Bilkent University

Department of Computer Engineering

Senior Design Project

Project short-name: Pandora

Low Level Design Report

Taner Baygün – 21300987

Mert Armağan Sarı – 21401861

Berfu Anıl – 21401502

Serhan Gürsoy – 21400840

Ege Yosunkaya – 21402025

Supervisor: Halil Altay Güvenir

Jury Members:

Özgür Ulusoy

Muhammet Mustafa Özdal

Innovation Expert: Cem Çimenbiçer

Website: https://thepandora.github.io/boxproject/

Feb 12, 2018

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Table Of Contents

1. Introduction

Technology has affected our lives in a way that playing an interactive game without technology is almost impossible. There is an undeniable fact that technology has taken over the gaming industry, which are mostly computer based. People want to play games with their friends, yet machines lock them up on screens so that in a social manner, people stop talking with each other when they play games. The reason is that they need to follow the every single state of the game via computers or smartphones. Therefore, this creates a more asocial environment for those who want to play games interactively with their friends.

Pandora is designed as a small box that brings people together at any place. Pandora aims to improve face-to-face and interactive communication in a fun way for the players. Users need to connect the box via smartphones or computers just after they have finished the setup. They can register as an admin or guest. Afterwards, they can select a game and enter the game lobby. Once every single player is connected to the selected game, people can start to play the game interactively.

In this report, we stated the low-level architecture and the fundamental design of the project in detail. We made a point of trade-off issue, followed by package and class interface designs and all the functionalities are briefly described.

1.1 Object Design and Trade-offs

Performance vs Scalability

Pandora is a box that serves games via Raspberry Pi Zero W. It is essential to stay above a certain response time, not to annoy users. While players are playing the game, latency is undesirable. Number of players and response time of the box is inversely proportional. Raspberry Pi Zero W has 512 MB RAM and 1GHz single-core CPU [1]. So, it can cause a problem to depend on its hardware specifications to run a server with high requests. We have to optimize the performance with this respect.

Complexity vs Usability

Pandora tries to optimize users' enjoy. In the background Pandora should handle the server and client related business. However Pandora should be playable to any user. To achieve that goal Pandora's GUI will be simple. While keeping the GUI simple, another challenge is serving system functionality. We are going to feature usability over complexity because main purpose of the system is to provide high quality experience.

Reliability vs Cost

Pandora consists of both hardware and software, therefore the sale price depends on both engineering work and hardware cost. Pandora uses Raspberry Pi Zero which has the most stable wifi connector among Raspberry family, yet there are more costly chips. In order to keep the cost under an affordable limit of 150 TL we choose Raspberry Pi Zero and we sacrificed slightly from reliability.

Space vs Cost

Pandora is using a Raspberry microPC to serve as a host and its operating system is stored in an SD card with relevant data about the servers, games, admin user and his/her customizations. The available space in the SD card can cause a problem because it is always finite. In order to store the operating system and Raspberry Pi's own files, 4GB is needed in an SD card but 8GB is recommended by the Raspberry Pi's developers [2]. So, we chose to use 16GB SD cards in the Pandora to store our server files with the required database.

1.2 Interface Documentation Guidelines

In the 3rd section, the class interfaces are explained as in the following format:

Class Name
Class Description
Properties
Methods

1.3 Engineering Standards

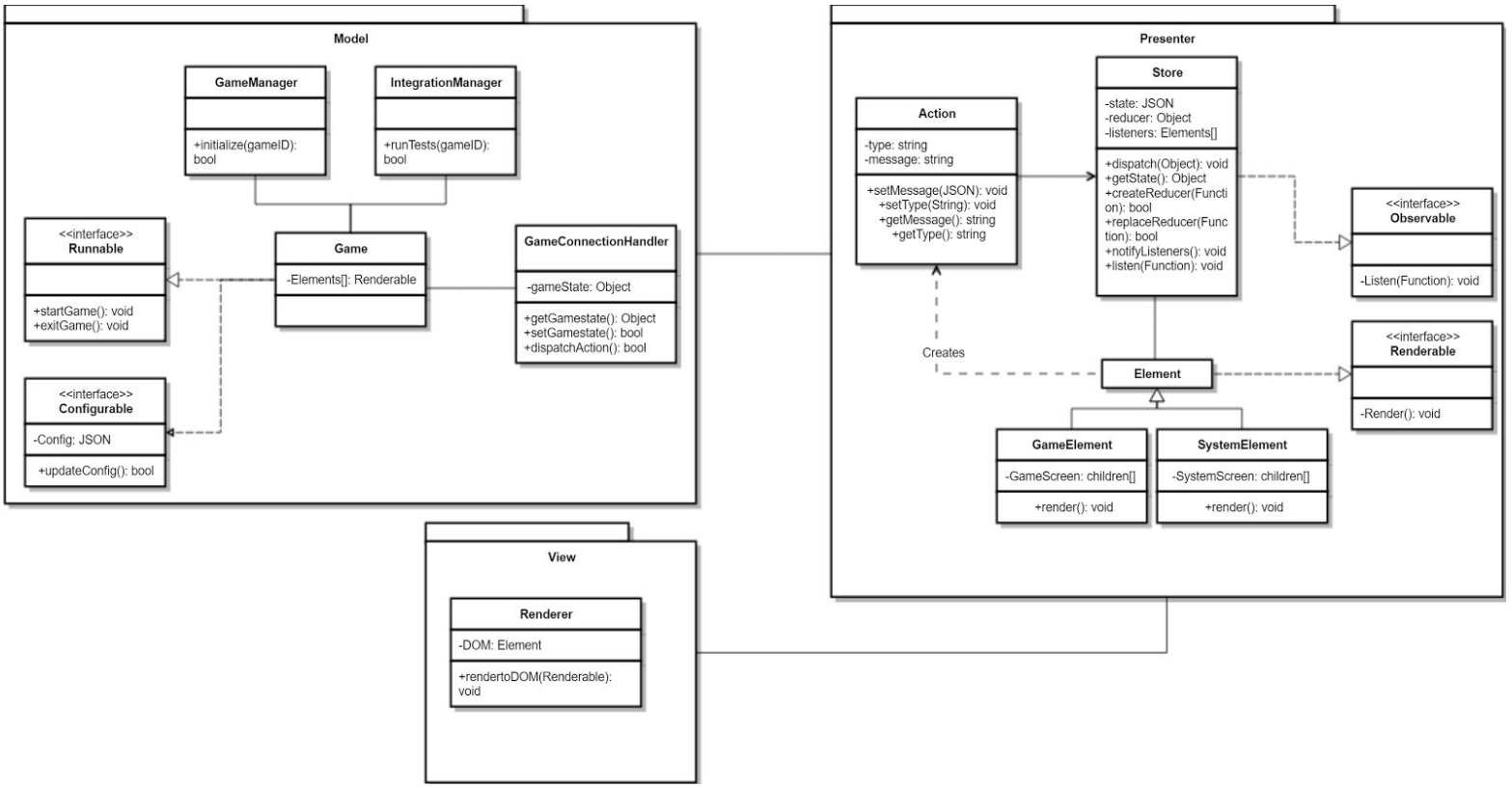
To show class interfaces, diagrams, and use cases we used UML [3] notation in all reports. The UML notation is commonly used to explain a software system in detail among IT professionals and developers. The citation in the reports are done within IEEE standards [4]. The hardware of the product is used under the standards of Raspberry Pi Zero W [5], which includes health & safety subjects and etc.

