
Joel Landis' *.NET PORTFOLIO*

joeldlandis.com



Introduction

This document provides a description of the projects I worked on at SetFocus (<http://www.setfocus.com>) as part of the .NET Master's Program from September to December 2009. The projects used Microsoft Visual Studio 2008 along with Microsoft SQL Server 2008. The primary programming language was C# in the .NET Framework version 3.5.

All of the projects utilized multiple code layers (n-tier architecture). Code layer separation helps to promote code clarity, ease of maintenance, and reusability. Below is an example of how the layers were typically organized:

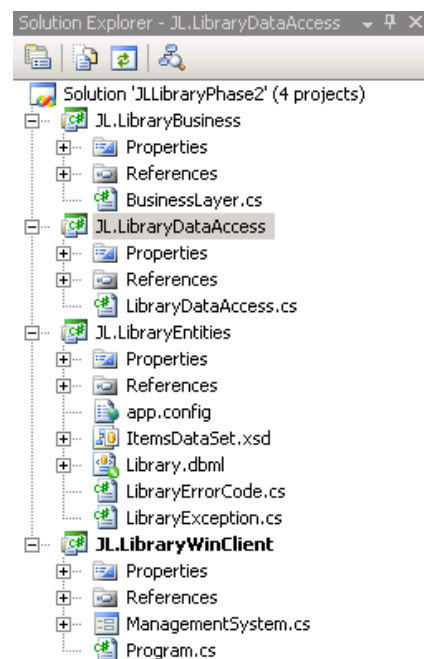
Presentation Layer: The presentation layer is the graphical user interface seen by the end user. It displays information to the user and relays input from the user. Examples include an ASP.NET web site interface or a Windows application. The presentation layer makes calls to the business layer to accomplish tasks and receives information back from the business layer.

Entities: Entities are objects with properties that describe something like a user or an item. The entities layer simply defines the entity objects which are then instantiated, populated, and passed back and forth between the other layers.

Business Layer: The business layer receives and processes information calls from the presentation layer and in turn makes calls to the data access layer as needed before returning information to the presentation layer for display.

Data Access Layer: The data access layer is what communicates with the database to store and retrieve data. It may utilize a combination of stored procedures in the database along with SQL commands or LINQ to SQL queries. Since this is the only layer to directly access the database, this layer could be rewritten later to use a different type of database without needing to rewrite the rest of the program.

Figure 1. A sample project.



.NET Framework Project

Overview

The first project focused on core programming skills and the features of the C# language. It was my job to translate detailed project specifications into code adhering to naming conventions, coding conventions, commenting, use of regions, and encapsulation.

The goal of the project was to build parts of the business tier for a retail company by creating and testing two assemblies. The first assembly was a class library containing interfaces and base classes. The second assembly contained various entity, collection and exception classes used by various business processes.

Properties

C# makes it easy to define class properties that can be written to or read from. It is possible to qualify the value being set, as shown in Figure 2.

```
/// <summary>
/// string field
/// </summary>
protected string companyName;
/// <summary>
/// Read and write property of type string named CompanyName
/// </summary>
public virtual string CompanyName
{
    get
    {
        return companyName;
    }
    set
    {
        if (value != null && value.Length >= 1 && value.Length <= 40)
            companyName = value;
        else
            throw new ArgumentOutOfRangeException("You must provide a value that is not
            longer than 40 characters for CompanyName.");
    }
}
```

Figure 2. Setting a custom property.

Interfaces

Interfaces are a contract or template that gets implemented in subsequent classes, ensuring that certain functionality will be met in the classes. This project implemented several interfaces that come with .NET, including `IList<T>`, `ICollection<T>`, `IComparable<T>`, `IComparer<T>`, `IEnumerable`, `IEnumerator`, and `ISerializable`. Custom interfaces were created for contact information, and to define and implement a custom collection.

`IList<T>` from `System.Collections.Generic` provides a way to access a collection of objects by index. `IList<Product>` was used in the `Products` class to provide access to `Product` objects in the collection by index.

Also part of `System.Collections.Generic`, `ICollection<T>` provides methods such as `Add`, `Remove`, `Contains`, `Count`, `CopyTo`, and `Clear` to manipulate generic collections. This was implemented in the `Products` class to add new `Product` objects to the collection.

