











CRITERION C: DEVELOPMENT

| Essay |
|---|
| <div>- name: String</div> <div>- grade: int</div> <div>- numOfParagraphs: int</div> |
| <div>- StoreEssay(essay: String, essayName: String): void</div> <div>+ StoreEssay(essay: String, essayName: String, essayGrade: String, essaysLoadIF: JInternalFrame, numOfParagraphs: int): void</div> <div>+ setName(): void</div> <div>+ setGrade(): void</div> <div>+ setNumOfParagraphs(): void</div> <div>+ SaveEssayNames(): void</div> <div>+ SaveEssayGrades(): void</div> <div>+ SaveEssayNumOfParagraphs(): void</div> <div>+ getName(): String</div> <div>+ getGrade(): int</div> <div>+ getNumOfParagraphs(): int</div> <div>+ AddStoredToEssays(): void</div> <div>+ RemoveEssay(): void</div> <div>+ getIndexOfEssay(): int</div> |

The program contains an *Essay* class. Each time a essay is loaded into the program, an object of the *Essay* class is instantiated. There is one *JFrame*, containing several internal frames. These internal frames contain further components (buttons, textboxes, labels etc). At one point only one internal frame must be visible.

| | |
|---|---|
|  | [JFrame] |
|  | startingWindowIFrame [JInternalFrame] |
|  | viewWindowIFrame [JInternalFrame] |
|  | statisticsWindowIFrame [JInternalFrame] |
|  | essaysLoadIFrame [JInternalFrame] |
|  | wordVarietyIFrame [JInternalFrame] |
|  | wordCountIFrame [JInternalFrame] |
|  | mostCommonWordsIFrame [JInternalFrame] |
|  | paragraphIFrame [JInternalFrame] |
|  | quotesIFrame [JInternalFrame] |
|  | firstPersonIFrame [JInternalFrame] |

PROGRAM PERSISTENCE

The *storeEssay(String, String)* function is key to program persistence.

The function takes the entire essay as a string parameter, as well as essay name as another parameter.

Next, a new object of the *StringTokenizer* is created, called *wordSeparator*. *wordSeparator* can separate the *text* String into separate Strings, with a space separating the Strings from each other. The function removes punctuation and makes all characters in the String lower case, using a loop. This is necessary for the word stemming algorithm to work. In the same loop, the word is stemmed using a method from the *EnglishSnowballStemmerFactory* class. (Porter 1980)

Word stemming is changing a word to its unconjugated form. For example, the word “crawling” would be stemmed to “crawl”. This is essential in calculating word variety, as otherwise “crawling” and “crawl” used in the same essay would be treated as two different words, when in reality they are simply conjugated.

Finally, the stemmed word is added into the *essayWords* ArrayList.

Once there is a ArrayList of separated, stemmed words, the algorithm stores these words in a separate, unique .txt file for each essay, using the *PrintWriter* and *FileWriter* classes: Each *WordStorageFile* is unique as *essayName* is used in naming of the file. Since the program only allows to load essays with a name that has not earlier been used, the file name is unique.

The same is done to form a *OriginalStorageFile*, which instead of storing the stemmed words, stores the original String essay as a whole.

Both of these storage files are key in forming statistics.

```

private void storeEssay(String text, String essayName)
{
    ArrayList<String> essayWords = new ArrayList<String>();
    StringTokenizer wordSeparator = new StringTokenizer(text, " ");
    while (wordSeparator.hasMoreTokens()) {
        String Word = wordSeparator.nextToken();
        String noPunctuationInWord = Word.replaceAll("[^a-zA-Z ]", "");
        String wordToLowerCase = noPunctuationInWord.toLowerCase();
        try {
            String stemmedWord = EnglishSnowballStemmerFactory.getInstance().process(wordToLowerCase);
            essayWords.add(stemmedWord);
        } catch (StemmerException ex) {}
    }
    try {
        PrintWriter writer = new PrintWriter(new FileWriter(essayName+"WordStorageFile.txt"));
        for (int k = 0; k < essayWords.size(); k++)
        {
            writer.println(essayWords.get(k));
        }
        writer.close();
    } catch (IOException ex) {}
    try {
        PrintWriter writer = new PrintWriter(new FileWriter(essayName+"OriginalStorageFile.txt"));
        writer.println(text);
        writer.close();
    } catch (IOException ex) {}
}


```

The program contains an ArrayList called essays. Each time a new Essay is loaded, a object is added to the ArrayList.

```
public static ArrayList<Essay> essays = new ArrayList<Essay>();
```


Each essay contains a name, grade and number of paragraphs. If the program contains this information, it generates all statistics, as it can access each word or original storage file by using essay name.

