Criterion C

List of simple techniques to code functionality and not included due to word count restrictions
Encapsulation If, else if, else statements For and while loops Data hiding Polymorphism: toString, compareTo OOP Techniques: Constructors, Accessors and Mutators

Technique: Tutor Ranking Algorithm- Moving Average, Parallel Insertion Sort
Link to Success criterion/Criteria: The system should be able to rank the tutors based on the feedback received from the tutees and allow the client to influence the ranking
Source: **Student original idea and algorithm** Insertion Sort: (Drien Vargas, Marcos)

Justification

The tutors must be ranked to optimally determine what tutor should a tutee pair with. This is to allow the client to easily pair the best tutor available with a tutee and prevent the time-consuming method of randomly selecting tutors to pair with tutees.

Explanation

Firstly, ratings (doubles between 0-10) are added to the **LinkedHashMaps subjectUnderstandingRatings, explanationAbilityRatings, attitudeAbilityRatings,** and **progressRatings** using the **addRating()** method. After that, the **rank()** method in the **Ranking** class is called in the **RankingController** class. The **rank** method calls the **calculateOverallRating** method, which calls the calculate **movingAverage** method. These methods together create an overall rating of the tutor by taking a weighted moving average of each of the individual ratings and then taking a weighted average of the 4 ratings. This Map is then placed into two **ArrayLists**, and, using the **insertionSort()** method in parallel, the tutors are sorted according to their overall ratings, ranking these tutors. Insertion sort was chosen because, despite its inefficient time complexity of $O(n^2)$, when the ratings are added to the system, insertion sort is adaptable and efficient and sorting the list with the new rating. Using **Stacks**, this sorting is reversed since the formatting of the **HashMaps** means the ranking is in reverse. In the **RankingController**, these rankings are then printed.

