IPPO Model Design Proposal

Antonios Kolovos B158495

October 2019

1 Basic Structure

The world is organized in rooms(Locations). Each Location has 4 directions(Views) the player can look at, and he has 3 controls, left, right and forward. Forward is only is active when there is a door in the view we are looking, and there can only be one door per View. The player can pick up and drop items in each room at a specified view. The item is visible only in the view that it was dropped and not in the entire location.

2 Class structure

The program is based on the MVC idea. The View is based on the javaFX library, and it interacts with the World Controller Class, which in turn, interacts with the Model(World Class), in order to get the current status of the world. Note that for all the following classes there will also be setter/mutator methods for the fields.

1. **World Controller**.
   - The controller for the world, s the handlers for the user actions from the GUI and modifies the Model(World). Based on user input, it commands the World, to turn left, right, or go forward(if available).
   - Creates the World object.
   - Has handlers to respond to UI action
   - All the information about the UI elements(which buttons are active and which images to display) are ascertained by asking the World Object(using it;s methods
2. **World**
   - Contains the World topology and status.
   - Method Initialize() Creates the world topology(Directions and Locations). Also initializes the Player Object, which stores the world state(Current Location and Direction as well as any picked up items)

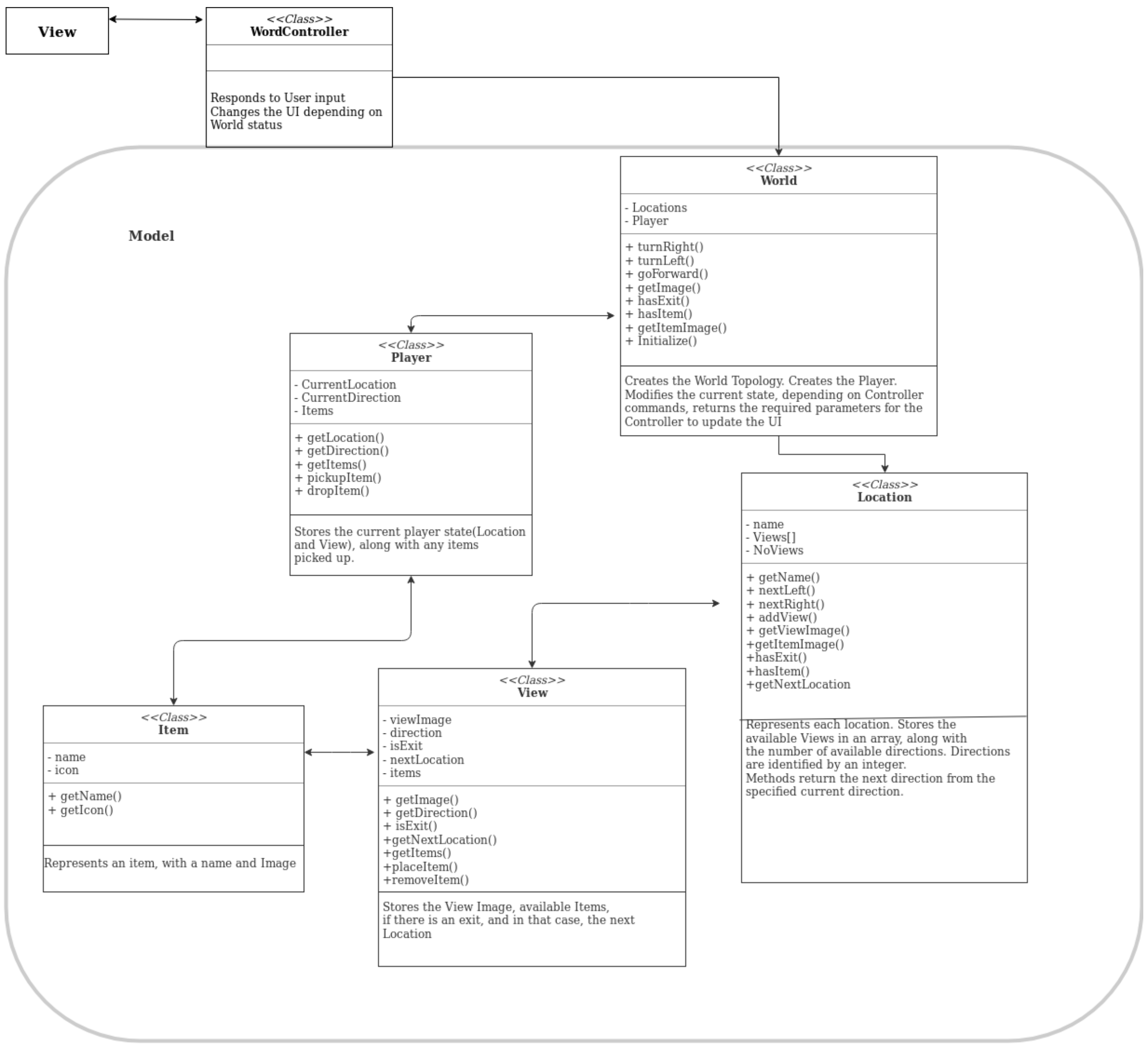

Figure 1: Class Diagram

- Methods: turnRight(), turnLeft(), goForward(). These methods move the player around the world . After every action, the Player state(currentLocation and currentView) are updated. : getViewImage(), getItemIcon(), hasExit(), hasItem() These methods are used by the Controller to update ne cessary UI elements.

3. **Player**

- Represents the current Player state. Is instantiated by the World Object
- Method: getLocation() Returns the current location
- Method: getDirection(): Returns the current direction
- Method: getItems(): Returns the items that the player has picked up(Not the ones that are placed in a View)
- Method: pickupItem() Adds the specified item to the items that have been picked up
- Method: dropItem(): Removes the specified item from the items that have been picked up
- There will also be setter methods for the CurrentLocation and CurrentDrection fields.

4. **Location**

   - Represents a Location each location has an array of the available Views as well as a name. The Location object has no knowledge of which View is currently being used, so in all of the methods which reference a view object, the view direction(an integer) must be specified.
   - Method: getName() Returns the location name
   - Method: getViewImage(Int): Retuns the image of the requested View
   - Method: hasExit(Int): Returns if the requested view has an exit
   - Method: hasItem(Int): Returns if the requested view has an item
   - Method: getItemIcon(Int): Returns the requested view item
   - Method: getNextLocation(Int): Returns the next locaiton from the specified view
