Spike 2 Outcome & Notes

Spike Outcomes

Set: ***2b***
Languages ***C#***
Name: Reuben Wilson and Yan Chernikov

Goals:
As a team, it was decided that a generalised database framework had to be developed, which would provide for the majority of the required functionality for this spike (spike 2), and perhaps future spikes. The idea was to develop a backbone that could be used for any data driven program written in C#. The adjective generalised is used to describe the class, because it has been designed and implemented in a way that it can be applied to any database, not just the archery database that has been provided for the semester's task requirements.

In terms of specific spike related goals, the first goal is to ensure that a clear understanding of the DataSet class, included in the ADO .NET framework is gathered and how it applies to the required outcomes of this spike. As well as developing an understanding of the DataSet class, it is the goal of the group to implement a simple program that utilises initialised DataSet objects that represent real data from a database. In regards to this spike, a DataSet object will be utilised to catalogue all of the information related to one or more entities within the archery database.

In relation to DataSet objects, another goal is to ensure that data contained within an initialised DataSet can be manipulated using the common sql query types. I.e. select, insert, update and delete. Each table contained within a DataSet object is actually represented by a DataTable object, that is to say, there is a unique DataTable object within a DataSet for each unique database table stored within the DataSet. Therefore, the goal is to successfully manipulate data contained within a DataTable object and push those changes to the sql database after successful data manipulation.

The final goal for this spike is to ensure that a clear understanding of the DataRelation class is gathered in terms of how an instantiated DataRelation object can be used to represent entity relationships contained within a DataSet object. Remember, a DataSet is populated using one or more tables from an sql database and if there are entity relationships present in one or more of those tables, there needs to be a programmatic way to represent those relationships. That is where the DataRelation class comes into play. In regards to this spike, the two tables used are multieventrule and mutlieventcompetition, both of which are from the archery database provided. There is a one to many relationship present in these two tables and the idea is to utilise a DataRelation object to enforce the relationship.

Some secondary goals include establishing a clear understanding of how MySqlCommandBuilder objects can be used to dynamically populate sql queries rather than having to hardcode queries in the program's source code.

Personnel:
Primary - Yan Chernikov 9991379
Secondary - Reuben Wilson 9988289

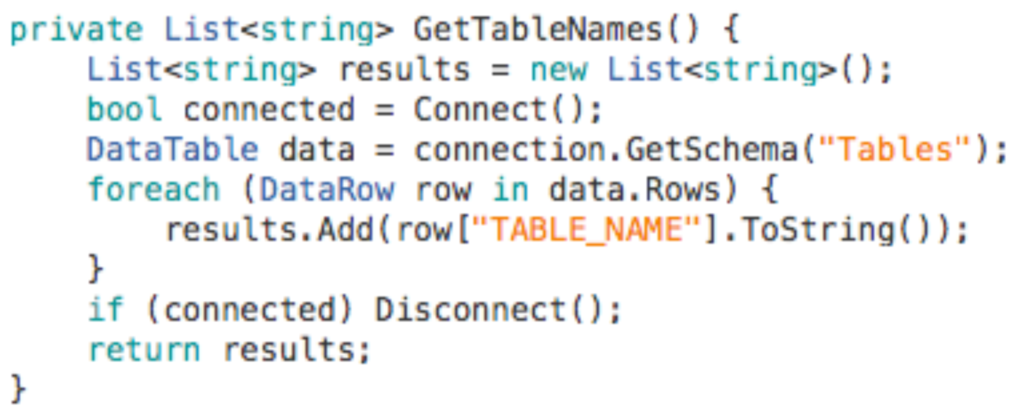
Technologies, Tools, and Resources used:
The technologies used to develop the overall result for this spike are defined below:
C#
.NET Framework 4.5
Mono Framework
MySql
ADO .NET

The tools used to complete this spike are defined below:
Xamarin IDE
Terminal
Mac OSX

The resources that were used to gain the knowledge required to complete this spike are defined below:
MSDN C# Documentation Portal:
http://msdn.microsoft.com/en-AU/library/618ayhy6.aspx
MSDN ADO .NET Documentation Portal:
http://msdn.microsoft.com/en-us/library/e80y5yhx(v=vs.110).aspx
Stack Overflow Community Discussion Board:
http://stackoverflow.com

Tasks undertaken:

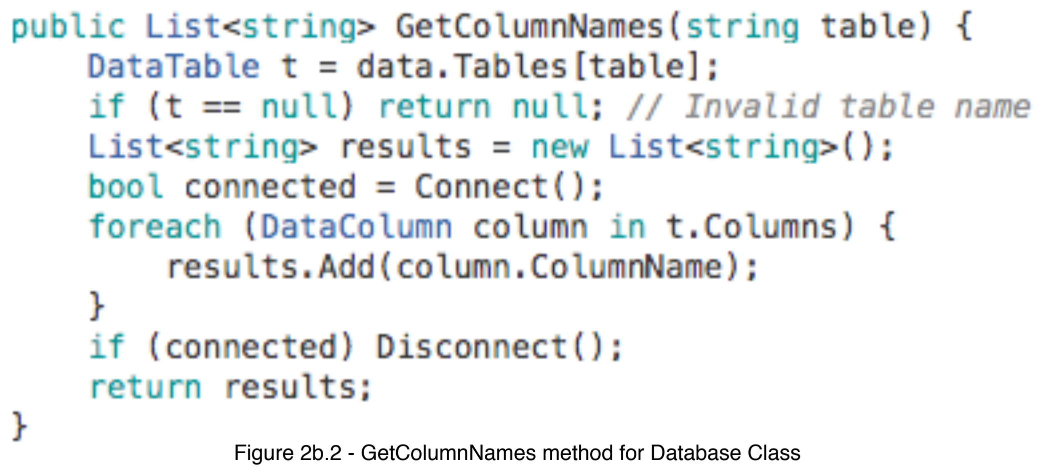
1. As previously defined in the goals section, the central theme behind the development phase of this spike was ensure that a stable, generalised class was developed that could be utilised with any database. The generalised class implements all of the functionality that is required for the outcome of this spike and the development of this class was the main task undertaken. The class will be briefly defined here, highlighting some of the methods that have been developed and can demonstrate the generalised nature of the class. For the complete source code, please refer to the Spike 2 Source Code appendix at the end of this portfolio.



```
private List<string> GetTableNames() {
    List<string> results = new List<string>();
    bool connected = Connect();
    DataTable data = connection.GetSchema("Tables");
    foreach (DataRow row in data.Rows) {
        results.Add(row["TABLE_NAME"].ToString());
    }
    if (connected) Disconnect();
    return results;
}
```

Figure 2b.1 - GetTableNames method for Database Class

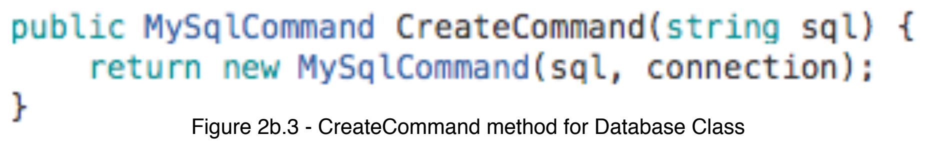
The method defined in the figure above is called GetTableNames. Based on the database that has been connected to, this method will return a List of strings. Each string value within the list represents a unique table within the database schema.



```
public List<string> GetColumnNames(string table) {
    DataTable t = data.Tables[table];
    if (t == null) return null; // Invalid table name
    List<string> results = new List<string>();
    bool connected = Connect();
    foreach (DataColumn column in t.Columns) {
        results.Add(column.ColumnName);
    }
    if (connected) Disconnect();
    return results;
}
```

Figure 2b.2 - GetColumnNames method for Database Class

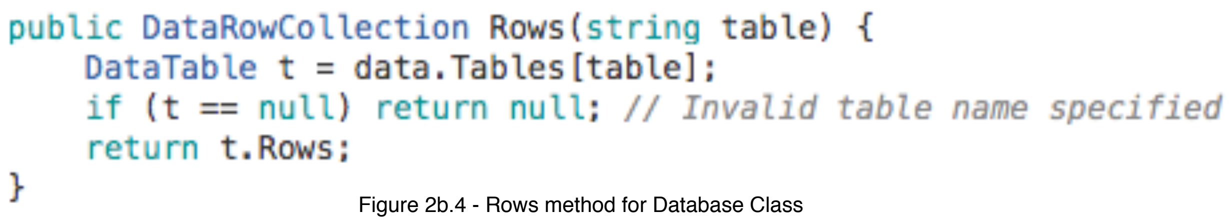
The method defined in the figure above is called GetColumnNames. The method takes one parameter as a string. The value of the string parameter represents the name of a single table within the database schema. Recall, as defined above, the program offers the functionality to reliably and accurately return the names of every table within the connected database. Considering this, GetColumnNames could be passed a table name, one of which is returned from the method GetTableNames, and as a result, the method GetColumnNames returns a List of strings representing all of the columns. Each string within the list represents the name of a single column in regards to the table name that was passed as a parameter.



```
public MySqlCommand CreateCommand(string sql) {
    return new MySqlCommand(sql, connection);
}
```

