REFLECTION:

My overall class design for Project 2 didn't stray from the original UML I developed, in fact the only change to the super structure was the introduction of a new Sprite subclass Explosion. In my original design it was intended that when an explosion occurred it would simply take over as the preferred rendering of the TNT class however during development this became more difficult to implement than originally foreseen. Problems that I encountered included maintaining TNT in the game map without it interfering in the Player class' ability to make legal moves and how to code the "switch" criteria between rendering a TNT block and an explosion. In order to simplify and eliminate these problems I developed a sprite introduction and removal system that was administered by the GameMap class, with this system in place triggering an explosion became much more convenient and to be honest it made much more sense. Encapsulating an explosion in a separate Sprite subclass meant triggering an explosion was just the introduction of a sprite and the removal of TNT and CrackedWall in the same operation which solved the problem of Player interference but also ensured simplicity in coding the render methods in both TNT and Explosion and allowed for the use of the super method for the most part.

The only other major difficulty I faced in this project was being able to undo moves prior to the Explosion without undoing the explosion itself, with the undo and recording system I implemented this was extremely difficult and would have involved several layers of deletion within the game map copies, I decided that this overhead wasn't necessary and instead took the opportunity to add an additional layer of complexity to the game play in so much as preventing the player from undoing any moves prior to an explosion trigger. This modification meant that triggering an explosion also triggered the deletion of all prior moves, this not only simplified the game from a coding viewpoint while simultaneously meeting the implementation checklist but it also made more sense with the copy-store-undo technique I developed and it meant making a mistake was less forgiving, games shouldn't be too easy in any case.

The biggest lesson I've learned from this project is that copying should never be underrated, especially the use of deep copying as a recovery or in this case undo technique. The simplicity added to my game by simply copying the game state after every move and then rolling it back when undo is called can't be overstated it prevented messy command recording and rollback but it also meant identifying a bug was much easier. If I was testing the undo and restart features and the game state jumped back too far or didn't change at all I was able to identify an issue with the queue implementation much more easily because I already knew that the game state was a self contained object and so couldn't cause the corruption, likewise if a particular class was copying incorrectly it was very easy to discern from the rollback of an undo as it wouldn't change with the rest of the game which allowed a much swifter bug identification than some other techniques may have.

When I think about what I would do differently a few things come to mind the first of which is my approach to copying. In this project copying the whole game state requires relatively small amounts of storage and is efficient enough to work however in a larger system or a more complex game, the storage required grows quite exponentially and this very simple copying technique may become more hassle than it's worth. When I think about this what immediately jumps to mind is reducing what's copied or modifying how certain classes are copied. Immobile and completely immutable classes such as Wall and Floor probably don't need to be deep copied and could simply be reference copied, another way to reduce the storage needed would be to use a sort of incremental copy, only copy what's changed instead of copying the whole state in a sort of OSX TimeMachine style approach. The other major issue that comes to mind is the way in which I use the GameMap class as a conduit between GameObjects and the World class, passing instructions or changes between them which when I think about it is really outside the scope of what a GameMap should be. For my implementation this worked very well as GameMap was for the most part the only common class between the two however I think in another life this usability should have been contained in either another class designed specifically for this purpose or inserted into the World and GameObject classes so as not to bloat the role and responsibilities of what in essence should simply be a class for storing and looking up the game state.