The original problem was that when the user closes the program, all of the data is lost from the ArrayList. The solution to this problem was saving all information in separate .txt files each time the program is closed.



EssayGradesStorageFile.txt
 EssayNamesStorageFile.txt
 EssayParagraphStorageFile.txt

Each .txt file would have the next item in the ArrayList stored in the next line, so that when the program is opened all data can be fed back into the ArrayList, in the same order as before.

 EssayNamesStorageFile.txt — Notatnik

Plik Edycja Format Widok Pomoc

test cs ia document.docx

sample word file.docx

Extended Essay.docx

Communist Speech.docx

Another File.docx

This was done using methods similar to the one below for all 3 class variables for saving data and one method to feed data back into ArrayLists:

```
public void saveEssayNames() throws IOException
{
    File file;
    if (essays.size() > 0)
    {
        PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter("EssayNamesStorageFile.txt")));
        String tmp = "";
        for (int er = 0; er < essays.size(); er++)
        {
            tmp = essays.get(er).name;
            writer.println(tmp);
            writer.flush();
        }
        writer.close();
    }
    else
    {
        file = new File("EssayNamesStorageFile.txt");
        file.delete();
        file.createNewFile();
    }
}
```

```

public void addStoredToEssays() throws FileNotFoundException, IOException
{
    String tmp;
    ArrayList<Integer> tmpEssayGrades = new ArrayList<Integer>();
    ArrayList<String> tmpEssayNames = new ArrayList<String>();
    ArrayList<Integer> tmpNumsOfParagraphs = new ArrayList<Integer>();
    BufferedReader reader = new BufferedReader(new FileReader("EssayGradesStorageFile.txt"));
    int intTmp = 0;
    while ((tmp = reader.readLine()) != null){
        intTmp = Integer.parseInt(tmp);
        tmpEssayGrades.add(intTmp);
    }
    reader.close();
    reader = new BufferedReader(new FileReader("EssayNamesStorageFile.txt"));
    while ((tmp = reader.readLine()) != null){
        tmpEssayNames.add(tmp);
    }
    reader.close();
    reader = new BufferedReader(new FileReader("EssayParagraphStorageFile.txt"));
    intTmp = 0;
    while ((tmp = reader.readLine()) != null){
        intTmp = Integer.parseInt(tmp);
        tmpNumsOfParagraphs.add(intTmp);
    }
    reader.close();
    for (int i = 0; i < tmpEssayNames.size(); i++){
        Essay essay = new Essay();
        essay.name = tmpEssayNames.get(i);
        essay.grade = tmpEssayGrades.get(i);
        essay.numOfParagraphs = tmpNumsOfParagraphs.get(i);
        essays.add(essay);
    }
}

```

FORMATION OF STATISTICS

The inclusion of an abstract parent class reduces length of the code in each of the 4 child inheriting classes, as well as makes it easier for future developers to add more statistics into the program. It also ensures the *getStatistic* method is overridden in a new added statistic, in case a developer forgets to override it.

