SDV602 Assessment Milestone 3

Tim Gentry

16/11/2017

Contents

[1. Summary 3](#_Toc498637831)

[2. Game Concept Design 4](#_Toc498637832)

[2.1 Design Considerations 4](#_Toc498637833)

[2.2 Design Development 4](#_Toc498637834)

[3. Game Interface Design 5](#_Toc498637835)

[4. Game System Design 9](#_Toc498637836)

[4.1 Use Case Diagram 9](#_Toc498637837)

[4.2 Functional Test Cases 12](#_Toc498637838)

[4.3 Architectural Pattern 14](#_Toc498637839)

[4.4 Analysis Class Diagram 15](#_Toc498637840)

[4.5 Data Model - Entity Relationship Diagram 17](#_Toc498637841)

[4.6 Class Responsibilities and Collaboration Cards 19](#_Toc498637842)

[4.7 Class Diagram 25](#_Toc498637846)

[4.8 Sequence Diagrams 26](#_Toc498637847)

[5. Multiplayer System Design 27](#_Toc498637848)

[5.1 HTTP Login System 27](#_Toc498637849)

[5.2 MQTT Client Syncing 28](#_Toc498637850)

[6. Using the Device’s Sensor 28](#_Toc498637851)

[7. Summary of Development Process 29](#_Toc498637852)

# 1. Summary

This report covers the work done for Milestone One, which covers three areas development for the game.

The first section covers development of the game’s concept. How the game will play, what the player will be expected to do.

The second section covers development of the game’s user interface. How the player will see the game and how they will be able move between different views within the interface.

The third section covers the development of the game’s system. Going into development of the functionality the player will be able to access and brief development of the organisation for the individual components of the system and how the all relate to each other.

The project files used to develop the application for this project are too large to upload to the Moodle dropbox and should be downloaded from the GitHub repository found at

**https://github.com/tgentrynz/SDV602-Assessment**

# 2. Game Concept Design

## 2.1 Design Considerations

The basic idea behind the game being developed is a text based adventure game. Through research of games the fit similar descriptions the following key points were identified as the focus of this game’s system.

* The user will control the game through text input.
* The game should simulate an environment and allow the user to move between locations within this environment.
* The game should allow for interactions between the player and objects within the environment to create some form of puzzle mechanic.

## 2.2 Design Development

Based on these considerations and research of similar game concepts, it was decided that the game for this project would focus on an escape room scenario.

The environment for this kind of game would take the form of some indoor space with an exit that is sealed based on some condition. The players’ goal would be to move around the environment and find the correct key item to unseal the exit and escape the enclosed environment.

Despite the name, “escape room”, the environment for this game should simulate a building or a dungeon, some larger structure that can be made of more than a single room. This allows for more player movement, creating multiple stages for puzzles to reduce them becoming cluttered and keep the objectives clear.

The following section provides an example of how this game concept could play.

The player could start in a room directly facing the exit, which takes the form of a locked door. Other than the door, there is a lever in the room, but it’s handle is missing. There is an existing exit, which leads to a side room.

Being unable to use the locked door or broken handle, the player will move to the side room, which contains a lever handle. Returning to the first room, the player can use the lever handle with the lever. This doesn’t unlock the exit, instead opening a secret doorway.

Entering through the secret doorway, the player enters a room that contains a key. After returning to the first room, the player can use the key on the exit door to unlock it. Allowing them access to the victory area.

