Homework #7

Complete By: Wednesday, November 23rd @ 11:59pm
Assignment: completion of N-tier CTA Ridership Analysis app
Policy: Individual work only, late work accepted
Submission: electronic submission via Blackboard

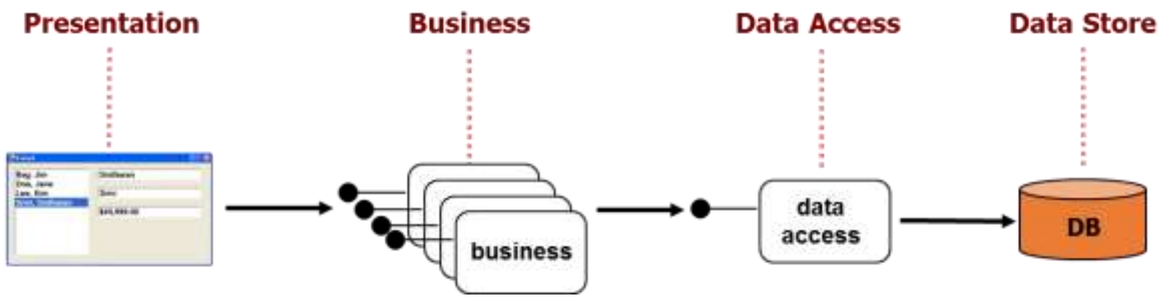
Assignment

The previous homework (HW #6) focused on building a data-driven GUI application accessing CTA Ridership data. If you look back on this application, it's probably a mess of GUI + C# + SQL code. How much of that code is redundant? How easily can you switch from one database file to another? Or imagine if you had to build a different client-side user interface, such as a console app or a web app. How easily could you extract and reuse the database access code vs. the GUI display code?

In this homework you're going to redo the solution to HW #6, taking a much more disciplined approach. We discussed the approach in class on 11/16 (PDF, PPT), but I'll also summarize here in the next section. We will provide the design, along with a skeleton code implementation. Your job will be to fill in the skeleton, using bits and pieces of code you have already written. You'll also need to rewrite the GUI.

N-tier Design

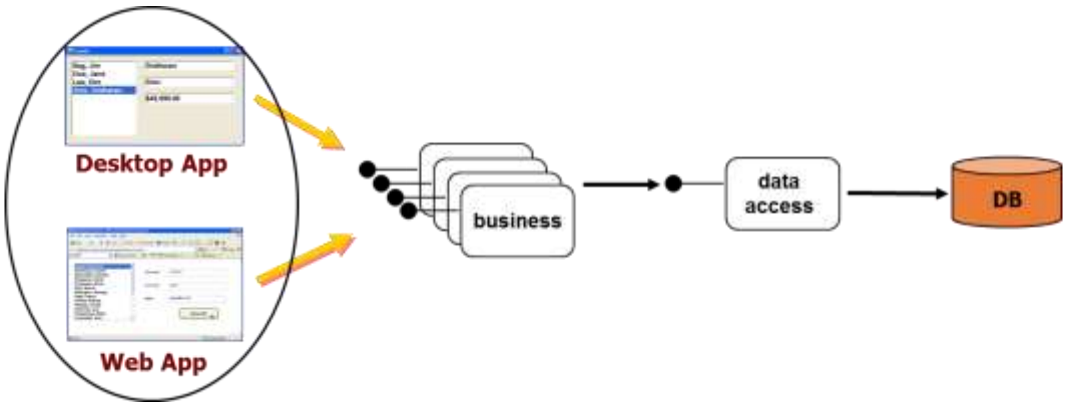
Data-driven applications, whether on a phone, laptop, or a web server, are generally built from at least 2 layers of software: the **front-end user interface** that interacts with the user, and the **back-end data access** code that interacts with the data store (typically a database). Since most organizations offer a variety of apps — imagine a bank that has customer-facing software as well as employee-facing software — the standard design consists of four layers or **tiers**:



Here's a summary of what each tier does:

1. **Presentation**: interacts with the user
2. **Business**: supports the security, business rules, and data processing needed by this particular application. [*What's a business rule? Imagine a sales app; there would be rules about how to charge sales tax, who gets discounts, etc*.]
3. **Data Access**: interface between business tier and data store — the data access tier does not manage nor store data, it only facilitates access to the data
4. **Data Store**: actual data repository

