

Andrey Grzegorzewski
 Programming Environments
 Ohio Northern University - Fall 2017
 Progress Report

Introduction and Status

For Programming Environments, I have been working on a C# program called JobFairsApp that is meant to perform data management and interview scheduling for job fairs. So far, I have modified the database to reflect my approach to this task, begun implementing data entry, and outlined an algorithm for scheduling interviews.

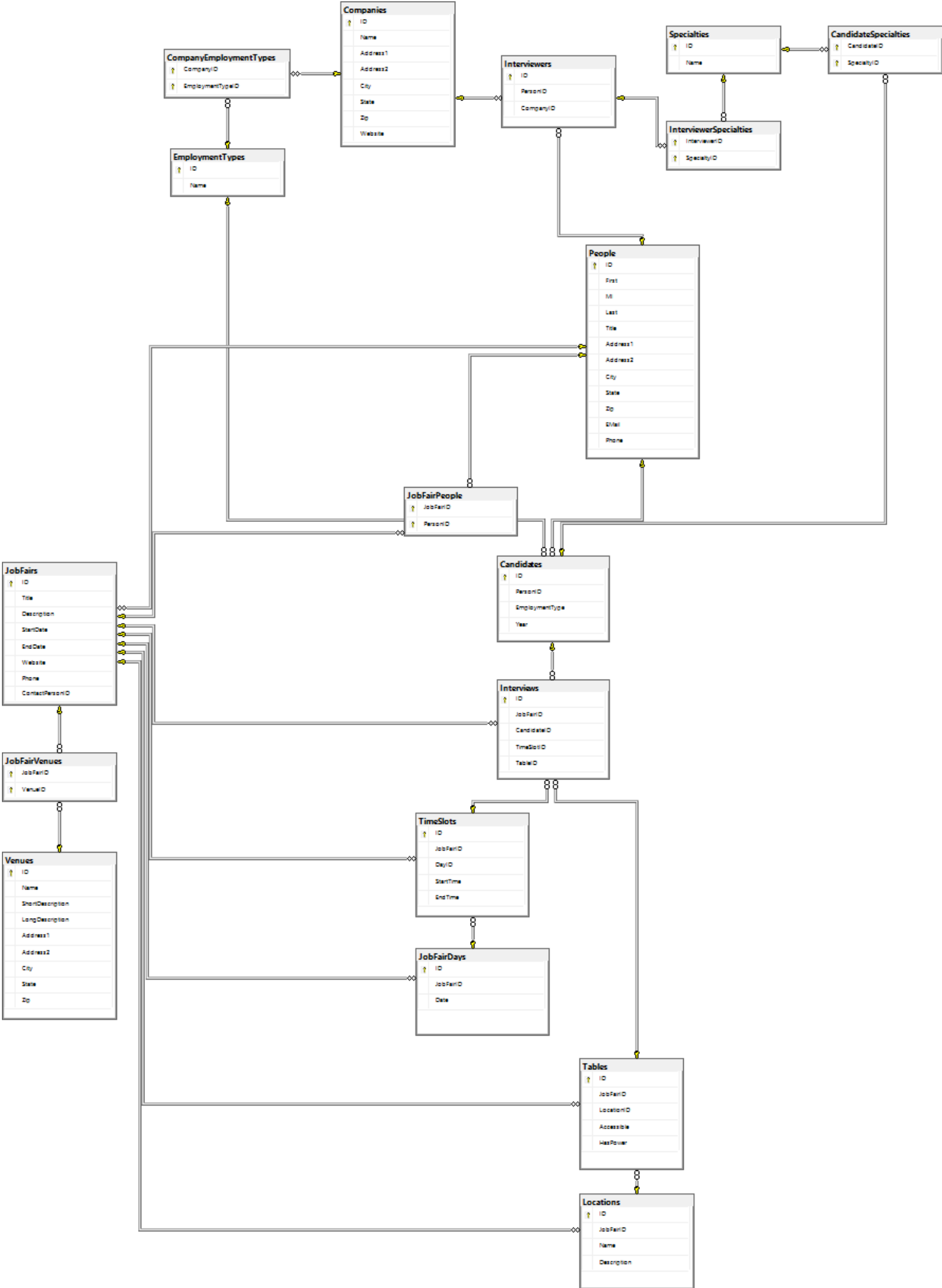
Database Design

I have added several tables to the JobFair database, and I've edited a few others. Following is a summary of my changes to each table and my reasons for the changes.