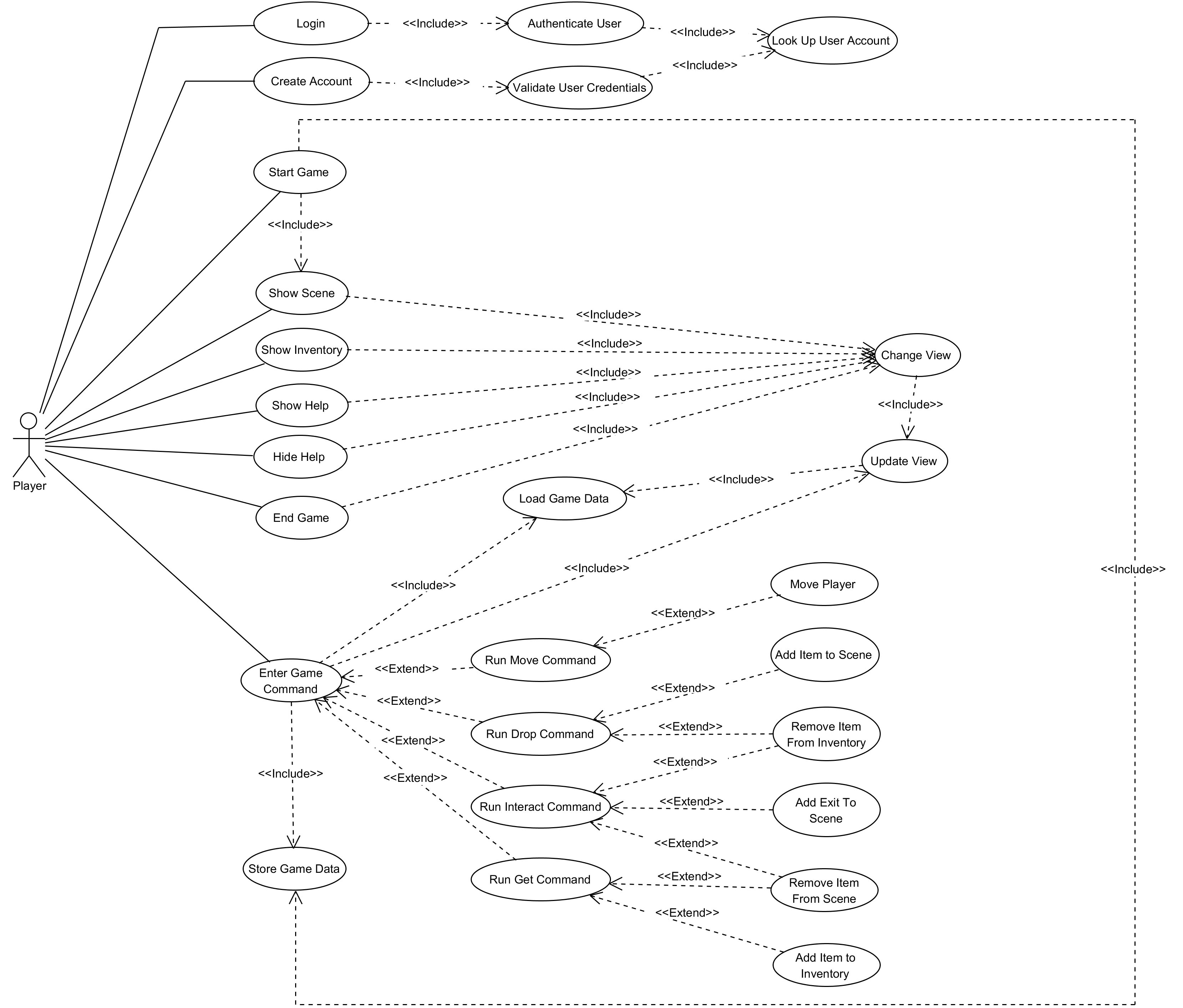
# 3. Game Interface Design

|  |  |
| --- | --- |
| C:\Users\TimTen\AppData\Local\Microsoft\Windows\INetCache\Content.Word\s1.png | **Board 1 – Start Screen**  This is the first screen visible upon launching the application.  The top of the screen displays the game’s title.  The middle of the screen has the two main buttons, the login button will go to **Board 2** when pressed and the create button will go to **Board 3** when pressed.  The bottom of the screen has the return button, which will end the application when pressed. |
|  |  |
| C:\Users\TimTen\AppData\Local\Microsoft\Windows\INetCache\Content.Word\s2.png | **Board 2 – Login Screen**  This is the screen show to the player when they want to log in.  The top of the screen displays the game’s title. and underneath that is a text label which will update to explain to a user what they need to do.  The middle of the screen has two entry fields for the username and password and the login button will go to **Board 4** when pressed.  The bottom of the screen has the return button, which will go to **Board 1** when pressed. |
| C:\Users\TimTen\AppData\Local\Microsoft\Windows\INetCache\Content.Word\s3.png | **Board 3 – Account Creation Screen**  This is the screen show to the player when they want to log in.  The top of the screen displays the game’s title. and underneath that is a text label which will update to explain to a user what they need to do.  The middle of the screen has two entry fields for the username and password and the create button will go to **Board 4** when pressed.  The bottom of the screen has the return button, which will go to **Board 1** when pressed. |
|  |  |
|  | **Board 4 – Game Title Screen**  This is the screen visible to the player after logging in.  The main content on the screen shows the name of the game and provides a list of commands the player can enter to start the game, receive further instruction or end the game.  The bottom of the screen holds two text boxes, one is read only and displays feedback to the player’s inputs and the other is the text box the player uses to enter inputs.  Typing “new” or “continue” will cause the application to proceed to **Board 5**, typing “help” will have the application proceed to **Board 7** and typing “exit” will return to **Board 3**. |
|  | **Board 5 – Scene Screen**  This screen is how the player views the game environment.  The main content on the screen provides a description of the location, a list of items present in the location and a list of exits to other locations in the game environment. It is backed by an image the represents the game environment.  The bottom of the screen holds two text boxes, one is read only and displays feedback to the player’s inputs and the other is the text box the player uses to enter inputs.  Typing “show items” will cause the application to proceed to **Board 6**, typing help will have the application proceed to **Board 7** and typing “exit” will return to **Board 4**. |
|  |  |
|  | **Board 6 – Inventory Screen**  This screen displays the items that a player has in their inventory.  The main content shows a list of items that are in the player’s inventory backed by an image of a bag.  The bottom of the screen holds two text boxes, one is read only and displays feedback to the player’s inputs and the other is the text box the player uses to enter inputs.  Typing “show room” will cause the application to return to **Board 3**, typing “help” will have the application proceed to **Board 7** and typing “exit” will return to **Board 4**. |
|  | **Board 7 – Help Screen**  This screen displays a list of commands the player can use to operate the game.  The main content shows a list of how each command is typed and what it does.  The bottom of the screen holds two text boxes, one is read only and displays feedback to the player’s inputs and the other is the text box the player uses to enter inputs.  Typing “close” will cause the application to return to the last board before it landed on this one. |

# 4. Game System Design

## 4.1 Use Case Diagram

To operate the game, users need to be able to switch between views, enter game commands and see updates to the game information. These use cases were further developed into the following use case diagram.



The following section contains a description for each use case presented in the diagram.

**Login**

This use case is accessible from the initial view. A user presses the login button, enters their username and password and then presses a second login button. This logs the player in and then changes the UI to the game’s title screen.

**Create Account**

This use case is accessible from the initial view. A user presses the create account button, enters their desired username and password and then presses a second create account button. This creates a new account and logs the player into that account. The UI then changes to the game’s title screen.

**Authenticate User**

This use case is accessible to the user by inclusion in the **Login** use case. It takes the username and password that the player entered and searches for a matching account in the game’s data storage through inclusion of the **Look up User Account** use case.

**Validate User Credentials**

This use case is accessible to the user by inclusion in the **Create Account** use case. It takes the username and password that the player entered and makes sure they are within the system’s constraints. Then, through inclusion of the **Look up User Account** use case, it makes sure the username entered does not already exist in the system.

**Look up User Account**

This use case is accessible to the user through inclusion in the **Authenticate User** and **Validate User Credentials** use cases. It searches the game’s data storage for user accounts that match the information provided by the player.

**Start Game**

This use case is accessible from the main menu view. A user types “start” into the input field and a new game instance is created, so the **Store Game Data** use case is included. After a game instance is created, the UI changes so that is now displaying the information about the game scene, this requires that the **Show Scene** use case is included.

**Show Scene**

This use case is directly accessible to the player from the inventory view. A user types “show scene” into the input field and the view changes to the scene view by including the **Change View** use case.

**Show Inventory**

This use case is accessible to the player from the scene view. A user types “show items” into the input field and the view changes to the inventory view by including the **Change View** use case.

**Show Help**

This use case is accessible to the player from all of the other views. A user types “help” into the input field and the view changes to the help view by including the **Change View** use case.

**Close Help**

This use case is accessible to the player from the help view. A user types “close” into the input field and the view returns to the view the user was on before accessing the help menu by including the **Change View** use case.

**End Game**

This use case is accessible to the player from the scene view or the inventory view. A user types “exit” into the input field and the display is changed back to the main menu view by including the **Change View** use case.

**Change View**

This use case is accessible to the player from one of the graphic user interface focussed use cases, **Show Scene**, **Show Inventory**, **Show Help**, **Close Help** or **End Game**. The graphic user interface changes to show the view that the user requested. To make sure that the screen is updated with the most recent information, the **Update View** use case is included by this one.

**Update View**

This use case is accessible to the player from use cases that change information being viewed by the player, **Change View**, or use cases that need to display changes to the game data, **Enter Game Input**. By including the **Load Game Data** use case, it retrieves the most up to date game data and then changes the information presented on the user interface elements.

