The database [SchoolDB](https://web.csulb.edu/~pnguyen/cecs475/labs/SchoolDB.mdf)  has has different tables, stored procedures and views.

As you can see in the above figure that SchoolDB database includes tables with following relationship.

* **One-to-One:** Student and StudentAddress have a one-to-one relationship eg. Student has zero or one StudentAddress.
* **One-to-Many:** Standard and Teacher have a one-to-many relationship eg. many Teachers can be associate with one Standard.
* **Many-to-Many:** Student and Course have a many-to-many relationship using StudentCourse table where StudentCourse table includes StudentId and CourseId. So one student can join many courses and one course also can have many students.

Below is the Entity Data Model (EDM) for SchoolDB database.

**Repository Pattern and Unit of Work with Entity Framework**

Repository pattern is used to create an abstraction layer between the data access layer and the business logic layer. This abstraction layer contains data manipulation methods which communicate with the data access layer to serve data as per the business requirements to the logical layer. The main purpose to create this layer is for isolating data access layer so that changes cannot affect the business logic layer directly.

For example, a POS application is usually deployed to different stores. A few new clients wanted to use the Microsoft SQL Server database system and some others wanted Oracle and other databases. Also they had a few different relational database systems to store their data but business logic was almost the same. Repository pattern helps developers to detach the layer and add a new data access layer.

**Unit of work** is a pattern to handle transaction during data manipulation using the Repository pattern. So we can say, according to Martin Fowler, *Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problem.*

Implementing these patterns can help insulate the application from changes in the data store and also gives advantages to automate unit testing. T

In this lab assignment, you are required to create a Repository pattern.

1. Create a Repository interface which will contain the CRUD functionality along with Queryable search.

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Linq.Expressions;  
using System.Text;  
using System.Threading.Tasks;

namespace DataAccessLayer  
{  
public interface IRepository<T> : IDisposable  
{  
void Insert(T entity);

void Delete(T entity);

void Update(T entity);

T GetById(int id);

IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate);

IEnumerable<T> GetAll();

**T GetSingle(Func<T, bool> where, params Expression<Func<T, object>>[] navigationProperties);**

}  
}

2. Create a Repository class that implements the interface IRepository.

namespace DataAccessLayer  
{  
public class Repository<T> : IRepository<T> where T : class  
{  
protected Dbcontext context;

public Repository(DbContext datacontext)  
{  
  //You can use the cpmt  
}

public void Insert(T entity)  
{  
**//Use the context object and entity state to save the entity**  
}

public void Delete(T entity)  
{  
**//Use the context object and entity state to delete the entity**  
}

public void Update(T entity)  
{

**//Use the context object and entity state to update the entity**

}

public T GetById(int id)  
{

}

public IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate)  
{  
return context.Where(predicate);  
}

public IEnumerable<T> GetAll()  
{  
  
}

**public T GetSingle(Func<T, bool> where, params Expression<Func<T, object>>[] navigationProperties)  
//This method will find the related records by passing two argument  
//First argument: lambda expression to search a record such as d => d.StandardName.Equals(standardName) to search am record by standard name  
//Second argument: navigation property that leads to the related records such as d => d.Students  
//The method returns the related records that met the condition in the first argument.  
//An example of the method GetStandardByName(string standardName)  
//public Standard GetStandardByName(string standardName)  
//{  
//return \_standardRepository.GetSingle(d => d.StandardName.Equals(standardName), d => d.Students);  
//}   
{  
T item = null;  
IQueryable<T> dbQuery = null;  
foreach (Expression<Func<T, object>> navigationProperty in navigationProperties)  
dbQuery = DbSet.Include<T, object>(navigationProperty);**

**item = dbQuery  
.AsNoTracking()  
.FirstOrDefault(where);  
return item;**  
  
}

}  
}

3. Add interfaces and classes to present specific repositories for the *Standard*and *Student* entities.

namespace DataAcessLayer  
{   
 public interface IStandardRepository: IRepository<Standard>  
{

}

public interface IStudentRepository: IRepository<Student>  
{

}

public class StandardRepository: Repository<Standard>, IStandardRepository  
{

}

public class StudentRepository: Repository<Student>, IStudentRepository  
{

**public StudentRepository()  
: base(new SchoolDBEntities())  
{   
}**

}

4. Add a new project (BusinessLayer) to the solution.

namespace BusinessLayer  
{  
   public interface IBusinessLayer  
  {  IList<Standard> getAllStandards();  
      Standard GetStandardByID(int id);  
      void addStandard(Standard);  
      void updateStandard(Standard)  
      void removeStandard(Standard);  
     IList<Student> getAllStudents();  
      Student GetStudentByID(int id);  
      void addStudent(Student);  
      void UpdateStudent(Student);  
      void RemoveStudent(Student);  
  }   
    public class BusinessLayer: IBusinessLayer  
  {  
     private readonly IStandardRepository   \_standardRepository;  
    private readonly IStandardRepository   \_studentRepository;  
    public BusinessLayer()  
    {  \_standardRepository = new StandardRepository();  
        \_studentRepository = new StudentRepository();  
    }

   //Implement other methods

  }

5. Write a client to implement CRUD operations  to test all the methods defined in the class BusinessLayer.

1. Table Teacher
   * Create, update, and delete - Update by seaching teacher id or teacher name
   * Input the teacher id and then display all courses that has that teacher id.
   * Display all standards
2. Table Courses
   * Create, update, and delete - Update by searching course id or course name.
   * Display all courses