Abstract Statistic class

| Statistic |
|---|
| <ul style="list-style-type: none">- barChartTitle: String- panel: JPanel- xAxisLabel: String- yAxisLabel: String |
| <ul style="list-style-type: none">+ getAverage(grade: int): int+ getStatistic(name: String): int+ setPanel(panel: JPanel): void+ setBarChartTitle(barChartTitle: String): void+ setXAxisLabel(label: String): void+ setYAxisLabel(label: String): void+ showChart(): void |

```
public void showChart()
{
    DefaultCategoryDataset barChartData = new DefaultCategoryDataset();
    for (int ii = 0; ii < 8; ii++)
        try
        {
            barChartData.setValue(getAverage(ii), "", Integer.toString(ii));
            JFreeChart chart = ChartFactory.createBarChart(barChartTitle, xAxisLabel, yAxisLabel, barChartData, PlotOrientation.VERTICAL, false, true, false);
            CategoryPlot plotter = chart.getCategoryPlot();
            plotter.setRangeGridlinePaint(Color.BLUE);
            ChartPanel barPanel = new ChartPanel(chart);
            panel.removeAll();
            panel.add(barPanel, BorderLayout.CENTER);
            panel.validate();
        }
        catch (IOException ex) {
        }
    }
}
```

```

public int getAverage(int grade) throws IOException
{
    int averageNumberForGrade = 0;
    if ((grade >= 1) || (grade <= 7))
    {
        averageNumberForGrade = 0;
        ArrayList<Integer> numsOfNumbersPerGrade = new ArrayList<Integer>();
        for (int yy = 0; yy < essays.size(); yy++)
        {
            if (grade == essays.get(yy).getGrade())
            {
                int intTmp = getStatistic(essays.get(yy).getName());
                numsOfNumbersPerGrade.add(intTmp);
            }
        }
        for (int tt = 0; tt < numsOfNumbersPerGrade.size(); tt++)
        {
            averageNumberForGrade += numsOfNumbersPerGrade.get(tt);
        }
        try
        {
            averageNumberForGrade = averageNumberForGrade / numsOfNumbersPerGrade.size();
        }
        catch (Exception ex)
        {
            averageNumberForGrade = 0;
        }
    }
    return averageNumberForGrade;
}

public abstract int getStatistic(String name);

```

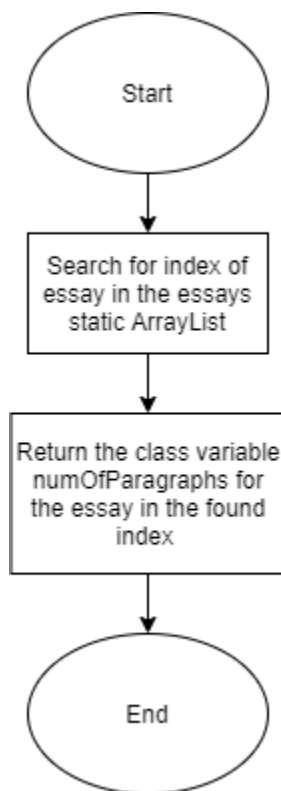
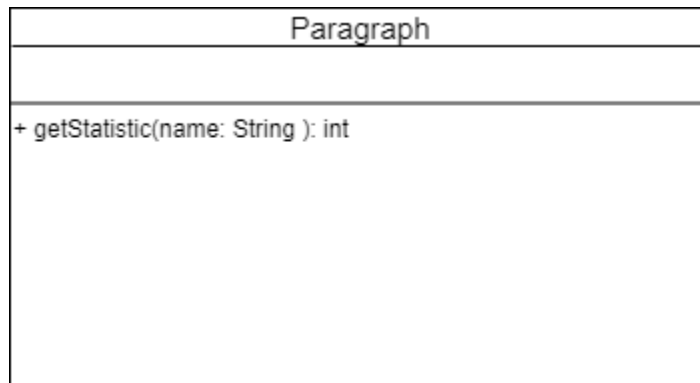
The 4 statistics inherit the *getAverage*, *showChart* and *getStatistic* methods.

The *getAverage* method uses the abstract *getStatistic* method to calculate an average statistic for a given grade. The fact that the *getStatistic* method is abstract, means it will be overridden in each of the child classes, meaning each of these child classes will be able to have their unique way of calculating statistics by overriding the *getStatistic* method.

The *showChart* method loops the *getAverage* method through all possible IB grades and plots a bar chart accordingly using the *JFreeChart* class.

The next section will explain how each statistic is calculated in child classes of the Statistics class.