**Enter Game Command**

This use case is accessible to the user from the scene and inventory views. A user types one of the game environment manipulation commands into the input field. The system then loads the data that needs to be changed through the inclusion of the **Load Game Data** use case. Following this, the specific command entered is performed by one of the extending use cases, **Run Move Command**, **Run Drop Command**, **Run Interact Command** or **Run Get Command**. After the game data is changed the display is updated to show the changes by including the **Update View** use case and then saved through the inclusion of the **Store Game Data** use case.

**Run Move Command**

This use case is accessible to the user by extension of the Enter Game Command use case. The system checks that the game scene the user would like to move to is accessible and, provided it is, the game system updates the player’s position through extending to the **Move Player** use case.

**Run Drop Command**

This use case is accessible to the user by extension of the Enter Game Command use case. The system checks that the item the user would like to drop is in their player inventory and, provided it is, the game system removes the item from the player’s inventory through extending to the **Remove Item from Inventory** use case and places it in the current game scene through extending to the **Add Item to Scene** use case.

**Run Interact Command**

This use case is accessible to the user by extension of the Enter Game Command use case. The system checks the door item the user would like to interact with is present in the current scene and the key item required by it is present in the player’s inventory then, provided these conditions are met, the game system removes the key item from the player’s inventory through extension to the **Remove Item from Inventory** use case, removes the door item from the game scene through extension to the **Remove Item from Scene** use case and finally adds a new exit in the game scene through extension to the **Add Exit to Game Scene** use case.

**Run Get Command.**

This use case is accessible to the user by extension of the Enter Game Command use case. The system checks that the item the player wants to pick up is present in the current game scene and, provided it is, the game system removes the item from the game scene through extension to the **Remove Item from Scene** use case and then add that item to the player’s inventory through extension to the **Add Item To Inventory** use case.

**Move Player**

This use case is accessible to the user by extension of the **Run Move Command** use case. It changes the game scene that the player is positioned in within the game environment.

**Add Item to Scene**

This use case is accessible to the player by extension of the **Run Drop Command** use case. It adds an item to the current game scene.

**Remove Item from Scene**

This use case is accessible to the player by extension of the **Run Interact Command** and **Run Get Command** use cases. It removes an item reference from the current game scene.

**Add Item to Inventory**

This use case is accessible to the player by extension of the **Run Get Command** use case. It adds an item reference to the player inventory data.

**Remove Item from Inventory**

This use case is accessible to the player by extension of the **Run Drop Command** and **Run Interact Command** use cases. It removes an item reference form the player inventory data.

**Add Exit to Scene**

This use case is accessible to the player by extension of the **Run Interact Command** use case. It adds a reference to another game scene to the current game scene.

**Load Game Data**

This use case is accessible to the player through inclusion by the **Enter Game Command** and **Update View** use cases. It retrieves the game’s information from the data storage.

**Save Game Data**

This use case is accessible to the player through inclusion by the **Start Game** and **Enter Game Command** use cases. It saves game information to the data storage.

## 4.2 Functional Test Cases

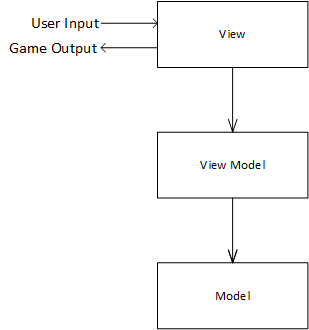
To develop a better understanding of the functionality behind the use cases, some initial test cases were created.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function Tested** | **Use Case** | **System State** | **Input** | **Output** |
| Player starts game. | Start Game | Game is on the title screen. | Player types “start” into input field and presses the enter key. | The game starts and the view changes to the scene screen. |
| Player views the inventory. | Show Inventory | Game is on the scene screen. | Player types “show items” into the input field and presses the enter key. | The game view changes to the inventory screen. |
| Player views the scene. | Show Scene | Game is on the inventory screen. | Player types “show room” into the input field and presses the enter key. | The game view changes to the scene screen. |
| Player views the help screen. | Show Help | Game is on any of the non-help screen screens. | Player types “help” into the input field and presses the enter key. | The game view changes to the help screen. |
| Player closes the help screen. | Close Help | Game is on the help screen. | Player types “close” into the input field and presses enter | The game view changes to the screen prior to the help screen being opened. |
| Player moves. | Run Move Command | Game is running and player is not looking at the help screen. | Player types “goto” and an exit name into the input field and presses enter. | The current scene is changed and the scene screen is updated. |
| Player picks up an item. | Run Get Command | Game is running and player is not looking at the help screen. | Player types “get” and an item name into the input field and presses enter. | The item is removed from the list of items in the scene and added to the list of items in the player’s inventory. |
| Player drops an item. | Run Drop Command | Game is running and player is not looking at the help screen. | Player types “drop” and an item name into the input field and presses enter. | The item is removed from the list of items in the player’s inventory and added to the list of items in the scene. |
| Player uses an item. | Run Interact Command | Game is running and player is not looking at the help screen. | Player types “use” and an item name into the input field and presses enter. | The door item is removed from the list of items in the scene, the door’s key item is removed from the list of items in the player’s inventory and the door’s referenced game scene is add to the list of exits in the current scene. |

## 4.3 Architectural Pattern

To help organise the system behind the game, some elements of the **Model-View-View Model** pattern were used. This involves separating different areas of functionality into individual sections of the codebase. This allows for the different functions of the application to be developed independently of each other. The specific functions that are separated are the **Model,** which represents functionality related to storing and retrieving data; the **View**, which represents functionality related to user interactions; and the **View Model,** which represents the communication functionality between the view and the model.

The relationship between these three sections of the codebase works as follows.

