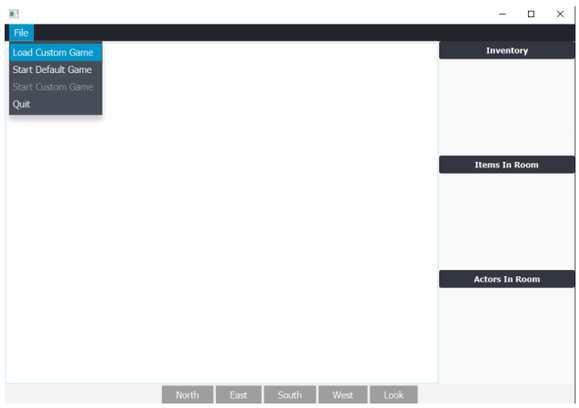
1 Setup

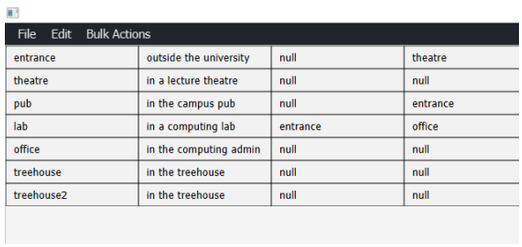
The project was completed using Eclipse IDE (Using JavaFX 11 and OpenJDK 11 but tested as working with OpenJDK 13) on both Linux and Windows. A combination of FXML (partially created with Scenebuilder) and 'pure' JavaFX was used to provide the GUI. The report was created with LaTeX. The application builds upon my previous submission for the 'zuul-bad' project with the code being simplified in various areas. There is **minimal** coupling in the main game with the JavaFX view. In the `Room` class and `InventoryModel` class there are methods to return an `ObservableList` to connect to the `ListView` objects of the GUI. Apart from this the underlying game is largely the same, calling the same methods as the previous submission. The coupling could have possibly been avoided by converting the `List` to an `ObservableList` in the view when necessary but the negatives of this approach seemed to outweigh the positives.

2 Summary Of Application

The application consists of two main parts: the main game (`FXController`) and a CSV editor (`CSVEditorController`). On loading the program the user is greeted with the main game screen and can choose to start a default game or load a custom game. On loading a custom game the CSV Editor is presented.



(a) Loading options



(b) The CSV Editor

2.1 Default vs Custom Game

It should be noted that loading from the default CSV and loading a custom CSV is handled a little differently. A default game loads from the CSV as a `List<List<String>` (see section 2 for more details of this) and directly instantiates the data as various instances of `Room`. However, the CSV Editor loads the data, again, as a `List<List<String>` but then later converts it to `ObservableList<ObservableList<CSVEditorCell>>` which is used to populate the `GridPane`. The `GridPane` is just a visual representation of the two-dimensional array of the CSV's data. Likewise, `CSVEditorCell` binds its data with the corresponding element of the underlying `ObservableList`. Only when the user presses "Finish Editing" in the CSV Editor's menu is it then converted to instances of `Room` like the default game.

3 Loading the CSV

```
...
public List<Object> loadCSV(Function<Object, Object> makeRow, String path) {
    File inputF = new File(path);
...
    BufferedReader br = new BufferedReader(new InputStreamReader(inputFS));
    List<Object> records = br.lines().map(makeRow).collect(Collectors.toList());
...
```

Listing 1: CSVParser.java

Loading from any CSV file and handling its data is done using streams and mapping a function to each element of the stream. This pattern is also used in `AllRoomDataController` to instantiate all instances of `Room`. This allows the parser to be extremely compact and the main logic of how to process the CSV/`List` is tied to the class that needs it. The parser is completely blind to its input, the caller of the parser class must cast onto the result what is expected. For example:

```
private Map<String, Room> rooms;
...
rooms = csvData.stream().map(customMap).collect(Collectors.toMap(e -> ((Room) e).getName(),
    e -> (Room) e));
```

Listing 2: AllRoomDataController.java. Edited for clarity.