This approach has numerous benefits. For example, if 2 different apps access the same data store, they can use the same Data Access Tier (DAT) to access the data store. As another example, suppose you need to support the same app on both a desktop and a mobile device. Then you should be able to reuse the Data Access and Business Tiers, and need only build 2 different Presentation Tiers:

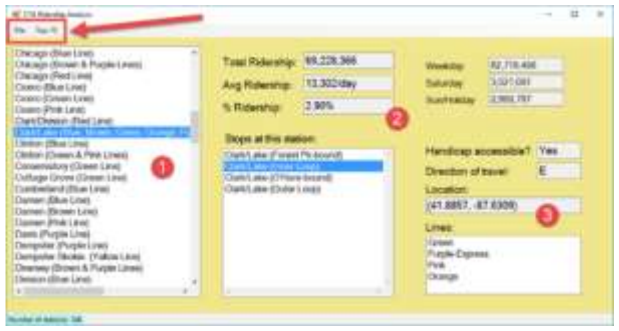


That's the theory anyway :-) But most organizations practice this approach in data-driven apps.

Step 1: Your solution or ours?

[*If you worked in class on Friday 11/18, you may have already done this. If so, skip to the next section.*]

In HW #7 we're going to rewrite the CTA Ridership program using an N-tier design. The first decision is whether to build off your solution, or our solution. Once you decide, do the following:



1. Create a folder on your desktop called "hw07".
2. If you plan to build on your solution for the next HW, make a copy of your entire Visual Studio solution folder, and place this copy into the "hw07" folder you just created. If you plan to use our solution, download a copy from the course web site "Homeworks\hw06-solution\CTA-Analysis.zip". Open the .zip, extract the "CTA-Analysis" folder and store the extracted folder into the "hw07" folder. Delete the .zip file.
3. At this point, your "hw07" folder should contain a solution to HW #6. Is there a CTA database file

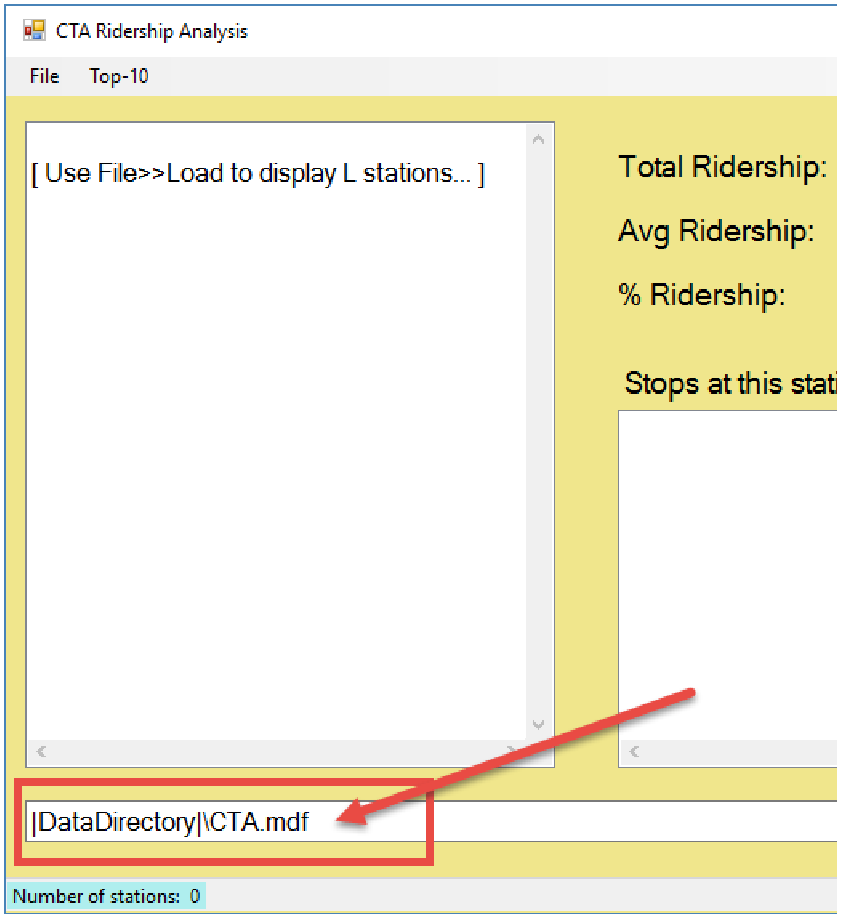
installed? If you downloaded our solution, you will need to find a local copy of the CTA database files (.mdf and .ldf) and copy them down into the bin\Debug sub-folder. If necessary, you can download copies of the CTA datbase from "hw06-files" on the course web page; don't forget you'll need to extract, then connect via Server Explorer in order to upgrade the files to your version of SQL Server.

