**Extension:** Task 16

**Title:** Configuration Files

**Author:** Sam Huffer, 101633177

# Goals / deliverables:

* An expanded world / adventure file which should:
  + Be easy to produce with different values; demonstrate by showing at least two working configurations.
  + Go beyond graph implementation to include one of the following:
    - Entities (items, containers, etc.)
    - Commands
    - A unique idea confirmed with your tutor

# Technologies, Tools, and Resources used:

* Visual Studio 2019
* Microsoft Word
* Draw.io

# Tasks undertaken:

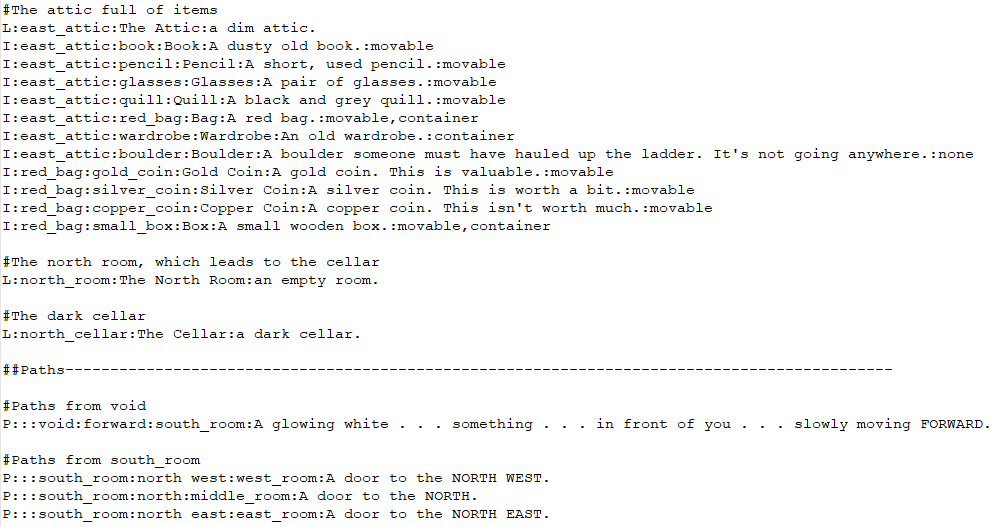
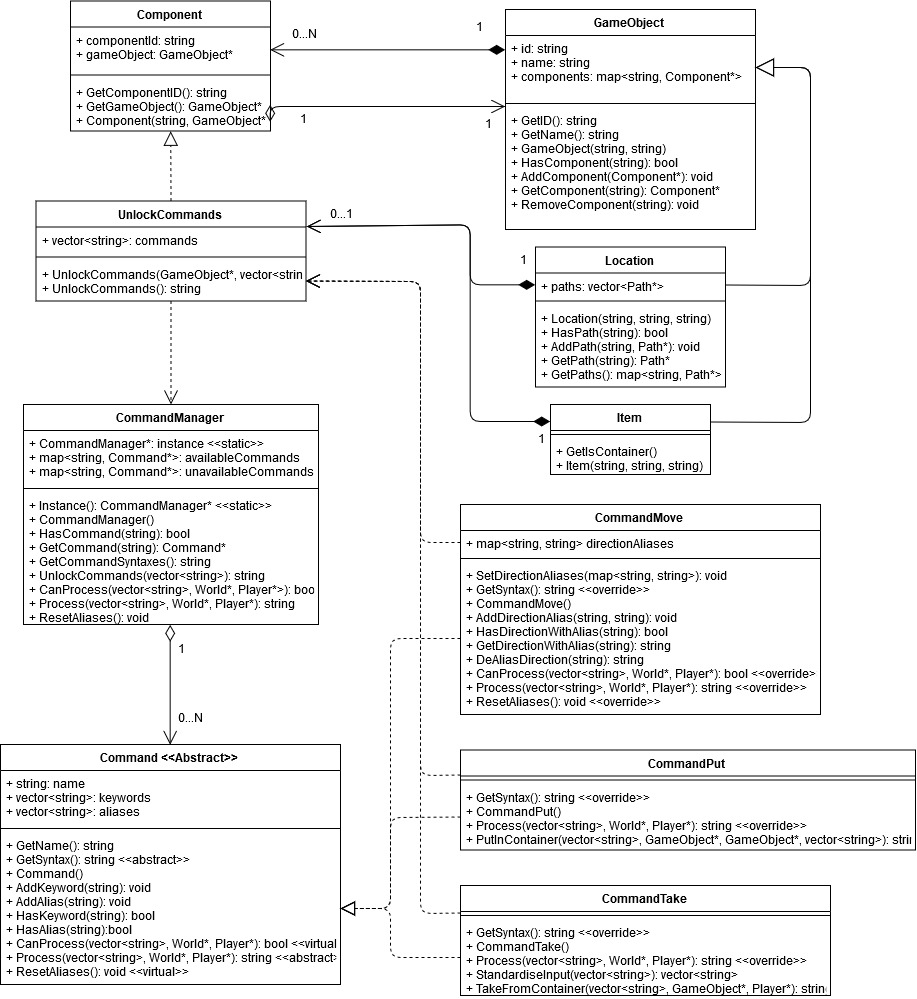
* I copied the “Zorkish Adventure” project and the task 15 spike report into the task folder, stripping out the spike report’s original content and replacing it with goals and resources pertaining to the task at hand.
* I had a look at the extension’s requirements, and found that I had already implemented the first suggestion in earlier spikes (fig. 1, fig. 2), specifically Task 12 – Game Graphs from Data, as I saw no  

