Program Overview
This project was to design and implement a text adventure game. The game had to be menu based, not a text parser. The game must consist of at least 6 rooms, 3 of which must be unique. Each room must have 4 pointers that either point to another room or to NULL to simulate a connection between the spaces of the game world. There must be an inventory system with a limited capacity, and some of the rooms will require player interaction beyond moving through them or picking up items. Finally, the game required a time limit mechanic.

The final design for this project was a space themed game with 7 rooms, 2 objective based timer mechanics, and an inventory system that allowed the player to swap items with the currently occupied room when his or her inventory is full.

The player is on a ship in space that has been struck by micrometeorites. There are holes in the hull that are leaking air rapidly, and an electrical short causing power fluctuations that are damaging the ship's reactor. The player must patch the holes in the hull and repair the short before restarting the reactor for the good ending. It is also possible to only patch the hull and restart the reactor without fixing the short but this has a 90% chance of destroying the reactor.

Program Design
After a few design iterations the implemented design consisted of a Game class, which covered player actions as well as general game mechanics, along with a virtual Room class and 7 different child classes of the Room class, one for each type of room used in the game. The program also uses the DataValidation utility class.

Game class
The variables for the Game class are:
2 ints air and power
4 bools airpatch, elecpatch, gambling, exit
2 strings clear, events
2 string vectors inventory, exchange
1 Rooms pointer currentLocation

The ints air and power were used as a time mechanic for the game. They decreased at varying rates for every action the player takes. If either one reaches 0 the game ends.
The bools tracked objectives being completed (patching the hull, fixing the short) or end conditions (if the player gambled by not fixing the short and if the game is over).
The string clear is determined at compile time. Since this program was written on a Windows machine but has to be able to run in a Linux environment a variable was required to give the proper system command to clear the terminal screen. On a Windows machine clear will equal "cls", on Linux "clear".
The events string will change based on player and room actions and will be used to update the display.
The vectors inventory and exchange are used for player inventory. Since the player has an inventory limit of 2 items and vectors are easily used when taking the last item the vector exchange was added to allow for transfer of items to and from the player and a Room. In the event of a swap, an item will be

placed in exchange, then there is a transfer of items between the player and the Room, finally the item in exchange is put where it belongs.
The Room pointer currentLocation is exactly as it sounds, a pointer to the Room the player is currently in. It is used in all methods that involve player to Room interactions.

The Game class has 15 different methods:
tick which decrements the air and power variables.
clearScreen which clears the terminal screen.
writeStatus which writes the current air and power remaining as well as the current room name.
writeDes which writes the current room's description.
writeEvents which writes any events such as taking an item or performing a room action.
writeMenu which writes the list of actions the player can choose from.
endGame which writes out the ending the player has reached.
addItem which will add an item from the Room to the player inventory.
exchangeItem which will be called by addItem if the player inventory is full and will allow the player to swap one of their items with the item in the room or cancel the item exchange.
travel which allows the player to change the currentRoom to one that is attached to the room they're in.
search which is update event to notify the player of any actions they can perform or items in the room.
action which will perform an action in the room if the player is able.
menu which will call other methods based on player inputs.
play is the main method loop that runs the game until an end condition is reached then calls endgame.
startup is an introduction method with a menu that allows the player to play the game, see a walkthrough, or exit the program. If play is chosen introductory text is displayed that gives the setting and objectives of the game before calling the play method.

Room class
The variables for the Room class are:
4 Room pointers north, south, east, west
1 bool finished
3 strings description, altDescription, name
1 string vector inventory

The Room pointers are the Rooms that are accessible from the current Room.
The bool denotes the action has been performed for this Room and the altDescription should be used
The description and altDescription strings are used to describe the room to the player both before and after actions have been performed. The name string is the name of the Room.
The inventory vector is the Room's inventory

The Room class has 17 different methods:
1 set and 1 get method for each direction i.e. *setNorth*, *getNorth* etc.
hasAction which returns a string to the game class of what item is required to perform an action in the Room.
actionDescription returns a string to the Game class for the Game events variable to describe the action that occurred in the room.

action which returns an int to the Game class to be used to change the objective and end game bools.
getDescription which returns a string of a description of the room that is either description or altDescription to the Game class to be used with the writeDes Game method.
getName which returns a string of the Room's name to the Game class.
itemReport which returns an int of the number of items in the Room.
getItem which pops and returns the item string from the Room inventory.
itemName which returns the string of the item without popping it from the Room inventory.
setItem which adds an item to the Room inventory.

ComRoom class
ComRoom is a child class of Room and overrides the *hasAction, actionDescription,* and *action* methods.
The *hasAction* method returns "tool kit".
The *actionDescription* method returns a string that suggests a tool kit could be used to repair the short.
The *action* method returns 2 and sets finished to true. The returned 2 tells the Game class that the "fix short" objective has been achieved.

Engine class
Similar to ComRoom, Engine is a child class of Room and overrides the *hasAction, actionDescription,* and *action* methods.
The *hasAction* method returns "engine lockout key".
The *actionDescription* method returns a string that suggests an engine lockout key can be used to restart the reactor.
The *action* method returns 3 and sets finished to true. The returned 3 tells the Game class that the exit condition has been achieved.

Engineering class
Engineering is a child class of Room and does not override any methods.