Paragraph class



First, number of paragraphs is calculated by the *XWPFDocument* class when the essay is first loaded. Next, it is stored as a class variable in *Essay*.

```
List<XWPFPParagraph> paragraphList = docx.getParagraphs();  
numOfParagraphs = paragraphList.size();
```



```

public int getStatistic(String name)
{
    Essay essay = new Essay();
    int indexOfNumber = essay.getIndexOfEssay(name);
    int numOfParagraphs = essays.get(indexOfNumber).getNumOfParagraphs();

    return numOfParagraphs;
}

```

The *getStatistic* method in the *paragraph* class searches for the essay in the *essays* static ArrayList and returns the essay's *numOfParagraphs*.

```

public int getIndexOfEssay(String essayName)
{
    int indexOfEssay = 0;
    for (int i = 0; i < essays.size(); i++)
        if (essays.get(i).name.equals(essayName))
            indexOfEssay = i;
    return indexOfEssay;
}

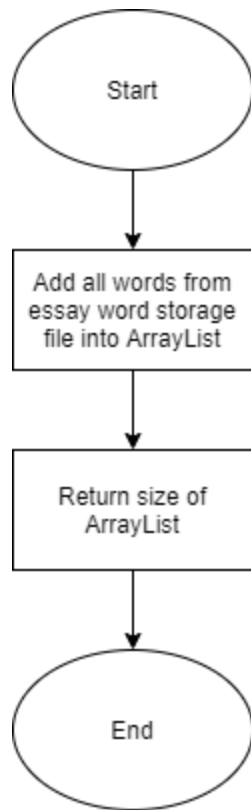
```

The essay is found in the ArrayList with the use of *getIndexOfEssay*, present in *Essay*.

WordCount class

| WordCount |
|------------------------------------|
| |
| + getStatistic(name: String): int |

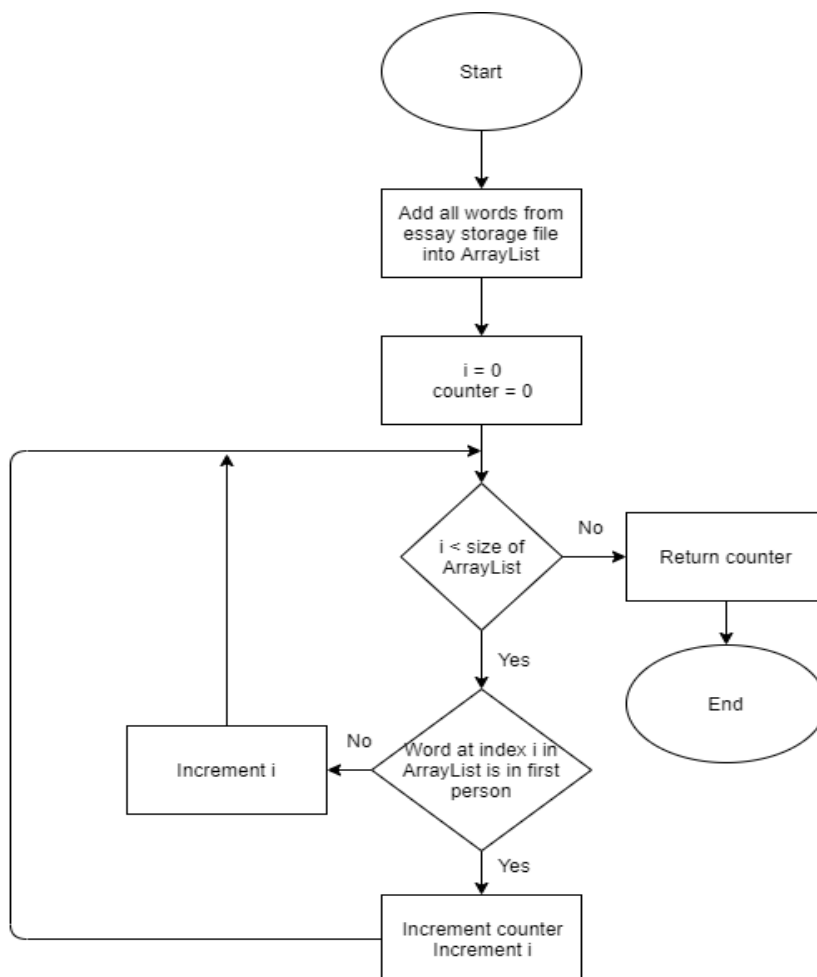
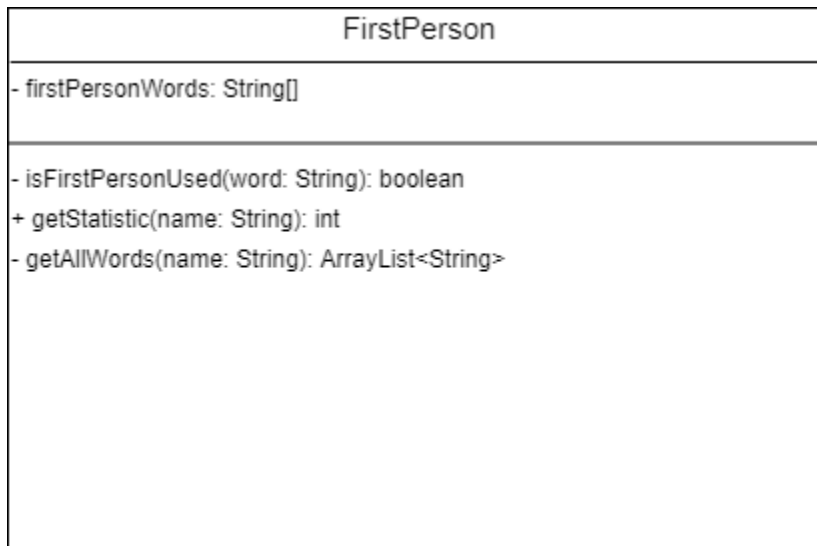
The *nameWordStorageFile.txt* is a .txt file with all the words in the essay with name *name*, separated into separate lines of the document.



```
@Override
public int getStatistic(String name)
{
    String tmp;
    ArrayList<String> essayWords = new ArrayList<String>();
    BufferedReader essayStoreReader;
    try {
        essayStoreReader = new BufferedReader(new FileReader(name+"WordStorageFile.txt"));
        while ((tmp = essayStoreReader.readLine()) != null)
        {
            essayWords.add(tmp);
        }
        essayStoreReader.close();
    } catch (IOException ex) {
    }
    return essayWords.size();
}
```