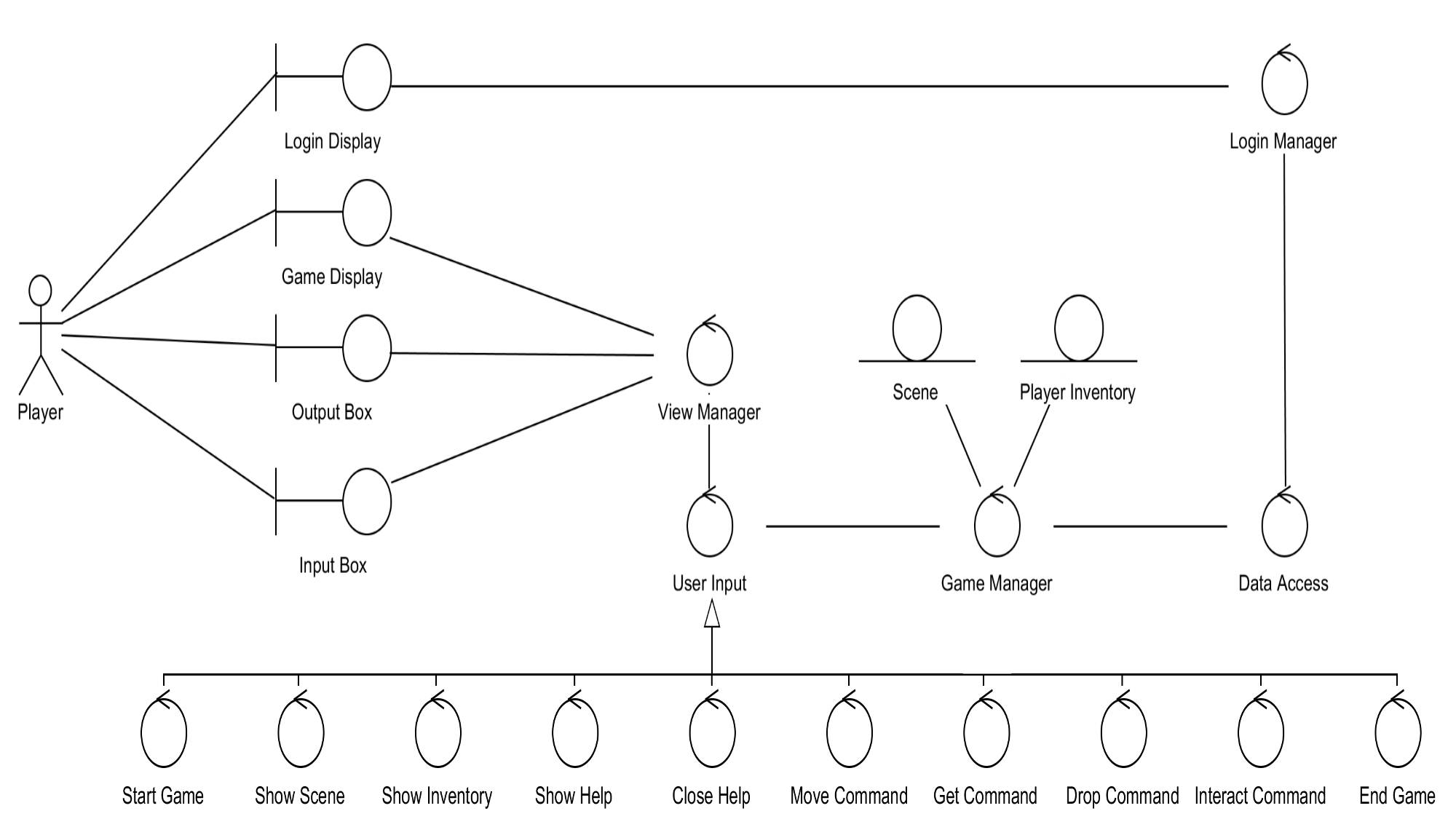


As shown here, the **View** is the aspect of the program that the user interfaces with. It will cover use cases related to the user interface, such as Start Game, Show Scene, Show Inventory, Show Help, Close Help, Change View, Update View and Enter Game Command. The view communicates with **View Model**, which is responsible for the game manipulation use cases, such as Run Move Command, Run Get Command, Run Drop Command, Run Interact Command, Move Player, Add Item to Scene, Add Item to Inventory, Add Exit to Scene, Remove Item from Scene and Remove Item from Inventory. The view model communicates with the **Model**, which handles the data access use cases, Load Game Data and Store Game Data.

These relationships are unidirectional so that the **Model** does not need information about the **View Model** and the **View Model** does not need information about the **View**. It should also work so that the **View** only needs to know enough about the **View Model** to request data from it and shouldn’t need to know anything about how the **Model** operates.

## 4.4 Analysis Class Diagram

Building on the use cases, a map of the potential system was created to help develop the separation of responsibilities and relationships between system components.

****

The following section details each element presented in this diagram.

**Game Display**

This is a boundary class that represents the user interface elements that the player uses to view information about the game environment. It is updated by the **View Manager**.

**Login Display**

This is a boundary class that represents the user interface elements that the player uses to login to the game.

**Login Manager**

This controller class manages the validation and authentication of user account information.

**Output Box**

This boundary class represents the user interface elements that provide feedback to the player’s inputs. It is updated by the **View Manager**.

**Input Box**

This is a boundary class that represents the user interface elements that the player uses to enter input into the system. It updates the **View Manager**.

**View Manager**

This is a controller class that manages the passing of input and output between the user interface classes and the game manipulation classes. It has direct access to the **Game Display**, **Output Box** and **Input Box** classes for user interface processes and direct access to the **Game Manager** class for game information processes. It is also in charge of the creation of **User Input** instances for game manipulation processes.

**User Input**

This is a controller class that manages the changes to the system made by user inputs. It can update both the **View Manager** and **Game Manager** classes. The different realisations of this class, **Start Game**, **Show Scene**, **Show Inventory**, **Show Help**, **Close Help**, **Move Command**, **Get Command**, **Drop Command**, **Interact Command** and **End Game** relate to specific use cases and alter the data in different ways, however the way they communicate with the other components remains the same.

**Game Manager**

This controller class manages changes to the game data. It is updated by the **User Input** class, can send updates to the **View Manager** class and request updates from the **Data Access** class. It can also read and write **Scene** and **Player Inventory** classes.

**Scene**

This entity class holds information about a game scene.