Normally you can run a sort method on an object like an array, however, with a collection of objects, C# needs to know how to compare the objects in order to sort them. This is where `IComparable` comes in. Implementing a `CompareTo` method provides a default sorting mechanism when calling the `Sort` method on the collection. See Figure 3 for an example in the `Product` class.

```
public int CompareTo(Product p)
{
    if (p == null)
        throw new ArgumentException("Project object is null.");
    if (this.ID < p.ID)
        return -1;
    else
```

```
    if (this.ID > p.ID)
        return 1;
    else
        return 0;
}
```

Figure 3. An example of an `IComparable.CompareTo()` implementation providing a default comparison based on product id.

It may be desirable to provide multiple alternative means of sorting in addition to a default method. To do this you implement `IComparer`, which consists of a method which returns a nested class object implementing a `Compare()` method as shown in Figure 4. So to sort the `Products` collection by supplier id instead of product id, you could make a call like this `myProducts.Sort(GetSortByCategoryID());` where you pass in the `IComparer` object.

```
private class SortBySupplierID : IComparer<Product>
{
    public int Compare(Product x, Product y)
    {
        if (x == null && y == null) return 0;
        if (x == null) return -1;
        if (y == null) return 1;

        return x.SupplierID.CompareTo(y.SupplierID);
    }
}
public static IComparer<Product> GetSortBySupplierID()
{
    return new SortBySupplierID();
}
```

Figure 4. `IComparer` uses a method to return an instance of a nested `IComparer` class.

Just like it is necessary to use **`IComparable`** to sort a collection of objects, **`IEnumerable<T>`** provides a way to iterate over a collection of objects of a certain type in a `foreach` loop. As shown in Figure 5, a simple `yield return` statement gets the job done. For a `Suppliers` class, I implemented `IEnumerable` the “old” manual way where the `IEnumerable` `GetEnumerator()` method returns a new instance of a nested class object containing `IEnumerator` methods to iterate through the collection such as `Current()`, `MoveNext()`, and `Reset()`.

```
public IEnumerator<Product> GetEnumerator()
{
    foreach (Product p in productsCollection)
    {
        yield return p;
    }
}
```

Figure 5. `IEnumerable<T>` implementation.

The project implemented the **`ISerializable`** interface in several classes to provide serialization capabilities. This could allow the state of an object to be saved or restored as needed.

Custom Attributes and a Custom Exception Class

Custom attributes were created, allowing meta information such as developer info to be attached to classes in the project. A custom exception class was created inheriting from System.Exception which provided an additional property for the line number of the exception.

Library Phase 1: Windows Front-End Application

Overview

The purpose of this project was to create a Windows forms-based front-end and a business layer to support the principle functions of a lending library's day-to-day operations, including adding new members (adult and juvenile), and checking books in and out. Existing .dll files were used (no source code was provided) for the data access layer and for the entities objects such as members and items.

Graphical User Interface

The first thing I did was to sit down with the specifications, pencil and paper to design a user friendly interface. I wanted the application to be easy to use and have all of the features integrated nicely together. To accomplish that I used one form with a tabbed interface. Through the use of status text label areas (see Figure 9) I also avoided using modal forms of communication where you throw up a dialog box and force the user to make a decision before they can continue.

Title	Author	Checked Out	Due Date
The Adventures of Robin Ho...	Howard Pyle	11/7/2009	11/21/2009
Treasure Island	Robert Louis Stevenson	11/7/2009	11/21/2009
Julius Caesar's Commentarie...	Julius Caesar	11/7/2009	11/21/2009

Figure 6. Check Out

To check out an item (a book), you enter the Member ID. Tabbing over to the ISBN box triggers an event that populates the member information, including showing which items are currently on loan. Entering an ISBN, Copy Number, and clicking Verify Book brings up information about the item. If the item was never checked in previously, the Check In Now button can be used to check the item in before

it is checked out to the member. Once an item is checked out, the item is displayed in the items currently on loan grid.