The `getStatistic` method in the `WordCount` class adds each of the lines of the `nameWordStorageFile` into the `ArrayList` `essayWords`. Finally, the method returns the size of `essayWords`, which is the number of words in the essay.

FirstPerson



```

private ArrayList<String> getAllWords(String name) throws FileNotFoundException
{
    ArrayList<String> essayWords;
    String tmp;
    essayWords = new ArrayList<String>();
    BufferedReader essayStoreReader = new BufferedReader(new FileReader(name+"WordStorageFile.txt"));
    try {
        while ((tmp = essayStoreReader.readLine()) != null)
        {
            essayWords.add(tmp);
        }
        essayStoreReader.close();
    } catch (IOException ex) {
    }
    return essayWords;
}

```

To calculate the number of times first person is used, the *FirstPerson* class first uses *getAllWords* to read the essay, returning an ArrayList containing all individual words in the essay with name *name*.

```

@Override
public int getStatistic(String name)
{
    ArrayList<String> essayWords = new ArrayList<String>();
    try {
        essayWords = getAllWords(name);
    } catch (FileNotFoundException ex) {
    }
    int numberOfFirstPersonUsed = 0;
    for (int i = 0; i < essayWords.size(); i++) {
        String tmp = (String) essayWords.get(i);
        if (isFirstPersonUsed(tmp))
            numberOfFirstPersonUsed++;
    }
    return numberOfFirstPersonUsed;
}

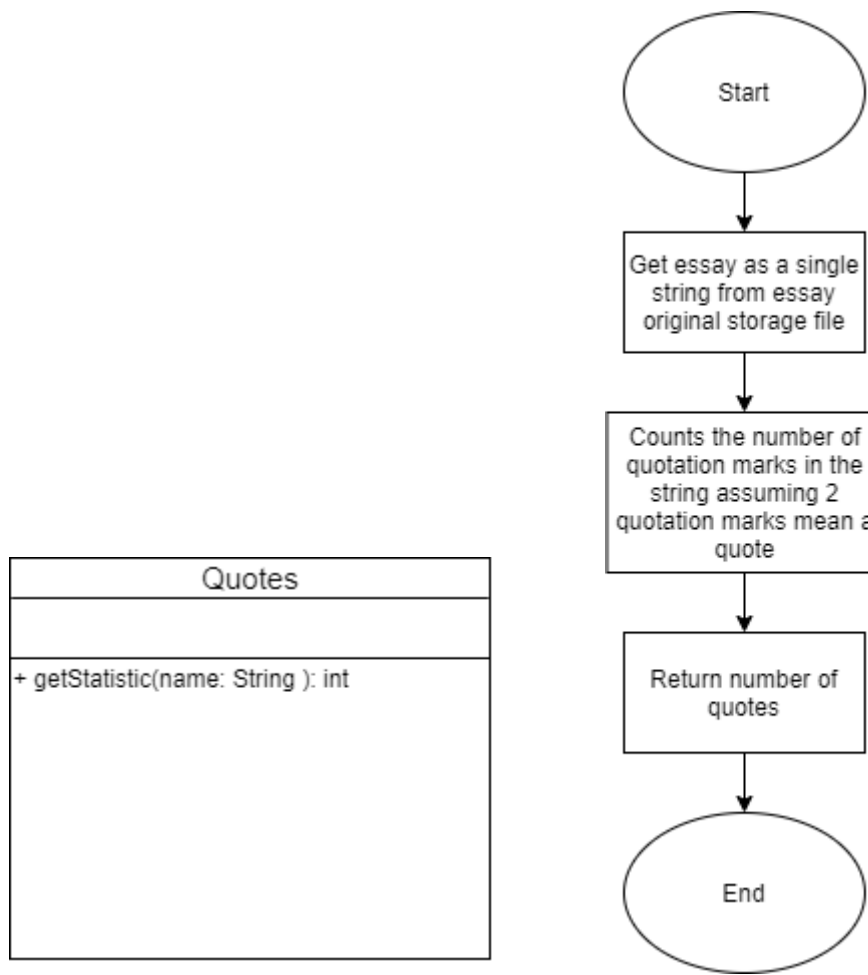
private boolean isFirstPersonUsed(String word)
{
    for (int i = 0; i < firstPersonWords.length; i++) {
        if (word.equals(firstPersonWords[i])) {
            return true;
        }
    }
    return false;
}

