using System;

using System.Collections.Generic;

using System.Globalization;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Model

{

public class Book

{

public int BookId { get; set; }

public string Name { get; set; }

public string Author { get; set; }

public string Ganre { get; set; }

public int Size { get; set; }

public string Path { get; set; }

}

}

and

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Model

{

public class Category{

public Category()

{

Books = new List<Book>();

}

public int CategoryId { get; set; }

public string CategoryName { get; set; }

virtual public ICollection<Book> Books { get; set; }

}

}

But my question is, **how to add the book to a category?**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Data.Entity;

using Model;

using DataAccess;

namespace TestDB

{

class Program

{

static void Main(string[] args)

{

Database.SetInitializer(

new DropCreateDatabaseIfModelChanges<BookShelfContext>());

using (var db = new BookShelfContext())

{

var book = new Book

{

Author = "Author Name",

Ganre = "Ganre",

Name = "Book Name",

Path = @"Path",

Size = 10

};

var category = new Category

{

CategoryName = "Interesting"

};

var bookrepository = new Repository<Book>(db);

var categoryrepository = new Repository<Category>(db);

IEnumerable<Book> books = bookrepository.GetAll();

IEnumerable<Category> categories = categoryrepository.GetAll();

//get all books for example

foreach (var b in books)

{

Console.WriteLine(b.Name);

}

}

Console.ReadKey();

}

}

}

Add the context to your Repository so that you can implement the SaveChanges method:

protected readonly DbContext context;

public Repository(DbContext datacontext)

{

DbSet = datacontext.Set<T>();

context = datacontext;

}

public void SaveChanges()

{

context.SaveChanges();

}

Then in order to add a book to an existing BookCategory simply add the book to the Category's collection and save the category:

var categoryrepository = new Repository<Category>(db);

var myCategory = categoryrepository.GetById(1);

myCategory.Books.Add(book);

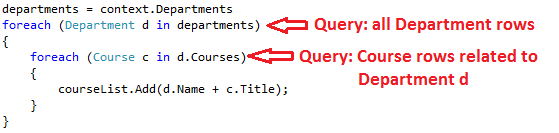
categoryrepository.SaveChanges();

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

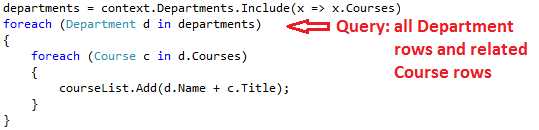
**Lazy, Eager, and Explicit Loading of Related Data**

There are several ways that the Entity Framework can load related data into the navigation properties of an entity:

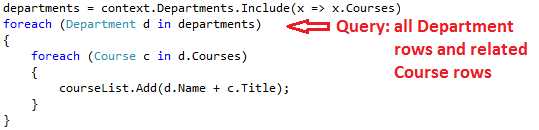
* *Lazy loading*. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. This results in multiple queries sent to the database — one for the entity itself and one each time that related data for the entity must be retrieved.



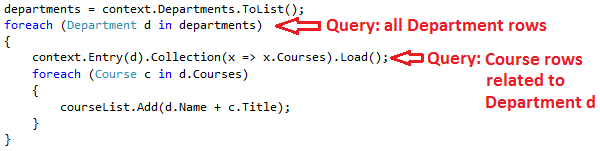
* *Eager loading*. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading by using the Include method.



* Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading by using the Include method.



* Explicit loading. This is similar to lazy loading, except that you explicitly retrieve the related data in code; it doesn't happen automatically when you access a navigation property. You load related data manually by getting the object state manager entry for an entity and calling the [Collection.Load](http://msdn.microsoft.com/en-us/library/gg696220%28v=vs.103%29.aspx) method for collections or the [Reference.Load](http://msdn.microsoft.com/en-us/library/gg679166%28v=vs.103%29.aspx) method for properties that hold a single entity. (In the following example, if you wanted to load the Administrator navigation property, you'd replace Collection(x => x.Courses) with Reference(x => x.Administrator).)



Because they don't immediately retrieve the property values, lazy loading and explicit loading are also both known as deferred loading.