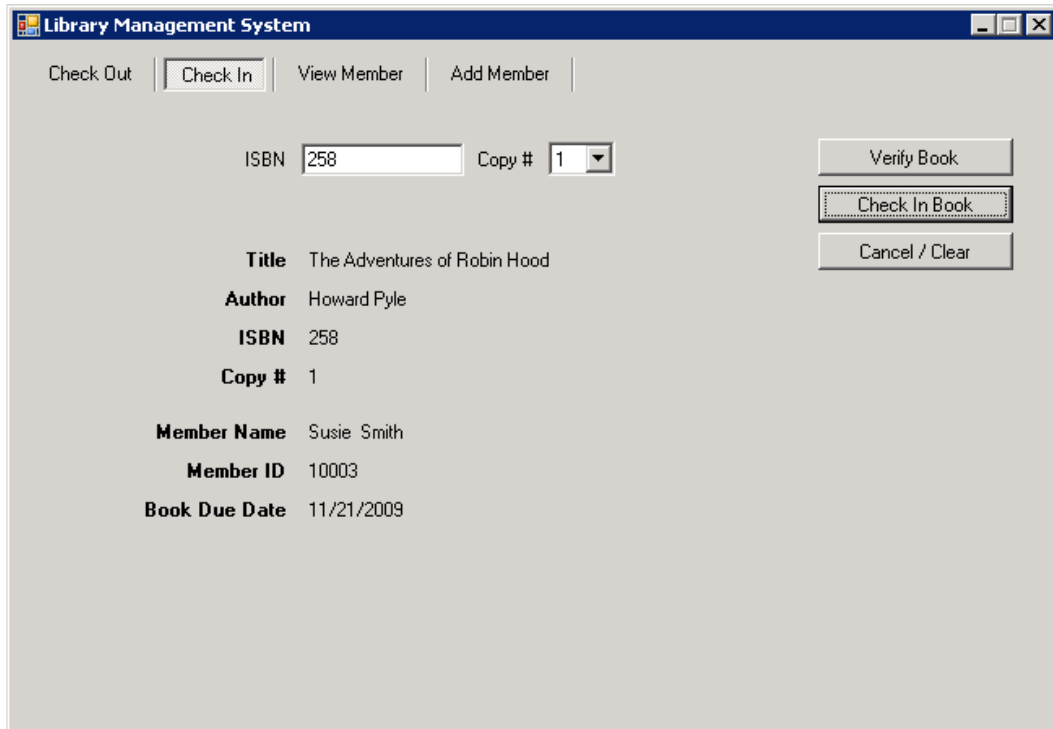


Figure 7. Check In

The ISBN and Copy Number are entered. After the item has been verified, it can be checked in.