```

Next, the class uses *isFirstPersonUsed* in the *getStatistic* method to search through the returned *ArrayList*, increasing the *numberOfFirstPersonUsed* variable whenever the *String* in the *essayWords* *ArrayList* is equal to any of the elements of the *firstPersonWords* array, which is a class variable.

```
private final String[] firstPersonWords = {"i", "my", "me", "we", "us", "our", };
```

Quotes



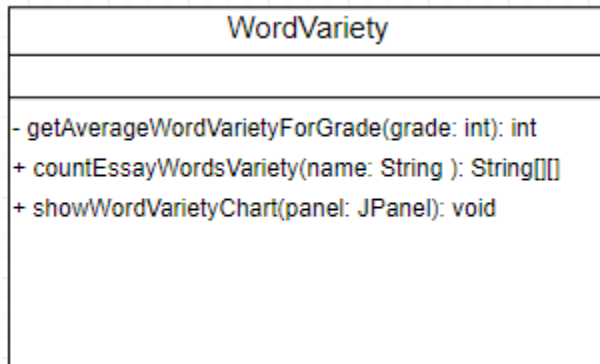
```

@Override
public int getStatistic(String name)
{
    String tmp;
    char[] essayCharArray;
    int numberOfQuotationMarks = 0;
    int numberOfQuotes = 0;
    //step 1 - loading the essay string from storage .txt files
    BufferedReader essayOriginalStoreReader;
    try {
        essayOriginalStoreReader = new BufferedReader(new FileReader(name + "OriginalStorageFile.txt"));
        String text = essayOriginalStoreReader.readLine();
        while ((tmp = essayOriginalStoreReader.readLine()) != null){
            text += tmp;
        }
        essayOriginalStoreReader.close();
        //step 2 - counts the quotation marks and assumes two quotation marks mean a quote
        essayCharArray = text.toCharArray();
        for (int i = 0; i < essayCharArray.length; i++){
            if ((essayCharArray[i] == '"' ) || (essayCharArray[i] == '\'' ) || (essayCharArray[i] == '`')){
                numberOfQuotationMarks++;
            }
            if (numberOfQuotationMarks == 2){
                numberOfQuotes++;
                numberOfQuotationMarks = 0;
            }
        }
    } catch (IOException ex) {
    }
    //return statement
    return numberOfQuotes;
}

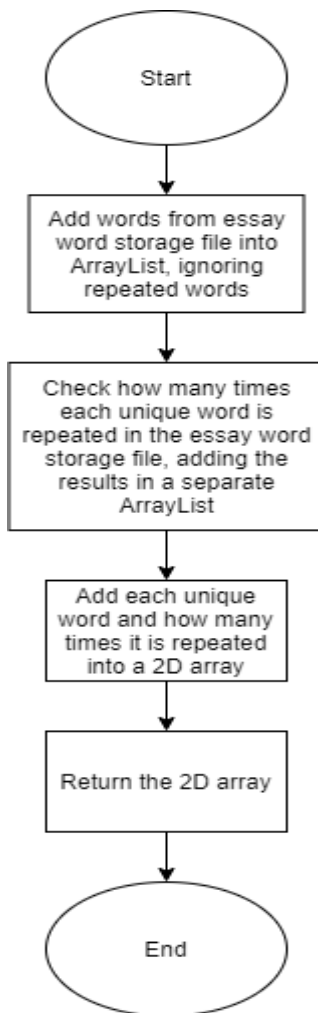
```