Figure 2b.3 - CreateCommand method for Database Class

The method defined in the figure above is called CreateCommand. The method returns a single instance of class, MySqlCommand. The method takes a single parameter as a string. The passed parameter is intended to represent a valid sql query. The method returns a MySqlCommand object related to the sql query string that is passed as a parameter.



```
public DataRowCollection Rows(string table) {
    DataTable t = data.Tables[table];
    if (t == null) return null; // Invalid table name specified
    return t.Rows;
}
```

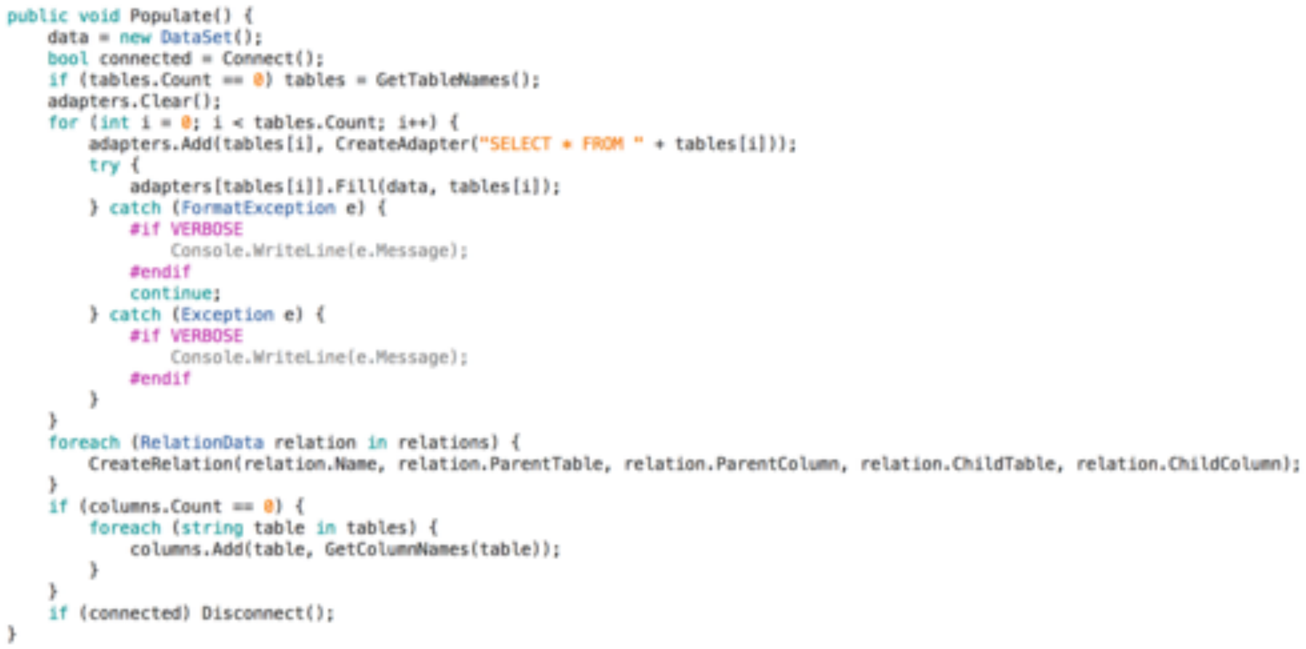
Figure 2b.4 - Rows method for Database Class

The method defined in the figure above is called Rows. The method is passed a single parameter as a string. The value of the parameter is intended to represent the name of a single table within the connected database schema. Using the private DataSet member, data, a DataTable object is initialised and it’s rows are returned, hence the return type of DataRowCollection.

Please note that this method is only ever invoked AFTER the private member data (of type DataSet) has been instantiated appropriately.

This concludes the brief insight into some of the generalised methods within the generalised Database class. The remainder of the tasks defined within this report also depict methods from the same class that have all been developed using the same generalised theme.

2. The next task involved developing the required method for the generalised Database class, that could instantiate the required DataSet object for the connected database. In regards to this spike, the required DataSet object is initialised using all of the tables within the archery database schema. Remember, as defined earlier, the method GetTableNames returns all of the table names for the connected database. This method is therefore utilised in the process of populating the DataSet object. The figure below offers and insight into the Populate method, which is responsible for appropriately instantiating the DataSet object.

