Criterion C: Product Development

Techniques used to create the database

- Searching and Sorting
- Algorithmic Thinking
- Java Console Input/Output
- Dynamic Lists
- Error Control
- File I/O

Searching and Sorting

The relational database uses both searching and sorting algorithms to maximize its efficiency when doing tasks.

```
/**
 * Prints out in the console the volunteers that need to be trained.
 */
public void displayVolunteersWithoutJobs()
{
    final String NO_JOB = "no job. needs to be trained!";
    for( int volunteerIndex = 0; volunteerIndex < database1.getVolunteerList().size(); volunteerIndex++)
    {
        if (database1.getVolunteerList().get(volunteerIndex).getActualJob().getJobTitle().equals(NO_JOB))
        {
            System.out.println(database1.getVolunteerList().get(volunteerIndex).toString());
        } // end of  if (database1.getVolunteerList().get(volunteerIndex).getActualJob().getJobTitle().equals(NO_JOB))
    } // end of for( int volunteerIndex = 0; volunteerIndex < database1.getVolunteerList().size(); volunteerIndex++)
} // end of displayVolunteersWithoutJobs()
```

The method above uses a sequential search to linearly search through the volunteer list to print out all the volunteers without jobs. A sequential search is necessary as there is no way of sorting between if a volunteer has or does not have a job (i.e. Boolean instance variables).

```
 * Bubble sorts the given job list in order from least number of volunteers neeeded, to greatest.
 *
 * @param jobList the list of jobs<br>
 * <i>precondition:</i> <code>jobList</code> cannot be null
 * @return the list of jobs
 */
public LinkedList bubbleSortJobList(LinkedList <Job> jobList)
{
    int j = 0;
    boolean swapWasMade = true;
    Job temporaryJobHolder;
    while (swapWasMade)
    {
        j++;
        swapWasMade = false;
        for (int i = 0; i < jobList.size()-1; i++)
        {
            if (jobList.get(i).getNumberOfVolunteersNeeded() > jobList.get(i+1).getNumberOfVolunteersNeeded())
            {
                temporaryJobHolder = jobList.get(i);
                jobList.set(i, jobList.get(i+1));
                jobList.set(i+1, temporaryJobHolder);
                swapWasMade = true;
            } // end of if (jobList.get(i).getNumberOfVolunteersNeeded() > jobList.get(i+1).getNumberOfVolunteersNeeded(
        } // end of for (int i = 0; i < jobList.size()-1; i++)
    } // end of while
    return jobList;
} // end of bubbleSortJobList(LinkedList <Job> jobList)
```

```java
/**
 * Prints out in the console the jobs that need volunteers, and the number of volunteers each job requires, in order from jobs that need the most volunteers to jobs that need the least volunteers.
 */
public void displayExtraJobs()
{
    System.out.println("\n");
    database1.bubbleSortJobList(database1.getJobList());
    int jobIndex = database1.getJobList().size()-1;
    boolean jobIndexIsNegative = false;
    while (!jobIndexIsNegative && database1.getJobList().get(jobIndex).getNumberOfVolunteersNeeded() != 0)
    {
        int jobNumber = jobIndex + 1;
        System.out.println(jobNumber + ": " + database1.getJobList().get(jobIndex).getJobTitle() + " - " + database1.getJobList().get(jobIndex).getNumberOfVolunteersNeeded() +
            " volunteers needed");
        jobIndex--;
        if (jobIndex == - 1)
        {
            jobIndexIsNegative = true;
        } // end of if (jobIndex == - 1)
    } // end of while(!jobIndexIsNegative && database1.getJobList().get(jobIndex).getNumberOfVolunteersNeeded() != 0)
} // end of displayExtraJobs()
```

The bubbleSortJobList method sorts the job list in terms of number of volunteers needed. This allows easier traversal in other methods. For example, the displayExtraJobs method displays the jobs that still need volunteers in order from jobs that need the most volunteers, to jobs that need the least number of volunteers. Without a sorting algorithm, the linear search would have to traverse through the whole job list multiple times to iteratively print out the jobs and volunteers needed in terms of priority. With the bubble sort with complexity $O(n^2)$, the console simply has to traverse linearly through the job list once to print out the jobs and their volunteers needed in terms of priority.

