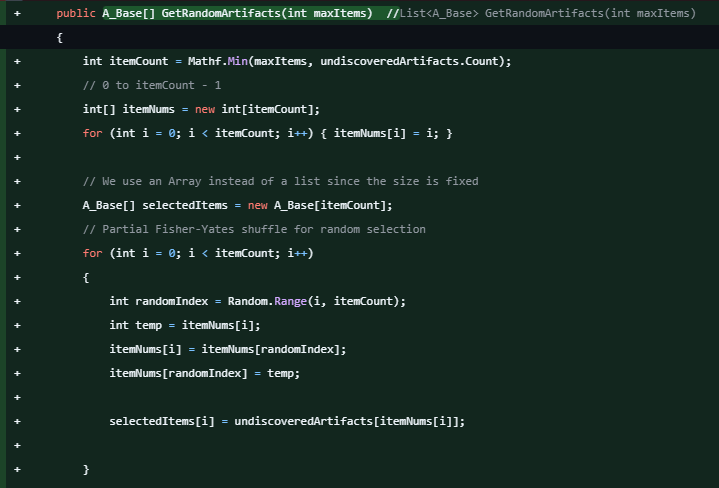
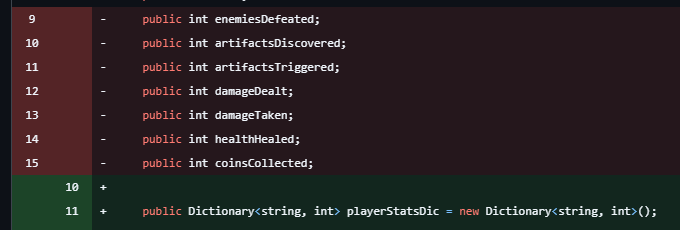
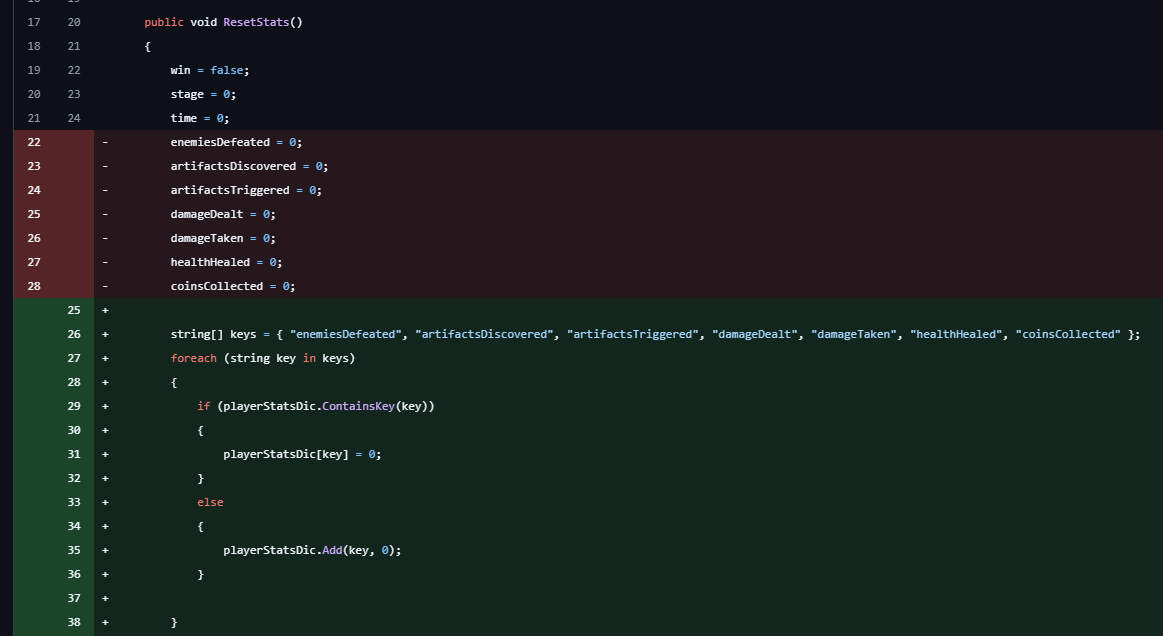
**Data Structures Final Project Reflection Report**