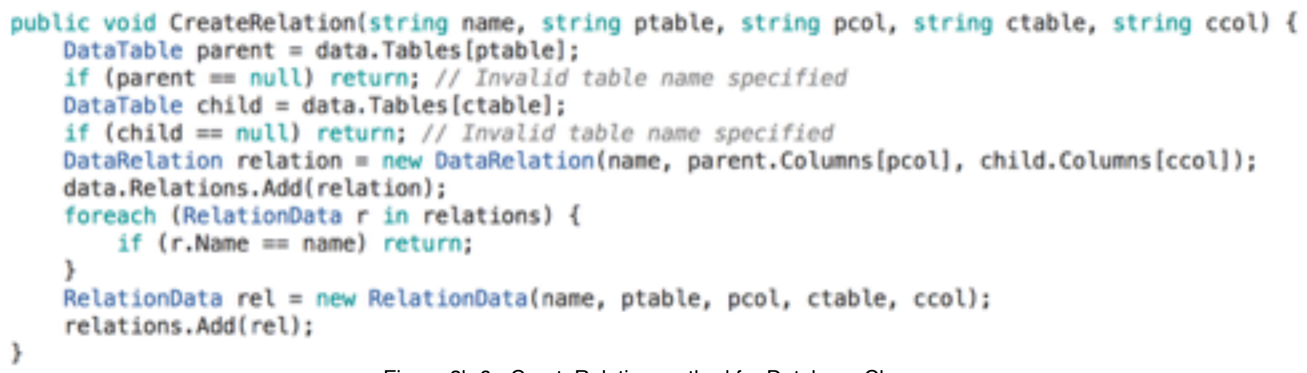


```
public void Populate() {
    data = new DataSet();
    bool connected = Connect();
    if (tables.Count == 0) tables = GetTableNames();
    adapters.Clear();
    for (int i = 0; i < tables.Count; i++) {
        adapters.Add(tables[i], CreateAdapter("SELECT * FROM " + tables[i]));
        try {
            adapters[tables[i]].Fill(data, tables[i]);
        } catch (FormatException e) {
            #if VERBOSE
                Console.WriteLine(e.Message);
            #endif
            continue;
        } catch (Exception e) {
            #if VERBOSE
                Console.WriteLine(e.Message);
            #endif
        }
    }
    foreach (RelationData relation in relations) {
        CreateRelation(relation.Name, relation.ParentTable, relation.ParentColumn, relation.ChildTable, relation.ChildColumn);
    }
    if (columns.Count == 0) {
        foreach (string table in tables) {
            columns.Add(table, GetColumnNames(table));
        }
    }
    if (connected) Disconnect();
}
```

Figure 2b.5 - Populate method for Database Class

The method, Populate, as depicted above utilises the method GetTableNames to instantiate a unique MySqlAdapter object for each table within the connected database. The adapters are then used to populate the DataSet object, which is identified with the private member identifier, data. As you can see, the Populate method also conforms to the generalised theme surrounding the spike's implementation.

3. The next task was to ensure that a DataRelation object could be instantiated, representing the entity relationships within the database. This spike works with two tables that are related, and they are multieventrule and multieventcompetition. There is a many to one relationship, where one entry in multieventcompetition may refer to many entries in multieventrule. One of the requirements of this spike was to utilise the DataRelation class appropriately in order to represent the relationship using objects. The code below offers and insight into how this was achieved.



```
public void CreateRelation(string name, string ptable, string pcol, string ctable, string ccol) {
    DataTable parent = data.Tables[ptable];
    if (parent == null) return; // Invalid table name specified
    DataTable child = data.Tables[ctable];
    if (child == null) return; // Invalid table name specified
    DataRelation relation = new DataRelation(name, parent.Columns[pcol], child.Columns[ccol]);
    data.Relations.Add(relation);
    foreach (RelationData r in relations) {
        if (r.Name == name) return;
    }
    RelationData rel = new RelationData(name, ptable, pcol, ctable, ccol);
    relations.Add(rel);
}
```

Figure 2b.6 - CreateRelation method for Database Class

The method, CreateRelation defined above, is passed five parameters, the four most important ones specify both the parent and child table's names as well as the column within the parent that needs to be matched within the child. There is some basic error handling performed by the method, just to ensure that the tables specified actually exist. If both the parent and child tables exist, the DataRelation object is instantiated and added to a private class member called data (data is a DataSet object).

4. The next critical task was to ensure that the data contained within a single DataSet object could be manipulated. After successfully manipulating any of the DataTables within the DataSet, it was required to ensure that those changes could be pushed, and ultimately reflected by the connected database. This task also factored the importance of the role that the DataRelation object plays. The DataRelation object enforces entity relationships. The role of the DataRelation object is to ensure that any changes made to a DataTable within the DataSet conform to any relationships present within the entities that are contained within the DataSet. The outcome of this task incorporated functionality for inserting, updating and deleting data from a DataTable within a DataSet and having the changes applied to the connected database. Each is defined briefly below, with code snippets provided for each.

```
public bool InsertRow(Dictionary<string, object> row, string table) {
    DataTable t = data.Tables[table];
    if (t == null) return false; // Invalid table name specified
    DataRow r = t.NewRow();
    foreach (KeyValuePair<string, object> entry in row) {
        r[entry.Key] = entry.Value;
    }
    // TODO: This can probably be removed
    adapters[table].InsertCommand = new MySqlCommandBuilder(adapters[table]).GetInsertCommand();
    data.Tables[table].Rows.Add(r);
    try {
        adapters[table].Update(t);
    } catch (Exception e) {
        #if VERBOSE
            Console.WriteLine(e.Message);
        #endif
    }
    Populate();
    return true;
}
```