In the *Quote* class, the *getStatistic* method first creates a single String *text* containing the entire text of the essay. Next, the method searches through text for quotation marks, incrementing *numberOfQuotationMarks* by one each time one is found. Each time a second quotation mark is found, the *numberOfQuotationMarks* is set to 0 and *numberOfQuotes* is incremented by 1. Finally, *numberOfQuotes* is returned.

WordVariety



The *WordVariety* class does not inherit as its methods are slightly different.



```

public String[][] getEssayWordVariety(String name) throws FileNotFoundException
{
    String[][] uniqueWordsAndCount = {};
    String tmp = "";
    ArrayList<String> uniqueWords = new ArrayList<String>();
    BufferedReader reader = new BufferedReader(new FileReader(name+"WordStorageFile.txt"));
    try{
        while ((tmp = reader.readLine()) != null){
            if (!uniqueWords.contains(tmp))
                uniqueWords.add(tmp);
        }
        reader.close();
        int[] uniqueWordCount = new int[uniqueWords.size()];
        for (int q = 0;q<uniqueWordCount.length;q++){
            uniqueWordCount[q] = 0;
        }
        for (int d = 0;d<uniqueWords.size();d++){
            reader = new BufferedReader(new FileReader(name+"WordStorageFile.txt"));
            String checker = uniqueWords.get(d);
            while ((tmp = reader.readLine()) != null){
                if (checker.equals(tmp))
                    uniqueWordCount[d]++;
            }
            reader.close();
        }
        reader.close();
        uniqueWordsAndCount = new String[uniqueWords.size()][2];
        for (int r = 0;r<uniqueWordCount.length;r++){
            uniqueWordsAndCount[r][0] = uniqueWords.get(r);
            uniqueWordsAndCount[r][1] = Integer.toString(uniqueWordCount[r]);
        }
    }
    catch (IOException ex){
    }
    return uniqueWordsAndCount;
}

