Throughout writing our code for the project, there were useful patterns that we applied to help the code run smoother. However, there are definitely more that we could've added.

In terms of delegation of responsibilities, we came across a decision for the FileManager class. While we could've followed the Information Expert approach and allowed Game to deal with all the saving, this would lead to lower cohesion in the Game class. As well, since the Game class is not immediately created upon starting the Game, the load game functionality would likely have to be in the StartUp class (as this is the only active when this system asks to load a game), which would lead to duplicate code across the two classes, as well as low cohesion and high coupling for StartUp. The solution for our project was Pure Fabrication. By this pattern, we created a new class titled FileManager to handle all file related actions. This class separates all of the reading and writing into files from the actual game functionality, and in turn creates a high cohesion amongst the FileManager class.

For managing the UI/user input events, we attempted to implement a bit of a Controller class, as both StartUp and ChoosePieces classes are made with the sole purpose of gathering user input and passing this along to the Game class (an example of the Creator pattern, see below). This would also follow the Indirection model, by placing an intermediate class between the user, Board, and Game. Once the game begins, all user input talks directly to the Board, which has led to a bit more of the gameplay to have to be focused in the Board class then would be ideal, causing the Board to have to be made aware of most of the classes in the program, which increases the coupling. This central basis is probably best focused in the Game class, and given more time, the addition of a user input class that could talk straight to game might decrease this coupling.

This connection between board and the game functionality also means that on many occasions, Board is skipping over the Game class with calls to Turn that for the most part appear in the form "game.getTurn().(some method in turn)". While there are a few instances of calls that follow the suggested pattern, it is mostly the opposite of what Protected Variation suggests (similar to the example). Following this patterns suggestion, it would improve code structure to hand over more responsibilities to Game in terms of facilitating the transition to Turn.

The Creator pattern comes in handy for the ChoosePieces class. ChoosePieces contains all the necessary information to create the Game class, so it is the creator. For creating the Board and Turn classes, each of these aggregate Game, so the Creator pattern would assign Game to create these, which is what happens in the code.