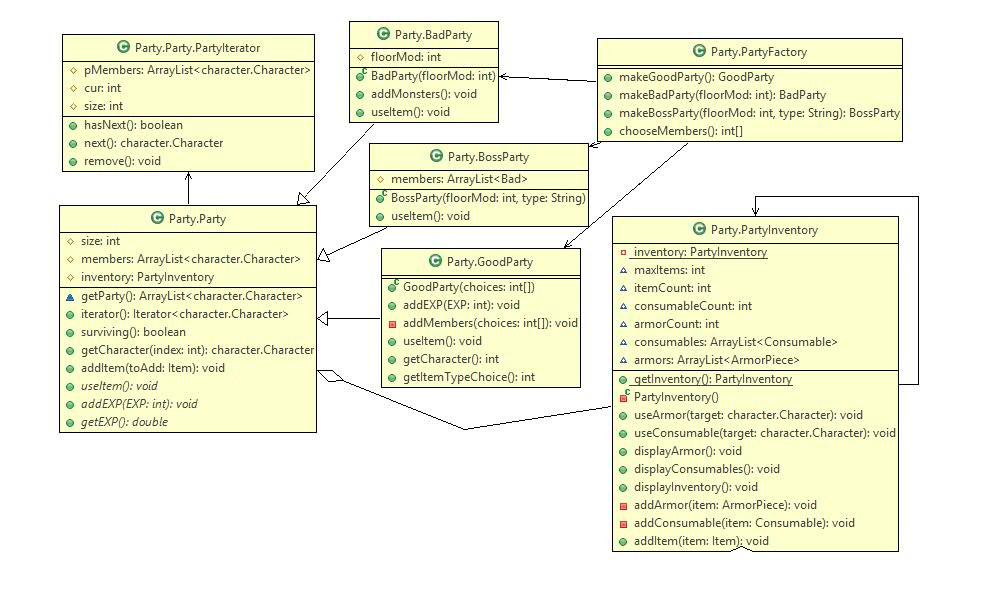
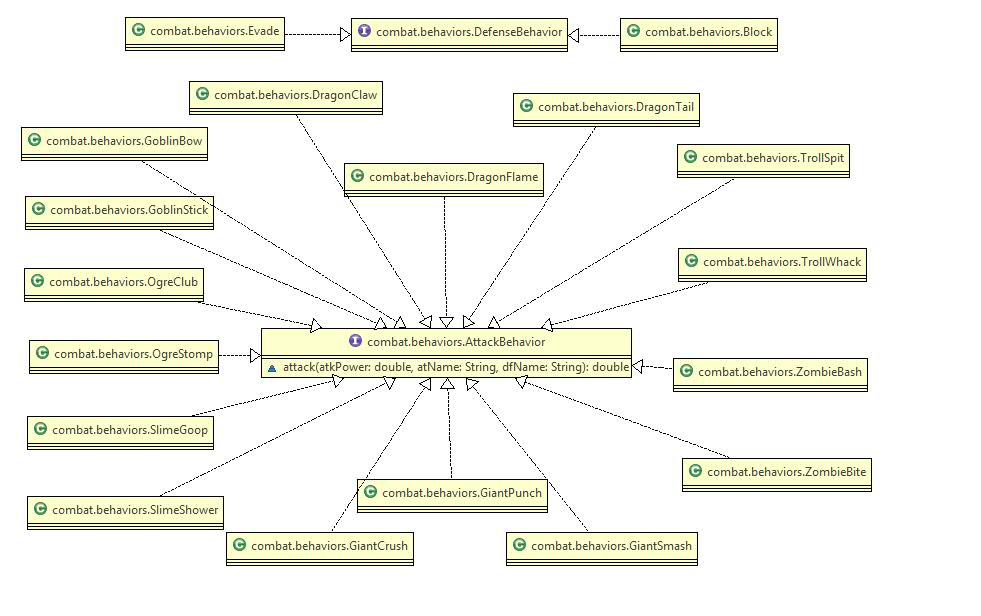
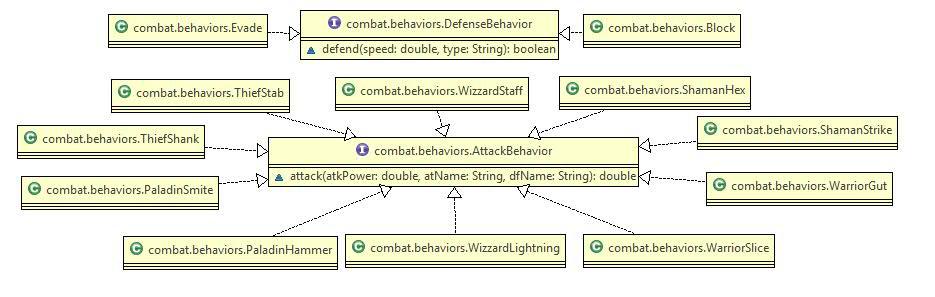
Final Project Description Of Used Patterns

Throughout the development of this project, several design patterns were used and/or considered. Some patterns did not work out and were changed during development. Others (such as patterns like factory) were used several times throughout the different pieces of the project. This document will attempt to cover all design patterns that were used, in the final completed project.