In general, if you know you need related data for every entity retrieved, eager loading offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, in the above examples, suppose that each department has ten related courses. The eager loading example would result in just a single (join) query and a single round trip to the database. The lazy loading and explicit loading examples would both result in eleven queries and eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios lazy loading is more efficient. Eager loading might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of entities you're processing, lazy loading might perform better because eager loading would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Typically you'd use explicit loading only when you've turned lazy loading off. One scenario when you should turn lazy loading off is during serialization.  Lazy loading and serialization don’t mix well, and if you aren’t careful you can end up querying significantly more data than you intended when lazy loading is enabled. Serialization generally works by accessing each property on an instance of a type. Property access triggers lazy loading, and those lazy loaded entities are serialized. The serialization process then accesses each property of the lazy-loaded entities, potentially causing even more lazy loading and serialization. To prevent this run-away chain reaction, turn lazy loading off before you serialize an entity.

The database context class performs lazy loading by default. There are two ways to disable lazy loading:

* For specific navigation properties, omit the virtual keyword when you declare the property.
* For all navigation properties, set LazyLoadingEnabled to false. For example, you can put the following code in the constructor of your context class:

this.Configuration.LazyLoadingEnabled = false;

Lazy loading can mask code that causes performance problems. For example, code that doesn't specify eager or explicit loading but processes a high volume of entities and uses several navigation properties in each iteration might be very inefficient (because of many round trips to the database). An application that performs well in development using an on premise SQL server might have performance problems when moved to Azure SQL Database due to the increased latency and lazy loading. Profiling the database queries with a realistic test load will help you determine if lazy loading is appropriate.

|  |
| --- |
| using Mm.DomainModel;  using System;  using System.Collections.Generic;  using System.Linq.Expressions;    namespace Mm.DataAccessLayer  {      public interface IGenericDataRepository<T> where T : class      {          IList<T> GetAll(params Expression<Func<T, object>>[] navigationProperties);          IList<T> GetList(Func<T, bool> where, params Expression<Func<T, object>>[] navigationProperties);          T GetSingle(Func<T, bool> where, params Expression<Func<T, object>>[] navigationProperties);          void Add(params T[] items);          void Update(params T[] items);          void Remove(params T[] items);      }  } |

### IList vs IQueryable

Note that the return type of the two Get\* methods is IList<T> rather than IQueryable<T>. This means that the methods will be returning the actual already executed results from the queries rather than executable queries themselves. Creating queries and return these back to the calling code would make the caller responsible for executing the LINQ-to-Entities queries and consequently use EF logic. Besides, when using EF in an N-tier application the repository typically creates a new context and dispose it on every request meaning the calling code won’t have access to it and therefore the ability to cause the query to be executed. Thus you should always keep your LINQ queries inside of the repository when using EF in a disconnected scenario such as in an N-tier application.

### Loading related entities

EF offers two categories for loading entities that are related to your target entity, e.g. getting employees associated with a department in this case. Eager loading uses the Include method on the DbSet to load child entities and will issue a single query that fetches the data for all the included entities in a single call. Each of the methods for reading data from the database in the concrete sample implementation of the IGenericDataRepository<T> interface below supports eager loading by accepting a variable number of navigation properties to be included in the query as arguments.

10. Add a new class named GenericDataRepository to the MM.DataAccessLayer project and implement the IGenericDataRepository<T> interface .

|  |
| --- |
| using Mm.DomainModel;  using System;  using System.Collections.Generic;  using System.Data.Entity;  using System.Data.Entity.Infrastructure;  using System.Linq;  using System.Linq.Expressions;    namespace Mm.DataAccessLayer  {      public class GenericDataRepository<T> : IGenericDataRepository<T> where T : class      {          public virtual IList<T> GetAll(params Expression<Func<T, object>>[] navigationProperties)          {              List<T> list;              using (var context = new Entities())              {                  IQueryable<T> dbQuery = context.Set<T>();                    //Apply eager loading                  foreach (Expression<Func<T, object>> navigationProperty in navigationProperties)                      dbQuery = dbQuery.Include<T, object>(navigationProperty);                    list = dbQuery                      .AsNoTracking()                      .ToList<T>();              }              return list;          }            public virtual IList<T> GetList(Func<T, bool> where,               params Expression<Func<T,object>>[] navigationProperties)          {              List<T> list;              using (var context = new Entities())              {                  IQueryable<T> dbQuery = context.Set<T>();                    //Apply eager loading                  foreach (Expression<Func<T, object>> navigationProperty in navigationProperties)                      dbQuery = dbQuery.Include<T, object>(navigationProperty);                    list = dbQuery                      .AsNoTracking()                      .Where(where)                      .ToList<T>();              }              return list;          }            public virtual T GetSingle(Func<T, bool> where,               params Expression<Func<T, object>>[] navigationProperties)          {              T item = null;              using (var context = new Entities())              {                  IQueryable<T> dbQuery = context.Set<T>();                    //Apply eager loading                  foreach (Expression<Func<T, object>> navigationProperty in navigationProperties)                      dbQuery = dbQuery.Include<T, object>(navigationProperty);                    item = dbQuery                      .AsNoTracking() //Don't track any changes for the selected item                      .FirstOrDefault(where); //Apply where clause              }              return item;          }            /\* rest of code omitted \*/      }  } |