4. Back in "hw07" folder, open the program in Visual Studio. Run and confirm it works. Do not continue until you have a working solution to HW #6, either yours or ours.

Before continuing with Step 1.5, you must have a working solution to HW #6.

Step 1.5: Add a text box to make life easier for the TAs

The TAs use a different database when they grade, so it's very helpful if it's easy to retarget the app towards a different database. Notice the provided solution has a text box along the bottom of the window with the path to the database file:



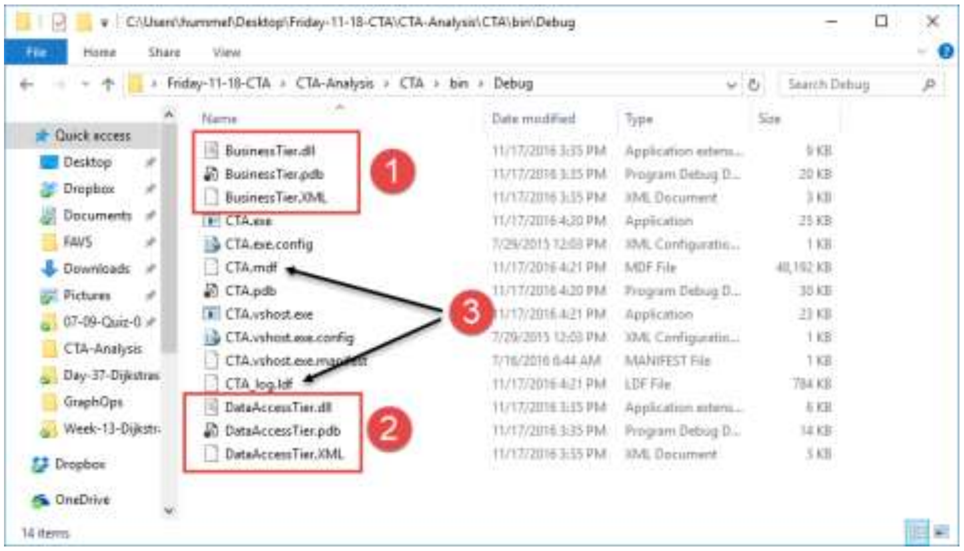
If you are using our solution, the text box is already there — skip to the next section. If you are using your own solution, please add a text box to the bottom of your main window, and initialize to "|DataDirectory|\CTA.mdf", without the quotes. Name the text box something like **txtDatabaseFilename**.

Step 2: Start Rewriting the GUI

[*If you worked in class on Friday 11/18, you may have already done this. If so, skip to the next section.*]

Step 2 is to re-write some of the GUI using the provided N-Tier design. You must use the design as given — we are providing the code in compiled form initially so that it cannot be changed. You are not only rewriting the GUI, but also learning the API of the provided Business Tier.

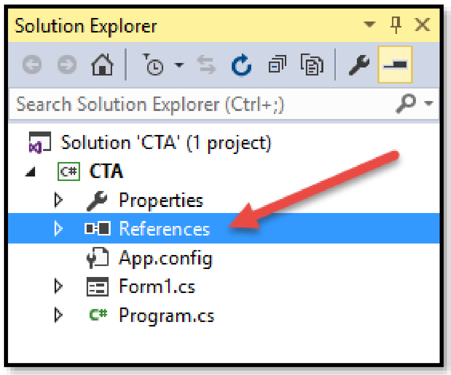
Right now you have 2 tiers: the presentation and the data store, where the presentation is doing all the work. We're going to add 2 more tiers — Business and Data Access — and divide the workload. Browse back to the course web page, in particular "Homeworks\hw07-files\Step02". You'll see 6 files, download all 6 files to your machine — the goal is to download and move to the bin\Debug project folder, i.e. the same folder that contains the database. When you're done, the bin\Debug folder should look like this:



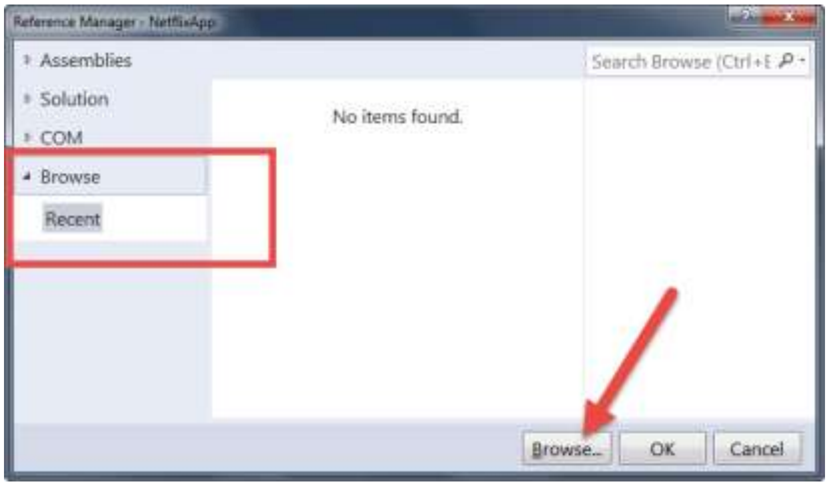
Name	Date modified	Type	Size
BusinessTier.dll	11/17/2016 3:35 PM	Application extens...	9 KB
BusinessTier.pdb	11/17/2016 3:35 PM	Program Debug D...	20 KB
BusinessTier.XML	11/17/2016 3:35 PM	XML Document	3 KB
CTA.exe	11/17/2016 4:20 PM	Application	25 KB
CTA.exe.config	7/29/2015 12:03 PM	XML Configuratio...	1 KB
CTA.mdf	11/17/2016 4:21 PM	MDF File	40,192 KB
CTA.pdb	11/17/2016 4:20 PM	Program Debug D...	30 KB
CTA.vshost.exe	11/17/2016 4:21 PM	Application	23 KB
CTA.vshost.exe.config	7/29/2015 12:03 PM	XML Configuratio...	1 KB
CTA.vshost.exe.manifest	7/16/2016 6:44 AM	MANIFEST File	1 KB
CTA_log.ldf	11/17/2016 4:21 PM	LDF File	784 KB
DataAccessTier.dll	11/17/2016 3:35 PM	Application extens...	6 KB
DataAccessTier.pdb	11/17/2016 3:35 PM	Program Debug D...	14 KB
DataAccessTier.XML	11/17/2016 3:35 PM	XML Document	5 KB

Business tier is #1, Data Access tier is #2, and CTA database files are #3. Note that "DLL" stands for dynamic linked library, i.e. compiled code that is dynamically loaded and linked into your program. If you also have a copy of the database in bin\Release, copy the DLLs and related files into bin\Release.

Back in Visual Studio, look at your Solution Explorer window (upper-right typically). If not visible, use View menu to expose. Expand your project, and you'll see a small folder named "References":

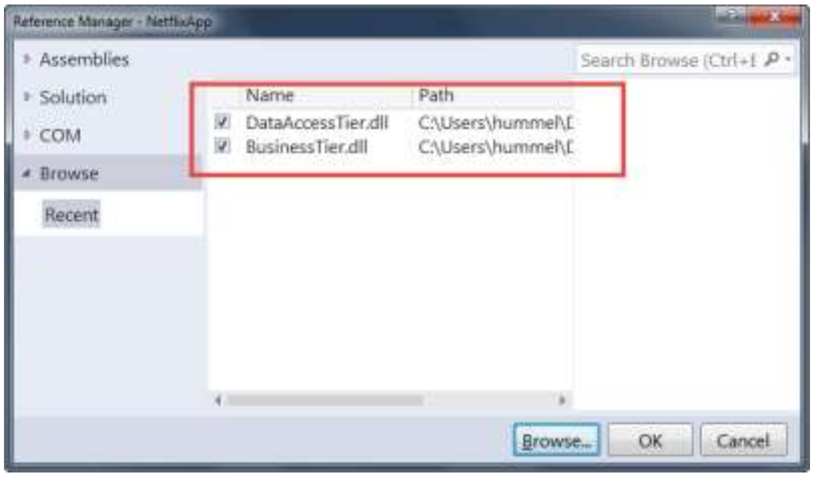


Right-click, select “Add Reference”, and in the dialog that opens, click on the “Browse” tab along the left. Now click the “Browse” button at the bottom:



No items found.