Example

```
private static double movingAverage(double subjectRating, double newRating) {
    double alpha = 0.1; // alpha - indicates the weight of the newest piece of data, with older pieces of data getting multiplied with the ratio 1-alpha
    if (subjectRating <= 0) {
        subjectRating = newRating;
    } else {
        subjectRating = (alpha * newRating) + ((1-alpha) * subjectRating);
    }
    return subjectRating;
}
```

```java
//calculates the overall rating of a certain attribute using the list of doubles provided
4 usages  Savit10 *
public static LinkedHashMap<Tutor, Double> calculateOverallRating(LinkedHashMap<Tutor, ArrayList<Double>> ratings) {
    LinkedHashMap<Tutor, Double> overallRatings = new LinkedHashMap<>();
    double overallRating = 0;
    for (Tutor tutor : ratings.keySet()) { //iterates through hashmap's keys
        ArrayList<Double> ratingsList = ratings.get(tutor);
        for (Double rating : ratingsList) { //iterates through the arraylist
            overallRating = movingAverage(overallRating, rating); // calls moving average method for each new piece of data
        }
        overallRatings.put(tutor, overallRating);
    }
    return overallRatings;
}
```

```java
public void rank() {
    //calculates the overall rating of the list of doubles and instantiates a new HashMap with key: Tutor, Double: overall attribute rating
    LinkedHashMap<Tutor, Double> overallSubjectUnderstanding = calculateOverallRating(this.subjectUnderstandingRatings);
    LinkedHashMap<Tutor, Double> overallExplanationAbility = calculateOverallRating(this.explanationAbilityRatings);
    LinkedHashMap<Tutor, Double> overallTutoringAttitude = calculateOverallRating(this.attitudeAbilityRatings);
    LinkedHashMap<Tutor, Double> overallTuteeProgress = calculateOverallRating(this.progressRatings);
    //creates a Hashmap of Maps with all the ratings to add to
    List<LinkedHashMap<Tutor, Double>> listOfMaps = new ArrayList<>();
    listOfMaps.add(overallSubjectUnderstanding);
    listOfMaps.add(overallExplanationAbility);
    listOfMaps.add(overallTutoringAttitude);
    listOfMaps.add(overallTuteeProgress);
    //LinkedHashMap to map all the overall ratings from each category of a tutor in each category to the tutor
    LinkedHashMap<Tutor, ArrayList<Double>> tutorOverallRatings = new LinkedHashMap<>();
    // list of tutors to map to
    ArrayList<Tutor> tutorArrayList = new ArrayList<>(overallSubjectUnderstanding.keySet());
    for (Tutor tutor: tutorArrayList) { // iterates through the tutors list
        ArrayList<Double> ratings = new ArrayList<>();
        for (LinkedHashMap<Tutor, Double> currentHashMap : listOfMaps) { // iterates through the List of Hashmaps to get the rating of each tutor in one category
            ratings.add(currentHashMap.get(tutor));
        }
        tutorOverallRatings.put(tutor, ratings);
    }
```

```java
//LinkedHashMap mapping tutor to their overall rating
LinkedHashMap<Tutor, Double> tutorOverallRating = new LinkedHashMap<>();
double overallTutorRating = 0;
for (Tutor tutor: tutorOverallRatings.keySet()) {
    overallTutorRating = 0.4*tutorOverallRatings.get(tutor).get(0) + 0.3*tutorOverallRatings.get(tutor).get(1) + 0.2*tutorOverallRatings.get(tutor).get(2) + 0.1*tutorOverallRatings.get(tutor).get(3);
    tutorOverallRating.put(tutor, overallTutorRating);
}
// list of tutors
ArrayList<Tutor> tutors = new ArrayList<>(tutorOverallRating.keySet());
//list of their overall ratings to add to
ArrayList<Double> doubles = new ArrayList<>();
for (Tutor tutor: tutors) {
    doubles.add(tutorOverallRating.get(tutor));
}
```

```java
//parallel sorts tutors with their ratings using insertion sort
for(int i = 1; i < doubles.size(); i++)
{
    double temp = doubles.get(i);
    Tutor tempTutor = tutors.get(i);
    int j = i-1;
    while(j >= 0  && (temp < doubles.get(j)))
    {
        doubles.set(j+1, doubles.get(j));
        tutors.set(j+1, tutors.get(j));
        j--;
    }
    doubles.set(j+1, temp);
    tutors.set(j+1, tempTutor);
}
```

```java
//reversing list using stacks
Stack <Double> doubleStack = new Stack<>()
Stack <Tutor> tutorStack = new Stack<>();
for (Double aDouble : doubles) {
    doubleStack.push(aDouble);
}
for(int i = 0; i < doubles.size(); i++) {
    doubles.set(i, doubleStack.pop());
}
for (Tutor tutor : tutors) {
    tutorStack.push(tutor);
}
for(int i = 0; i < tutors.size(); i++) {
    tutors.set(i, tutorStack.pop());
}
```

Technique: Data Structures - LinkedHashMaps, HashMaps, ArrayLists, Stacks
Link to success criteria: the client must be able to enter, edit, and delete the tutors' and tutees' names, grade levels, subjects taught/wanting to learn and session availabilities
Source: GeeksForGeeks. "HashMap in Java with Examples." GeeksforGeeks, 28 Apr. 2017, www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/. GeeksForGeeks. "ArrayList in Java." GeeksforGeeks, 6 Oct. 2016, www.geeksforgeeks.org/arraylist-in-java/. Oracle. "LinkedHashMap (Java Platform SE 8)." Docs.oracle.com, docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html.

Justification

By using **HashMaps** and **Arraylists**, the system was not restricted to having a fixed number of tutees, tutors, sessions, subjects etc, which it would've been if the system implemented static data structures like **arrays**. This allowed the client to enter, edit and delete from the various classes, meeting the success criteria. **LinkedHashMaps** were used to not only store the ratings of the tutors for ranking purposes but to preserve the order of the ratings to do the rankings and save the order in which the tutor and ratings were paired. **Stacks** were used to reverse the order of the **ArrayLists** of ratings and tutors since the **insertion sort** provided a ranking in reverse.