1.4 Definitions, Acronyms and Abbreviations

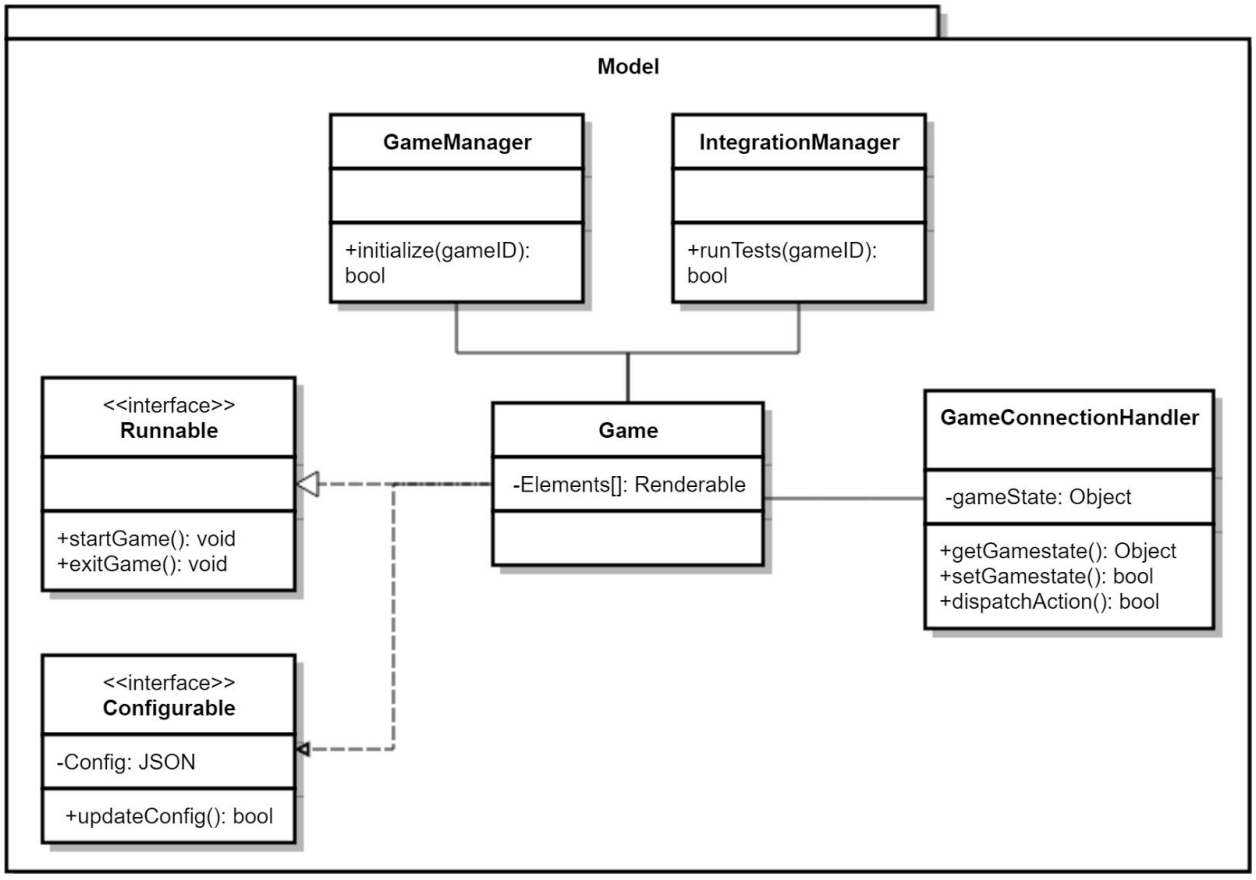
API: Application Programming Interface
RAM: Random Access Memories
CPU: Central Processing Unit
GHz: Gigahertz
Client: The part of the system the users interact with
Server: The part of the system responsible from logical operations, scheduling, and data management
UML: Unified Modeling Language
JS: JavaScript
DB: Database
GUI: Graphical User Interface
DOM: Document Object Model

2. Packages

2.1 Application Package



2.1.1. Model



GameManager: GameManager class is responsible for initializing installed games.

IntegrationManager: IntegrationManager class is responsible for running tests for installed games , make sure games' configuration files are structured properly.