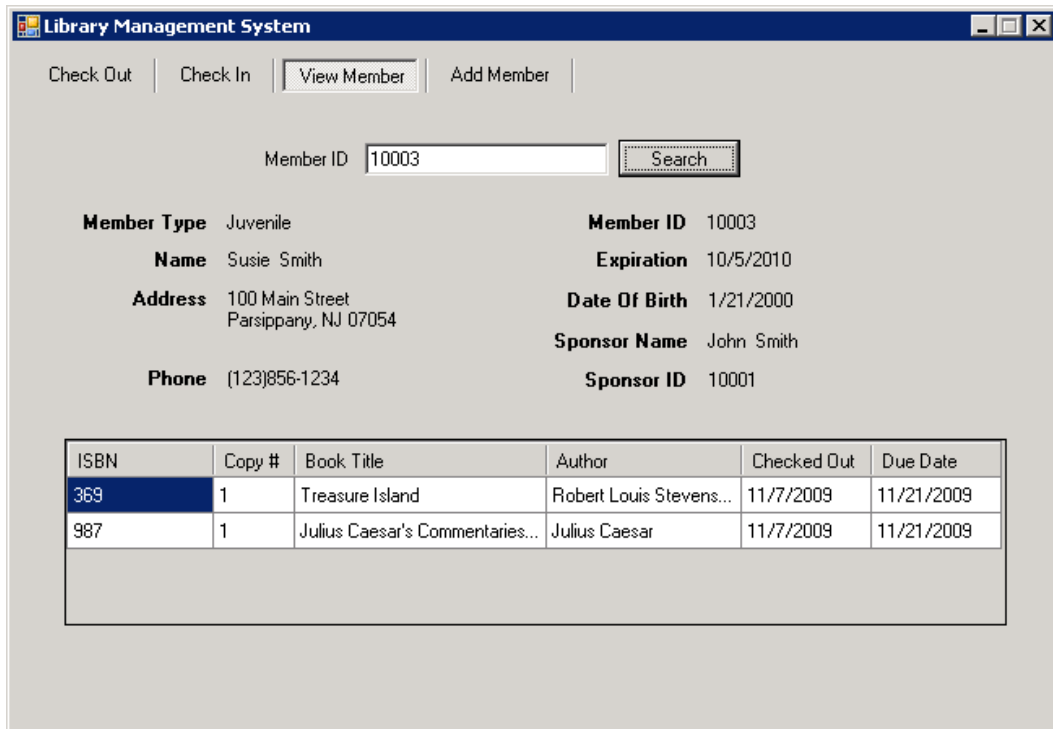
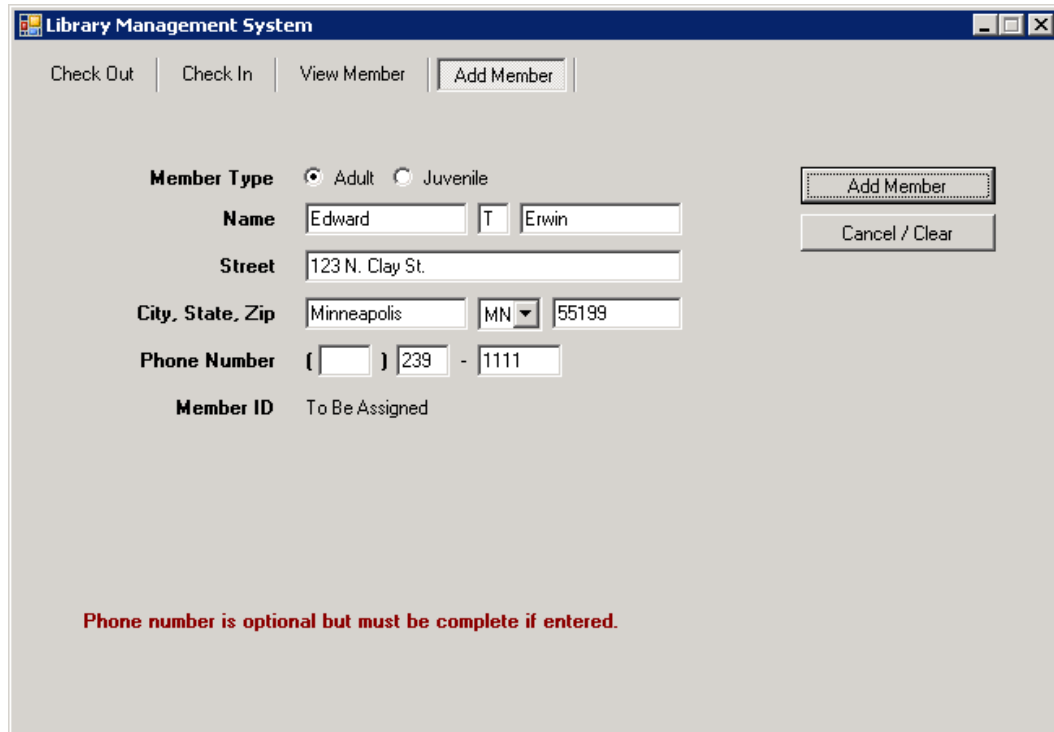


Figure 8. View Member



Library Management System

Check Out | Check In | View Member | Add Member

Member Type Adult Juvenile

Name Edward T Erwin

Street 123 N. Clay St.

City, State, Zip Minneapolis MN 55199

Phone Number () 239 - 1111

Member ID To Be Assigned

Add Member

Cancel / Clear

Phone number is optional but must be complete if entered.

Figure 9. Add Member

New adult and juvenile members can be added. A juvenile member is linked to an adult member id and includes birth date information. After a new member is added, a member id is assigned and displayed in on the View Member screen.

Validating User Input

A big portion of the project was validating user input based on exact specifications. For example, certain inputs like name or address could only be a certain number of characters long and the zip code had to be in the format of either ##### or #####-####. Member ids and ISBNs had to be within a certain range. Figure 10 shows a method to validate the zip code using a regular expression. Before checking out an item (Figure 6), the member id, isbn, copy number, item availability, and the quantity of items able to be checked out by the member are all verified. Context error messages are shown if there are any problems.

```
public bool ValidZipCode(string value, out string errorMessage)
{
    if (Regex.IsMatch(value, "^([0-9]{5})|([0-9]{5}-[0-9]{4})$"))
    {
        errorMessage = "";
        return true;
    }
    else
    {
        errorMessage = "Enter a valid zipcode ##### or #####-####. ";
        return false;
    }
}
```

Figure 10. Validating user input using a Regular Expression.