The file dialog should start at the top-level of your project folder, so drill down to the bin\Debug sub-folder that contains the 2 DLLs that you copied earlier — BusinessTier.dll and DataAccessTier.dll. Select them both, and click the “Add” button. You should see this:



	Name	Path
✓	DataAccessTier.dll	C:\Users\hummel\D
✓	BusinessTier.dll	C:\Users\hummel\D

Make sure there are checkmarks next to both DLLs. Click “OK”, and your program will now have access to the compiled classes inside these DLLs. You’ll see the DLLs listed under the “References” folder in the Solution Explorer.

Run your program — it should still work, nothing has changed other than you have additional functionality you can use when you’re ready.

At this point, rewrite the following portions of the GUI to use the functionality provided by the Business and Data Access tiers:

1. In the Form_Load event, perform a test connection to startup SQL Server.
2. Get the stations using the Business Tier.
3. When the user selects a station, get the stops using the Business Tier.

4. When the user requests the top-10 stations, use the Business Tier to retrieve the stations.

For example, here's how to test the database connection in #1:

```
try
{
  string filename = this.txtDatabaseFilename.Text;

  BusinessTier.Business bizTier;
  bizTier = new BusinessTier.Business(filename);

  bizTier.TestConnection();
}
catch
{
  //
  // ignore any exception that occurs, goal is just to startup
  //
}
```

Now complete #2, #3, and #4 to use the Business tier to display the stations, stops, and top-10. When you are done, the app should work exactly as before, only now some of the app is written using the Business tier, while the rest is executing SQL directly.

Step 2.5: Business Tier API

As a reference, here are the functions available in the provided Business tier:

```
public class Business
{
  ///
  /// <summary>
  /// Constructs a new instance of the business tier.  The format
  /// of the filename should be either |DataDirectory|\filename.mdf,
  /// or a complete Windows pathname.
  /// </summary>
  /// <param name="DatabaseFilename">Name of database file</param>
  ///
  public Business(string DatabaseFilename)

  ///
  /// <summary>
  ///  Opens and closes a connection to the database, e.g. to
  ///  startup the server and make sure all is well.
  /// </summary>
  /// <returns>true if successful, false if not</returns>
  ///
  public bool TestConnection()

  ///
  /// <summary>
```

```
    /// Returns all the CTA Stations, ordered by name.
    /// </summary>
    /// <returns>Read-only list of CTAStation objects</returns>
    ///
    public IReadOnlyList<CTAStation> GetStations()

    ///
    /// <summary>
    /// Returns the CTA Stops associated with a given station,
    /// ordered by name.
    /// </summary>
    /// <returns>Read-only list of CTAStop objects</returns>
    ///
    public IReadOnlyList<CTAStop> GetStops(int stationID)

    ///
    /// <summary>
    /// Returns the top N CTA Stations by ridership,
    /// ordered by name.
    /// </summary>
    /// <returns>Read-only list of CTAStation objects</returns>
    ///
    public IReadOnlyList<CTAStation> GetTopStations(int N)

  }//class
```

The Business tier also defines classes for the objects being returned: **CTAStation** and **CTAStop**. Here are their definitions:

```
  ///
  /// <summary>
  /// Info about one CTA station.
  /// </summary>
  ///
  public class CTAStation
  {
    public int ID { get; private set; }
    public string Name { get; private set; }

    public CTAStation(int stationID, string stationName)
    {
      ID = stationID;
      Name = stationName;
    }
  }

  ///
  /// <summary>
  /// Info about one CTA stop.
  /// </summary>
  ///
  public class CTAStop
  {
    public int ID { get; private set; }

    public string Name { get; private set; }

    public int StationID { get; private set; }

    public string Direction { get; private set; }

    public bool ADA { get; private set; }

    public double Latitude { get; private set; }
    public double Longitude { get; private set; }

    public CTAStop(int stopID, string stopName, int stationID, string direction, bool
ada, double latitude, double longitude)
    {
      ID = stopID;
      Name = stopName;
      StationID = stationID;
      Direction = direction;
      ADA = ada;
      Latitude = latitude;
      Longitude = longitude;
    }
  }
```