Figure 1: My existing Test World specification, which outlines how to specify various game entities to be loaded into Zorkish Adventures, then features examples of specifying the world name, starting location, items, and components.

Figure 2: Another excerpt from my existing Test World specification, which specifies locations, items and their components, and some paths between locations. Some of the items are placed in locations, others in other items.

Figure : a UML diagram of the classes relevant to an UnlockCommands Component, as well as the Component itself.

sense in specifying locations without specifying the items in those locations when the commands for moving them about had already been created for Task 10 – Game Data Structures, and Task 15 – Composite and Component Patterns, which required the implementation of the component pattern. As all game objects were being specified in the text file already, I saw no reason not to add components to the specification as well.

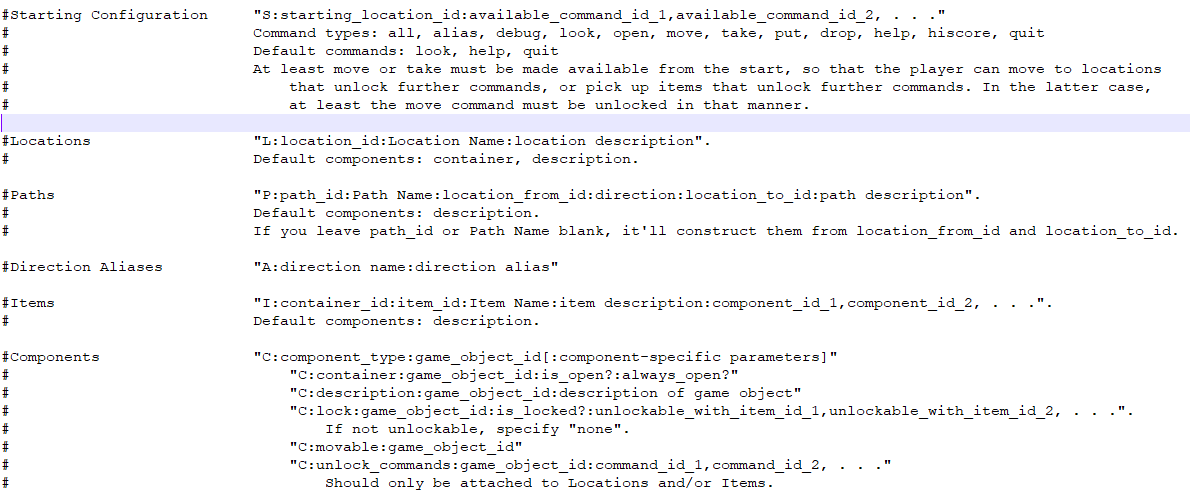
* I discussed what I could do for this task with Tien, my tutor, and settled on specifying commands to be unlocked when a player gets to a particular location or adds a particular item to their inventory. As such, I put together a UML outlining how that might be done (fig. 3).
* I updated the text file specification to accommodate the UnlockCommands Component.

Figure : the updated text file specification. The starting configuration now lets users list Commands (or all Commands) to be available from the start, and an UnlockCommands Component can be attached to GameObjects as a custom Component.