Library Phase 2: Business Entities & Data Access Tiers

Overview

In Phase 1, existing .dll files (no source code was provided) were used for the data access layer and business entities. In Phase 2, I created new entity object models using LINQ-to-SQL, a new data access layer, as well as views and stored procedures in SQL Server. Project goals included adequate error handling and implementation of a custom exception class so that any SQL database errors would be caught at the data access layer and translated as appropriate into custom Library Exceptions. By implementing the data access layer in this way, it would be possible at a future time to swap out the SQL database with a different type of database and the only code that would need to be changed would be in the data access layer.

Database Views & Stored Procedures

In SQL server I started by creating an item view and a member view. As you can see in Figure 11, the database was already set up in a normalized form with information spread across multiple tables for efficient storage purposes. By creating a view for item and member, it made it easy to select information later. The views were also used later in Visual Studio with LINQ-to-SQL classes mapped to relational objects. Figure 12 shows an example of using inner joins and a union to create the member view.

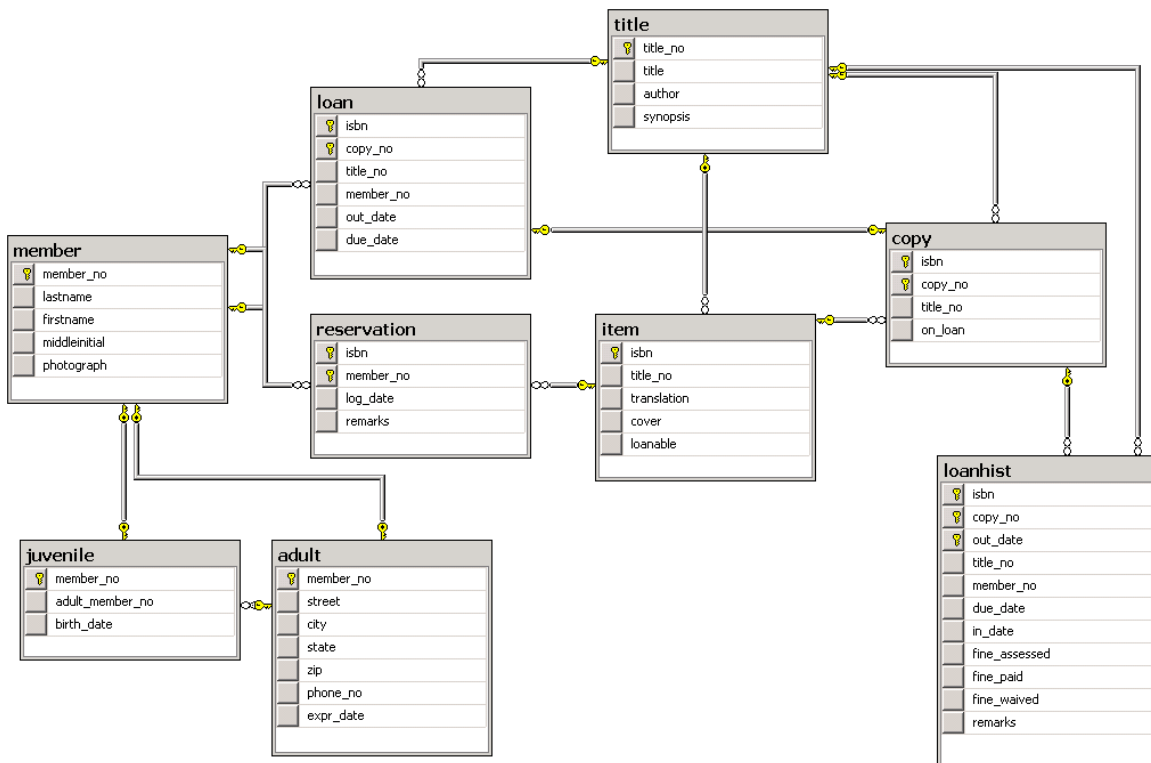


Figure 11. Database table layout that was provided.

```

SELECT
    dbo.member.member_no AS MemberID,
    dbo.member.firstname AS FirstName,
    dbo.member.middleinitial AS MiddleInitial,
    dbo.member.lastname AS LastName,
    NULL AS AdultMemberID,
    NULL AS BirthDate,
    dbo.adult.street AS Street,
    dbo.adult.city AS City,
    dbo.adult.state AS State,
    dbo.adult.zip AS ZipCode,
    dbo.adult.phone_no AS PhoneNumber,
    dbo.adult.expr_date AS ExpirationDate,
    'A' AS MemberType
FROM    dbo.member INNER JOIN
        dbo.adult ON dbo.member.member_no = dbo.adult.member_no
UNION
SELECT
    m.member_no AS MemberID
