We developed a GUI for our game, so it's use is more-or-less evident from that. There are two available buttons after starting the game: "New Game" and "Load Game". Admittedly the load game button does nothing right now. We were planning on putting that in, but never had time. Clicking the new game button takes you to a character creation window. Here you enter the names for the two characters you will control as well as their class. Your options are Fighter, Mage, and Healer. The "Ready Up" button on this screen starts the game. Your options here are to "Fight", "Boss Fight", view your inventory, visit the store, and save the game. Clicking the save button doesn't do anything, but it may in the future if anybody in the group continues development on the project after this class. Starting a fight takes you to the battle screen where 1-3 randomly generated enemies appear. Starting a boss fight does the same thing, but these enemies are much more powerful.

For each of your characters you have the option to attack, defend, use the character's special, use an item, or run. With attack, defend, special and use item you pick a target. This allows you to do things like attack an enemy, use a potion on one of your characters, or even attack one of your own characters! Attack uses your character's weapon and some of your base attributes to determine the damage to the enemy. These attributes are health/damage, stamina, mana, intelligence, strength, and defense. Defend increases the character's defense attribute. This attribute mitigates damaged done to you by an attack. Special is different for each class. The fighter's special does two normal attacks in a row. The mage's special does damage according to how much intelligence she has. The healer's special heals according to how much intelligence he has. Using an item allows your character to select an item from the party's inventory (a party is your two characters) and use it on one of your characters or one of the enemies. An item could heal or do damage.

The game is turn based among characters and the two teams. The fight is over once your two characters get to 0 health or your enemies all get to 0 health. Each defeated enemy gives experience points and loot. The loot is added to the party's inventory and the experience is given to each character.

All characters get the same amount of experience points, which is determined by the enemy. After a character gets a certain amount of experience points it levels up and its attributes are increased. The amount of experience required to level up again is increased with each level up. With each new battle the enemies get stronger and the loot gets better.

We used the command, actor, strategy, factory, and composite patterns. We probably used some other patterns, but these are the ones we explicitly planned on using. The command pattern is used to execute different kinds of commands, which include attack, defend, special, and use item. These are added to one of two queues and can be executed later. We implemented the actor pattern by having two separate queues for commands, allowing us to mimic two threads.

The specific implementations of attack and defend are easily customizable via a strategy. The abstract Character class, which is the parent class for the Player and Enemy classes, contains an interface with Attack() and Defend() methods. We implemented one version of this interface to give players and enemies an implementation of these, but completely different versions of attack and defend could be implemented. The abstract class Player extends the character class and implements some additional features like experience points, inventory, and leveling up. Classes Fighter, Mage, and Healer extend the Player class and each implement the Special() method, which is an abstract method defined in Player. Each of these classes implement this quite differently.

The factory pattern is used to create items like weapons, armor, and usable items, e.g, potions. It is also used to automatically create enemies based on the current level. This way stronger and stronger enemies can be created as the game progresses.

We implemented the composite pattern in a CompositeArmor class, which extends Armor and contains two Armor classes as members. We ended up not using this because it caused problems for us, particularly in inventory control.

I regret choosing to work in a group, especially a group of 5. By far the hardest part was working with four other people. We had a hard time organizing all of our ideas into a single coherent design in a timely manner. I think this was for two reasons. First, 5 people is too many for a project of this size. Second, we had no group leader. When working with just yourself or one or person, plans and designs can get effectively communicated and implemented with simple conversations. A leader is necessary with a dynamic of 5 people. The bright side is that besides getting some good design pattern practice, the chaos that was this project was a valuable lesson for me. In any future group of more than 3 people, I will make sure a leader is explicitly established.

In the first few weeks of this class I thought I had made a mistake because it was too easy. Later I changed my mind. I found this class valuable because it gave me an opportunity to concentrate solely on code quality. In my other classes there was always something difficult I needed to accomplish in a limited amount of time. I tried to use good OOP practices were I could, but I often had to sacrifice good code quality because of limited time or because I was not able to develop good code in conjunction with solving a hard problem. With this class I could easily see the benefits of the design patterns and concentrate on a good design because the actual programs we implemented were simple. It made it easy on my small brain.

There's a difference between writing good code and writing code that solves a difficult problem. Both are hard things to do. Writing good code is obviously a good thing for testing, maintainability, reusability and all that other great stuff, but is not strictly necessary. And doing both at the same time is sometimes too hard in limited time. In your class I got a chance to concentrate on code quality while my other classes did not seem to care so much about that.

Object oriented coding is awesome. I love designing classes and their interactions. Design patterns are like revelation. Something as simple as the combination of the strategy and composite patterns to create arbitrarily complex tree structures is such a powerful tool. I really do not know if I

would ever have thought of such a thing myself. Having explicitly learned that such a thing is so easily possible makes it that much more likely I will use it in the future to create better, simpler, cleaner, and more powerful code. I think design patterns allow for creativity in my programs without requiring me to be creative myself. They make it easy to provide simplicity and power simultaneously, which is great because it makes code both easier to understand and more robust in its functionality.

Think about all of the amazing things that computer programs are doing right now and even how much better they will likely be in just a handle of years from now and you will get an appreciation of OOP. These awesome programs would not be here today it were not for OOP because it allows us to create better software with less mental effort. We can write extremely complex code while still being not that smart.