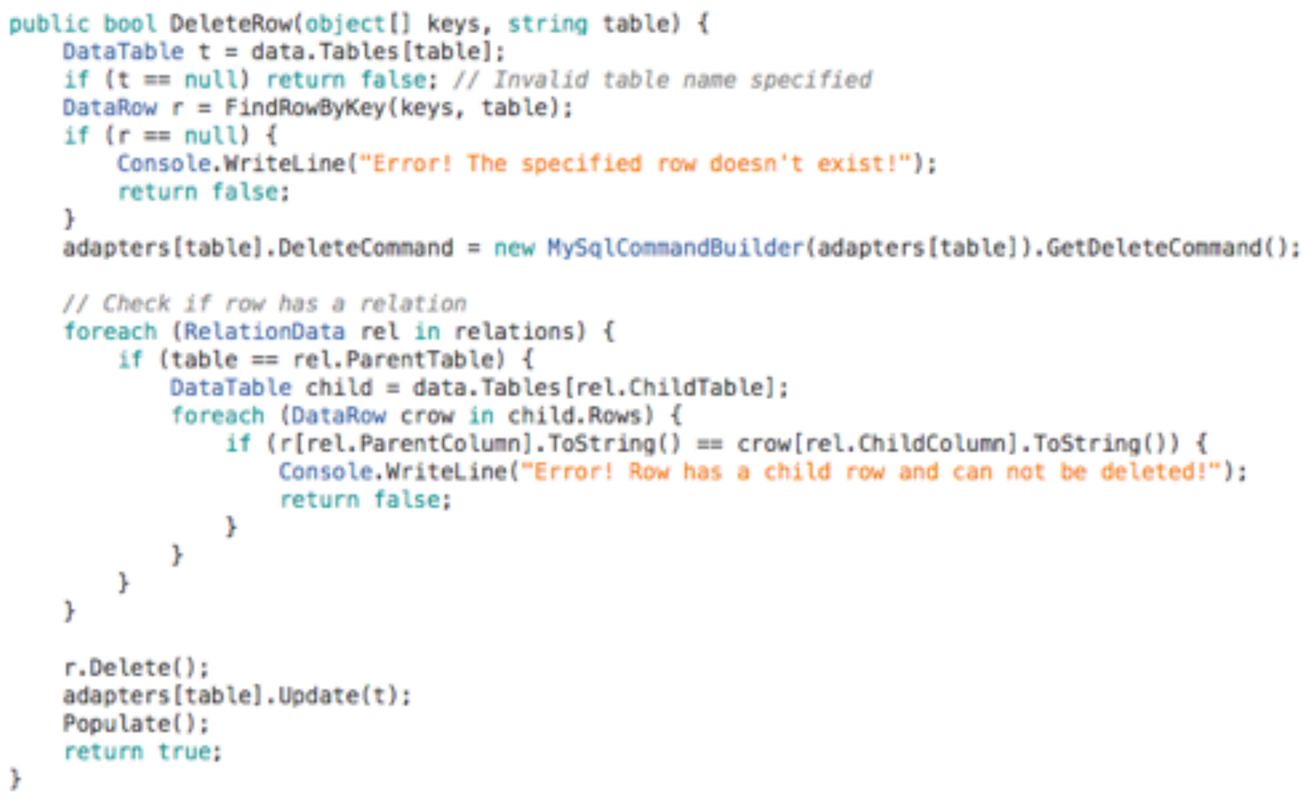
Figure 2b.7 - InsertRow method for Database Class

The InsertRow method, defined above, is used to insert a new row into a DataTable object contained within a DataSet object. After the insertion has been completed successfully, the appropriate DataAdapter is used to update the connected database, pushing the changes to the DataSet to the live database. After the changes are pushed, the private method Populate is called so that the DataSet object and the DataRelation object both reflect the changes and current state of the connected database. This method returns a bool based on the outcome. I.e. If the insert was successful, true is returned, otherwise it will return false.

```
public bool UpdateRow(object[] keys, Dictionary<string, object> row, string table) {
    DataTable t = data.Tables[table];
    if (t == null) return false; // Invalid table name specified
    DataRow r = FindRowByKey(keys, table);
    foreach (KeyValuePair<string, object> entry in row) {
        r[entry.Key] = entry.Value;
    }
    adapters[table].UpdateCommand = new MySqlCommandBuilder(adapters[table]).GetUpdateCommand();
    adapters[table].Update(t);
    Populate();
    return true;
}
```

Figure 2b.8 - UpdateRow method for Database Class

The UpdateRow method, defined above, is used to update data in a DataTable object contained within a DataSet object. After the update has been completed successfully, the appropriate DataAdapter is used to update the connected database, pushing the changes to the DataSet to the live database. After the changes are pushed, the private method Populate is called so that the DataSet object and the DataRelation both reflect the changes and current state of the connected database. This method returns a bool based on the outcome. I.e. If the update was successful, true is returned, otherwise it will return false.