    ,m.firstname AS FirstName
    ,m.middleinitial AS MiddleInitial
    ,m.lastname AS LastName
    ,j.adult_member_no AS AdultMemberID
    ,j.birth_date AS BirthDate
    ,a.street AS Street
    ,a.city AS City
    ,a.state AS State
    ,a.zip AS ZipCode
    ,a.phone_no AS PhoneNumber
    ,a.expr_date AS ExpirationDate
    ,'J' AS MemberType
FROM    dbo.member m INNER JOIN
        dbo.juvenile j ON m.member_no = j.member_no LEFT OUTER JOIN
        dbo.adult a ON a.member_no = j.adult_member_no

```

Figure 12. Member SQL View.

In the database I also created stored procedures to AddJuvenileMember, AddAdultMember, CheckInItem, and CheckOutItem. The stored procedures used transactions where appropriate so that when multiple tables needed to be updated, if an error occurred, any updates to other tables would be rolled back so that the database can be kept in a stable condition. The stored procedures also checked incoming parameters and raised errors using custom state values, which would then be checked and processed in C#.

Business Entities & LINQ-to-SQL

Business entities refers to objects like Member and Item that get passed between the different application layers (Figure 1). After creating the views in SQL Server, the views were then used in Visual Studio to create entities in a LINQ-to-SQL class (.dbml file) by dragging the SQL Views from the Server Explorer panel in Visual Studio into the class. The results, shown in Figure 13, are entities objects that are mapped to the relational database. This means that when data is taken from the database for say a library item, it is automatically mapped to an item object. If the item is changed, it can then be saved back to the database.

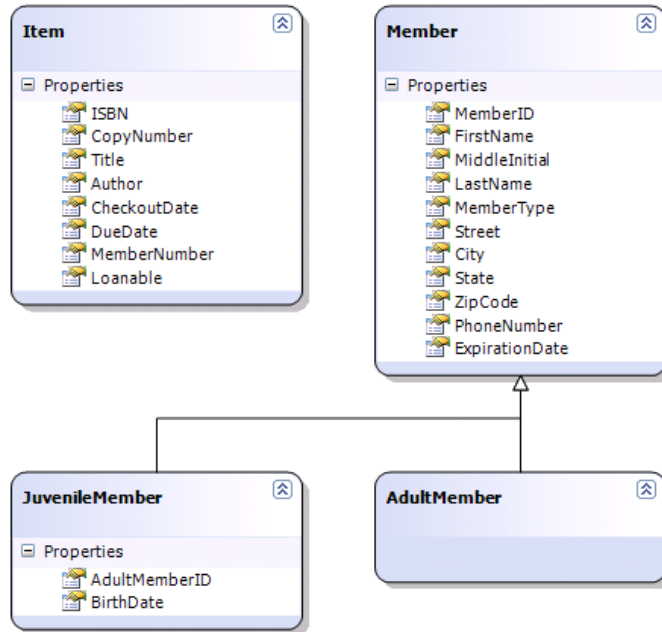


Figure 13. LINQ-to-SQL entities mapped to the relational database.

Another benefit of using LINQ-to-SQL is that it abstracts away the underlying data provider code such as ADO.NET calls and setting up a data reader, etc. The stored procedures also get accessed through LINQ-to-SQL.