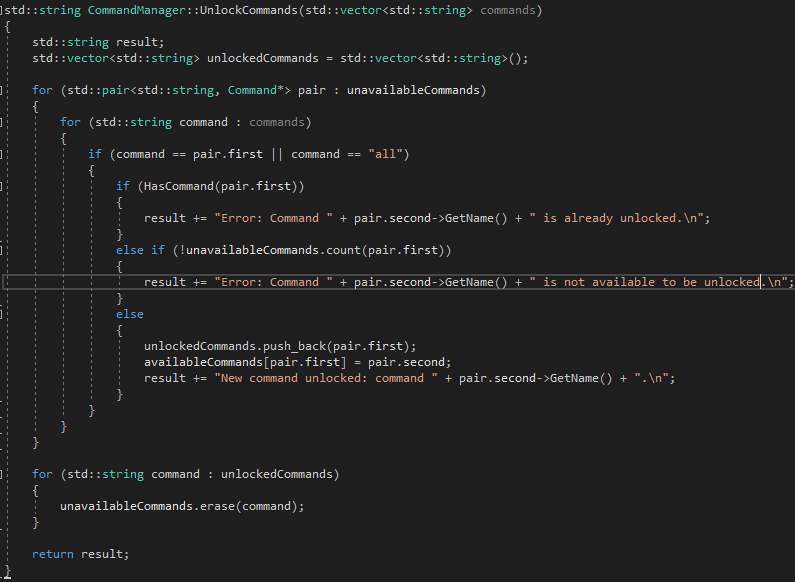
* I created the UnlockCommands Component’s .h and .cpp files, although I swapped out the non-constructor UnlockCommands() method for a GetCommands() public property. I then added to CommandManager the UnlockCommands() method, which iterates over each command id, checks unavailableCommands for a matching command (or just accepts it if the id listed is “all”), adds it to availableCommands if that Command type isn’t already in availableCommands, adds to an output string a “Command Unlocked” message listing the new command, and removes the Command from unavailableCommands. If the Command is already in availableCommands or does not exist in unavailableCommands, an appropriate error message is appended instead.

Figure : CommandManager.UnlockCommands()

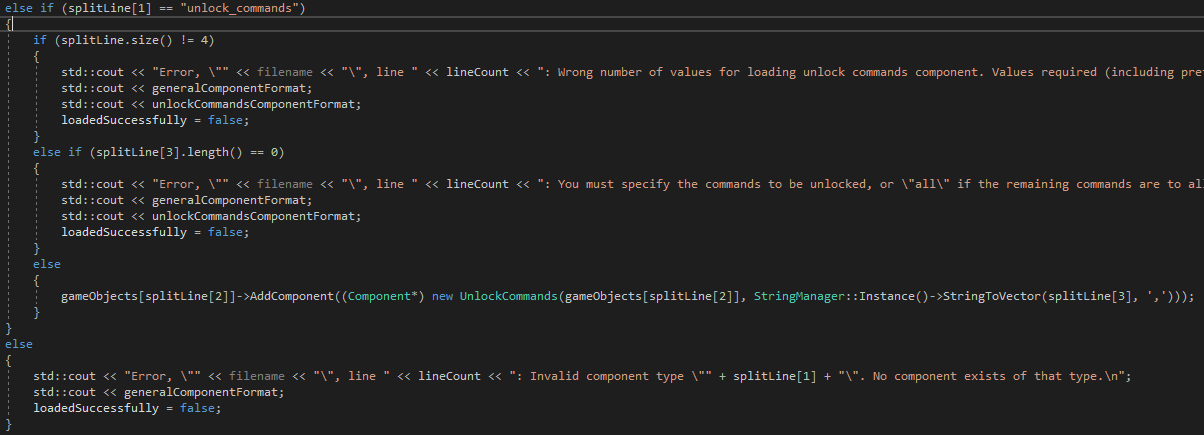
* I added to CommandMove.Process() a check when the player moves to a new Location for whether or not that Location has an UnlockCommands Component, passing its Command ids to CommandManager.UnlockCommands() and printing the result if so, before deleting the Component.
* I modified “Test World.txt” such that the starting location only had the default Commands (look, help, quit) and move available, and that the next location the player would reach would unlock the remaining commands.
* I modified World.World()’s handling of the starting configuration to validate that starting commands were specified, to unlock commands specified, and verify that the commands were all unlocked, printing an error message if they were not unlocked.
* To allow printing of correct formatting for any line or object at any point in the file loading loop, I copied an instance of each set of formatting into a string variable outside of World.World()’s while loop, and replaced each instance of that formatting with that string variable.
* I added to the components checks a set of checks for the UnlockCommands Component, checking the appropriate information was entered in the read line of the text file, before constructing a new UnlockCommands component with the list of Command ids, and adding it to the specified GameObject.

Figure 7: World.World()’s checks for the UnlockCommands Component.

* I updated CommandPut.PutInContainer() and CommandTake.TakeFromContainer()if the item moved had an UnlockCommands Component if it was put in the player’s inventory. However, I found PutInContainer()’s containerTo parameter is a GameObject\*, not a Player\*, giving no way of knowing if the item is being put in the player’s inventory. To fix this, I added to GameObject a public property GetType() to allow for this to be checked, and made GameObject’s constructor require child classes to specify their type.
* TODO: fix names of Commands to not be blank and to be in all caps.
* TODO: adjust formatting of CommandManager.UnlockCommands()’s output to add blank lines at the front and back of the list of valid / erroneous commands to separate from “Moving [direction] . . .” and the new location’s description.

# What we found out:

* C++ doesn’t like non-constructor methods with the same name as the class’s constructor.