```

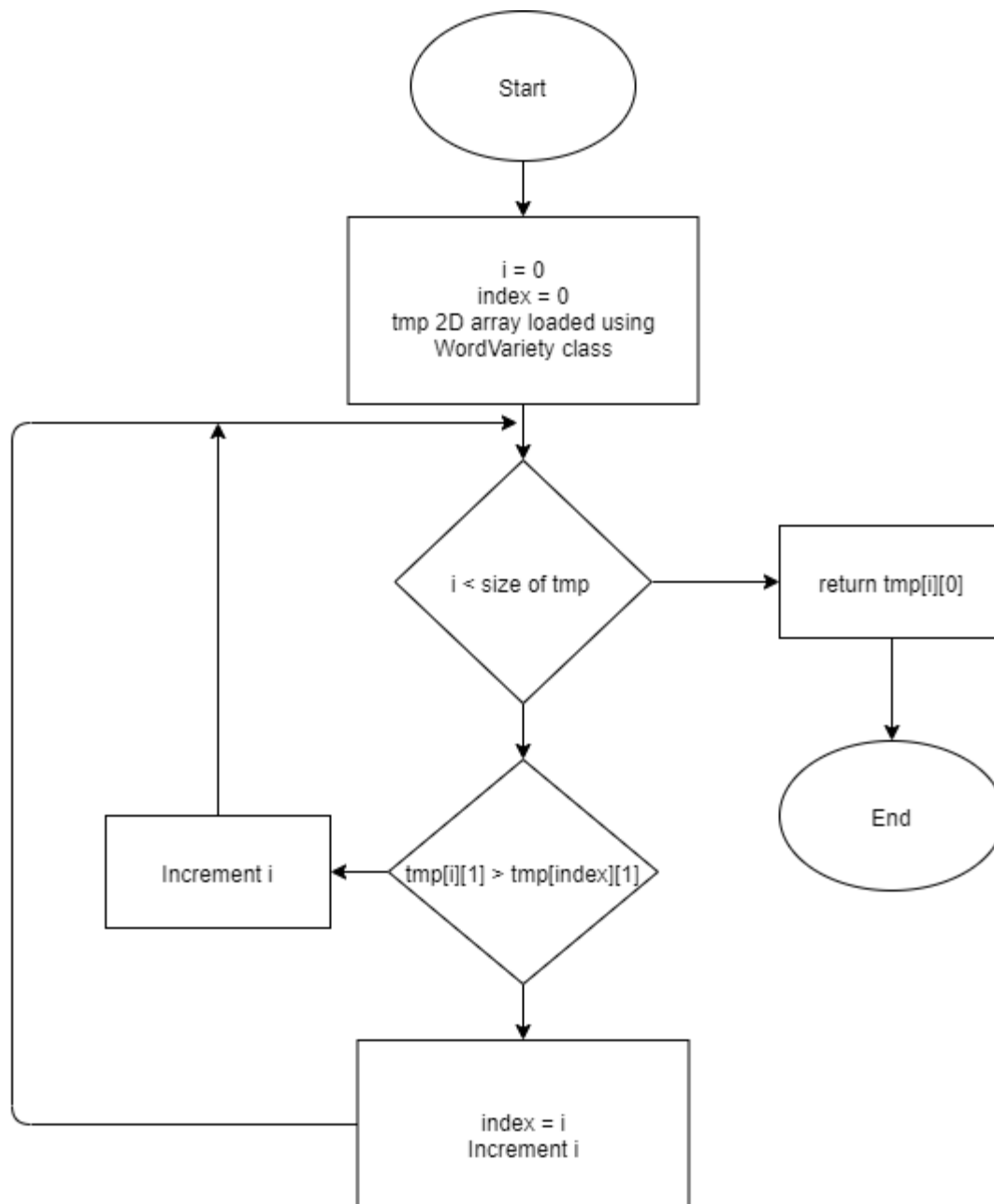
First, the method searches through *nameWordStorageFile.txt*, adding words found to the ArrayList *uniqueWords*. If *uniqueWords* already contains the word to be added, the word will not be added, resulting in an array of unique words. Next, the method searches for how many times each element of *uniqueWords* appears in *nameWordStorageFile.txt* and adds it to the array *uniqueWordCount*. Both of these arrays will be required in the *MostCommonWords* class later, which is why these two arrays are converted into the 2-dimensional array *uniqueWordsAndCount*, which is returned by the method. This is also why the method is *public*.

The *getEssayWordVariety* method code would usually be stored in a *getStatistic* method, however the method returns a 2-dimensional array instead of an integer, which makes inheritance impossible.

Most Common Words

| MostCommonWords |
|--|
| - commonWords: String[] |
| + showMostCommonWordsTable(table: JTable): void - isCommonWord(word: String): boolean |

The algorithm for finding most common words works as follows, with the difference of being repeated for grades 1-7, for all essays within each grade and values in tmp being converted into integers from strings before comparison.



Bibliography

Porter. 1980. "Program." 130-170. <http://snowball.tartarus.org/algorithms/porter/stemmer.html>.