**Player Inventory**

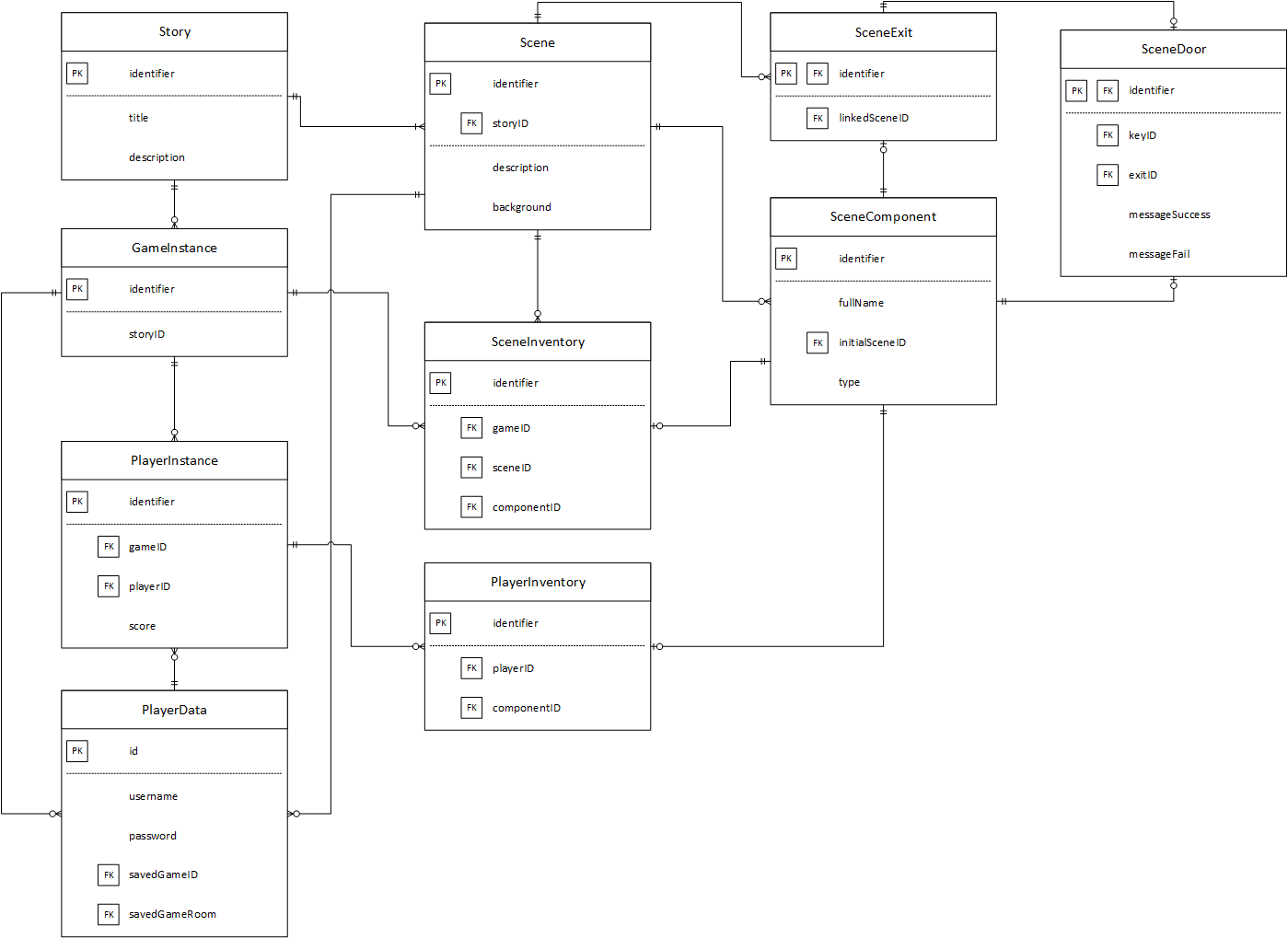
This entity class holds information about items a player has collected.

**Data Access**

This controller class handles requests from the **Game Manager** class for retrieval of game data from the data store.

## 4.5 Data Model - Entity Relationship Diagram

The data storage for the game will be implemented using SQLite, which is a relational database management system. As such, the game’s data including player accounts, game instances and game state need to be stored within a relational database mapping. This ERD was created to develop how the game’s data would be stored.



The following section provides descriptions for the entities in the diagram.

**PlayerData**

This entity represents a user account within the system. It contains a numerical identifier, username and password values for login functionality and ids for the last game the player played and the last room they were in, for use in the save system.

**PlayerInstance**

This entity represents a user playing a game. It contains a numerical identifier, game and player ids, to link a player account to a game instance and a score value, to track a player’s progress in a game instance.

**GameInstance**

This entity represents an instance of the game being played. It has a numerical identifier and a story identifier to help identify and categorise it.

**Story**

This entity represents a story in the game model, which is a collection of related scenes. It has a numerical identifier, a title and a description.

**Scene**

This entity represents a scene in the game model. It has a numerical identifier and a story ID as well as a description, which contains the text displayed to players, and a background, which contains the image displayed to players.

**SceneComponent**

This entity represents an object within the game model that can be placed in a scene or a player’s inventory. It has a numerical identifier, a name for the object, a scene identifier, which is used to place it into the correct scene at the start of a new game, and a type, used to categorise each component.

**SceneExit**

This entity represents a link in the game model, from one scene to another. It acts as an extension to the SceneComponent entity. It has a numerical identifier, which it gets from the SceneComponent entity and an id for the scene that it links to.

**SceneDoor**

This entity represents a locked door in the game model. It acts as an extension to the SceneComponent entity. It has a numerical identifier, which it gets from the SceneComponent entity; a key identifier, which also references the SceneComponent entity, an exit identifier, which references the SceneExit entity, and two message fields which are displayed to the player when they interact with this entity.

**SceneInventory**

This entity represents SceneComponents placed inside a Scene within the game model. It has a numerical identifier and then Scene, SceneComponent, and GameInstance identifiers to represent the ternary relationship.

**PlayerInventory**

This entity represents SceneComponents picked up by a PlayerInstance within the game model. It has a numerical identifier and identifiers for a PlayerInstance and a SceneComponent to repres3ent the binary relationship.

## 4.6 Class Responsibilities and Collaboration Cards

Building on what was developed in the analysis classes, a more concrete implementation was developed. The following CRC cards describe the classes that will be developed for the game system, briefly cover their role within the system and list classes they interact with.

### Used within the View

|  |  |
| --- | --- |
| Class **ViewManager** | |
| **Responsibilities**   * Handle moving between different application views. * Accept user’s game input and pass it to the input manager. * Update display based on user’s input. | **Collaborators**   * InputManager * CommandOutput |

|  |  |
| --- | --- |
| Class **GameViewManager** | |
| **Responsibilities**   * Handle the retrieval of data to update game related views. * Pass user’s input commands for starting, continuing and ending games to the view model. | **Collaborators**   * InputManager * GameManager * Command |

|  |  |
| --- | --- |
| Class **LoginViewManager** | |
| **Responsibilities**   * Handle moving between different subviews within the login view. * Pass user’s login and account creation requests to the view model. | **Collaborators**   * LoginManager * LoginResult |

|  |  |
| --- | --- |
| Class **InputManager** | |
| **Responsibilities**   * Parse user’s text input and turn it into commands usable by the system. * Pass commands to other classes that act upon them. | **Collaborators**   * Command * CommandDelegate |

|  |  |
| --- | --- |
| Data Structure **Command** | |
| **Responsibilities**   * Hold data related to a game command. * Allow a command method to be called from the view. * Return the result of a command to a command caller. | **Collaborators**   * CommandDelegate |

|  |  |
| --- | --- |
| Data Structure **CommandOutput** | |
| **Responsibilities**   * Hold data related to the result of a game command being called | **Collaborators** |

|  |  |
| --- | --- |
| Delegate **CommandDelegate** | |
| **Responsibilities**   * Contain a reference to a method capable of being used as a game command. | **Collaborators**   * ViewManager * GameViewManager * GameManager |

|  |  |
| --- | --- |
| Delegate **AccelerometerManager** | |
| **Responsibilities**   * Accept input from a device’s accelerometer. | **Collaborators** |

