Criterion C: Development

Techniques Used -

- Processing Files and Saving Data
- Graphical User Interface
- Object Oriented Programming
- Parameter Passing
- Modular coding
- Overloading methods
- Complex Data Structures

Processing Files and Saving Data:

The TeamList object contained all the information necessary to recreate the state of the program. This, along with the showwelcomemessage boolean which contained information about which screen should be shown on startup, was saved to a file. To allow the program to be used on multiple operating systems, such as Windows, Mac and Linux, the java File class was used. By creating a modular filepath using this class, the program could work with file systems in all three operating systems.

Graphical User Interface:

The windowbuilder plugin for Eclipse was used to create a basic layout of the program. Certain techniques were used to ensure ease of use for the end user and reduce data entry errors. Tables were used to display complex data, and data in these tables was connected to click listeners that could be used to modify their contents. Dropdown lists were used in place of text boxes to reduce data entry errors and allow for quicker input of data, and a message was displayed to the user in case the data entered was not of the correct format. The GUI is important because it allows an easier interface that can be used by people with less computer experience, and it is the easiest, most robust method of entering complex data.

Object Oriented Programming:

The programming language used, Java, is an object oriented language. I first identified the different types of objects that would interact in my program - Criterion, Game, Team and Player. I then created a different class to define each of these objects. Defining custom objects allowed me to store all pertinent data as one object instead of as a group of different data structures. Some of these objects such as the Criterion object were simpler and contained various simple data structures, while some such as the Team object were more complex, and contained their own methods along with more complex data structures. The team object, for example, contained the Player, Game and Criterion.

Complex Data Structures –

A more complex data structure, an arraylist of arraylists, was used to store the history of criterion change over time. The inner arraylist stored the ratings of the player for each criterion value, and the outer arraylist stored all instances of arraylist – with each instance representing the rating of the player at a given point in time. Arraylists were used because they are dynamic data structures, so can accommodate a change in the number of criterion or length of criterion history, and they allow for fast random access, so the user can modify any part of the history instantly.

Algorithmic Thinking -

- Playercompare
- CreateRoster
- FindImportance
- CreateCriterionView

PlayerCompare

```
PlayerCompare accepts two inputs (p1 and p2) and implements Comparator
Calculate Rating of p1 and p2
If the ratings are not equal
        Return the difference in their ratings
Else
        If one of the player has not played a game yet
                return the difference in the number of games played
        If the ratio of their games won to games played is not the same
                Return the difference in this ratio
        Remove all unexpected results from the game list:
                Calculate the rating of each team in the game by adding the individual ratings
                of all players on the team. When doing so, add the doubles ratings if the game
                is doubles, and singles rating if the game is singles.
                Parse the character array that holds the score of the game as entered by the
                user to obtain the total number of points earned in that game by each team.
                Divide the number of points earned by each team by the overall rating of the
                team. Compare these numbers to a predetermined threshold to test for the
                game's importance.
        Calculate the number of points each player has earned from the new game list
        If the ratio of points earned to games played is not the same
                Return the ratio of points earned to games played
        Return the difference in total amounts of games won and tied
```

This complex algorithm extends the comparator class and is used by the CreateRoster method to compare two players to see who deserves a higher spot on the team. Firstly, it attempts to compare the players using their scores on the criterion ratings. If these scores are the same, the algorithm uses the players' performance on games to compare them. During this comparison, a second algorithm is used to remove games with unexpected results from the calculation. This is so games where players were not performing at their best, perhaps due to an injury, are not taken into account in the comparison. The complexity of this algorithm stems from the multiple, nested if commands that are used to compare the players in various manners. The method used to ascertain which games have unexpected results includes computing the ratings of the players and comparing these ratings to the player's game results.

CreateRoster

```
Sort all four team lists using the PlayerCompare comparator
If there is a player that is both on the JV team and Varsity team of their sex
        If the player is a singles player (the highest rating) on the Varsity team
                Remove the player from the JV team
        If the player plays doubles on varsity team and singles on JV
                If the player is a better doubles player than a singles player
                        Remove the player from the JV team
                If the player is a better singles player than doubles player
                        Remove the player from the Varsity team
        If the player is a doubles player in both teams
                Remove the player from the JV team
Traverse the player lists for all teams (Boys JV, Varsity, Girls JV, Varsity)
        If there is a player playing a high seed of doubles who is a better singles player
than a player playing a low seed of singles
                Move the doubles player to singles and the singles player to doubles
```

This algorithm sorts the players into teams using the PlayerCompare comparator, then modifies these teams to ensure that there is no overlap between the varsity and JV teams, and if there is overlap then it removes the player from one of the teams based on which team the player would have a greater impact on. It then changes the position of players on each team so that player is playing in a position best suited to their skills. Finally, it removes players from the team until the team is of the correct size. The complexity stems from the use of multiple, nested for loops, within which are multiple, nested if commands.