4 CSV Editor

4.1 The 'Header' Class

When converting the raw CSV data strings into a `CSVEditorCell` the cell is given a `Header`. A big advantage of using `Header` classes is that we somewhat avoid the problem of hard-coding indexes in the game. For example, in the future if we want to add more columns to the CSV then it would just be a matter of redefining the indexes of the columns in all implementations of `Header`. This means the indexes are defined only once, making it easier to maintain. Likewise, instead of using indexes when instantiating the instances of `Room` we use a dedicated `HeaderEnum`. This provides a more robust solution that is more future-proof at the slight expense of readability and speed. An even better solution, which would free us of the need to hard-code any index (but not possible in this project), would be to have the first row as a 'header' row which could be used to define the expected indexes of each column.

```
public class DescriptionHeader extends Header {
  private final static int INDEX = 1;

  public DescriptionHeader() {
    super(HeaderEnum.DESCRIPTION);
  }

  @Override
  public String validateFieldText(String textFieldValue) {
    if (textFieldValue.isEmpty()) {
      return "Field must not be blank.";
    }
    return null;
  }

  public static boolean matchesIndexCondition(int csvIndex) {
    return csvIndex == INDEX;
  }
}
```

Listing 3: A simple example of a header

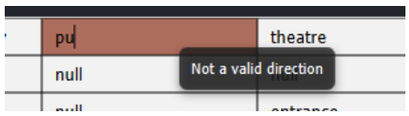
```
String description = line.stream().filter(e -> e.getHeader().getEnum().equals(HeaderEnum.
    DESCRIPTION)).map(csvCell -> csvCell.getProperty().getValue()).findFirst().orElse(null);
```

Listing 4: Using the same header when instantiating a Room object

4.2 Editor Features

4.2.1 Error Checking

As previously mentioned, the `CSVEditorCell` is used to represent the underlying element in the `ObservableList` in the `GridPane`. Each cell contains a validator method that gives a series of if statements and checks that returns a custom error tooltip/colour if an error is found and `null` if not. Each `TextField` of the grid has an `InvalidationListener` which checks the validity of the new text content after each change, allowing for on-the-fly error checking in the editor. This feature could be extended further with various different checks, autocompletion and more.

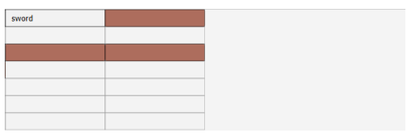


(a) Checks all room names to see if it exists.

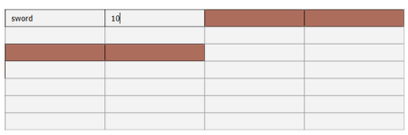
(b) Default tooltip (Header name).

4.2.2 Dynamically Adding Columns

When the grid is formed the longest array in the matrix is used as the value for the length of all rows. This allows for all rows to be of an even length and more aesthetically pleasing. Each row of the CSV has two optional extra columns for the user to fill out if necessary. When the user fills out both of these columns - without error - then two extra columns are appended/enabled.



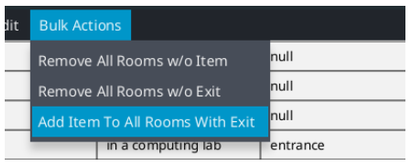
(a) About to enter text in the final column.



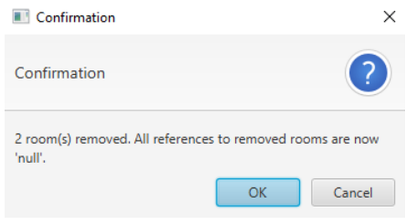
(b) Two columns added when text entered.

4.2.3 Bulk Actions

The 'bulk actions' requested in the assignment can be achieved by selecting them from the appropriate menu.



(a) Bulk actions menu.



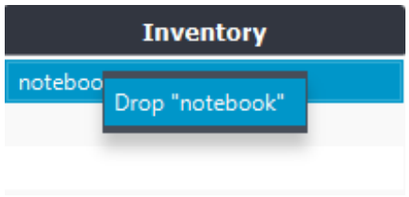
(b) Confirmation of removal.

Each bulk action is its own class and a 'command design pattern' has been used. Any future bulk actions simply have to place their new class in the appropriate directory and implement the `BulkAction` interface. The `BulkActionInstantiator` class will automatically populate the `Menu` with these new `MenuItem`s relieving the developer any headaches of figuring out how to add the item to the menu. I believe this will make future developments of the CSV editor a lot easier to manage.