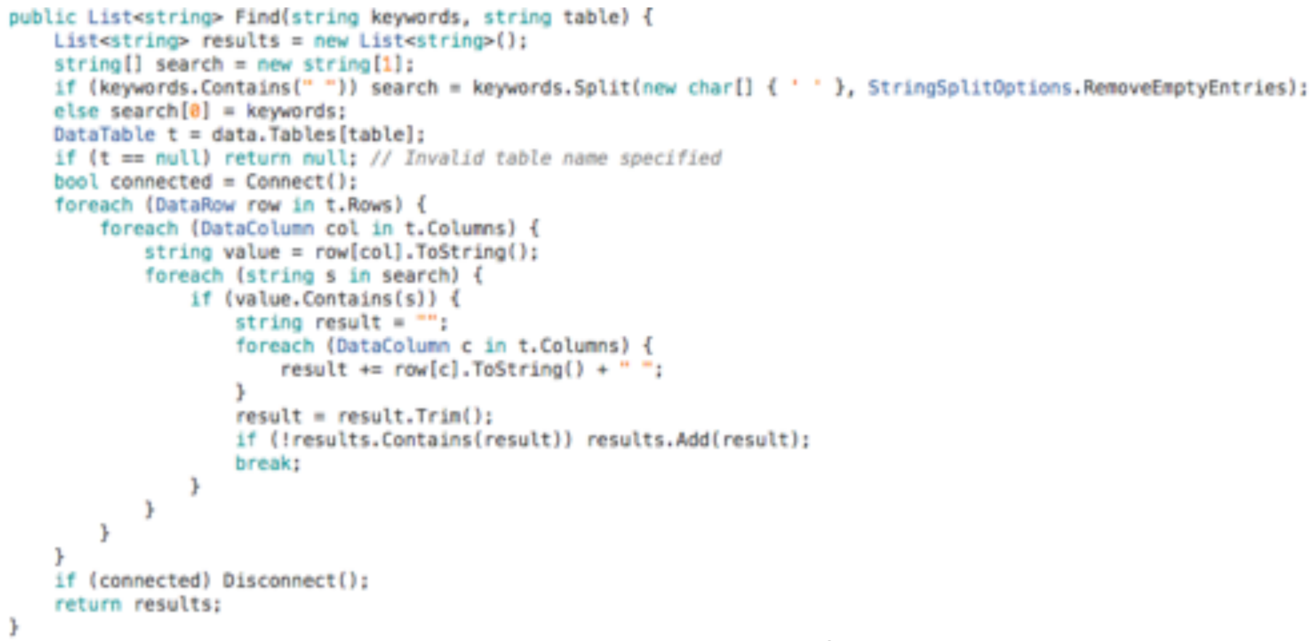


```
public bool DeleteRow(object[] keys, string table) {
    DataTable t = data.Tables[table];
    if (t == null) return false; // Invalid table name specified
    DataRow r = FindRowByKey(keys, table);
    if (r == null) {
        Console.WriteLine("Error! The specified row doesn't exist!");
        return false;
    }
    adapters[table].DeleteCommand = new MySqlCommandBuilder(adapters[table]).GetDeleteCommand();

    // Check if row has a relation
    foreach (RelationData rel in relations) {
        if (table == rel.ParentTable) {
            DataTable child = data.Tables[rel.ChildTable];
            foreach (DataRow crow in child.Rows) {
                if (r[rel.ParentColumn].ToString() == crow[rel.ChildColumn].ToString()) {
                    Console.WriteLine("Error! Row has a child row and can not be deleted!");
                    return false;
                }
            }
        }
    }

    r.Delete();
    adapters[table].Update(t);
    Populate();
    return true;
}
```

Figure 2b.9 - DeleteRow method for Database Class

The DeleteRow method, defined above, is used to delete data in a DataTable object contained within a DataSet object. After the delete has been completed successfully, the appropriate DataAdapter is used to update the connected database, pushing the changes to the DataSet to the live database. After the changes are pushed, the private method Populate is called so that the DataSet object and the DataRelation both reflect the changes and current state of the connected database. This method returns a bool based on the outcome. I.e. If the delete was successful, true is returned, otherwise it will return false.