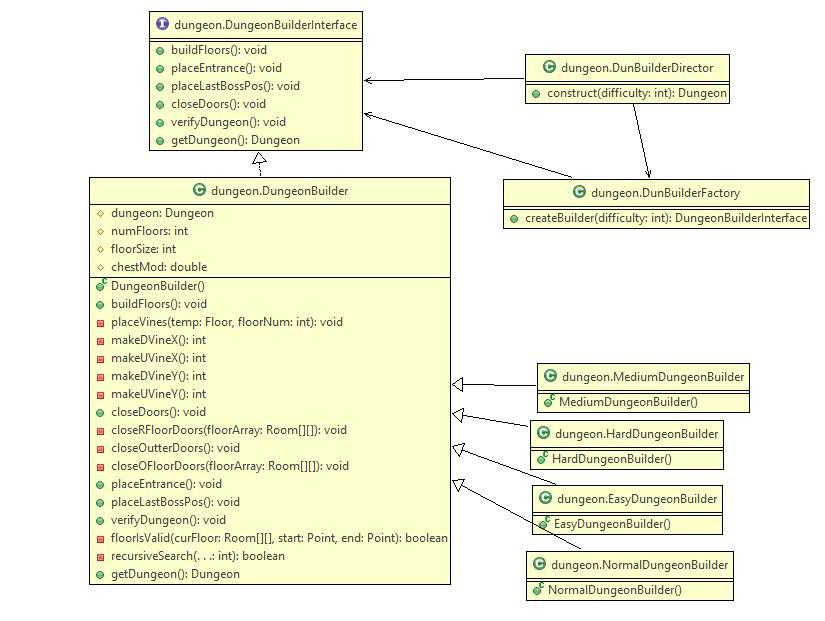
Beginning with the party system of the project, three separate design patterns were used. Singleton pattern was used for the party inventory. This allowed for the inventory to remain consistent from many access points, both from the party level and individual character level. Simple factory was also used in the party system. All of the different types of parties shared similar attributes and characteristics; for this reason all of the logic necessary for building a party of any type, was encapsulated within the factory. Iterator pattern was also used within the party system. This allowed outside classes to iterate through that party members contained within the party system, while staying decoupled from the outside classes. Below is the corresponding UML that illustrates these three patterns in the party system:



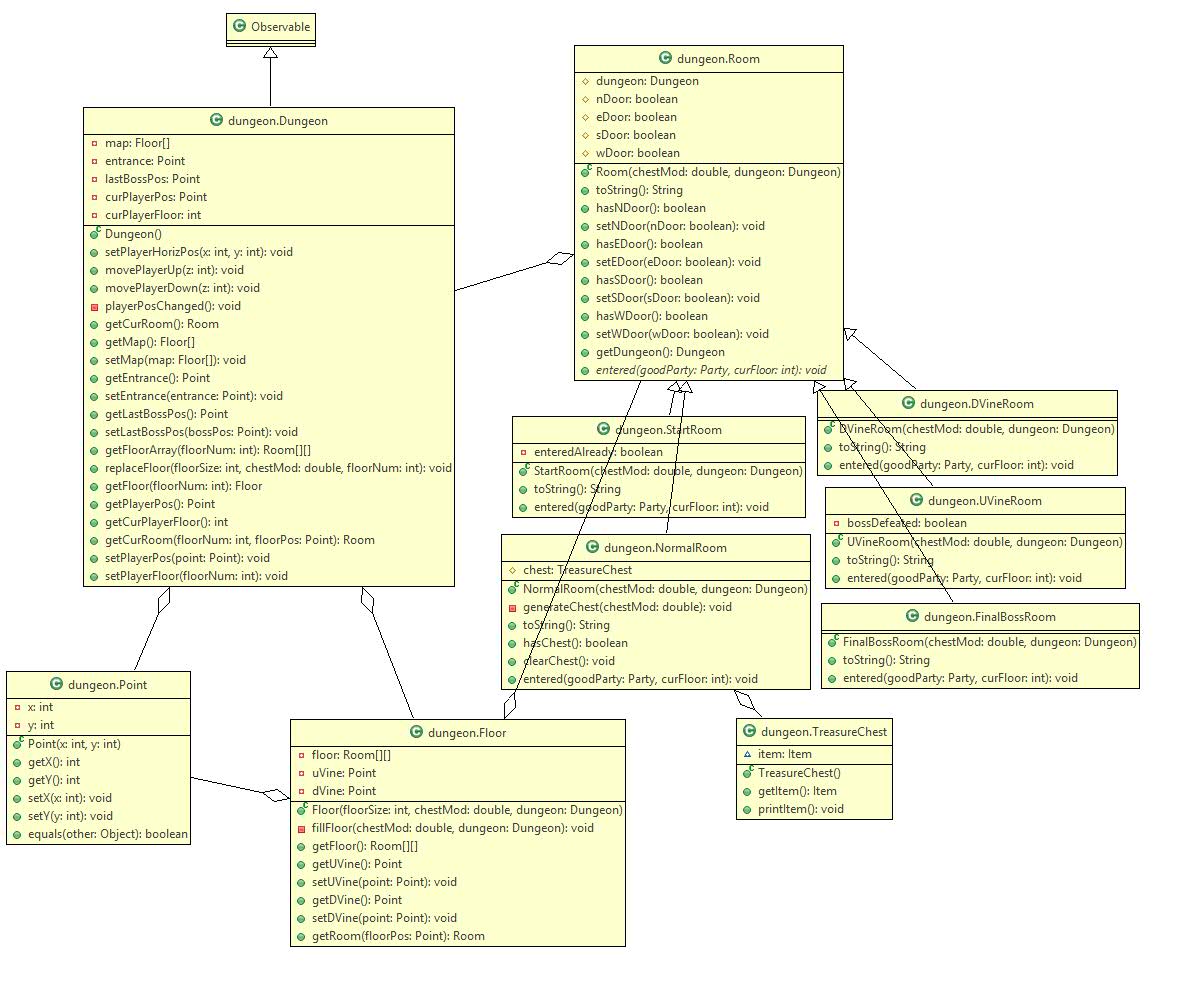
Next, in the character/combat systems Strategy pattern was used to determine attack and defense behaviors. This pattern worked well here, because it was very easy to change attack and defense behaviors in combat, based on a users choice. Additionally this system is easily extensible for new attack and defense behaviors. Below are the corresponding UML diagrams that illustrate this pattern used for both hero character attack and defense behaviors, and also monster character attack and defense behaviors:



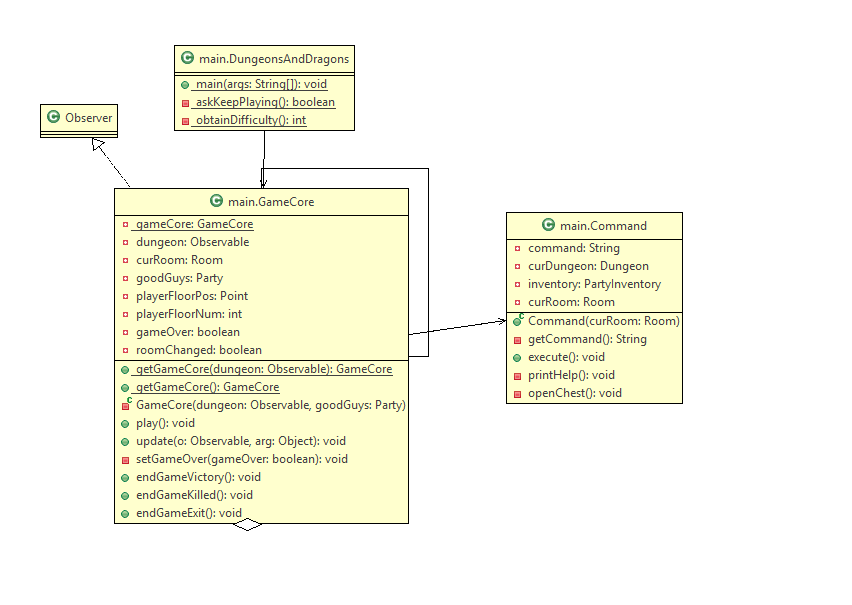
The dungeon was a fairly complex structure that utilized several patterns. To construct the dungeon we used a builder pattern. There were several steps involved with creating the dungeon that needed to be completed in a specific order, so the steps were laid out in a director class. By using a builder to create the dungeon we were able to minimize the size of the dungeon class, as well as compartmentalize the creation of this complex object. There was also a strategy pattern used to pick which type of builder would be used to build the dungeon (easy, medium, etc.) and a simple factory to retrieve a builder of the correct type.