**By Stephen Zoccoli.**

Tier 1:

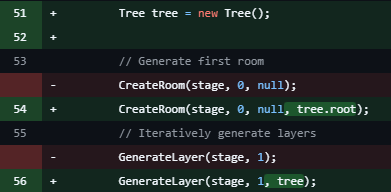
* ShopManager.cs used to use a List<> of Artifacts for the shop. The list was fixed to size 6, or the maximum number of undiscovered artifacts remaining (whichever is less). Since the size is fixed and pre-determined, I switched to a fixed size Array.
* The old algorithm shuffled all undiscovered artifacts, and took the first 6 of the shuffle (Fisher Yates). I changed to do a partial Fisher Yates shuffle, only up to the first 6 items.
* (Insert algorithm into PDF. Insert Array change).
* 
* The change in ShopManager.cs required a small change in ArtifactManager.cs, now getting an Array from the respective function instead of a List.

Tier 2:

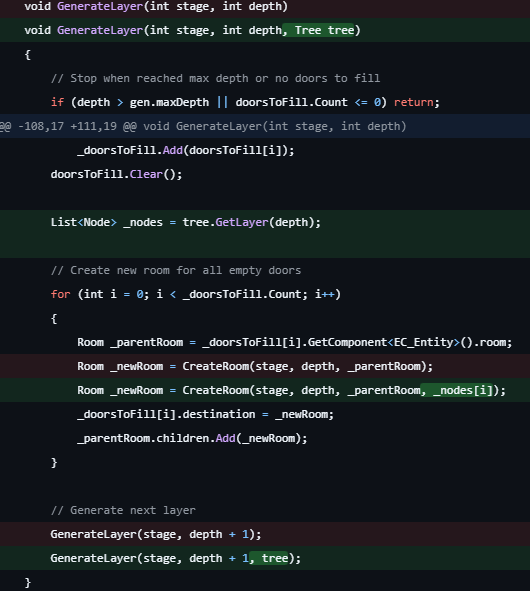
* PlayerStats.cs had several separate variables for the “# enemies defeated”, “artifacts discovered”, “damage dealt”, “health remaining”, etc. Instead, I used a hash table (dictionary) called “playerStatDict” for all of these.
* 
* All variables were initialized to 0. Instead of doing it manually for each variable, now with a dictionary, I can for-loop “foreach key” and set all to 0. (include code).
* 
* This change propagated to several other files that used these variables. That took some time to find everything.

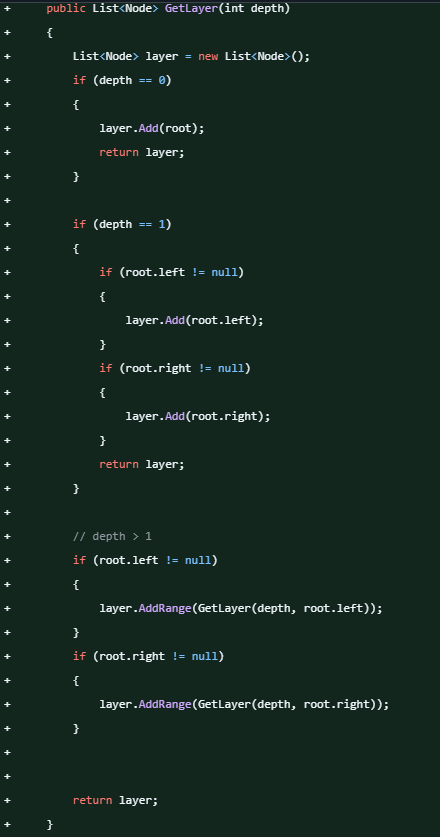
Tier 3:

DungeonManager.cs randomly generated the dungeon as a tree. This is done layer by layer up to a max tree depth. I pre-defined a tree structure, then made the dungeon follow the pre-defined tree.



To get this to work with the current tree building algorithm, I needed to write a getLayer function for the tree, to integrate into the current dungeon build.





In all cases, the end result is the same, but the code should work faster, i.e., optimized.