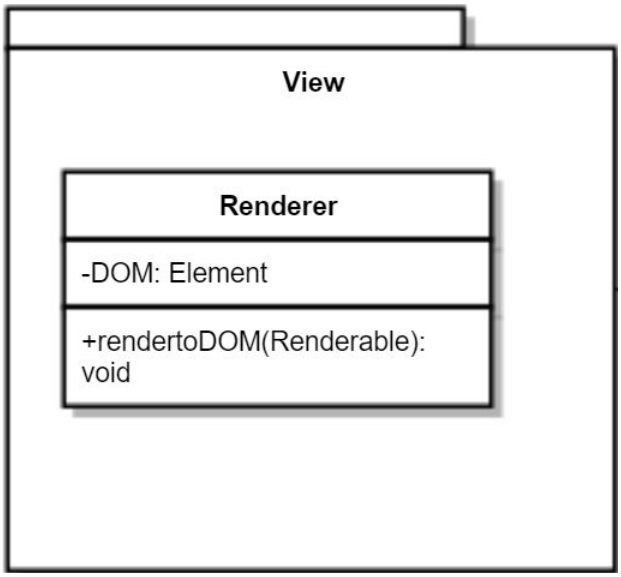
GameConnectionHandler: GameConnectionHandler class handles the connection between the game and the system management subsystem. Games can synchronize their state by pooling from server the current state and sending the state for other game instances to update their states using this class.

Runnable: This interface provides start and exit methods to the game and makes GameManager class' initialize method more generic

Configurable: This interface again is to create more generic approach to game class , providing a configuration interface to update game configurations.

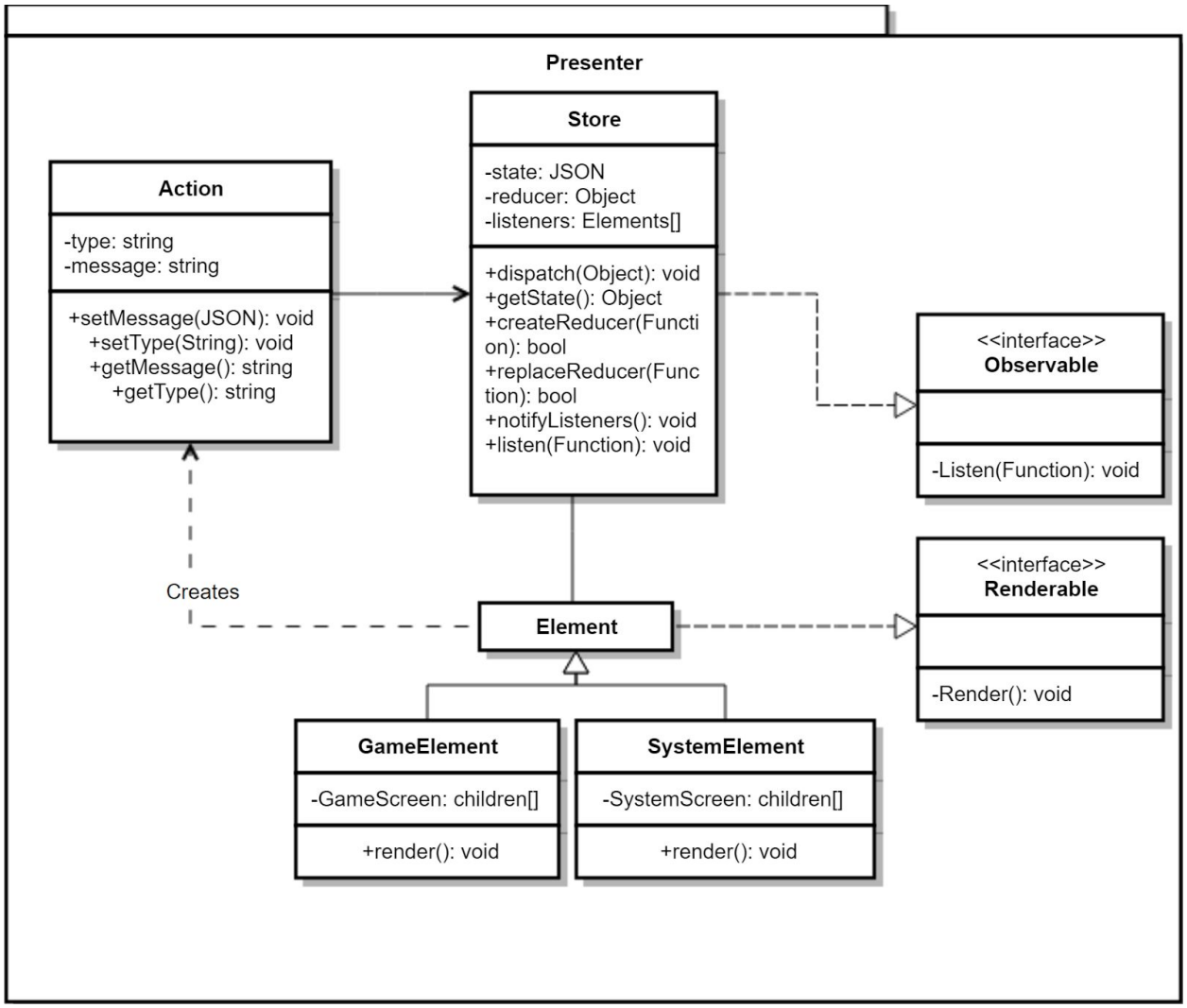
Game: Game class is a representation of a game. Every game has its own implementation and this class is only a generic class for the entry point , every game implements runnable and configurable interfaces to integrate with the system, however game logic differs from game to game. Our system designed that way to making game development for the system easier.

