03-29	Renamed sequence diagrams to activity diagrams, added 4. INTERFACE DESCRIPTION, fixed UML diagram link	KAC12
0.86	N/A	2017-03-30	Added Interaction Activity diagram and interface descriptions	KAC12
0.9	N/A	2017-04-01	3.1, 3.2 - added Model-View-Controller and package description; removed 5.2.1 GameEngine-related states, removed "game" state string as not necessary nor used; added sortInc() and sortDec() to CardHolder	KAC12
0.91	N/A	2017-04-03	Added UI classes description	MAJ56
1.0	N/A	2017-04-04	Release	TEAM