In C#, it's common for objects to make data available in a read-only fashion. This is commonly done using a public "Getter" function, but a private "Setter". In C#, the calls to the "Getter" functions are generated by the compiler, so the programmer can think in terms of data instead of function calls. For example, notice that stations have a public **Name property**. We use this to get the name for display purposes, as highlighted below:

```
      try
      {
        BusinessTier.Business bizTier;

        bizTier = new BusinessTier.Business(this.txtDatabaseFilename.Text);

        var stations = bizTier.GetStations();

        foreach (var station in stations)  // display stations:
        {
          this.lstStations.Items.Add(station.Name);
        }
      }
      catch (Exception ex)
      {
        string msg = string.Format("Error: '{0}'.", ex.Message);
        MessageBox.Show(msg);
      }
```

the database connection each time it is called by the Business Tier.

Step 3: Implement Business and Data Access Tiers

At this point you should have a working Presentation tier that is calling into the Business Tier. It has not been completely rewritten, but now it's time to write the Business and Data Access Tiers yourself. You'll then extend the Business Tier as you see fit to complete the rewriting of the GUI.

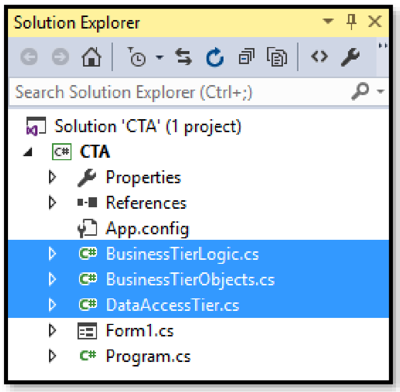
Back in Visual Studio, view the Solution Explorer. Expand the "References" folder, right-click on "DataAccessTier.dll", and select "Remove". Then right-click on "BusinessTier.dll" and remove that DLL as well.

Note that this removes access to the DLLs, but does *not* delete the files themselves. To be safe, minimize Visual Studio, locate your project folder, drill down into the bin\Debug sub-folder, and delete the DLLs "BusinessTier.dll" and "DataAccessTier.dll". If you copied these files into bin\Release, delete the files from that folder as well.

Back in Visual Studio, try to run your program — it should fail to run due to missing class definitions. This is a good thing :-) Minimize Visual Studio. Now browse to the course web page, under "Homeworks", open "hw07-files", open "Step03", and download the three C# source files:

1. BusinessTierLogic.cs
2. BusinessTierObjects.cs
3. DataAccessTier.cs

Copy these files into your project folder —**<u>the same folder that contains your other C# source files</u>**. Back in Visual Studio, let's add these C# files to your project: Project menu, select "Add Existing Item…", select all three files, and click "Add". The files should now appear in the Solution Explorer:



Now try again to run your program — it should compile and run. However, when you click the buttons nothing should happen, because these C# files are just skeletons — they don't actually do anything. Note that your program should not crash, since you should be handling scenarios where the business tier returns null or empty lists; if your program crashes, this is a symptom that you missed an error check in the Presentation tier.

I'd recommend starting with the Data Access Tier since it contains only 3 functions, and it's very clear what each function does. The provided code sets up the connection string for you in a private field, so each function needs to open a connection, perform the operation, close the connection, and return the result. See the comments in "DataAccessTier.cs" for more information. <u>Do not add change the API</u> in any way — do not change / add parameters, and do not change the return types. The job of the Data Access Tier is to execute the SQL as given, that's it. [*For help with function #3, which executes an action query — i.e. insert, update, or delete — see the lecture notes from day 33, Monday 11/07 (<u>PDF</u>, <u>PPT</u>).*]

The last step is to implement the Business Tier. The Business Tier is more complex, spread across two source files and 4 classes:

1. BusinessTierLogic.cs
2. BusinessTierObjects.cs

The 2nd file, “BusinessTierObjects.cs” contains the definitions of the Data Transfer Objects: **CTAStation** and **CTAStop** for now. These classes are already written — do not modify them. In the other file, “BusinessTierLogic.cs”, implement the 3 functions as given:

1. GetStations
2. GetStops
3. GetTopStations

The provided code creates a Data Access tier object, and stores a reference in a private field. For each function, you job is to create the appropriate SQL query, execute it via the Data Access tier, and then process the results — typically by creating objects, adding them to a list, and returning this list when it’s done. Once again, do not add change the API in any way — do not change / add parameters, and do not change the return types. This holds for all classes in the Business Tier; work within the design as given since we will be grading against the API using our UI, not yours.