5. To conclude, the final task was to ensure that data could be returned, or selected from a DataTable object contained within a DataSet object. The code snippet portrayed in the figure below demonstrates how this functionality was achieved.

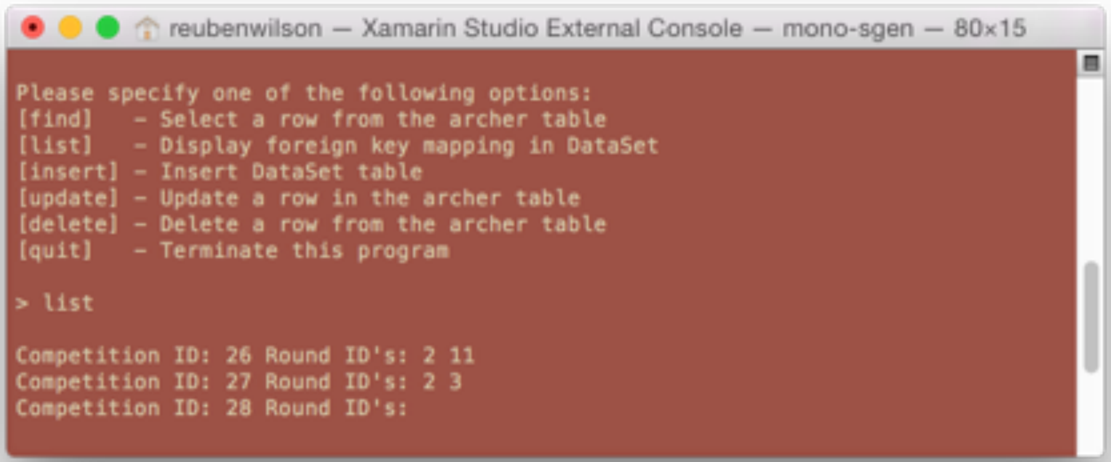


```
public List<string> Find(string keywords, string table) {
    List<string> results = new List<string>();
    string[] search = new string[1];
    if (keywords.Contains(" ")) search = keywords.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
    else search[0] = keywords;
    DataTable t = data.Tables[table];
    if (t == null) return null; // Invalid table name specified
    bool connected = Connect();
    foreach (DataRow row in t.Rows) {
        foreach (DataColumn col in t.Columns) {
            string value = row[col].ToString();
            foreach (string s in search) {
                if (value.Contains(s)) {
                    string result = "";
                    foreach (DataColumn c in t.Columns) {
                        result += row[c].ToString() + " ";
                    }
                    result = result.Trim();
                    if (!results.Contains(result)) results.Add(result);
                    break;
                }
            }
        }
    }
    if (connected) Disconnect();
    return results;
}
```

Figure 2b.10 - Find method for Database Class

The method, Find as depicted above, takes two parameters. The first represents the keywords that need to be matched against the data that needs to be located, the second of which represents the name of the table that the data needs to be retrieved from. The method caters for the event where more than one result will be returned, returning a List object, containing strings, each string representing a single retrieved row.

What we found out:

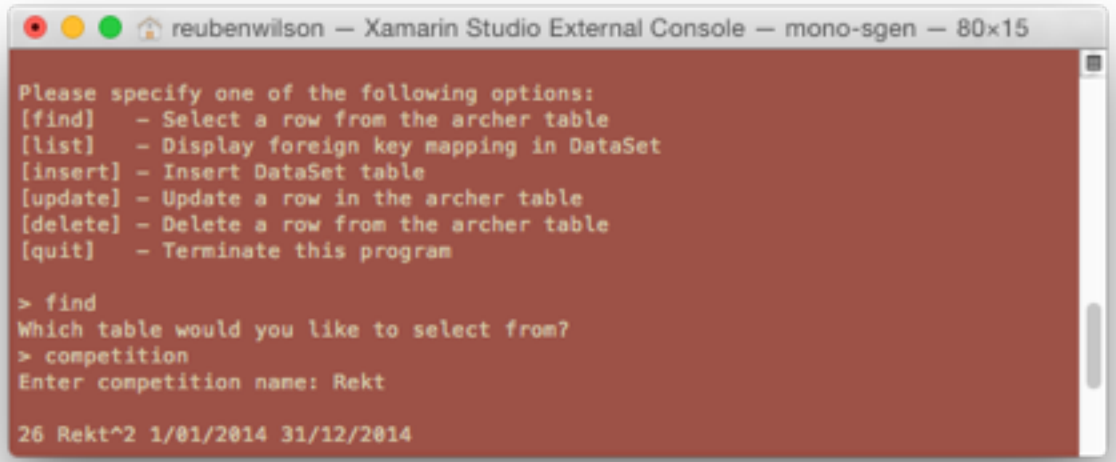


reubenwilson — Xamarin Studio External Console — mono-sgen — 80×15

```
Please specify one of the following options:
[find]   - Select a row from the archer table
[list]   - Display foreign key mapping in DataSet
[insert] - Insert DataSet table
[update] - Update a row in the archer table
[delete] - Delete a row from the archer table
[quit]   - Terminate this program

> list

Competition ID: 26 Round ID's: 2 11
Competition ID: 27 Round ID's: 2 3
Competition ID: 28 Round ID's:
```