Table Name	Description
Specialties	The specialties table contains an ID column and a name column. It is used to keep track of "specialties," such as majors, minors, concentrations, and areas of interest, that candidates have and that interviewers are looking for.
Employment Types	The EmploymentTypes table contains an ID column and a name column. It is used to hold the types of employment a company or candidate could be looking for—internship, co-op, or full-time.
Companies	I've added a Companies table, which contains a company name, address, and website. This can be used to link interviewers to the company they are interviewing for.
Candidates	I've updated the Candidates table to include an employment type and a year in school. The employment type column contains a foreign key to the EmploymentTypes table, and the year is an integer, usually from 1-4.
Interviewers	The Interviewers table now includes a CompanyID column, which is a foreign key to the ID column of the Companies table.
Candidate Specialties	I added this table to link candidates to their specialties. This table only contains two columns, both of which are foreign keys: the CandidateID column and the SpecialtyID column.

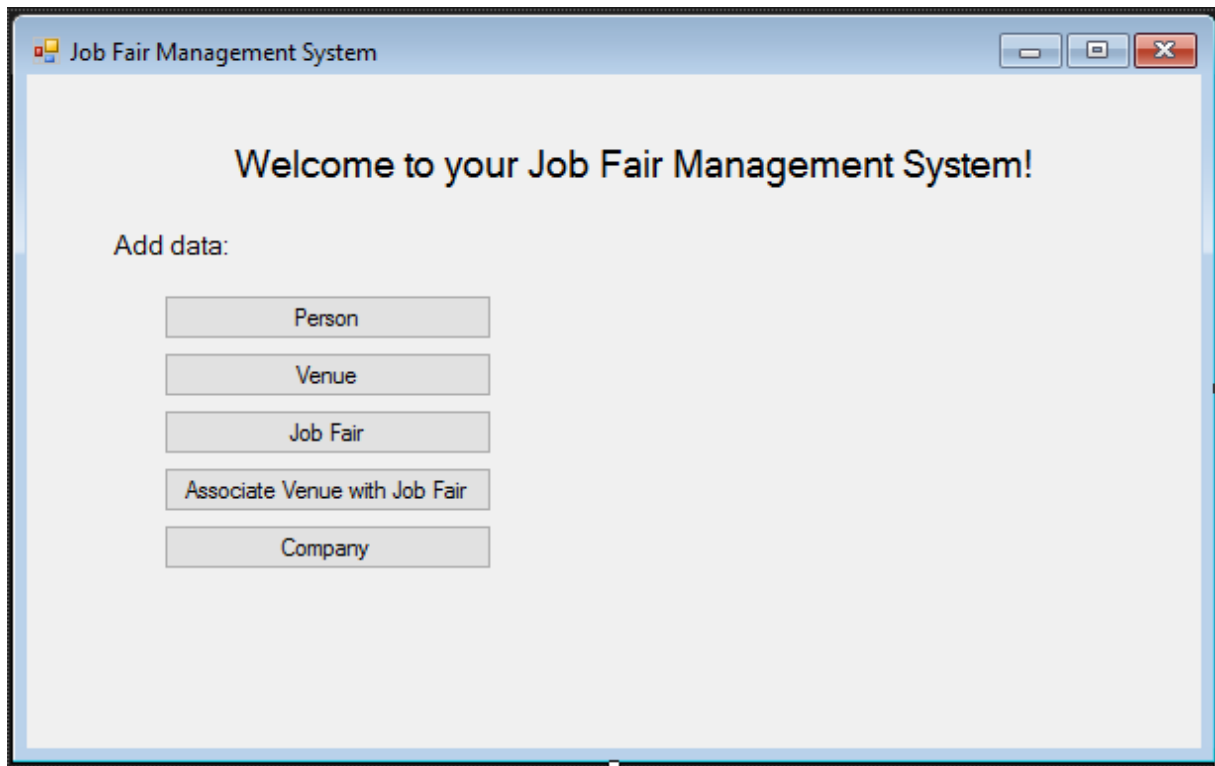
Interviewer Specialties	The InterviewerSpecialties table is similar to the CandidateSpecialties table, except that it links interviewers to their specialties.
Company Employment Types	This table is used to keep track of which types of employees a company is seeking to hire. A company can look for employees of as many types as it chooses.
Job Fair Interviews	I altered the JobFairInterviews table. I changed it to the JobFairPeople table. Instead of JobFairIDs, InterviewerIDs, and InterviewIDs, the table now has JobFairIDs and PersonIDs. This way, each person is linked to each job fair they attend. This will be useful later on for scheduling interviews. The interview scheduling procedure, as I will discuss later on, will only match candidates with interviewers that will be at the job fair they are attending. This table will ensure that that process can happen.