### Used within the View Model

|  |  |
| --- | --- |
| Class **LoginManager** | |
| **Responsibilities**   * Validate user’s account data. * Communicate with the model to retrieve or create user accounts. | **Collaborators**   * DataAccess * LoginResult |

|  |  |
| --- | --- |
| Class **GameManager** | |
| **Responsibilities**   * Handle the manipulation of game state based on a player’s input. * Communicate with the model to retrieve or update game state. | **Collaborators**   * DataAccess * FindSceneExitResult * FindSceneItemResult * Scene * SceneComponent * SceneExit * SceneDoor * SceneItem * SceneKey |

|  |  |
| --- | --- |
| Class DomainClasses.**Scene** | |
| **Responsibilities**   * Hold data related to a scene within the game state. * Return components within the scene based on search criteria. | **Collaborators**   * FindSceneExitResult * FindSceneItemResult * SceneComponent * SceneExit * SceneItem |

|  |  |
| --- | --- |
| Abstract Class DomainClasses.**SceneComponent** | |
| **Responsibilities**   * Represent the concept of an object existing in a scene. * Hold data related to an object existing inside a scene. | **Collaborators** |

|  |  |
| --- | --- |
| Class DomainClasses.**SceneDoor** | |
| **Responsibilities**   * Hold data related to a locked door within a scene. * Check if the doors key exists in the players inventory. | **Collaborators** |

|  |  |
| --- | --- |
| Class DomainClasses.**SceneExit** | |
| **Responsibilities**   * Hold data related to a link from one scene to another scene. | **Collaborators** |

|  |  |
| --- | --- |
| Abstract Class DomainClasses.**SceneItem** | |
| **Responsibilities**   * Represent the concept of something a player can interact with inside a scene. | **Collaborators** |

|  |  |
| --- | --- |
| Class DomainClasses.**SceneKey** | |
| **Responsibilities**   * Hold data related to an item that can unlock a door. | **Collaborators** |

|  |  |
| --- | --- |
| Class ActionResult.**FindSceneExitResult** | |
| **Responsibilities**   * Hold data related to the result of a search for exits within a scene. | **Collaborators** |

|  |  |
| --- | --- |
| Class ActionResult.**FindSceneItemResult** | |
| **Responsibilities**   * Hold data related to the result of a search for items within a scene. | **Collaborators** |

|  |  |
| --- | --- |
| Class ActionResult.**LoginResult** | |
| **Responsibilities**   * Hold data related to the result of a login attempt. | **Collaborators** |

|  |  |
| --- | --- |
| Class **MultiplayerManager** | |
| **Responsibilities**   * Monitor state updates from other clients. * Store state received from other clients. * Send state to other clients. | **Collaborators**   * MqttClient |

### Used Within the Model

|  |  |
| --- | --- |
| Class **DataAccess** | |
| **Responsibilities**   * Create and maintain a connection to the data storage for the game’s state. * Create initial game state within the data storage. * Allow data to be retrieved from the data storage. * Allow data to be inserted or updated in the data storage. | **Collaborators**   * DTO.GameInstance * DTO.PlayerData * DTO.PlayerInstance * DTO.PlayerInventory * DTO.Scene * DTO.SceneComponent * DTO.SceneDoor * DTO.SceneExit * DTO.SceneInventory * DTO.Story |

|  |  |
| --- | --- |
| Class DTO.**GameInstance** | |
| **Responsibilities**   * Represent a database record for an instance of a game being played. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**PlayerData** | |
| **Responsibilities**   * Represent a database record for a player account. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**PlayerInstance** | |
| **Responsibilities**   * Represent a database record for a player playing a game instance. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**PlayerInventory** | |
| **Responsibilities**   * Represent a database record for items held by a player instance. | **Collaborators** |

|  |  |
| --- | --- |
| Class **JsonDrop** | |
| **Responsibilities**   * Allow data to be retrieved from the server-based data storage. * Allow data to be sent to the server-based data storage. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**Scene** | |
| **Responsibilities**   * Represent a database record for a scene. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**SceneComponent** | |
| **Responsibilities**   * Represent a database record for scene component information. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**SceneDoor** | |
| **Responsibilities**   * Represent a database record for scene component information specific to a door. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**SceneExit** | |
| **Responsibilities**   * Represent a database record for scene component information specific to an exit. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**SceneInventory** | |
| **Responsibilities**   * Represent a database record for components held within a scene. | **Collaborators** |

|  |  |
| --- | --- |
| Class DTO.**Story** | |
| **Responsibilities**   * Represent a database record for an organisation of scenes. | **Collaborators** |