Algorithmic Thinking

The most brilliant aspect of the solution is its algorithm for assigning jobs to volunteers to minimize the number of volunteers that don't have jobs. The complete algorithm is shown below.

```java
/**
 * Assigns the jobs to the vounteers to minimize the number of volunteers without jobs.
 */
public void assignJobsToVolunteers()
{
    final int INVALID_JOB_INDEX = -1;
    volunteer = bubbleSortVolunteerList(volunteer);
    for (int volunteerIndex = 0; volunteerIndex < volunteer.size(); volunteerIndex++)
    {
        double lowestVolunteerToVolunteersNeededPercentage = Integer.MAX_VALUE;
        int jobIndexOfLowestVolunteerToVolunteersNeededPercentage = INVALID_JOB_INDEX;
        for(int jobIndex = 0; jobIndex < job.size(); jobIndex++)
        {
            if(volunteer.get(volunteerIndex).getCanDoJob()[jobIndex] && getJobVolunteersToVolunteersNeededPercentage(jobIndex) < lowestVolunteerToVolunteersNeededPercentage)
            {
                lowestVolunteerToVolunteersNeededPercentage = getJobVolunteersToVolunteersNeededPercentage(jobIndex);
                jobIndexOfLowestVolunteerToVolunteersNeededPercentage = jobIndex;
            } // end of if(volunteer.get(volunteerIndex).getCanDoJob()[jobIndex] && getJobVolunteersToVolunteersNeededPercentage(jobIndex) < lowestVolunteerToVolunteersNeededPercentage)
        } // end of for(int jobIndex = 0; jobIndex < job.size(); jobIndex++)

        if (jobIndexOfLowestVolunteerToVolunteersNeededPercentage != INVALID_JOB_INDEX && job.get(jobIndexOfLowestVolunteerToVolunteersNeededPercentage).getNumberOfVolunteersNeeded() != 0)
        {
            volunteer.get(volunteerIndex).setActualJob(job.get(jobIndexOfLowestVolunteerToVolunteersNeededPercentage));
            job.get(jobIndexOfLowestVolunteerToVolunteersNeededPercentage).setNumberOfVolunteersNeeded
            (job.get(jobIndexOfLowestVolunteerToVolunteersNeededPercentage).getNumberOfVolunteersNeeded() - 1);
        }
        else
        {
            volunteer.get(volunteerIndex).getActualJob().setJobTitle("no job needs to be trained");
        } // end of if (jobIndexOfLowestVolunteerToVolunteersNeededPercentage != INVALID_JOB_INDEX && lowestVolunteerToVolunteersNeededPercentage < maximumCapacityPercentage)
    } // end of for (int numberOfJobsCanDo = 1; numberOfJobsCanDo < job.size(); numberOfJobsCanDo++)
    jobsHaveBeenAssigned = true;
} // end of assignJobsToVolunteers()
```

This algorithm uses the multiple nested for loops and conditional statements to best assign jobs to volunteers. The nesting of loops and conditional statements within each other iteratively check through each list to select the least flexible volunteers and jobs that are least popular so that those volunteers and jobs are assigned first. Jobs are iteratively assigned so that the database maximizes the number of volunteers with jobs. This algorithm uses a bubble sort to sort the volunteer list in terms of the job flexibility of volunteers, to once again improve the efficiency of the algorithm.