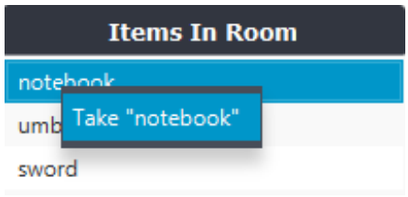
5 Features of the game

5.1 Go/Take/Drop Commands

Each time the player moves room the 'Items in Room' and 'Actors in Room' lists are set to the corresponding `ObservableList` of that current `Room` object. The user can drop and/or take an item by right clicking on the corresponding `ListView` and selecting the command which then updates the `ObservableList`.



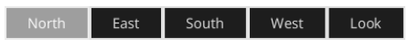
(a) Dropping an item.



(b) Taking an item.

When going from room to room only valid directions are enabled. This is to avoid having to give error messages to the player stating that they cannot go that way allowing for a better user experience.

(a) Before.



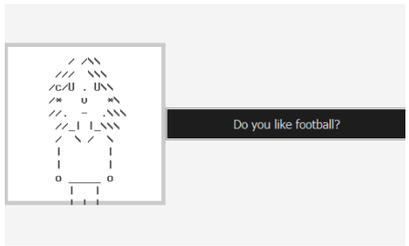
(b) After.

5.2 Character Features

Each character (`NPC`) moves around the rooms autonomously and on their own game thread. When they are in the same room as a player they do not move as to aid the player in interacting with them.

5.2.1 Talk Command

When talking to an NPC the user is offered various dialog options they can choose from, the NPC then gives a unique response. All of the dialog options and responses are written in the NPC CSV file.

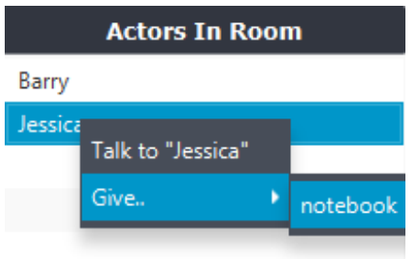


(a) Dialog Options.

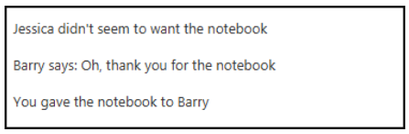
(b) Dialog Response.

5.2.2 Give Command

You can attempt to give a character an item from your inventory through right clicking and using the 'Give' context menu. The 'Give' menu is created dynamically when the main context menu is shown (See `view.ContextMenus.ActorContextMenu`). If the item matches the NPC's `validItem` field then they accept it and display a message accepting the item.



(a) The give menu



(b) NPCs accepting/rejecting items.

5.3 Summary

Overall I believe I have met (and hopefully surpassed) the requirements of the project. The user can edit the CSV (the world) in an interactive way, they can perform the bulk actions (the functionalities) requested in the report (and if needed, can later edit the data by clicking the cell). Further to this, the software company can easily add new 'bulk actions' by simply just creating a new class and dropping it into the directory. They can also extend the CSV file with more columns in a fairly painless way due to the `Header` class.

I also make extensive use of streams throughout the entire project (only in certain circumstances have I favoured a 'for loop'. The biggest example being in the `GridFactory` where I favour a standard 'C-Style' for loop as I believe it lends itself to create this type of structure). I use `Function`s and `Callback`s in combination with streams to make flexible methods to aid encapsulation and reuse code as much as possible.

5.4 Further Considerations

There are quite a few features that I would have liked to implement in the project that were ruled out to avoid 'feature creep'/scope issues. One of them was having an undo/redo option in the CSV editor. To achieve this I would have to make sure only to bind the `TextField`'s data with the underlying array once the user changes focus or presses return on their keyboard. The state of that cell/row could then be pushed into an undo/redo array that could be recalled later if the user decides they don't like their change.

I am also not entirely happy with the way the error tooltips are handled with the error checking in the CSV Editor. I would have preferred to have the `CSVEditorCell` be in control of its own `TextField` which the grid could call and use. Instead, there is coupling between these two classes. I ran into severe bugs where the `TextField` would lose focus when new cells are dynamically added to the row. Given longer, I would refactor this section further.

Overall I am quite proud of what I have made and believe with a few extra features and tweaks I have created quite a useful, if not quite specific, tool and the foundations of a game engine that could be extended and used in the future.