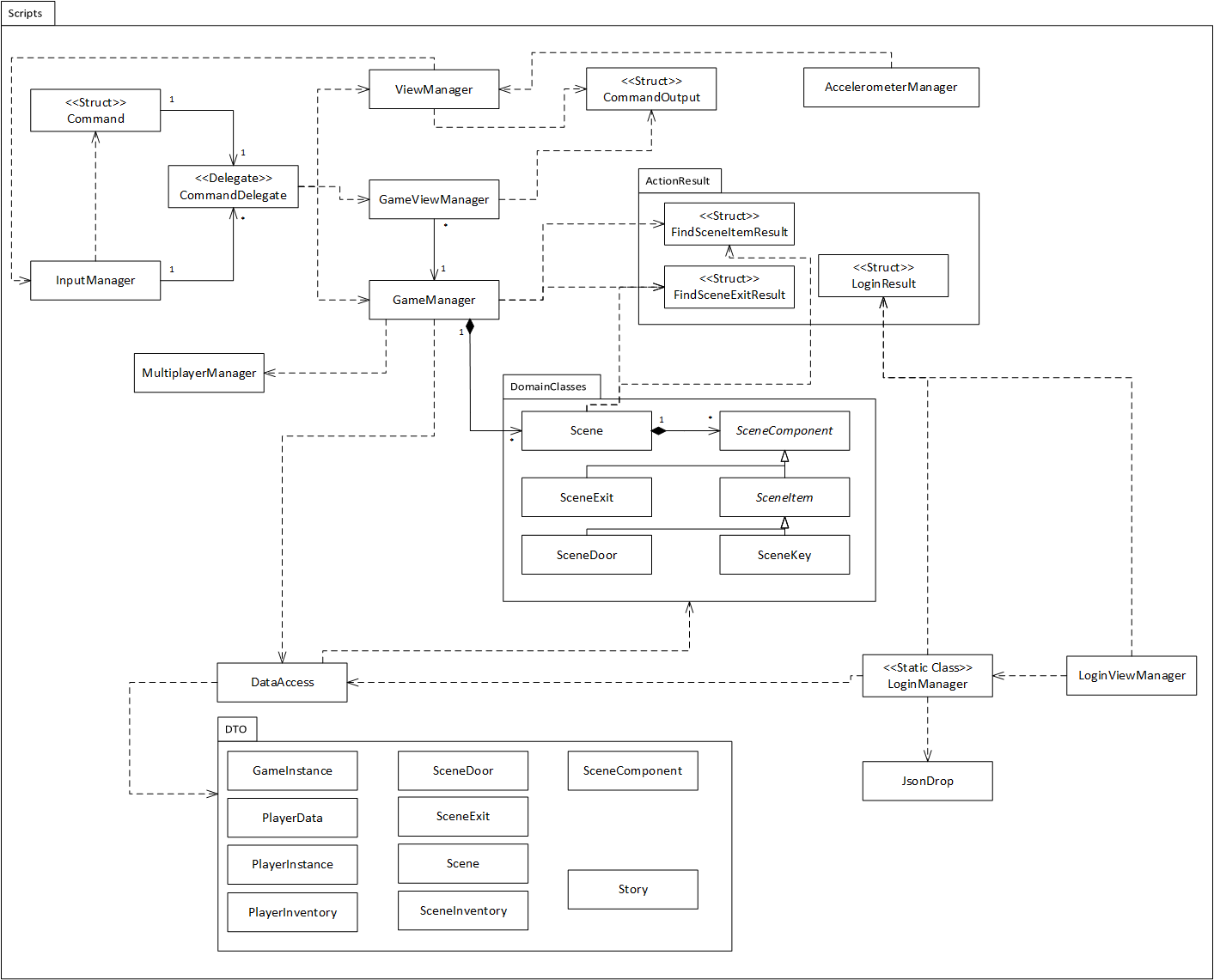
As shown in the last section, the classes can be organised by which section of the MVVM pattern they’re related to.

Something to note in these classes is that there is some overlap in data between the classes in the DomainClasses namespace and those the DTO name space. This is because the both model the same data entities. However, they have been kept as separate classes for the following reasons.

* The first reason is that, while they model the same data, they do so in different ways. The classes in DomainClasses are designed to model the data within an object-oriented system, while the classes in the DTO namespace are designed to model the data within a relational database system. This means that there are minor differences in the way entities interact or are related with each other.
* The second reason is based on the section of the MVVM pattern the classes are organised into. The DTO classes are part of the model section, so their purpose is to exist as pure data. The DomainClasses are part of the ViewModel. Along with storing data, they also need to contain behaviour to allow them to be manipulated from the view.
* The final reason is the way they are to be implemented. The DomainClasses are meant to be objects within the .NET OOP framework. This means they follow some practices about member access and mutable state that affect how their data is stored and accessed. The DTO classes exist solely as a mapping to and from the SQLite database, meaning they require public access to their member state.

## 4.7 Class Diagram

To further develop the interactions between classes in the game system, the following class diagram was created.



Something to note in this class diagram is there are only a few relationships that aren’t a dependency.

There is an association between the CommandDelegate delegate and the Command struct. Which represents the Command maintaining a reference to a method which it calls when it is told to run.

There is also an association between this delegate class and the InputManager class. This represents the InputManager managing a list of all the CommandDelegates usable in the system so it can properly assign commands.

There is an association between the GameViewManager and GameManager classes. This is a representation of the View-ViewModel relationship from the MVVM pattern. This means it’s a unidirectional association and that multiple GameViewManager instances can be associated with a GameManager instance.

There is a composition between the GameManager and Scene classes, which represents the GameManager being responsible for the scene it has loaded.

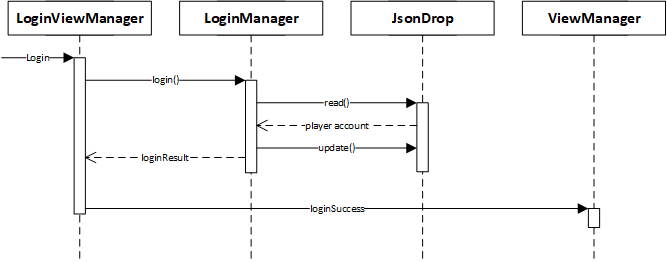
There is another composition between the Scene and SceneComponent class. This represents the Scene being responsible for all of the SceneComponents within it.

Other than these relationships, the other interactions are dependencies. One reason for the abundance of dependencies are the use of the singleton pattern in classes like the view managers, which means an instance reference only needs to be stored within that class. Another reason is that the game’s persistent state exists in a database instead of the game’s system. So instead of storing state on the heap and referencing it through associations, a lot of information is only retrieved when it is needed and its lifespan is entirely on the stack.