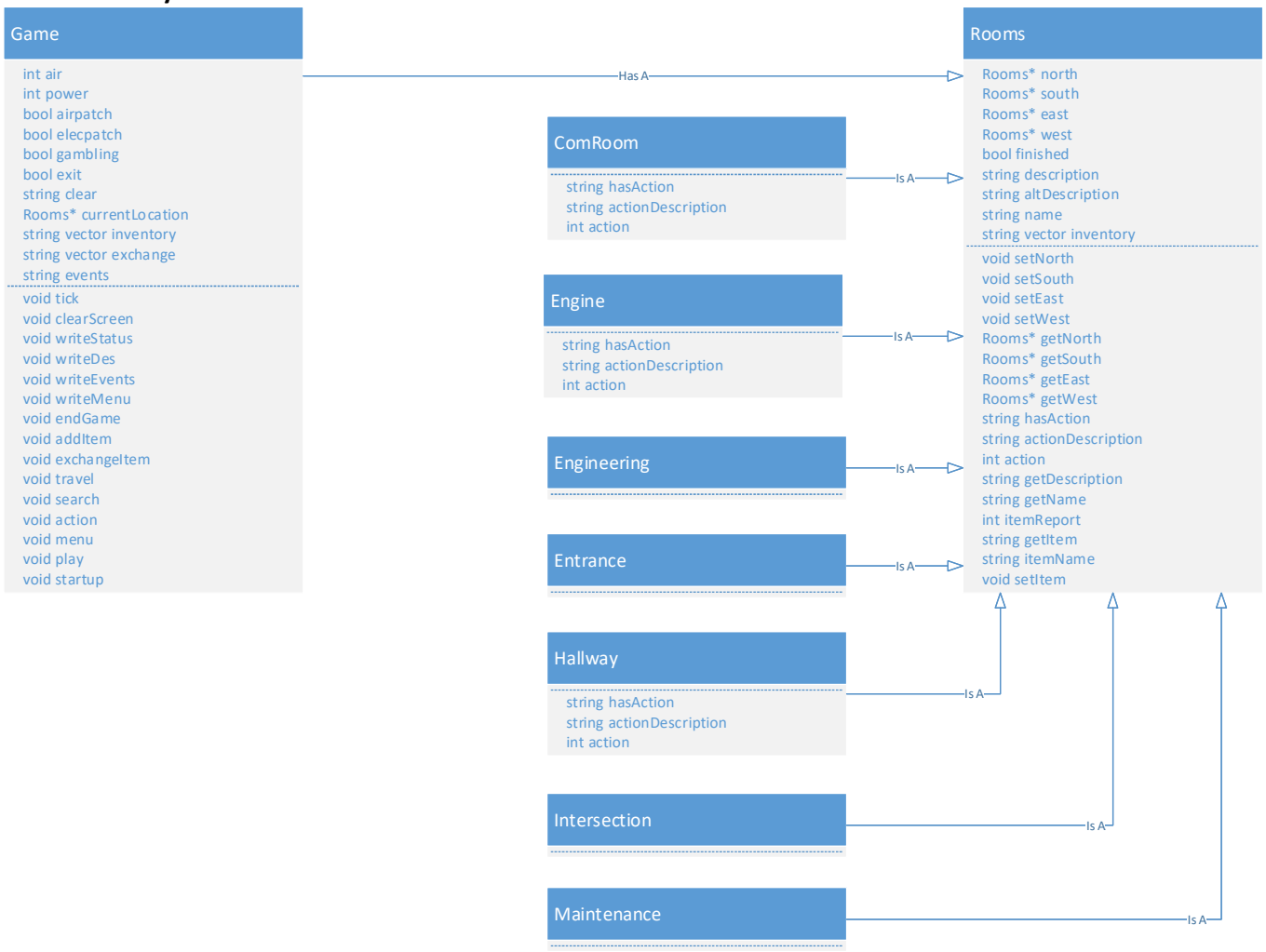
Entrance class
Entrance is a child class of Room and does not override any methods.

Hallway class
Similar to ComRoom, Hallway is a child class of Room and overrides the *hasAction, actionDescription,* and *action* methods.
The *hasAction* method returns "patch kit".
The *actionDescription* method returns a string that suggests a patch kit can be used to fix the hull.
The *action* method returns 1 and sets finished to true. The returned 1 tells the Game class that the "patch hull" objective has been achieved.

Hallway class
Hallway is a child class of Room and does not override any methods.

Maintenance class
Maintenance is a child class of Room and does not override any methods.

Class Hierarchy



Testing

For the testing of this program while in each room the search, action, take item were attempted as well as the 'Go' command for each direction. In each case, the program either allowed the action as appropriate or denied it with the appropriate explanation text. Likewise, allowing the air supply to reach 0 correctly ends the game with the appropriate ending message. Since the air supply will always be less than the power remaining there is no way in game to test the power loss ending. The "gambling" ending was tried until both endings, kind of good and bad, were achieved. The game was played as intended and the optimal ending was achieved. The menu options presented when the program first loads were also tested and worked as intended. Finally, valgrind was run and no memory leaks were found.

Reflections

This program took a bit longer than I originally anticipated. The process of getting the text onto the terminal screen, properly formatted, in distinct sections took roughly a third of the total design, coding, and debugging time. Another design item that took far long than expected was coming up with a solution for transferring data between the Room classes and the Game class. For example, when the player wants to perform an action in a Room a method had to be added to the Room class to tell the Game class what item, if any, is required to perform an action. Then the Game class checks its inventory and if that item is there, tells the Room to perform the action. Finally, the Room class may need to tell the Game that an objective has been met so another Room method was created to do this. Game methods were also developed to catch and process all of these new inputs from Room.

For the sake of time the originally planned Player and Item classes were cut. The Player class was absorbed by the Game class, and the Item class was replaced by item strings. As the coding phase began though, I noticed that the program was experiencing feature creep. What was once one timer, the air leak, became two, air and power drains. Likewise, the original design only called for enough items to reach the player's item capacity but not exceed it. The final version has double the original number of items and the ability to have the player swap a chosen item with the item in the current Room.

Overall, I am pleased with how this program turned out. I enjoyed having very few actual requirements and being able to largely do my own design.