The following page consists of an image of the diagram of the JobFair database so far.

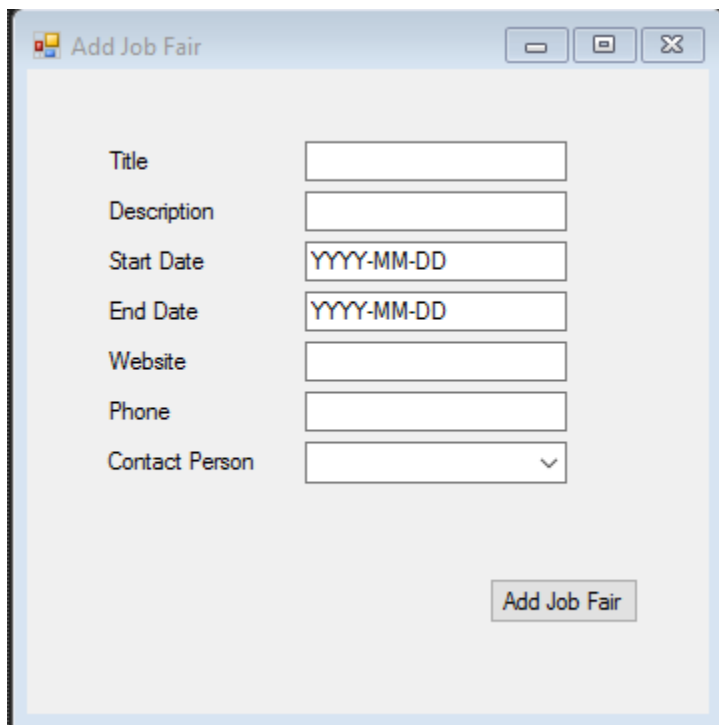


App Design

Although each data entry form looks different to reflect the data being entered, there is a common theme among each form. The main form allows the user to select which data they would like to enter. This form, as of the writing of this report, is as shown below.



The screenshot shows a window titled "Job Fair Management System". Inside the window, the text "Welcome to your Job Fair Management System!" is centered. Below this, the label "Add data:" is followed by five buttons stacked vertically: "Person", "Venue", "Job Fair", "Associate Venue with Job Fair", and "Company".



The screenshot shows a window titled "Add Job Fair". It contains a form with the following fields and labels: "Title", "Description", "Start Date", "End Date", "Website", "Phone", and "Contact Person". The "Start Date" and "End Date" fields have a placeholder text "YYYY-MM-DD". The "Contact Person" field is a dropdown menu. At the bottom right of the form is a button labeled "Add Job Fair".

The remaining forms are data entry forms. With some slight variation, they all look similar to the form to the left, which is the form for adding a job fair.

App Implementation

Since the last time I discussed my app in class, I have rewritten it to contain less repetitive code. I created a Column class to model a column in a database, and a Table class to model database tables. In each form, I create a new Table object. (I sometimes create more than one, depending on the context.) A table contains a list of Columns. I use the Table objects to perform insertion operations and to read data from the database. The Column code is summarized below:

```
public class Column
{
    private string name = "";
    private string currentValue = "";

    public Column(string name)
    {
        this.name = name;
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string CurrentValue
    {
        get { return currentValue; }
        set { currentValue = value; }
    }
}
```

