**Chidamber-Kemerer**

Our class with the worst rating for this was the Player class. This was quite expected as we already knew that the Player class has high coupling and low cohesion. The coupling between objects rating is 52, the highest in our entire project, which makes sense as the player class is dependent on quite a few different classes. This and its Weighted Method Complexity is at 74, also the highest rating in our entire project, as the player class has way too many methods that do things that the player class should not do. For example, the player class has its own move validator methods, it would be easy to integrate the player with the move validator class, as well as, utilizing the new methods we created in the BoardView class for the AI, in order to decrease the coupling and the cohesion simultaneously. The rest of our classes were actually about average in this metric and there were none in red. AI had one of the highest WMC but that is to keep its coupling and cohesion down. Player lobby also had a high CBO rating but again that one is fairly unavoidable as the Player Lobby class is one of the main connectors from the model to the UI so it is always going to have a lot of dependencies.

**Complexity**

We had a total of eight classes in Red for the Average Operation COmplexity (OCavg): BoardViewTest, MoveValidator, GetGameViewRoute, BoardRater, AI, GetTournamentRoute, Player, and GetHomeRoute. This is fairly expected as well as the are the main classes in our project that make all the decisions, they are the most well-connected class and have the most functions. To improve this rating, we could most likely break down our larger classes into smaller classes that have more specific functionality, but we can’t make it that much longer. The Weighted Method Complexity ratings are the same for this as they were above in the Chidambder-Kemerer section. In terms of methods all of our methods were fairly well within the acceptable ranges with notable outliers being turnHelper in the AI class and scanForFirstJump in the player class. These are to be expected as these are two of the largest methods in the entire project, turnHelper could certainly be broken down into multiple separate methods as it does an insane amount of work. ScanForFirstJump just has a lot of duplicated code, it could be simplified with loops and extra calls to new methods. Our overall project complexity metrics were well within the acceptable range as our V(G)avg was only 2.48 and our v(G)tot was 686, which are actually fairly low given the overall number of methods and functionality, again we would mostly need to fix the Player and AI classes to decrease that number even further.

**JavaDoc**

Our overall project percentages were as follows: 42.65% of our classes had Javadocs, 0.83% of our fields had Javadocs and 32.62% of our methods had Javadocs. This was also to be expected. We were so focused on getting the project we didn’t really consider the Javadocs to be that important, in hindsight that is always a mistake as there were times in our project where it was notably frustrating attempting to understand code done by another team member. In order to fix this we would need to go through and add Javadocs to most of our classes, as in the class view we can see that many classes are either fully covered or not at all.

**Lines of Code Metrics**

Nowhere in the metrics did we have any classes that showed up in Red, they were all within the bounds. Again, our classes with the most lines of code were the AI class and the Player class, with 366 and 251 respectively. This could easily be decreased again by dividing up responsibilities into helper classes and removing duplicate code. However, no matter what we do, those two classes will always be the largest, there is nothing we can do about that. Our total project had about 8,000 lines of code, excluding comments and whitespace, which seems to be fairly low considering the overall function of the project, but again if we wanted to lower this the first place to look would be the Player and AI classes. The tests also make up a large portion of this which can’t really be fixed that much and doesn’t really affect the overall project.

**Martin Package**

Right away we can see that we have basically zero abstract methods and classes, this is not necessarily a bad thing, and could probably not be easily solved. In terms of Afferent Coupling our model tier has the highest count, which makes some sense, then our Application tier, and finally the UI tier; however, all the values are within the range and nothing appears in Red. This could most likely be fixed by creating more specific classes that are the sole classes who interact with other tiers, but that would add a lot more complexity inside each tier’s self. In the Efferent Coupling rating, or exiting references, our UI tier has the most, which makes a lot of sense as the UI does not actually do any work; it is solely the way for the user to interact with the back end. I would not recommend making any changes to this. The Distance from the Main Sequence rating has its highest point in model with .84 and next in application with .74, which again makes sense as the main sequence is most connected to the UI tier, this could most likely be remedied by removing some of the middle men, as the lower the value is to 0 the better, but again this would not be a top priority to fix as it is largely inherently a product of the overall tier architecture. Finally, the Instability score has its highest value in the UI tier which also makes sense as the UI is completely dependent on the functionality of the other tiers, this cannot easily be fixed. Our other two tiers are .26 and .13, which are both fairly close to zero and acceptable in the range.