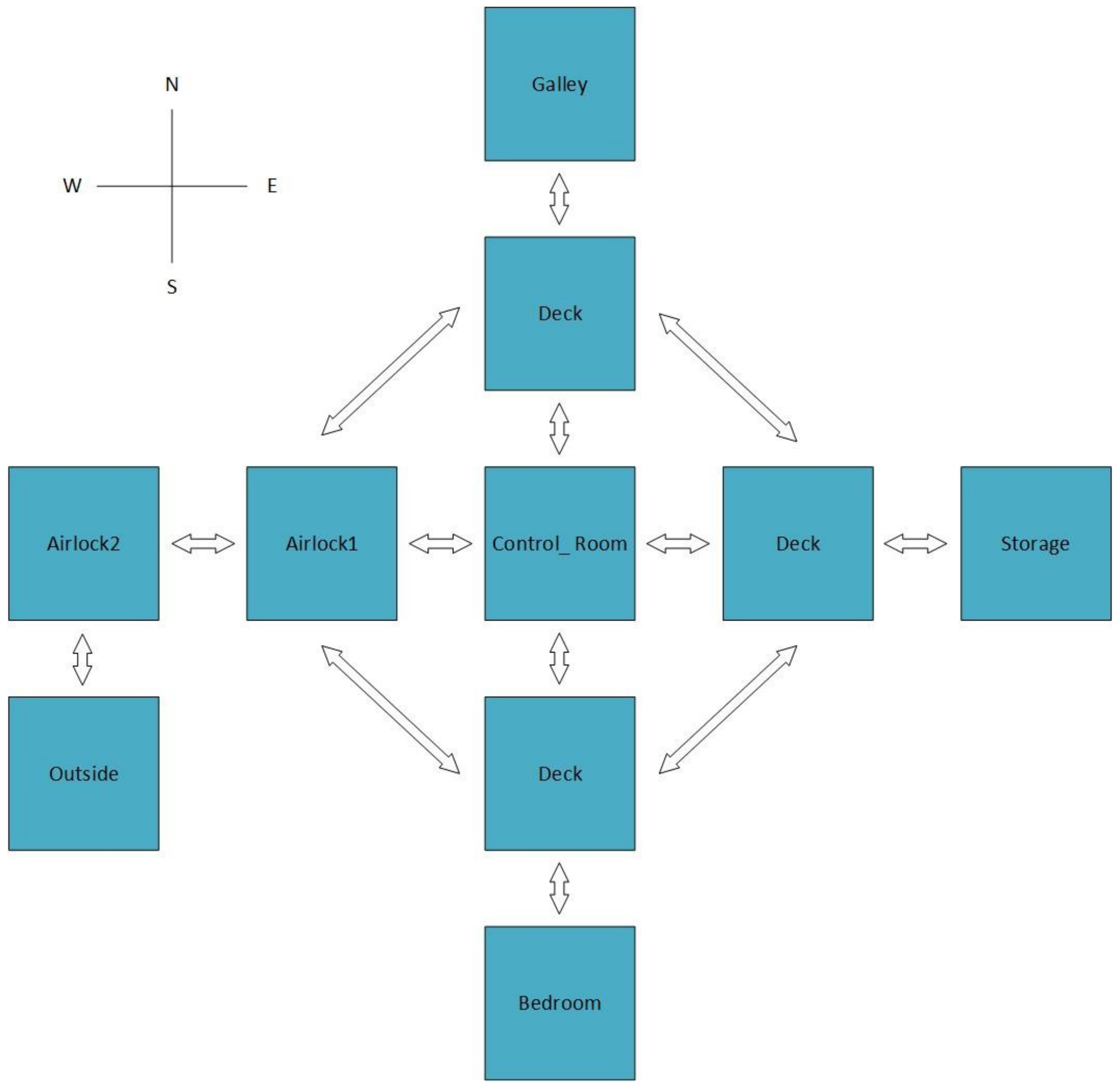
Joshua Kluthe
CS 162-400
Final Project

Design

My game begins with the player character, an astronaut, waking up on a space station to an emergency alert saying that oxygen levels are dropping. The player must then wander around the space station to determine what happened and fix the oxygen leak before their time runs out.