For example, here’s how you would call the GetAll method to get all departments with its employees included:

|  |
| --- |
| IGenericDataRepository<Department> repository = new GenericDataRepository<Department>();  IList<Department> departments = repository.GetAll(d => d.Employees); |

With lazy loading related entities are loaded from the data source by EF issuing a separate query first when the get accessor of a navigation property is accessed programmatically.

11. Lazy loading and dynamic proxy creation is turned off for all entities in a context by setting two flags on the Configuration property on the *DbContext* as shown below. Both these properties are set to true by default.

|  |
| --- |
| namespace Mm.DataAccessLayer  {      using System.Data.Entity;      using System.Data.Entity.Infrastructure;      using Mm.DomainModel;        public partial class Entities : DbContext      {          public Entities()              : base("name=Entities")          {              Configuration.LazyLoadingEnabled = false;              Configuration.ProxyCreationEnabled = false;          }            protected override void OnModelCreating(DbModelBuilder modelBuilder)          {              throw new UnintentionalCodeFirstException();          }            public DbSet<Department> Departments { get; set; }          public DbSet<Employee> Employees { get; set; }      }  } |

### ntityState

On the server side, things will get easier if you decide to not support graphs. In this case you could expose an Add method and an Update method for each entity type and these methods would only operate on a standalone instance rather than a graph of entities. EF makes it simple to implement these methods. It is all about setting the state of the passed in entity object. An entity can be in one of five states as defined by the System.Data.EntityState enumeration:

–Added: the entity is being tracked by the context but hasn’t been added to the database yet.

–Unchanged: the entity is being tracked by the context, it exists in the database but its property values have not been changed since it was fetched from the database.

–Modified: the entity is being tracked by the context, it exists in the database and some or all of its property values have been modified since it was fetched from the database

–Deleted: the entity is being tracked by the context, it exists in the database but will be deleted on the next call to the SaveChanges method.

–Detached: the entity is not being tracked by the context at all.

When the context’s SaveChanges method is called it decides what to do based on the entity’s current state. Unchanged and detached entities are ignored while added entities are inserted into the database and then become Unchanged when the method returns, modified entities are updated in the database and then become Unchanged and deleted entities are deleted from the database and then detached from the context.

### DbSet.Entry

You can explicitly change the state of an entity by using the DbSet.Entry method. There is no need to attach the entity to the context before using this method as it will automatically do the attachment if needed. Below is the implementation of the generic repository’s Add method. It explicitly sets the state of the entity to be inserted into the database to Added before calling SaveChanges to execute and commit the insert statement. Note that using the Entry method to change the state of an entity will only affect the actual entity that you pass in to the method. It won’t cascade through a graph and set the state of all related objects, unlike the DbSet.Add method.

|  |
| --- |
| public virtual void Add(params T[] items)  {      using (var context = new Entities())      {          foreach (T item in items)          {              context.Entry(item).State = System.Data.EntityState.Added;          }          context.SaveChanges();      }  } |

The implementation for the Update and Remove methods are very similar to the Add method as shown below. Note that all exception handling has been omitted for brevity in the sample code.

|  |
| --- |
| public virtual void Update(params T[] items)  {      using (var context = new Entities())      {          foreach (T item in items)          {              context.Entry(item).State = System.Data.EntityState.Modified;          }          context.SaveChanges();      }  }    public virtual void Remove(params T[] items)  {      using (var context = new Entities())      {          foreach (T item in items)          {              context.Entry(item).State = System.Data.EntityState.Deleted;          }          context.SaveChanges();      }  } |

Also note that all methods have been marked as virtual. This allows you to override any method in the generic repository by adding a derived class in cases where you need some specific logic to apply only to a certain type of entity. To be able to extend the generic implementation with methods that are specific only to a certain type of entity, whether it’s an initial requirement or a possible future one, it’s considered a good practice to define a repository per entity type from the beginning. You can simply inherit these repositories from the generic one as shown below and add methods to extend the common functionality based on your needs.