2.1.2. View



Renderer: Renderer is responsible for DOM manipulations.

2.1.3 Presenter



Store: Store class is the “single source of data” for the presenter. It holds the state, elements dispatch Actions to change the state. The store notifies the elements subscribed to it when the state has changed.

Action: Creating Action objects and dispatching it to the Store is the way of communication between the elements and the store. It provides queueing of Actions if different elements tries to communicate with the store concurrently.

Observable: This interface is for listening to store for changes, it was not necessary however it shows the Observable pattern more clear.

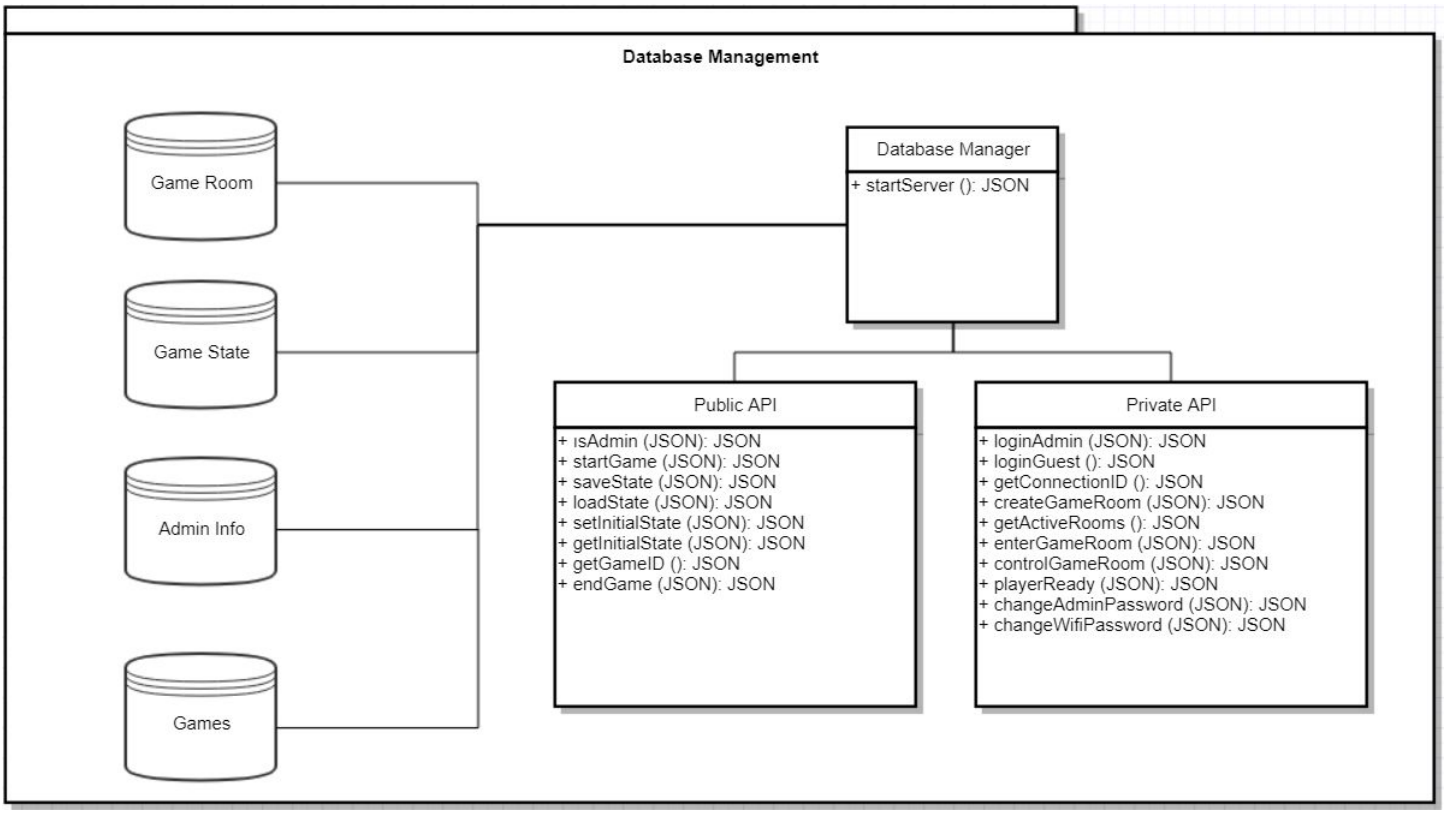
Renderable: The Renderable interface is for elements to be rendered for display. The render function returns a HTML element for the Renderer in the View package to get the element and render to the DOM and for the browser display.