To describe my design, I will first show the physical layout of the space station, representing all of the spaces the player can walk through. For simplicity, I will use the terms north, south, east, and west, despite the fact that these terms do not apply to a space station. These terms will not be used in the end user application, only in the program design and code. Each room or area will have a name, and I will then further explain the class hierarchies and functions underlying the spaces. The physical layout is as follows:



Here I have used "Outside" to mean the outside of the space station. I doing this to avoid using the term "Space" because "Space" is the abstract base class for each area. The Control_Room will allow you to talk to the station's computer. Each Deck will have a viewport showing a different view.

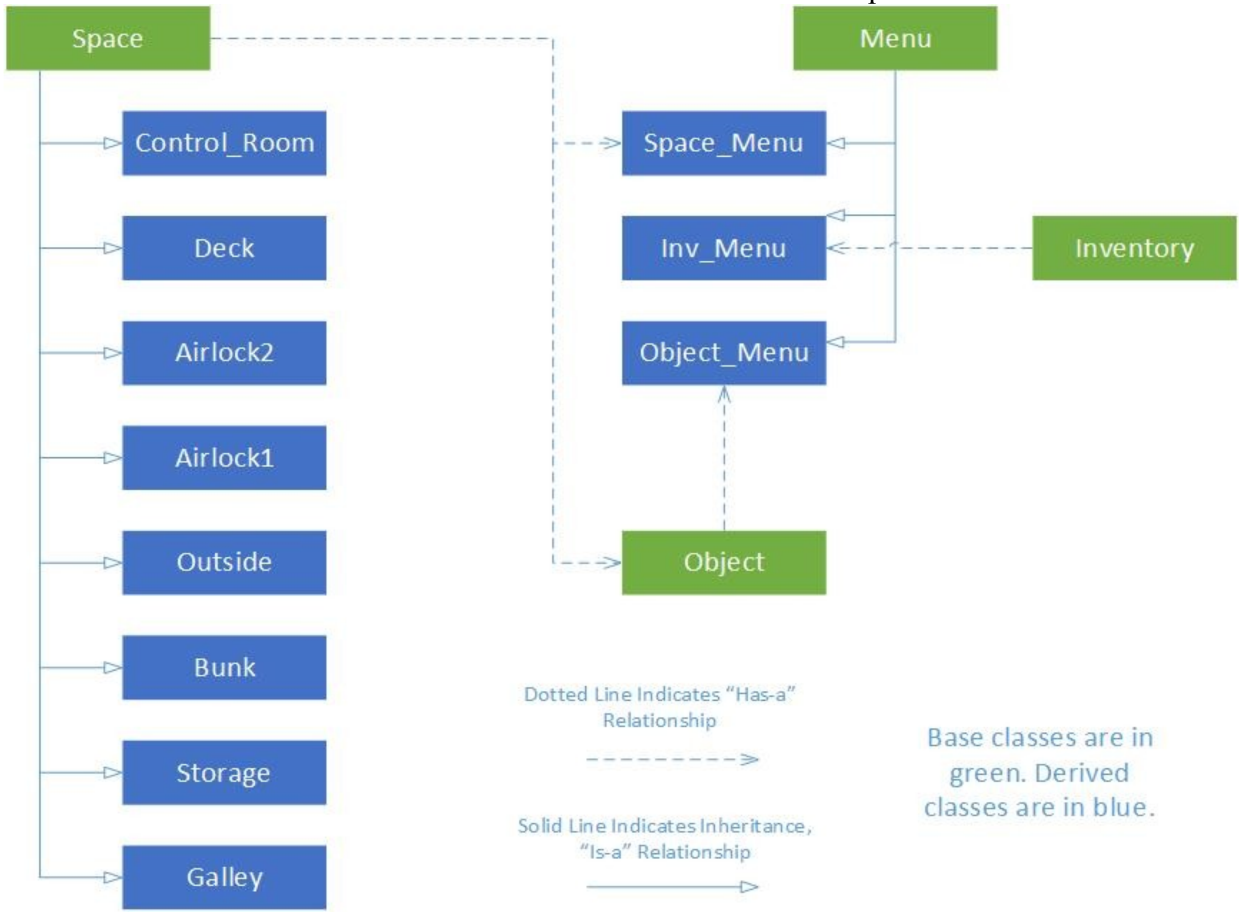
The Bedroom, Storage, and Galley will each have objects that will be necessary for the player to fix the oxygen leak. Airlock1 will have a door that must be opened to enter the actual airlock, and Airlock2 will have a robot that must be dealt with to open the final airlock door and get to the Outside, where the player can finally fix the leak. The entrances/exits for the areas are shown with arrows.