Java Console Input/Output Skills

Databases also have to fend off the problem of input error. The database limits input error in several different ways. For example, if the user tries to input a job title that he already inputted, the database will disregard the input and print out an error message. This is done by iteratively checking through the job list and seeing if the job title string matches with the new job title.

```
/**
 * Checks if the proposed job already exists, and if not, creates the given job with the given number of volutneers required and adds it to the end of the list of jobs.
 *
 * @param jobTitle the job title of this job to be added <br>
 * <i>precondition:</i> <code>jobTitle</code> cannot be null
 * @param numberOfVolunteersRequired the number of volunteers required of this job
 * @return true if the job was successfully created and added; false otherwise
 */
public boolean addJob(String jobTitle, int numberOfVolunteersRequired)
{
    if (jobTitle != null && numberOfVolunteersRequired > 0 )
    {
        boolean jobAlreadyExists = false;
        for (int jobIndex = 0; jobIndex < job.size(); jobIndex++)
        {
            if (job.get(jobIndex).getJobTitle().equals(jobTitle))
            {
                jobAlreadyExists = true;
            } // end of if (job.get(jobIndex).getTitle().equals(jobTitle)
        } // end of for (int jobIndex = 0; jobIndex < job.size(); jobIndex++)
        if (!jobAlreadyExists)
        {
            job.add(new Job(jobTitle, numberOfVolunteersRequired));
            return true;
        } // end of if (!jobAlreadyExists)
    } // end of if (jobTitle != null && numberOfVolunteersRequired > 0 )
    return false;
} // end of addJob(String jobTitle, int numberOfVolunteersRequired) throws IOException
```

The database does the same thing with volunteers that already have their names' in the database.

Dynamic Lists

```
// instance variables
private LinkedList <Job> job;
private boolean jobsHaveBeenAssigned;
private LinkedList <Volunteer> volunteer;
```

Like most databases, the solution uses dynamic lists. The database employs linked lists for both the volunteer and job list. Dynamic lists are necessary as their expandability allows for scaling number of jobs and volunteers. Arrays would be inefficient as their static sizes disallow changing number of volunteers and jobs which are against the use of databases. The solution uses linked lists over array lists as deletions and insertions are common actions done to the lists, and linked lists are very efficient at deleting and inserting.

. Error Control

```
/**
 * Receives the console input and the list of valid console inputs and invokes appropriate action.
 *
 * @param input the console input <br>
 * <i>precondition:</i> <code>input</code> cannot be null
 */
public void manageConsoleInput(String input,int[] validOption) throws IOException
{
    if (input != null)
    {
        inputIsValid = false;
        lastAction = LastAction.INVALID_INPUT;
        final int INVALID_OPTION_INDEX= Integer.MAX_VALUE;
        int option = INVALID_OPTION_INDEX;
        try
        {
            option = Integer.parseInt(input);
        }
        catch (NumberFormatException error)
        {
            if (database1.getVolunteerList().size()== 0)
            {
                addAllJobs();
            }
            else
            {
                addAllVolunteers();
            } // end of if (database1.getVolunteerList().size()== 0)
        } // end of try
```

The selection of options on the menu relies on parsing the string taken into an integer. If the user does not type a string that can be parsed into an integer, a NumberFormatException occurs. Thus, a try-catch statement is necessary to appropriately react to the invalid input. By catching the error instead of throwing it, the console can inform the user of the issue so that it does not happen again.