The Column class also contains some public methods to verify user input. These methods (VerifyPhone, VerifyZip, VerifyState, and VerifyDate) return true if the entered input is valid and false otherwise. The Table class is shown in its entirety below:

```
public class Table
{
    private string name = "";
    private List<Column> columns = new List<Column>();
    private string connStr = @"Data Source = localhost\SQLEXPRESS; Initial Catalog = JobFair; Integrated Security = True";

    public Table(string name, List<Column> columns)
    {
        this.name = name;
        this.columns = columns;
    }

    public string Name
    {
        get { return name; }
    }
}
```

```

        set { name = value; }
    }

    public List<Column> Columns
    {
        get { return columns; }
        set { columns = value; }
    }

    // Return the true if the insert was successful; false otherwise
    public bool InsertRow()
    {
        // Create the command
        string commandText = "INSERT INTO " + this.name + " (";

        // Make sure we don't accidentally insert an ID column
        int start;
        if (columns[0].Name == "ID")
            start = 1;
        else
            start = 0;

        // Add each column name to the command text
        for (int i = start; i < columns.Count - 1; i++)
            commandText += columns[i].Name + ", ";
        commandText += columns[columns.Count - 1].Name + ") VALUES (";

        // Add each column value to the command text
        for (int i = start; i < columns.Count - 1; i++)
            commandText += "'" + columns[i].CurrentValue + "', ";
        commandText += "'" + columns[columns.Count - 1].CurrentValue + "'";

        // Connect to the database
        SqlConnection sc = new SqlConnection(connStr);
        sc.Open();

        // Set command data
        SqlCommand cmd = new SqlCommand();
        cmd.CommandText = commandText;
        cmd.CommandType = System.Data.CommandType.Text;
        cmd.Connection = sc;

        // Try to execute the insert
        if (cmd.ExecuteNonQuery() == 1)
        {
            // Get the ID of the row we just inserted
            cmd.CommandText = "SELECT @@IDENTITY;";
            if (columns[0].Name == "ID")
                this.columns[0].CurrentValue = cmd.ExecuteScalar().ToString();

            // Close the connection - return true for success
            sc.Close();
            return true;
        }

        // Close the connection - return false for failure
        sc.Close();
        return false;
    }

```

```

    }

    /// <summary>
    /// Reads data from the database
    /// </summary>
    /// <param name="command">The CommandText for a SqlCommand object</param>
    /// <param name="columns">The number of columns that will be selected by the
command</param>
    /// <returns></returns>
    public List<string> Read(string command, int columns)
    {
        // Connect to the DB
        SqlConnection sc = new SqlConnection(connStr);
        sc.Open();

        // Select job fair title/venue name pairs
        SqlCommand cmd = new SqlCommand();
        cmd.CommandText = command;

        // Set command data
        cmd.CommandType = System.Data.CommandType.Text;
        cmd.Connection = sc;

        // List for holding the results
        List<string> results = new List<string>();

        // Iterate through data
        using (SqlDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())
            {
                for (int i = 0; i < columns; i++)
                {
                    results.Add(reader.GetString(i));
                }
            }
            sc.Close();
            return results;
        }
    }
}

```

In each data entry form, I create at least one Table object, which itself contains a list of Column objects. This is done as follows:

```

Column companyID = new Column("CompanyID");
Column employmentTypeID = new Column("EmploymentTypeID");

List<Column> CETcolumns = new List<Column>
{
    companyID,
    employmentTypeID
};

companyEmploymentTypes = new Table("CompanyEmploymentTypes", CETcolumns);

```

Then, in order to insert data into this table in the database, I assign values to the `CurrentValue` property of each `Column`, and call the `InsertRow` method.

```
companyEmploymentTypes.Columns[0].CurrentValue = companies.Columns[0].CurrentValue;
companyEmploymentTypes.Columns[1].CurrentValue = "1";
companyEmploymentTypes.InsertRow();
```

I follow this procedure for each insertion I have to do. I have a similar process for reading data from the database using the `Read` method shown above in the `Table` class.

Interview Automation Algorithm Outline

I have outlined the process I expect to follow for generating interviews below. I plan to use selects and joins in SQL to create a series of intermediate tables that I will use to fill in the final `Interviews` table.