Each area will be a subclass of the abstract Space class. The Space class will have four pointers-to-Space member variables, allowing Spaces to be connected with entrances/exits. The Space will also have a string name and string description, each with associated getters and setters. A Space can also contain an Object, so the space will have a pointer-to-Object variable with a getter and setter. The Space class will have a pure virtual function special(). This will represent some kind of action that can take place in the Space, and this action will have an associated string special_name with a getter and setter. Finally, there will be a print_space() function. This function will print out the description of the space and create a Space_Menu object that will show the user options in the Space.

The Object class will be an abstract base class with member variables for the Object string name and string description. It will have an examine() function that prints the description and creates an Object_Menu with the Object options. It will also have a pure virtual use() function that will allow the player to use the object for some purpose.

The Menu class will be an abstract base class with a vector for menu options. It will have a print_options() method and a pure virtual actions() method. The actions() method is intended to call a switch that calls the appropriate functions depending on the user input, but it must be overridden in the derived Space_Menu and Object_Menu classes.

Finally, their will be an Inventory class that will have a member vector of pointers-to-Object. These will be the objects that the player has picked up during the game, and they will be able to examine and use the objects in their inventory. The Inventory class will have a Inv_Menu object derived from the Menu base class. The class hierarchies and relationships are shown below:


Space
Control_Room
Deck
Airlock2
Airlock1
Outside
Bunk
Storage
Galley
Menu
Space_Menu
Inv_Menu
Object_Menu
Inventory
Object
Dotted Line Indicates "Has-a" Relationship
Solid Line Indicates Inheritance, "Is-a" Relationship
Base classes are in green. Derived classes are in blue.

Testing Plan

I will take a very incremental approach to implementing and testing the application. First, I will write up the Space abstract class and the Menu abstract class. Then, I will derive the Control_Room class and Space_Menu class from them, and test to see that the classes are functioning correctly together. I will then derive the Airlock1 class and connect it to the Control_Room, and verify that I can go back and forth between them and that the Space_Menu is functioning correctly. Next, I will derive the Deck class and connect all five areas, verifying that they are functioning correctly and debugging as necessary. Next, I will develop the Object class and derive the Object_Menu class, and ensure that I can add objects to an area and interact with them appropriately. Next, I will add the Inventory class and the derived Inv_Menu class, and debug as necessary until I can pick up and interact with Inventory objects.