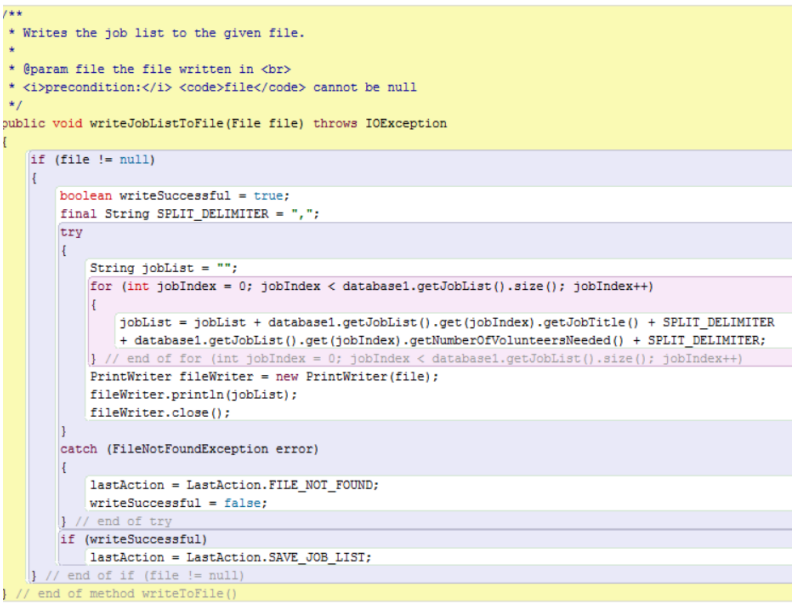
File I/O

The database can read from files. The database is able to read from comma-separated values files to import and export volunteers and jobs. Comma-separated values are elements that are separated by commas. The database reads the entirety of the file, and breaks down the string into substrings in an array. It then creates the volunteer/job based on the elements in the array. This is shown below.

```
**
* Adds jobs from the given file.
*
* @param filePath the file path of the file <br>
* <i>precondition:</i> <code>filePath</code> cannot be null
*/
ublic void addJobsFromFile(String filePath) throws IOException

    if (filePath != null)
    {
        try
        {
            BufferedReader fileReader = new BufferedReader(new FileReader(filePath));
            String concatenatedFileData = "";
            boolean fileIsCorrupt = false;
            boolean isEndOfFile = false;
            String fileData;
            while(!isEndOfFile)
            {
                fileData = fileReader.readLine();
                if (fileData == null)
                    isEndOfFile = true;
                if (!isEndOfFile)
                {
                    concatenatedFileData += fileData;
                } // end of if (!isEndOfFile)
            } // end of while(!isEndOfFile)
            final String SPLIT_STRING = ",";
            String[] job = concatenatedFileData.split(SPLIT_STRING);
            for (int numberOfWorkersNeededIndex = 1; numberOfWorkersNeededIndex < job.length; numberOfWorkersNeededIndex+=2)
            {
                try
                {
                    int numberOfWorkersForEachJob = Integer.parseInt(job[numberOfWorkersNeededIndex]);
                }
                catch (NumberFormatException error)
                {
                    fileIsCorrupt = true;
```

The database can write to the volunteer and job lists to files. It iteratively takes each volunteer/job and writes their instance fields to files to create a CSV file. This file I/O allows Joel to save his work as the CSV files can be read again later for further edits.



```
/**
 * Writes the job list to the given file.
 *
 * @param file the file written in <br>
 * <i>precondition:</i> <code>file</code> cannot be null
 */
public void writeJobListToFile(File file) throws IOException
{
    if (file != null)
    {
        boolean writeSuccessful = true;
        final String SPLIT_DELIMITER = ",";
        try
        {
            String jobList = "";
            for (int jobIndex = 0; jobIndex < database1.getJobList().size(); jobIndex++)
            {
                jobList = jobList + database1.getJobList().get(jobIndex).getJobTitle() + SPLIT_DELIMITER
                + database1.getJobList().get(jobIndex).getNumberOfVolunteersNeeded() + SPLIT_DELIMITER;
            } // end of for (int jobIndex = 0; jobIndex < database1.getJobList().size(); jobIndex++)
            PrintWriter fileWriter = new PrintWriter(file);
            fileWriter.println(jobList);
            fileWriter.close();
        }
        catch (FileNotFoundException error)
        {
            lastAction = LastAction.FILE_NOT_FOUND;
            writeSuccessful = false;
        } // end of try
        if (writeSuccessful)
            lastAction = LastAction.SAVE_JOB_LIST;
    } // end of if (file != null)
} // end of method writeToFile()
```

Word Count: 712