1. Select all candidates and interviewers who are registered for the specific job fair we're dealing with.
2. Create candidate-interviewer pairs wherever a candidate and interviewer have the same specialty. Store this data in an intermediate table called `Pairs1`. `Pairs1` will have columns for `CandidateID`, `InterviewerID`, `SpecialtyID`, and `CompanyID`.
3. Create candidate-company pairs wherever a candidate is seeking an employment type that a company is offering. (Employment type here refers to full-time, co-op, or internship.) Save this in an intermediate table called `Pairs2`. `Pairs2` will have columns for `CandidateID`, `CompanyID`, and `EmploymentTypeID`.
4. Join `Pairs1` and `Pairs2` together to find matches between candidates and interviewers where the employment type being sought by the candidate is being offered by the interviewer's company. Save this data in the `Interviews` table. Save `InterviewID` and `InterviewerID` pairs in a separate table to allow for one interview with multiple interviewers. One interview for each row in the `Interviews` table needs to be scheduled.
5. Iterate through the rows in `Interviews` joined with the `InterviewInterviewers` table, ordered by `InterviewerID`. (I'm ordering the interviews by `InterviewerID` based on the assumption that the Interviewers will have the most interviews, so we should schedule according to their needs.) For each interview,
 - a. Select the first available time slot that has at least one table available.
 - b. Check to see if the interviewer is busy and if the candidate is busy.
 - c. If both parties are available, add the `TimeSlotID` and the first `TableID` to the `Interviews` table. Also mark the time slot-table pair as taken.
 - d. If both parties are not available, jump to the next time slot and try again.

6. Use joins to generate a user-friendly table listing the following items:
 - a. The candidate's first and last name
 - b. The company name
 - c. The interviewer(s)' first and last name(s)
 - d. The time, table number, and location of the interview
 - e. The topic and type of employment the interview will cover

Necessary Updates

If I implement this algorithm, I will need to update the Interviews table, as well as create a new table for Interview-Interviewer pairs. I think I should also create a table for TimeSlot-Table pairs, which should include a boolean column stating if that pair is taken or available. It should also include a JobFairID column. I will need to create a new method in my app for generating the interviews, as well as implementing the algorithm with the SQL statements I described above. The method in my app should have the signature:

```
void GenerateInterviews(int jobFairID)
```

I can use my existing Read function to read all the data from the Interviews table, so no changes need to be made for that.

Now that my code structure reflects the database, significant changes shouldn't need to be made in my code. However, I will have to spend some time thinking about how to best present the data to the user and, more significantly, how to best allow the user to edit the generated interviews.

Next Steps

Here is an ordered list of the tasks I will be working on moving forward.

1. Create a data entry form for the Locations table and automate the data entry into the JobFairDays table. I'm starting here because these things need to be done before TimeSlots and Tables can be handled.
2. Create an algorithm to fill in the TimeSlots table without user input. This needs to be done next because other tables depend on the TimeSlots table.
3. Finish implementing data entry for the remaining tables. This needs to be done next because the interview generation algorithm will require the usage of each of these tables. The tables should be implemented in the following order:
 - a. JobFairPeople
 - b. Interviewers

- c. InterviewerSpecialties
- d. Candidates
- e. CandidateSpecialties
- f. Tables

I plan to start with JobFairPeople because it should be handled with the Person entry form, which is already started. Next, I'm moving on to Interviewers, Candidates, and their Specialties, because the forms for them should appear after the People data entry form. I'm ending with Tables because it is the only table left that doesn't have anything to do with people.

4. Test data entry for each of the tables implemented above and make sure foreign keys and primary keys are working as expected. This should be done before moving past the data entry portion of the app to ensure that I don't run in to any unexpected problems later on. I plan to try to break my code, and fix any breaks I cause.
5. Implement the interview generation algorithm. I will start by generating candidate-interviewer-company sets as described in Steps 1-4 of the algorithm I outlined above. I will check that the algorithm produces the correct sets and does not alter, remove, or skip over any data in the process. Then, I will implement Step 5 of my algorithm to assign each interview to a time slot and table. Finally, I will produce the user-friendly table and display it by turning each row into a string and displaying each string in a list box in my app.
6. Finally, I will implement data editing and deleting for the tables, and end by allowing the user to edit, add to, or delete interviews generated by my algorithm.

Changes to My App

As I discussed earlier, I have already rewritten my app to use Table and Column objects to simplify my interaction with data. I am currently in the process of creating forms for data entry for the remaining tables. As I do this, I am thinking about how the tables might need to be updated to allow for the best interview generation algorithm possible.

Conclusion

By class 10/18, I plan to have all of my data entry forms handled, and I will begin working on my interview generation algorithm. I hope to gain some insights on how to best proceed with the algorithm in class on 10/13. Although I feel that I have a good start on the algorithm, I anticipate finding unexpected problems at every step of its implementation. Since I don't have any experience with SQL procedures or functions, or the creation and deletion of intermediate tables, I would also like to learn more about that. I will research this on my own, but I also hope to hear about it in class 10/13.