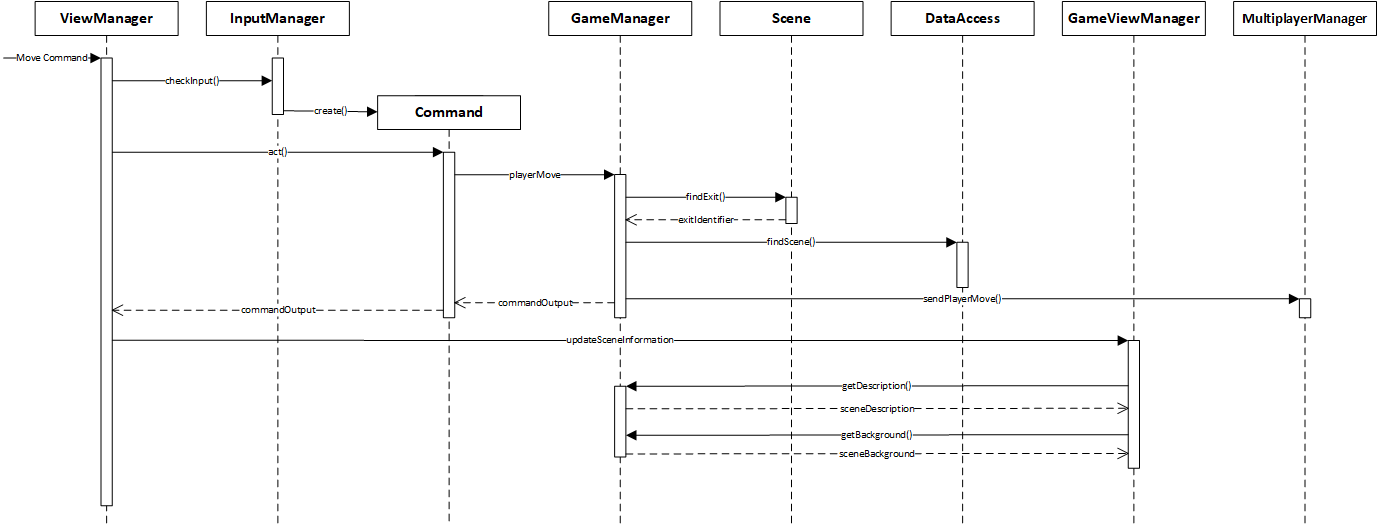
Another thing to note is that the DataAccess class has dependencies on both the DTO classes and the DomainClasses classes. This is because, while categorised as part of the model, the DataAccess exists more as a bridge between the view model and the real model, which is the database.

## 4.8 Sequence Diagrams

To further help show how the classes interact with each other, a couple of sequence diagrams were created. The focus of these is to show how a user’s input moves between the view, view model and model.



This diagram shows the sequence involved in a player logging in. The login view manager sends a message to the login manager informing it that the player is trying to log in. The login manager then sends a message to the JsnDrop service, to find the player’s account. JsnDrop returns a player account data structure, which the login manager uses for authentication. If the log in is successful, the login manger sends an update message back to the JsnDrop service so the log in is persistent and then sends the result back to the login view manager. Finally, the login view manager sends a message to the primary view manager informing it of the successful login.



This diagram shows the sequence involved in a player moving through the game environment. The view manager receives the input and get the input manager to check it and the input manager creates a command instance to further handle the input. The view manager then tells the command to act. The command informs the game manager that a move command has been entered. The game manager gets the exit from the scene then gets the exit’s information from the data access. After updating the local state, the game manager sends a message to the multiplayer manager so other active clients can be updated. The result of this move action is sent back to the command, which passes it back to the view manager. The view manager then sends a message to the game view manager, informing it that it needs to update the game display.

# 5. Multiplayer System Design

In the transition from a single-player game to a multi-player game, two new areas of functionality were added to the system.

* An HTTP based system, used to handle player account storage and logging in.
* A MQTT based system, used to sync data between clients.

## 5.1 HTTP Login System

The new login system uses the JSNDrop service to share data with a web server. In the implementation, a new class, JsonDrop, was created. This new class has similar responsibilities to the DataAccess class, but instead of the local storage, it provides an interface for the server-based storage.

To keep track of whether a player account is online or offline, the player entity stored in the database has a Boolean flag to keep track of this state. This flag is switched on when the player logs ins, and is switched off when the player exits from the game menu and returns to the log in menu. If an account is flagged as online, then other clients will be prevented from allowing that account to log in.

To prevent accounts from becoming permanently stuck in an online state due to situations where a client application ends unexpectedly without switching the flag to offline, a secondary check was added. This secondary check has the logged in clients update the server at minute intervals, storing the time that the update happened. In the case of a client application unexpectedly ending, the server will stop receiving these updates. When a client makes a login attempt, it will ignore the previous online flag, provided this last update time happened over a minute before the login attempt.

With the player account information now stored on the server, the only reason for player information to be stored locally is to keep track of saved games. Because of this, fields like username and password can be removed altogether.

## 5.2 MQTT Client Syncing

To allow players to see each other in the game world, the clients need a way to sync data between themselves. This functionality has been implemented using the MQTT protocol. The implementation uses a new class, MultiplayerManager, which is part of the ViewModel. This class uses the M2Mqtt Project - MQTT Client Library to send messages between clients.

The MultiplayerManager maintains a list of Usernames that it is currently receiving messages from. This data also includes which room they last moved to, which can be used to map out where every online player is within the game world.

When a player starts a game, moves between scenes or leaves a game, the MultiplayerManager sends a message to any other online clients to inform them of the change.

|  |  |  |
| --- | --- | --- |
| **Message** | **Sent When** | **Response** |
| JOIN message | A player starts a new game | Send an UPDATE message |
| UPDATE message | A client needs to share player’s position | Log the player and their position |
| LEAVE message | A player ends their game | Remove the player from the player log |

# 6. Using the Device’s Sensor

As part of the final requirements, the game needs to use one of the sensors commonly found on a mobile phone device.

For this implementation, the accelerometer was used. If the player flicks their phone and the speed exceeds the threshold set within the system, depending on what view is active, a command will execute.

If the game menu view is active, the game will log the player out and return to the login menu. The same as if the player had typed “exit”.  
If the help view is active, the game will close the help menu and return to the last view. The same as if the player had typed “close”.

Note: For this functionality to be operated correctly, an actual Android device was used. The Android emulator used during development, BlueStacks, did not feature a toolset robust enough to execute any of these commands.

# 7. Summary of Development Process

## Milestone One

For the first milestone, the game concept was developed and then a user interface and prototype game system were implemented in the Unity game engine. The development for this work is covered in sections 2 and 3, as well as the subsections 4.1, 4.2, 4.3 and 4.4.

## Milestone Two

For the second milestone the key development was the addition of account management within the game system. This involved the addition of a number of new screens to the user interface, to allow a user to access these new features, and classes to control the behaviour behind these screens. A database was also developed and implemented, not only to store this new user information, but also as a data storage for the entire game system. Another new development was to model the system in more detail, which led to changes in the existing codebase, changing the way some classes operate and adding new classes where needed.

During this development changes were made to the screen designs in section 3 and the diagrams in subsections 4.1 and 4.4.

New sections added to this report for milestone two are the subsections 4.5, which describes the new data model and subsections 4.6, 4.7 and 4.8, which describe class responsibilities and interactions.

## Milestone Three

For the third milestone, the key development was the multi-user functionality. This development is primarily documented in the newly added section 5.

A secondary development was the addition of device sensor functionality, which is documented in the newly added section 6.

To reflect changes made to the system’s structure and the way processes operate during this development, the various diagrams in section 4 were updated to show newly added classes.