Element: Element class is a generic class for the system and game elements. The various element objects composed to create a screen for the GUI. An element can be act like a container via having children elements.

GameElement: GameElements belong to a game shown on the screen according to the state of the store(e.g if a specific game is in progress).

SystemElement: SystemElements are shown on the screen according to the state of the store (e.g. if no game is in progress).

2.2 System Package



The subsystem is responsible for managing the database software and create an abstraction level for other subsystems to use.

Database Manager: The class that makes requests and retrieves responses from the database. It has a logic for the transactions , data retrieval and aliasing to prevent data loss.

Admin Info: Represents admin and its credentials.

Game State: Represents a saved game state in the database.

Game Room: Represents a current game rooms in the database.

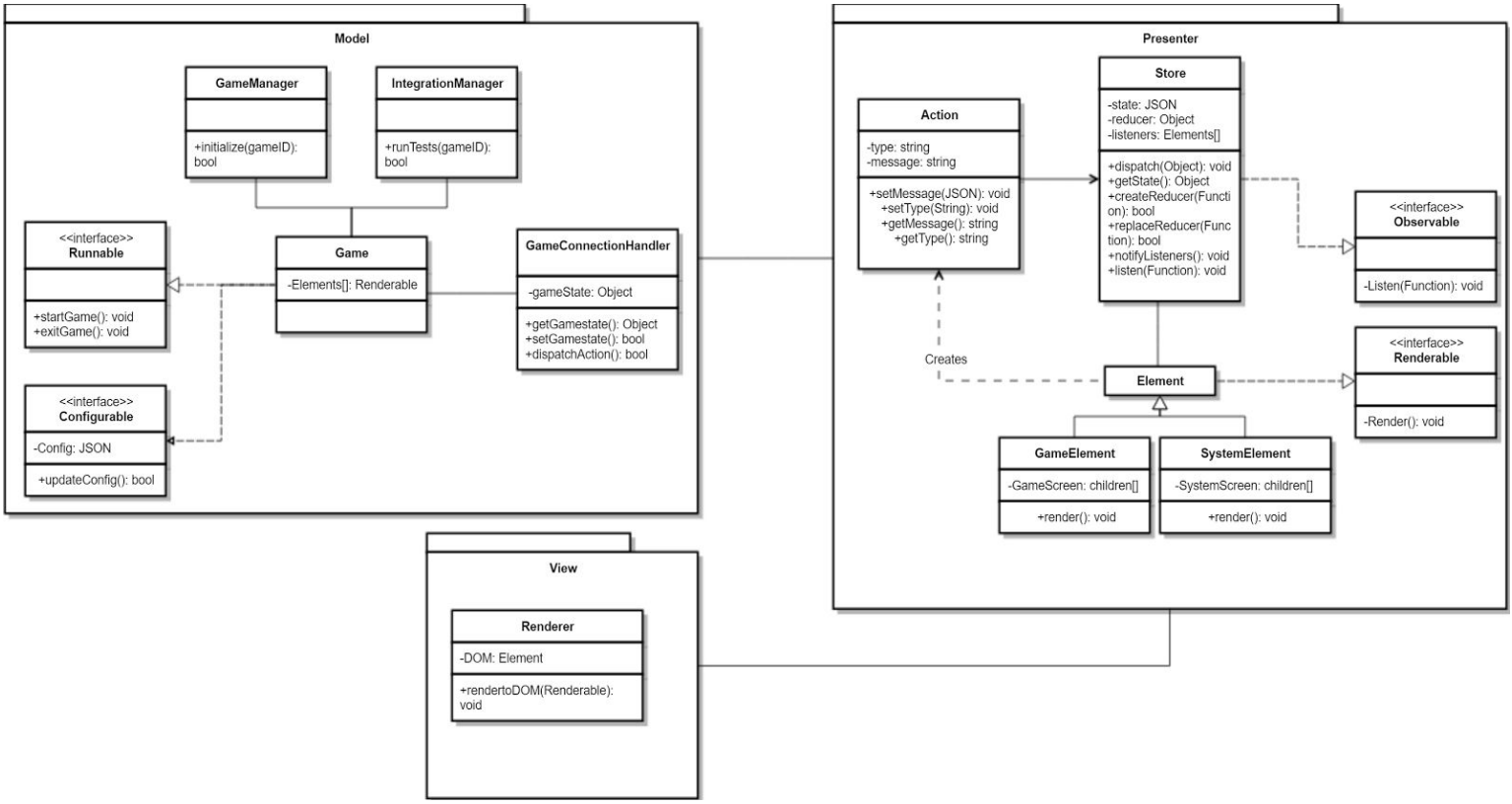
Games: Represents the available games to play in the database.

Public API: An Application User Interface for other subsystems and developers to use to make basic database operations.

Private API: An Application User Interface for the box, developers to use, works with a key for authentication purposes. It used to get the statistics, access system related documents etc.

3. Class Interfaces

3.1. Application Package



3.1.1. Model

class GameManager
GameManager class is responsible for initializing installed games.
Properties
private Game[] installedGames
Methods
initialize(gameID)

class IntegrationManager
IntegrationManager class is responsible for running tests for installed games , make sure games' configuration files are structured properly.
Properties
private Game[] installedGames
Methods
runTests(GameID)

class GameConnectionHandler
GameConnectionHandler class handles the connection between the game and the system management subsystem. Games can synchronize their state by pooling from server the current state and sending the state for other game instances to update their states using this class.
Properties
private JSON gameState
Methods
getGameState(gameID) sendGameState(gameID , state) dispatchActionToServer(JSON)

Interface Runnable
This interface provides start and exit methods to the game and makes GameManager class' initialize method more generic.
Methods
startGame() exitGame()

Interface Configurable
This interface again is to create more generic approach to game class , providing a configuration interface to update game configurations.
Methods
updateConfiguration(Configuration)

class Game implements Runnable , Configurable
Game class is a representation of a game.
Properties
private List<Renderable> Elements
Methods
/* Every game has its own implementation and this class is only a generic class for the entry point , every game implements runnable and configurable interfaces to integrate with the system however game logic differs from game to game */