As discussed in class, the Business tier is the interface between the Presentation tier and the Data Access tier. The functions in “BusinessTierLogic.cs” do not execute the SQL themselves. Instead, they create the SQL, and then call into the Data Access Tier for execution. The Data Access tier returns the appropriate result — scalar value or DataSet or integer — and then the Business tier transfers this data back to the Presentation tier using one or more Data Transfer Objects. For this reason, the functions in “BusinessTierLogic.cs” will need to create an instance of the Data Access tier, and then call the appropriate method. Here’s the idea:

```
DataAccessTier.Data dataTier = new DataAccessTier.Data(...);

string sql    = "Select ...";
object result = dataTier.ExecuteScalarQuery(sql);
```

The provided Business tier code already creates an instance of the Data Access tier; see the constructor in the **Business** class. Use this private instance to call the Data Access tier.

Step 4: Extend the Business Tier…

The next step is to rewrite the remainder of the GUI so that it calls into the Business tier instead of executing SQL itself. This means you will need to add additional functions to the Business API defined in “BusinessTierLogic.cs”, and you may also need to define additional objects in “BusinessTierObjects.cs” to transfer data. The design is up to you, but the idea is to follow in the spirit of what we are doing. In other words, the GUI may *not* pass SQL to the Business tier for execution. Instead, the GUI should call one or more functions in the Business tier to retrieve the data it needs, with no understanding of how this is done / CTA database. How many functions you have, and how the data is passed back and forth, is up to you.

When you are done, there should be no SQL code in your Presentation tier. You’ll know you are done rewriting the Presentation tier when you can delete the following using statement from the Form1.cs file:

~~**`using System.Data.SqlClient;`**~~

If removing this statement causes an error, then the GUI is still accessing the database directly, and you are not yet done.

Step 5: Add a few additional features to your app…

To make your design more interesting, add the following features to your application:

1. Add a “find” feature where the user can enter the partial name of a station, e.g. “Belmont”. In response, the app displays all stations that contain “Belmont” in the name. [*Hint: in SQL, you can use the* Like *operator with the pattern ‘%Belmont%’ to do this kind of search.*]
2. Display the top-10 stations based on weekday ridership. Likewise, an option to display the top-10 based on Saturday ridership. And an option to display top-10 for Sunday/Holiday ridership.
3. Allow the user to change the ADA field of a stop — i.e. whether that stop is handicap accessible or not. This is important because construction often temporarily closes handicap access. This change should “stick”, i.e. actually update the database, not just the GUI.

Electronic Submission

First, add a header comment to the top of your C# Form source code file (“Form1.cs”), along the lines of:

```
//
// N-tier C# and SQL program to analyze CTA Ridership data.
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS341, Fall 2016
// Homework 7
//
```

Exit out of Visual Studio, and find your project folder. Drill down into the bin\Debug and bin\Release folders, and delete the database files from each folder (*.mdf and *.ldf). If you are unable to delete the files, this means Microsoft SQL Server is still running in the background; use Task Manager to end the background process named “SQL Server Windows NT”. Then create an archive (.zip) of this entire folder for submission: right-click, Send To, and select “Compressed (zipped) folder”. Finally, submit the resulting .zip file on Blackboard under the assignment “HW07”.

Your program should be readable with proper indentation and naming conventions; commenting is expected where appropriate. You may submit as many times as you want before the due date, but we grade the last version submitted. This implies that if you submit a version before the due date and then another after the due date, we will grade the version submitted after the due date — we will *not* grade both and then give you the better grade. We grade the last one submitted. In general, do not submit after the due date unless you had a non-working program before the due date.

Policy

Late work *is* accepted. Given that the on-time due date bumps up against the Thanksgiving holiday, you may submit as late as 72 hours after the deadline for a penalty of 25%. After 72 hours, no submissions will be accepted. That means you can submit as late as midnight, Saturday November 26th.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. via Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is described here:

http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at http://www.uic.edu/depts/dos/studentconductprocess.shtml .