Explanation

In classes **Tutor** and **Tutee** attributes used **HashMaps** and **Arraylists** to store **Subjects** and **Sessions.** This allowed for dynamic access to the various attributes of Objects in the system. Furthermore, **LinkedHashMaps** were used to store tutors and their mapped ratings in a preserved order, allowing ranking calculations to occur. **Stacks** were used to reversing the parallel-sorted **ArrayList** tutors and double, generating tutor rankings.

Example

```
public class Tutee extends Person implements Serializable{
    // attributes
    3 usages
    private int gradeLevel;
    9 usages
    private ArrayList<Subject> subjectsLearning;
    5 usages
    private HashMap<String, Boolean> sessionsAvailable;
    3 usages
    private Tutor tutor;
```

```
//reversing list using stacks
Stack <Double> doubleStack = new Stack<>()
Stack <Tutor> tutorStack = new Stack<>();
for (Double aDouble : doubles) {
    doubleStack.push(aDouble);
}
for(int i = 0; i < doubles.size(); i++) {
    doubles.set(i, doubleStack.pop());
}
for (Tutor tutor : tutors) {
    tutorStack.push(tutor);
}
for(int i = 0; i < tutors.size(); i++) {
    tutors.set(i, tutorStack.pop());
}
```

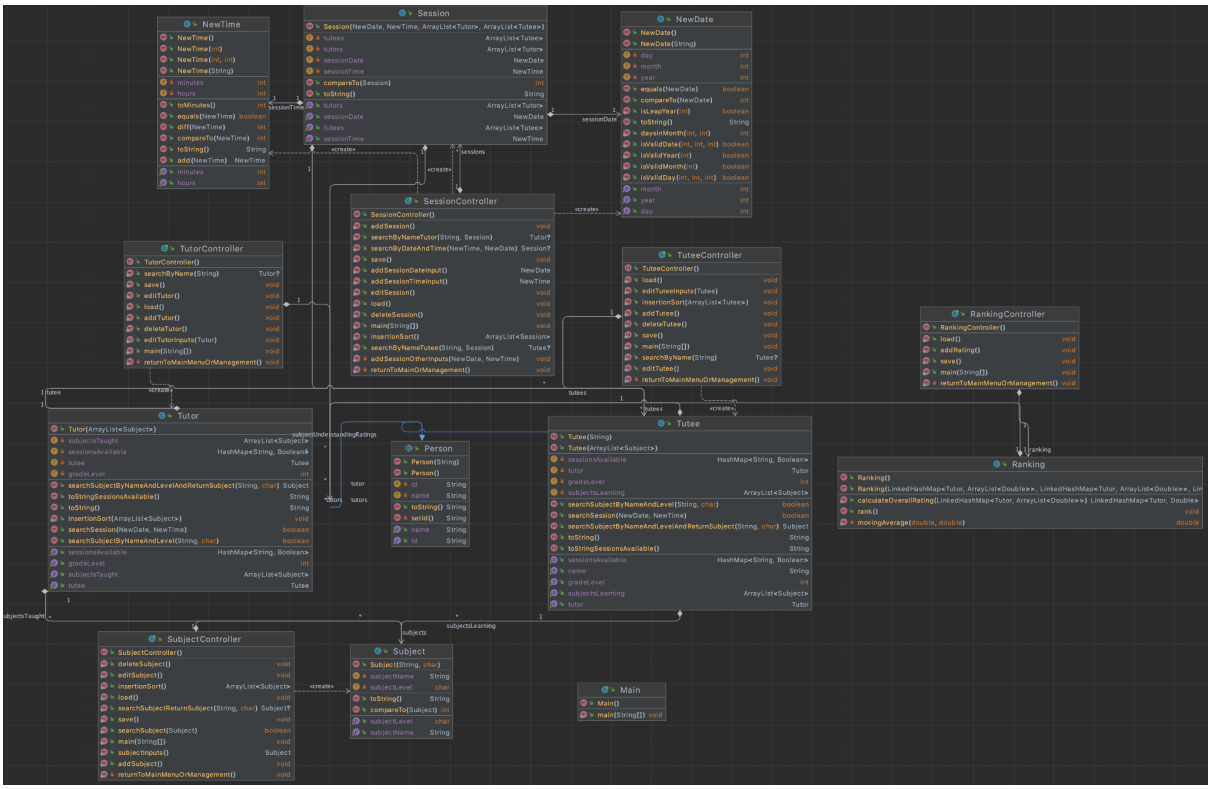
```
public void rank() {
    //calculates the overall rating of the list of doubles and instantiates a new HashMap with key: Tutor, Double: overall attribute rating
    LinkedHashMap<Tutor, Double> overallSubjectUnderstanding = calculateOverallRating(this.subjectUnderstandingRatings);
    LinkedHashMap<Tutor, Double> overallExplanationAbility = calculateOverallRating(this.explanationAbilityRatings);
    LinkedHashMap<Tutor, Double> overallTutoringAttitude = calculateOverallRating(this.attitudeAbilityRatings);
    LinkedHashMap<Tutor, Double> overallTuteeProgress = calculateOverallRating(this.progressRatings);
    //creates a Hashmap of Maps with all the ratings to add to
    List<LinkedHashMap<Tutor, Double>> listOfMaps = new ArrayList<>();
    listOfMaps.add(overallSubjectUnderstanding);
    listOfMaps.add(overallExplanationAbility);
    listOfMaps.add(overallTutoringAttitude);
    listOfMaps.add(overallTuteeProgress);
    //LinkedHashMap to map all the overall ratings from each category of a tutor in each category to the tutor
    LinkedHashMap<Tutor, ArrayList<Double>> tutorOverallRatings = new LinkedHashMap<>();
    // list of tutors to map to
    ArrayList<Tutor> tutorArrayList = new ArrayList<>(overallSubjectUnderstanding.keySet());
    for (Tutor tutor: tutorArrayList) { // iterates through the tutors list
        ArrayList<Double> ratings = new ArrayList<>();
        for (LinkedHashMap<Tutor, Double> currentHashMap : listOfMaps) { // iterates through the List of Hashmaps to get the rating of each tutor in one category
            ratings.add(currentHashMap.get(tutor));
        }
        tutorOverallRatings.put(tutor, ratings);
    }
```