LINQ-to-SQL also allows you to write queries. Figure 14 shows an example of a LINQ-to-SQL query in the data access layer. The query is performed by using the data context that was set up in the LINQ-to-SQL class. That data context abstracts away the underlying calls to the database. Any exceptions that are caught are re-thrown as library exceptions using the library exception class. This way, no SQL errors will go past the data access layer.

```

public Item GetItem(int isbn, short copyNumber)
{
    try
    {
        // get the library data context, set up query and execute it, return item
        LibraryDataContext ldc = new LibraryDataContext();
        Item myItem = null;
        var itemQuery = from item in ldc.Items
            where item.ISBN == isbn && item.CopyNumber == copyNumber
            select item;
        myItem = itemQuery.First();
        return myItem;
    }
    catch (InvalidOperationException ex)
    {
        throw new LibraryException(ErrorCode.ItemNotFound, "No items were found with that ISBN and copy
number.", ex);
    }
    catch (SqlException ex)
    {
        ThrowLibraryExceptionFromSQLException(ex);
        return null; //won't ever get to this
    }
}

```

```

    }
    catch (Exception ex)
    {
        throw new LibraryException(ErrorCode.GenericException, "Unexpected error has occurred.", ex);
    }
}

```

Figure 14. LINQ-to-SQL query.

DataSet

A dataset was created to represent items on loan to a member. The dataset was then filled using a stored procedure with a parameter taking in the member id. On the front end, the dataset was then bound to a data grid view as the data source.

SQL Exceptions

The SQL stored procedures were set up to raise errors if parameters were missing or if other conditions occurred like an item doesn't exist in the database. The errors were also assigned a state code. When the stored procedures were called in the data access layer, any SQL exceptions were caught and re-thrown as library exceptions. I developed a helper method in order to automate the error handling process as shown in Figure 15.

```

private void ThrowLibraryExceptionFromSQLException(SQLException ex)
{
    //Check if this is User Defined Error
    if (ex.Number >= 50000)
    {
        switch (ex.State)
        {
            case 1:
            case 2:
                // 'A parameter is missing.' 1
                // 'A parameter is out of range.' 2
                throw new LibraryException(ErrorCode.GenericException, ex.Message, ex);
                // break; - not necessary because throw breaks out of switch.
            case 10:
                // 'Member does not exist.'
                throw new LibraryException(ErrorCode.NoSuchMember, ex.Message, ex);
            case 11:
                // 'Membership has expired.'
                throw new LibraryException(ErrorCode.MembershipExpired, ex.Message, ex);
            case 20:
                // 'Item does not exist.' 20
                throw new LibraryException(ErrorCode.ItemNotFound, ex.Message, ex);
            case 21:
                // 'Item is not on loan.' 21
                throw new LibraryException(ErrorCode.ItemNotOnLoan, ex.Message, ex);
            case 22:
                // 'Item is already on loan.' 22
                throw new LibraryException(ErrorCode.ItemAlreadyOnLoan, ex.Message, ex);
            case 23:
                // 'Item is not loanable.' 23
                throw new LibraryException(ErrorCode.ItemNotLoanable, ex.Message, ex);
        }
    }
}

```

```

default:
    throw new LibraryException(ErrorCode.GenericException, "Unexpected Error Has Occured", ex);
}
}

```

Figure 15. SQL exceptions re-thrown as library exceptions.

Library Phase 3: Web Application

Overview

The final phase for the library project was to turn it into an ASP.NET web application. In addition to replacing the front end (presentation layer), new functional requirements were added, including implementing AJAX and forms-based authentication.

AJAX

AJAX is the ability to do partial page updates without refreshing or doing a postback of the entire page. Visual Studio provides various controls to accomplish this. A script manager control on the web form enables the use of update panels and other AJAX controls. Content within the update panel updates independently from the rest of the page. The main part of each page was placed in an update panel. An update progress control was also used to display "Processing..." text if the operation was taking too long.

Master Page / Layout

Since the header and navigation were the same for each page, I used a master page to contain those items. Subsequent pages only needed to populate the main content part. Figure 16 shows the overall layout. Although some CSS styling was applied, the main focus of the project was on functionality.

