**Exercise:**

In class we have been talking about object-oriented design using inheritance and interfaces. This assignment is a chance to practice OOD in the context of the real-world example: Database access using C#

* **Background:**

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 UI** that interacts with the user, and the **back-end DA** code that interacts with the data store (typically a database). Since most organization 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 of **tiers:**

Here’s a summary of what else 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.

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 anyways, but most organizations practice this approach to application design.

**The Assignment:**

The assignment is to design and build **Data Access** and **Business Tiers** for the Netflix database discussed in class. You have extensive freedom in how you design your classes, but the goal is to take advantage of inheritance, interfaces, and C# to enable Higher-end functionality in your final design. For example, most programmers working on the Presentation Tier will expect your design to allow one **foreach** through the result set of query:

**BusinessTier bt = new BusinessTier(“Netflix.sqlite”);**

**Users users = bt.Top10Reviewers();**

**foreach (User user in users)**

**Console.WriteLine(user.UserID + “: “ + user.NumReviews);**

Study this example very carefully… Notice first that there is no SQL visible—Presentation Tier programmers are not expected to know SQL. And even if they did, do you really want to embed database-specific details in the UI of the application? What if the Data Store Tier is changed in the future? So no Data Store-specific details in the Presentation Tier.

Second, how should the Business Tier return data from the Data Access Tier to the Presentation Tier? Arrays? Lists? Custom data structures? Structs? Classes? Even though the Netflix database only consists of Movies and MovieReviews, it also contains information about users. How should this information be represented? The code fragments above suggests a custom data structure (class) called **Users**, which is a collection of custom **User** objects (another class). The User class could be implemented from scratch, or note that it could inherit from the .NET System.Collection.Generic.List class—which would automatically enable foreach support (since you would inherit it!). Alternatively, a simpler approach would be to create just one custom User class, representing a single user, and then have the business tier return a List<User> as appropriate:

**BusinessTier bt = new BusinessTier(“Netflix.sqlite”);**

**List<User> users = bt.Top10Reviewers();**

**foreach(User user in users)**

**Console.WriteLine(user.UserID + “: “ + user.NumReviews)**

This works perfectly fine, the only drawback is that you are returning a List object back, which may have more functionality than you want to expose, and missing functionality that you would like to support.

**The Business Tier:**

The goal of the **Business Tier** is to provide access to data while hiding the implementation of details of the Data Store (e.g. hiding details about the database, such as tables and column names). Your Business Tier must provide the following functionality:

1. Ability to retrieve total # of movies

2. Ability to lookup a movie name from a movie id

3. Ability to lookup a movie id form a movie name

4. Ability to retrieve information about all the movies: ids, names number of reviews,   
 average rating, lowest rating, and highest rating;   
 - also the ability to foreach through this data

5. Ability to retrieve top-10 movies with the highest average ratings

6. Ability to retrieve total # of reviews

7. Ability to retrieve average rating across all the reviews.

8. Ability to retrieve information about a user based on user id: number of distinct movies   
 reviewed, number of reviews submitted, and average rating given by user across all

their reviews.

9. Ability to retrieve top-10 users who have submitted the most reviews

10. Ability to insert a new review into the database

How the information is conveyed into and out of the Business Tier is entirely up to you. For example, your “top-10 users” function could return an array of 10 userids, or a list of 10 user objects—whatever you prefer.

**The Data Access Tier:**

Think of the **Data Access Tier** as a generic .NET database engine. The Business Tier communicates with the Data Access Tier by forming and passing SQL strings, and the Data Access Tier executes this SQL against the data store and returns the results:

The Data Access Tier should be completely reusable from one application to the next. The Data Access Tier uses the ADO.NET objects and methods we discussed in class: **ExecuteNonQUery, ExecuteScalar,** and  **Filll.** Your DAT is free to return back scalars and DataSet objects, or scalars and DataTable onjects, or custom objects, whatever you want.

One important design consideration is the opening and closing of the database connection. Typically you would open the connection when an instance of your Data Access Tier is created, and then provide an explicit Close() method so users of your DAT can close the connection; this allows the connection to remove open across a series of data accesses, which is much more efficient. [*Like opening and closing a file vs. leaving it open.*] In other words, the Business Tier could do the following:

**DataAccessTier dat = new DataAccessTier(databasename);**

**var result1 = dat.ExecuteScalar(“Select …”);**

**var result2 = dat.ExecuteScalar(“Select …”);**

**.**

**.**

**.**

**dat.Close()**

When execution leaves the using block’s}—whether normally or because of an exception—the file’s **Dispose** method is actually called. The StreamReader class provides both Close() and Dispose() methods that do exactly the same thing. How can you enable “using” support for your own custom classes, such as the DataAccessTier? Implement the .NET interface **System.IDisposable.**

**Assignment Details:**

The course web page consists of two Netflix databases: the smaller 1MB file we used in class, and a larger 30MB file that can be used to stress-test your design. We are not concerned with efficiency in this exercise, but if time permits, you may want to see how well your design works on a more realistic database.

In terms of error handling, you are not required to do error handling in this exercise. You are encouraged of course to handle as many of the error cases you see fit: invalid movie ids, invalid movie names, invalid user ids, invalid filename for the database file, etc. The sheer number of possible errors is prohibitive in the time we have; as a rile of thumb, 90% of the real-world program code is error handling…

How will you go about testing your design? You will need to create some sort of testing framework—a rudimentary Presentation Tier?—to support the development and testing of your work. You will not be graded on the quality of your Presentation Tier, only on the quality of correctness of your Business and Data Access Tiers. But you must create and submit some sort of Presentation Tier/testing framework so we can test and evaluate your work. If we can’t run your submission against the Netflix database, we can’t grade it.

**Getting Started:**

Don’t even try to do it all at once. You certainly want to think about your design a bit before you start, but don’t spend a week thinking about the design and then start coding at the last minute. Most filks come up with an initial design, then start coding it up and see how well it works. Then they either continue or discard and start over. Some call this **prototype-based** design, others call it **agile** design. But the idea is to only design what you need right now to make functionality X work, and ignore everything else.

I would recommend picking one of the 10 requirements on page 3, then implementing that requirement from Presentation Tier down the Data Access Tier. What would you like the Presentation Tier to look like to call and retrieve the required data? That tells you how to design the Business tier, which is a separate class. Then focus on the Business Tier—what SQL do I need to retrieve the requested data? Based on what the SQL looks like, then you know what method you need in the Data Access Tier to execute it; note the Data Access Tier is yet another class.

**Electronic Submission:**

Using Blackboard (<http://uic.blackboard.com/>), submit your C# source files under assignment “HW8”. You will need to create a compressed folder /.zip file so that you can submit the files as a single entity. Before submission, your **Business** and **Data Access Tier** classes should contain a header comment along the lines of:

//

// N-Tier database app: Business Tier

// << Your Name >>

// U. of Illinois, Chicago

// CS341, Fall 2013

// Homework 8

//

Your individual methods should likewise contain header comments with a brief description of what functionality the method performs, along with other relevant information (parameters, return values, etc.). You may submit as many times as you want before the due date, but we see (and grade) only the last version submitted.