Technique: UML Relationships - Dependency, Association, Aggregation
Link to success criteria: the client must be able to enter, edit, and delete the tutors' and tutees' names, grade levels, subjects taught/wanting to learn and session availabilities
Source: (Drien Vargas, Marcos)

Justification: These techniques are essential to the product as it is the backbone of how the different stakeholders and classes interact and interface with each other to build the full system, allowing the client to use the system. Furthermore, all the data changes made by the client have many effects on other classes in the system as well, highlighting the interdependency between the classes.

Explanation: Many attributes and methods have been called in different classes and contain objects of other classes, creating all the different types of dependencies mentioned above between those classes. For example, both the **Tutee** and **Tutor** have an attribute as each other, highlighting the aggregation between the classes to create the mutual pairing between the two objects. Furthermore, a **Session** has an **ArrayList<Tutee>** and **ArrayList<Tutor>**, highlighting the different tutors and Tutees in a session. These relationships and more can be seen in the final UML class diagram below.

Example:



```java
public class Session implements Serializable {

    4 usages
    private NewDate sessionDate;
    4 usages
    private NewTime sessionTime;
    4 usages
    private ArrayList <Tutor> tutors;
    4 usages
    private ArrayList <Tutee> tutees;
```

Technique: File Management - Serialization and Deserialization
Link to success criteria: The system should automatically save the data entered by the client to prevent the data not getting deleted every time the system is shut down
Source *GeeksForGeeks. "Serialization and Deserialization in Java with Example - GeeksforGeeks." GeeksforGeeks, 22 Feb. 2019, www.geeksforgeeks.org/serialization-in-java/.*

Justification

By using this technique, the client can close and reopen the application without worrying about whether the data is stored in the systems. All tutees, tutors, subjects etc will be stored in the system automatically, increasing usability and convenience for the client.