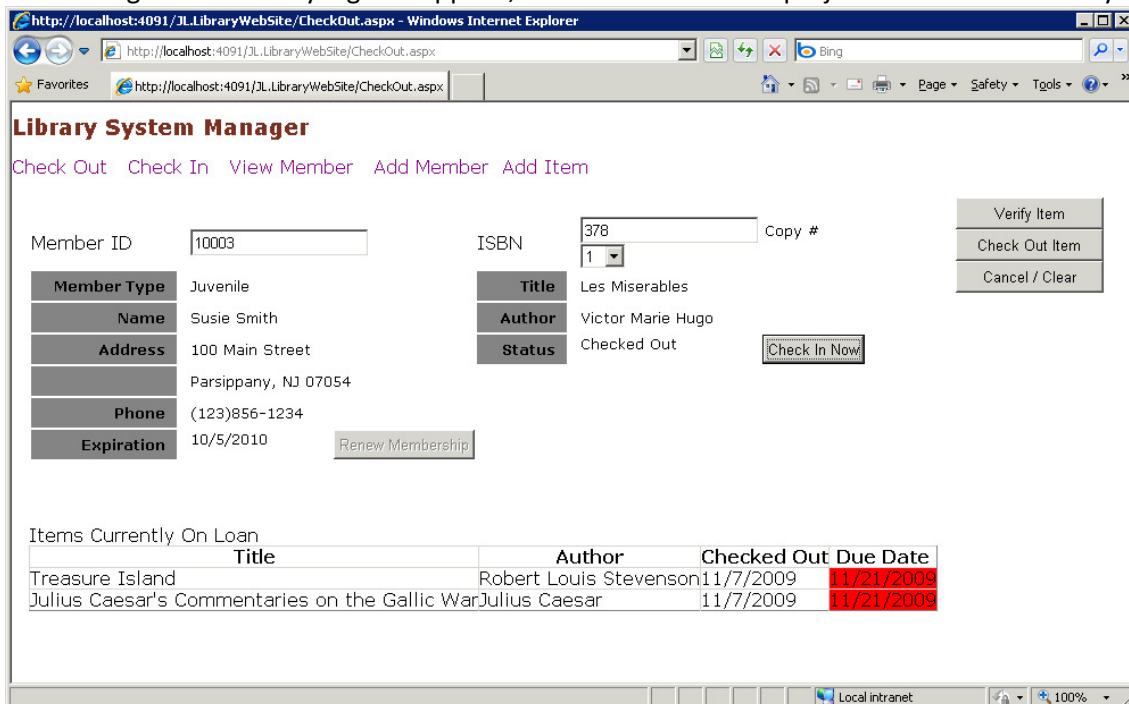


Figure 16. Overall layout.

Forms-based Authentication & Authorization

The web application made use of forms-based authentication and authorization. Authentication is identifying who the user is. Authorization is specifying who is allowed to do what. The ASP.NET Web Site Administration Tool was used to set up users and roles. Only members of the Librarian role were allowed to access the web application. Standard controls from the Login section of the Visual Studio Toolbox were used to create the login form. The command line `aspnet_regsql.exe` utility was used to set up the corresponding database tables used to track users.

Detect Expired Memberships, Overdue Items, & Juveniles Now Adults

Every time membership information was displayed, expired memberships were detected and the librarian was offered the opportunity to renew the membership. This was accomplished by displaying status messages and enabling a renew membership button to renew the membership.

Similarly, every time a juvenile member was displayed, they were validated to make sure that they are still under 18 years of age. If they are currently over the age of 18, they are automatically converted from a juvenile member to adult membership. The librarian is then notified with a status message that the upgrade took place. Under the hood this required making updates to the database and removing information from the juvenile table and inserting information into the adult table, including address information which was obtained from the juvenile's previous association with another adult member.

Finally, overdue books shown in any display were highlighted in red. Since checked out books were typically shown in a grid view control, an event handler was used to highlight the cell color of an overdue book any time the data was refreshed (bound to the control), as shown in Figure 17.

```
protected void memberGridView_RowDataBound(object sender, GridViewRowEventArgs e)
{
    //if it is a data row (as opposed to a header row)
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        if ((DateTime.Parse(e.Row.Cells[5].Text)).Date < DateTime.Now.Date)
        {
            //change the bgcolor of a cell if overdue
            e.Row.Cells[5].BackColor = System.Drawing.Color.Red;
        }
    }
}
```

Figure 17. Detecting and highlighting overdue items in a grid view control.

Insert Items Into Database

A new page was added to insert items into the database based on ISBN. After entering an ISBN, the number was checked in the database to see if there is an existing item. If so, the title information is displayed and another copy is added to the database. Otherwise an additional feature to be added is to enter information for a completely new item.