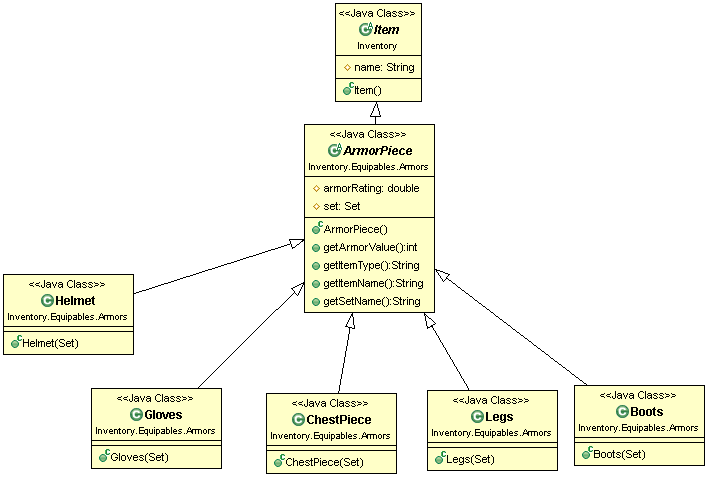
We needed several different types of rooms, so we implemented a strategy pattern based around a Room superclass to allow for the easy switching out of room types. The dungeon also needed to let the gamecore know when the player's position changed. To keep things as decoupled as possible, we implemented an observer pattern with the dungeon being the observable, and the gamecore being an observer.



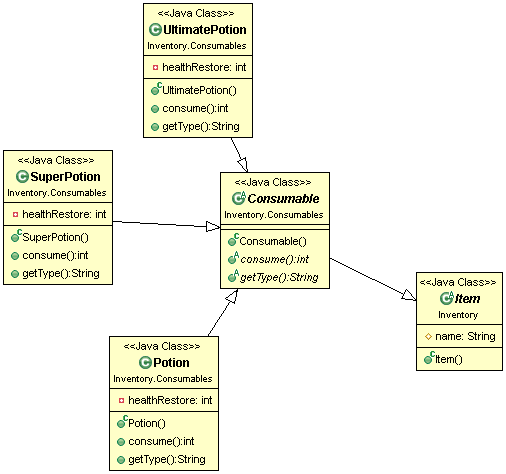
The gamecore system uses several patterns. As mentioned before, the gamecore is an observer of the dungeon. When the player's position gets changed, the update method will provide the gamecore with the current room the player is in. In addition, the gamecore has had a singleton implemented on it. The reason for this is that we only ever need one gamecore, and having global access to the gamecore was nice in that we could update it from the command class. Although we did not fully implement a command pattern in our command class, we did compartmentalize user commands while moving around the dungeon in it, which made testing user commands much simpler.



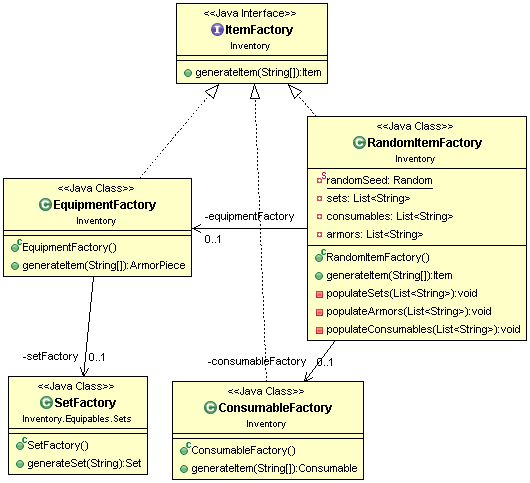
The armor implementation was a little rough around the edges, because we struggled with the hardship of determining which type of item a user is interacting with, so they don't end up placing a chest piece on their head. To avoid this, we decided to enforce this restriction based on type and abstract the armor piece. So, we effectively created a pseudo-template method pattern. The check we used to distinguish what type of item is being equipped, violates good OO design since we basically ask the item what it is, but there was no way around this at the time for us.



The consumable implementation was fairly straightforward and easy. We decided to use a strategy pattern here and make different types of items all containing consume() behavior.



The Item factory implementation was also pretty easy, since we decided to abstract the factories and let them handle the actual generation of the item. By doing this, we effectively generated specific item types without making one god factory.



The set implementation was another tricky one to implement, since sets had to know how many pieces they had, in order to give a proper set bonus. It also needed the the ability to know what that bonus was, and apply it effectively. The whole picture comes together (outlined below), and the armor piece contains a set, which effectively drives it's defensive capabilities. The set effectively implements a pseudo-template method pattern, and acts as a strategy pattern for the armor piece.