Explanation

The classes **Ranking, Tutor, Tutee, Subject, and Session** implement the interface Serializable. The classes **RankingController, TutorController, TuteeController, SubjectController, SessionController** each have a **save()** method to write an **ArrayList()** of all of the classes objects to a .txt file ie. **tutors, tutees, sessions, subjects** etc. Furthermore, these classes each also have a **load()** to read the **ArrayList()** of objects from the file and store it in the controller class attributes.

Example

```
public static void save() throws IOException
{
    System.out.println("Saving changes");
    try {
        FileOutputStream f = new FileOutputStream(new File( pathname: "ranking.txt"));
        ObjectOutputStream o = new ObjectOutputStream(f);
        o.writeObject(ranking);
        o.close();
        f.close();
        System.out.println("Changes saved to file");
    } catch (IOException e) {
        System.out.println("Error initializing stream");
    }
}

1 usage   Savit10
public static void load() throws Exception //
{
    System.out.println("------------------------------");
    ranking = null;
    try {
        FileInputStream fi = new FileInputStream( name: "ranking.txt");
        ObjectInputStream oi = new ObjectInputStream(fi);
        ranking = (Ranking) oi.readObject();
        oi.close();
        fi.close();
        System.out.println("Data Loaded");
    }
    catch (FileNotFoundException e) {
        System.out.println("File not found");
    }
    catch (IOException i) {
        System.out.println("Error initializing the data stream");
    }
    catch (ClassNotFoundException c) {
        c.printStackTrace();
    }
}
```

Technique: Error Validation - Exception Handling using Try-Catch Blocks
Link to success criteria: The system should prevent the input of data that fails the validation checks, is entered in an incorrect format, or is entered incorrectly, and handle these errors effectively.
Source: *Anderson, Julie, and Hervé Franceschi. Java Illuminated an Active Learning Approach. Burlington, Ma Jones & Bartlett Learning, 2019.*

Justification: This validation will prevent any abrupt breaks in the system, causing inconvenience to the client and prevent system stoppage.

Explanation: Certain methods throw Exceptions so **Try-Catch** blocks are effective at not only catching but handling these errors. It also allows the system to try completing the code inside the try block instead of immediately trying to catch the exception.

Example:

```
public static void save() throws IOException
{
    System.out.println("Saving changes");
    try {
        FileOutputStream f = new FileOutputStream(new File( pathname: "ranking.txt"));
        ObjectOutputStream o = new ObjectOutputStream(f);
        o.writeObject(ranking);
        o.close();
        f.close();
        System.out.println("Changes saved to file");
    } catch (IOException e) {
        System.out.println("Error initializing stream");
    }
}

1 usage  Savit10
public static void load() throws Exception //
{
    System.out.println("------------------------------");
    ranking = null;
    try {
        FileInputStream fi = new FileInputStream( name: "ranking.txt");
        ObjectInputStream oi = new ObjectInputStream(fi);
        ranking = (Ranking) oi.readObject();
        oi.close();
        fi.close();
        System.out.println("Data Loaded");
    }
    catch (FileNotFoundException e) {
        System.out.println("File not found");
    }
    catch (IOException i) {
        System.out.println("Error initializing the data stream");
    }
    catch (ClassNotFoundException c) {
        c.printStackTrace();
    }
}
```

Technique: Inheritance with Abstract Class
Link to success criteria: The client should not have to instantiate the ID of any person in the system
Source: *Oracle. "Abstract Methods and Classes (the Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance)." Oracle.com, 2019, docs.oracle.com/javase/tutorial/java/IandI/abstract.html.*

Justification

To make the code more modular and easier to use, **Person** (superclass) is **abstract** because it shouldn't be instantiated. This is because the **ID** of a person is a unique identifier and the client shouldn't need to remember all the **ID**s of corresponding people in the system.

Explanation

The Person template is abstract. Both **Tutor** and **Tutee** inherit the attributes **ID** and **Name** from the **Person** class. Furthermore, the **Person** class auto generates an ID for a **Tutor** and **Tutee** when instantiated, simplifying the client's work.