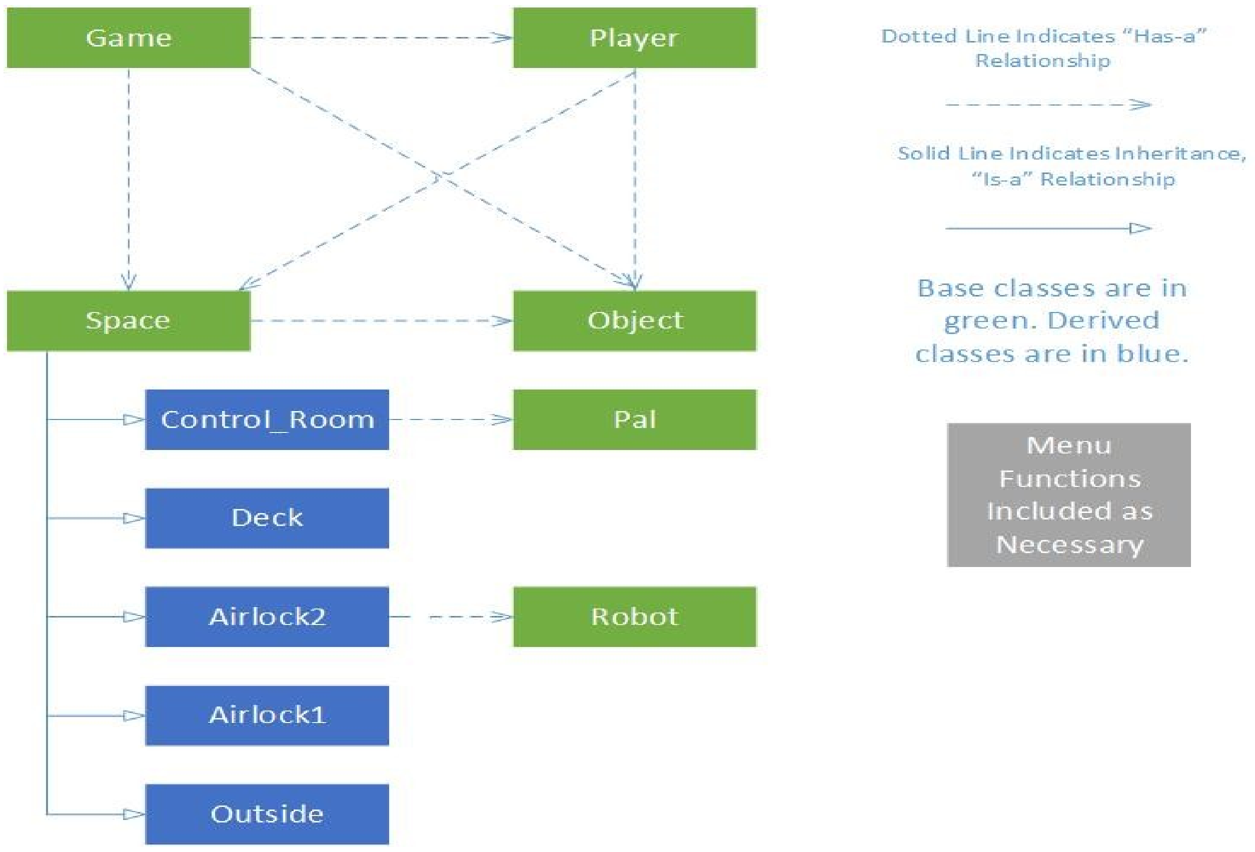
At this stage, most of the complicated tasks will have been dealt with, but I will maintain the incremental approach and add areas one at a time and test each. When this is complete, I will walk through the game and attemp to break it to find bugs. Finally, when I have a fully functioning collection of spaces and objects, I will flesh out all the details to make the game experience richer and more interesting.

Design Changes

I ended up making a number of changes during the implementation phase. One of the first big changes I realized was necessary was to create a Player object. I decided that the Player would then have a vector member for the inventory, so I wouldn't need a separate Inventory class. The Player would then have-a Space location pointer. Another big change was that I had to create a Game class containing all of the game objects. This class has a function play_game() that loops until the Player wins or loses. Additionally, and very importantly, it has a function that checks various variables in the game and updates other variables depending on various game conditions. This became necessary since several different objects that had no relationship with one another had to know what happened in the other, so this function fulfills that purpose.

I also realized that the Galley, Bunk, and Storage classes could actually just be instances of the Deck class instead of their own subclass. This simplified things considerably, although it did lead to the game being less interesting.

Instead of creating a menu class I simply put a couple of menu functions together and included them as necessary. Objects that have menus simply create their menu options and pass them to the menu functions, often including a vector that maps menu options to option indexes. This mapping vector is necessary because not all options should be printed out every time, depending on game conditions, so the mapping vector acts as a switch to prevent erroneous menu options. I also had to create two additional complex objects, a Pal object and a Robot object. The Pal is a mainframe computer that the Player can talk to for information and to make a necessary change during game play. The Robot object must also be interacted with in a certain way to progress in the game. An updated class hierarchy on the next page.