   - Method: nextLeft(Int): Returns the direction to the left of the specified
   - Method: nextRight(Int): Returns the direction to the right of the specified
   - Method: addView(View): Adds a View to this location
   - There will also be setter methods for the name and NoDirections fields.

5. **View**

   - Represents a View. This is the basic object that contains all the basic information to display. It stores the image to display, the items to display, and if there is an exit or not and the next Location we can move to

- Method: getImage() Returns the Image to display
- Method: getDirection() Returns the direction Id(integer)
- Method: isExit() Boolean that specifies if the player can exit from this direction
- Method: getNextLocation() Return the name of the next location(if there is an exit)
- Method: getItems() Returns the items stored in this location
- Method: placeItem() Adds an item to this location
- Method: removeItem() Removes an item from this location
- There will also be setter methods for the Image, id, exit and Next-Location fields

6. **Item**
   - Represent an item. Items can be pickeud up by Players and placed in Views
   - Method: getName(): Returns the item name
   - Method: getIcon(): Returns the item image
   - There will also be setter methods for the name and icon fields.

3 Design Compromises/Decisions

- In order to update the UI, the World Controller must know if there is an exit, if there is an item, and the View Image and the item Icon. All of those can be found using the accessor methods of the World Class, which in turn uses the Location's methods to get the result. This results in a lot of different methods, but decouples the Classes and leads to a more flexible design. For example it would be possible instead of requesting each field independently, to pass a View object, which contains all the necessary information. This Design however would lead to a more tightly coupled program, where a large part of the program would be dependent on the View class design.
- Every room has 4 Views(North, South, East, West) which are represented by incrementing integers(The direction field). For example right of View 1 is View 2 etc. This can be expanded to an arbitrary number of Views, and that's why each location can calculate its own next View(instead of doing that calculation at the World Class), depending on the total number of Views, for that location. However if we chose to implement differing number of views for different locations, we would have to find out in which direction to look at when we enter a new room.

- For every View, there can only be one single exit. If more exits are required, they have to be put in different views. This is done for simplicity, and given the current design proposal, it would be difficult ot implement multiple exits at the same view.
- The World, Location and View classes, hold no information about the current state(Which location and direction we are looking). All of this information in held in the Player class, and the World asks the player each time. These could be implemented inside the World class, however this design offers more flexibility in case we wish to expand the game with more than one players in the same world.