Example

```
public abstract class Person implements Serializable //superclass

{
    //attributes
    2 usages
    private String id; // unique id to each person
    2 usages
    private String name;
    4 usages
    public static List<String> idNumbers = new ArrayList<>(); //static arraylist containing the ids of all people in the system
    4 usages
    private static int lastId; // last ID in the idNumbers list
```

```
public class Tutor extends Person implements Serializable {
```

```
public class Tutee extends Person implements Serializable{
```

Technique: Menus/Menu Driven interface
Link to success criteria: - The client must be able to enter, edit, and delete the tutors' and tutees' names, grade levels, subjects taught/wanting to learn and session availabilities
Source: (Drien Vargas, Marcos)

Justification

To enter data into the system to create **Tutor, Tutee, Subject** objects etc, there must be multiple options to enter different types of objects. To do this, menu driven interfaces were used throughout the system to allow the client to complete many different tasks.

Explanation

The Main class outputs a list of options and the user inputs an integer that is read by the system and accordingly takes you to a controller class for entering/editing/deleting objects.

Example

```
do
{
    int input = 0;
    boolean inputMisMatch = true;
    while(inputMisMatch) {
        System.out.println("Press [1] to Manage Tutors");
        System.out.println("Press [2] to Manage Tutees");
        System.out.println("Press [3] to Manage Peer Tutoring Sessions");
        System.out.println("Press [4] to Manage Tutor Rankings");
        System.out.println("Press [5] to Manage Subjects");
        System.out.println("Press [6] to Quit Application");
        try {
            input = Integer.parseInt(sc.next());
            if (input != 1 && input != 2 && input != 3 && input != 4 && input != 5 && input != 6) {
                System.out.println("Incorrect input.");
            }
            else {
                inputMisMatch = false;
            }
        } catch (Exception e) {
                System.out.println("Incorrect input format");
            }
        }
    switch (input) {
        case 1:
            TutorController.main( args: null);
            break; // calling main method of tutor controller
        case 2:
            TuteeController.main( args: null);
            break;// calling main method of tutee controller
        case 3:
            SessionController.main( args: null);
            break;// calling main method of session controller
        case 4:
            RankingController.main( args: null);
            break;// calling main method of ranking controller
        case 5:
            SubjectController.main( args: null);
            break;// calling main method of subject controller
        case 6:
            continues = true;
            break;// quitting application
        default:
            System.out.println("Incorrect input");
            continues = false;
            break;
    }
}
```

Technique: Searching Algorithms
Link to success criteria: The client should be able to obtain the information of any tutor and tutee by searching the system using their name
Source: *Gayle Laakmann Mcdowell. Cracking the Coding Interview : 189 Programming Questions and Solutions. Palo Alto, Ca, Careercup, Llc, 2019.*

Justification: To obtain a tutee/tutor and check whether it exists, the **ArrayList()** of tutees/tutors must be searched. This allows the user to search for a tutor/tutee using the information they have (name). The binary search was chosen since it is efficient with the time complexity of O(log n). The linear sequential search was also used.

Explanation: Both **Tutor** and **Tutee** classes have **searchByName()** methods. Depending on the situation, a binary or sequential search was used.

Example:

```
23 usages   Savit10 +1 *
public static Tutor searchByName(String name){ // linear search by name since name is not unique
    for (int i = 0; i < tutors.size(); i++) {
        if (tutors.get(i).getName().equals(name)) {
            return tutors.get(i);
        }
    }
    return null;
}
```

```
public static Session searchByDateAndTime (NewTime time, NewDate date) {
    insertionSort();
    int low = 0;
    int high = sessions.size() - 1;
    while (low <= high) {
        int mid = (high+low)/2;
        if (sessions.get(mid).getSessionDate().equals(date) && sessions.get(mid).getSessionTime().equals(time)) {
            return sessions.get(mid);
        }
        else if (sessions.get(mid).getSessionTime().compareTo(time)< sessions.get(mid).getSessionDate().compareTo(date)) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return null;
}
```

Word Count: 971