Game
Player
Space
Object
Control_Room
Pal
Deck
Airlock2
Robot
Airlock1
Outside
Dotted Line Indicates "Has-a" Relationship
Solid Line Indicates Inheritance, "Is-a" Relationship
Base classes are in green. Derived classes are in blue.
Menu Functions Included as Necessary

Testing

I did fairly well sticking to my incremental testing plan. It fell apart a bit towards the end when I was fixing bugs. This was because of the approaching deadline, which caused me to try and fix multiple bugs in one go before testing to see if my fixes worked. However, during most of the development I used an incremental approach.

First I created the Space class, and derived the Control_Room from it. To test this, I needed a Player object, so I created that as well and tested by having the player point to its location as being the control room. Once I was satisfied this worked, I derived the Airlock1 class from Space and linked it to the Control_Room, and verified that I could go back and forth. It was at this point that I realized I needed an overriding Game class to contain this continuous loop.

The next step was to create the Game class. This was simple at first, and grew more complex as more objects were added to the game. For this reason, testing the Game class was an ongoing process. Next, I derived the Deck class and the three initially planned Deck objects were linked to my Airlock1 and my Control_Room. At this point, I did a bunch of testing to make sure that I could travel around the spaces with my Player.

The next phase involved creating the Pal mainframe and the Dodd-Paye doors control and decision structures. This were highly dependant on eachother and so were developed and tested in tandem. Along with these two portions of the game, I realized I had to implement a function in the Game class that would let the various objects in the game know what had happened to other objects. This control function ultimately became rather complex and extremely important, and was also developed and tested throughout game development.

Once the Pal and Dodd-Paye doors worked together, I moved on to create the Bunk object. At this point I realized the Bunk, along with the Galley and the Storage, could simply be Decks with different descriptions. I quickly linked and tested them, and moved on the the next big issue, creating the Object class. I first tested this class by adding a single object to the player inventory, and then trying to drop and use it. During this process, I realized my menu system was getting clunky, so I planned a universal menuing plan and created menu functions that could be included as necessary.

One of the trickier issues I ran into with the Objects had to do with indexes. Both the Space and Player hold Objects in vectors, and to simplify the process of removing them from the vectors I gave the Object objects an index member that was set to their position in the vector. However, this index was not getting updated when objects were removed. Since I had very few Objects in the game at this point the issues this caused were not readily apparent, and it took me a long time to track down the problem when I finally did notice it.

After I was satisfied with the Objects, I then created the Airlock2 class and made sure that the Dodd-Paye doors were controlling access between it and the Airlock1 object. I then created the Robot class held within the Airlock2, and tested the options and control variables in this area until they worked. Finally, I derived the Outside class from Space, and tested the win conditions. Testing the win-loss conditions involved consideral work in the update_variables() method of the Game class, but after a great deal of testing and debugging I had it working to my satisfaction.

At this point I had little time left to make the game interesting and pretty, so unfortunately it is probably pretty boring at this point. My final testing involved trying to break it, though, and I am reasonably confident it is largely bug-free.

Reflections

One lesson that was reinforced with this project was the importance of planning for a large program. Even though I made considerable changes to my inital plan, the initial plan gave me a solid foundation to start from. Once it became clear I needed to redesign things, I made sure to actually write out the new plan, and this also helped immensely going forward. I also was further reminded of how useful incremental testing is. Even small changes can result in tons of bugs, so the incremental approach definitely saved me a lot of headaches. I think, however, that the most important lesson I am taking away from this is that I am still do not have a great grasp on how long projects of this size will take me to complete. I think it took me about three times as long to finish this as I had thought it would, and I will try to multiply my expected timeframes by this factor of three for future projects until I get a better grasp on my abilities and the complexity of the tasks I am tackling.