3.1.2. Presenter

class Store implements Observable
Store class is the “single source of data” for the presenter. It holds the state, elements dispatch Actions to change the state. The store notifies the elements subscribed to it when the state has changed
Properties
private JSON state private Reducer reducer private Element[] listeners private Dispatcher dispatcher
Methods
processAction(Action) getState() createReducer(Function) replaceReducer(Function) notifyListeners() listen(Function)

interface Observable
This interface is for listening to store for changes, it was not necessary however it shows the Observable pattern more clear.
Methods
listen(Function)

interface Renderable
The Renderable interface is for elements to be rendered for display. The render function returns a HTML element for the Renderer in the View package to get the element and render to the DOM and for the browser display.
Methods
render()

object Dispatcher
The Dispatcher is a singleton used for dispatch Action objects to the store.
Properties
private Queue<Action> waitingActions
Methods
dispatch(Action)

class Element implements Renderable
Element class is a generic class for the system and game elements. The various element objects composed to create a screen for the GUI. An element can be act like a container via having children elements.
Properties
private children[] Element
Methods
render()

Class GameElement extends Element
GameElements belongs to a game shown on the screen according to the state of the store(e.g if a specific game is in progress)
Properties
private children[] GameScreen
Methods
render()

Class SystemElement extends Element
SystemElements shown on the screen according to the state of the store (e.g. if no game is in progress).
Properties
private children[] SystemScreen
Methods
render()

Class Action
Creating Action objects and dispatching it to the Store is the way of communication between the elements and the store. It provides queueing of Actions if different elements tries to communicate with the store concurrently.
Properties
private type private Message
Methods
setMessage(JSON) setType(String) getMessage() getType()

3.1.3. View

class Renderer
Renderer is responsible for DOM manipulations.
Properties
private DOM private Renderable[] screenElements
Methods
renderToDOM(Renderable) addElementToDOM(Renderable) removeElementFromDOM(Renderable)

3.2 Server

Public API:

Class Public API
Public API is designed to help developer in development stage. It contains essential methods to use access game states, determining admin (who can change the game settings) and game stages (start/end) etc.
Properties
Methods
isAdmin (JSON): JSON startGame (JSON): JSON saveState (JSON): JSON loadState (JSON): JSON setInitialState (JSON): JSON getInitialState (JSON): JSON getGameID (): JSON endGame (JSON): JSON

Private API:

Class Private API
Private API is designed to administrate backstage work of the Pandora. It contains methods that not users nor developers should interact with to keep Pandora secure. Private API controls game rooms, connections and authorizations.
Properties
Methods
loginAdmin (JSON): JSON loginGuest (): JSON getConnectionID (): JSON createGameRoom (JSON): JSON getActiveRooms (): JSON enterGameRoom (JSON): JSON controlGameRoom (JSON): JSON playerReady (JSON): JSON changeAdminPassword (JSON): JSON changeWifiPassword (JSON): JSON

Database Manager:

Class Public API
Database Manager is designed to run the server with the initial adjustments within the device.
Properties
Methods
+ startServer (): JSON

4. Glossary

Graphical User Interface: GUI is the interface that interacts with users with dynamic images, buttons and labels. Users can navigate between pages by using the interface with simple operations such as clicking a button. The device responds to the user according to the operation and the given input.

Model-View-Presenter: Model–View–Presenter (MVP) is a derivation of the Model View Controller (MVC) architectural pattern, and is used mostly for building user interfaces. In MVP, the presenter assumes the functionality of the "middle-man". In MVP, all presentation logic is pushed to the presenter [6].

5. References

[1] "RASPBERRY PI ZERO W", Raspberry Pi. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-zero-w/. [Accessed: 06- Feb- 2018].

[2] "SD Cards", Raspberry Pi. [Online]. Available: https://www.raspberrypi.org/documentation/installation/sd-cards.md. [Accessed: 09- Feb- 2018].

[3] "An introduction to the Unified Modeling Language", IBM, 2018. [Online]. Available: https://www.ibm.com/developerworks/rational/library/769.html. [Accessed: 02- Feb- 2018].

[4] "IEEE Editorial Style Manual", IEEE. [Online]. Available: https://www.ieee.org/conferences_events/conferences/publishing/style_references_manual.pdf. [Accessed: 05- Feb- 2018].

[5] J. Adams, European Declaration of Conformity. Cambridge: Raspberry Pi, 2017.

[6] "Model–view–presenter", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter. [Accessed: 20- Dec- 2017.