Figure 2b.11: Results from running 'list' option

The output represented above demonstrates the program's output set (collected from the tables multieventcompetition and multieventrule) when the 'list' menu command is run. It shows demonstrates the one to many relationship between the two tables.



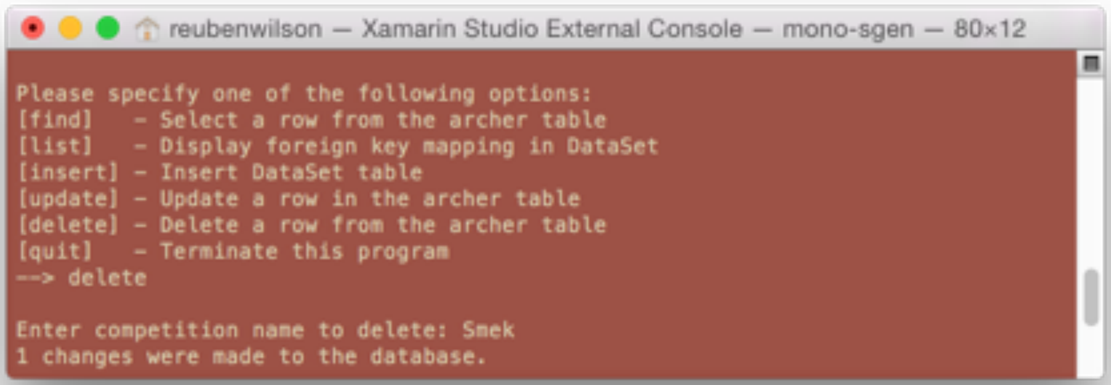
reubenwilson — Xamarin Studio External Console — mono-sgen — 80×15

```
Please specify one of the following options:
[find]   - Select a row from the archer table
[list]   - Display foreign key mapping in DataSet
[insert] - Insert DataSet table
[update] - Update a row in the archer table
[delete] - Delete a row from the archer table
[quit]   - Terminate this program

> find
Which table would you like to select from?
> competition
Enter competition name: Rekt

26 Rekt^2 1/01/2014 31/12/2014
```

Figure 2b.12: Results from running 'find' option

The output represented above demonstrates the program's output following a successful population of a DataSet object with a result set returned from the database after invoking a simple select query. This feature is invoked with the 'find' menu option is invoked.

The output generated by invoking the list options 'insert', 'update' and 'delete' is all similar. and is depicted in the figure below.



reubenwilson — Xamarin Studio External Console — mono-sgen — 80×12

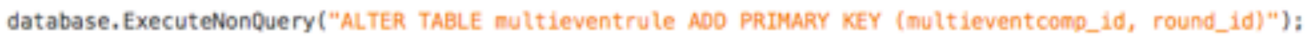
```
Please specify one of the following options:
[find]   - Select a row from the archer table
[list]   - Display foreign key mapping in DataSet
[insert] - Insert DataSet table
[update] - Update a row in the archer table
[delete] - Delete a row from the archer table
[quit]   - Terminate this program
--> delete

Enter competition name to delete: Smek
1 changes were made to the database.
```

Figure 2b.13: Results from running 'delete' option

The output represented above is generated from running the 'delete' menu option. A result set from a database query is used to populate a DataSet object. A user specified row from within the DataSet's DataTable is deleted and the changes to committed to the database. The 'insert' and 'update' menu options function in a similar manner and therefore are not portrayed here.

Open issues/risks:

The major issue that the team uncovered was the fact the multieventrule table does not have a primary key. This caused issues when trying to instantiate a DataRelation object, as DataRealtion objects require that the tables have primary keys. A simple work around was implemented to cater for this issue. The workaround includes updating the multieventrule table and applying a primary key constraint. The code snippet below depicts the query that was executed in order to apply the primary key constraint.



```
database.ExecuteNonQuery("ALTER TABLE multieventrule ADD PRIMARY KEY (multieventcomp_id, round_id)");
```

Figure 2b.14: Alter multieventrule

Recommendations:

The team has concluded that a more thorough design phase for the entities and their relationships should have been conducted and that in future, when designing databases, all appropriate constraints should be enforced where required.