12. Add interfaces and classes to represent specific repositories for the Department and Employee entities to the DAL project.

|  |
| --- |
| using Mm.DomainModel;    namespace Mm.DataAccessLayer  {      public interface IDepartmentRepository : IGenericDataRepository<Department>      {      }        public interface IEmployeeRepository : IGenericDataRepository<Employee>      {      }        public class DepartmentRepository : GenericDataRepository<Department>, IDepartmentRepository      {      }        public class EmployeeRepository : GenericDataRepository<Employee>, IEmployeeRepository      {      }  } |

### Business layer

As mentioned before, the repository is located somewhere between the DAL and the business layer in a typical N-tier architecture. The business layer will use it to communicate with the database through the EDM in the DAL. Any client application will be happily unaware of any details regarding how data is fetched or persisted on the server side. It’s the responsibility of the business layer to provide methods for the client to use to communicate with the server.

13. Add a new project (Mm.BusinessLayer) to the solution with references to the DAL project (Mm.DataAccessLayer) and the project with the domain classes (Mm.DomainModel). Then add a new interface and a class implementing this interface to it to expose methods for creating, reading, updating and deleting entities to any client application.

Below is a sample implementation. In a real world application the methods in the business layer would probably contain code to validate the entities before processing them and it would also be catching and logging exceptions and maybe do some caching of frequently used data as well.

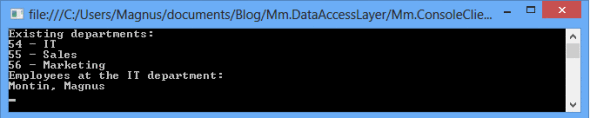
|  |
| --- |
| using Mm.DomainModel;  using System.Collections.Generic;  using Mm.DataAccessLayer;    namespace Mm.BusinessLayer  {      public interface IBusinessLayer      {          IList<Department> GetAllDepartments();          Department GetDepartmentByName(string departmentName);          void AddDepartment(params Department[] departments);          void UpdateDepartment(params Department[] departments);          void RemoveDepartment(params Department[] departments);            IList<Employee> GetEmployeesByDepartmentName(string departmentName);          void AddEmployee(Employee employee);          void UpdateEmploee(Employee employee);          void RemoveEmployee(Employee employee);      }        public class BuinessLayer : IBusinessLayer      {          private readonly IDepartmentRepository \_deptRepository;          private readonly IEmployeeRepository \_employeeRepository;            public BuinessLayer()          {              \_deptRepository = new DepartmentRepository();              \_employeeRepository = new EmployeeRepository();          }            public BuinessLayer(IDepartmentRepository deptRepository,              IEmployeeRepository employeeRepository)          {              \_deptRepository = deptRepository;              \_employeeRepository = employeeRepository;          }            public IList<Department> GetAllDepartments()          {              return \_deptRepository.GetAll();          }            public Department GetDepartmentByName(string departmentName)          {              return \_deptRepository.GetSingle(                  d => d.Name.Equals(departmentName),                  d => d.Employees); //include related employees          }            public void AddDepartment(params Department[] departments)          {              /\* Validation and error handling omitted \*/              \_deptRepository.Add(departments);          }            public void UpdateDepartment(params Department[] departments)          {              /\* Validation and error handling omitted \*/              \_deptRepository.Update(departments);          }            public void RemoveDepartment(params Department[] departments)          {              /\* Validation and error handling omitted \*/              \_deptRepository.Remove(departments);          }            public IList<Employee> GetEmployeesByDepartmentName(string departmentName)          {              return \_employeeRepository.GetList(e => e.Department.Name.Equals(departmentName));          }            public void AddEmployee(Employee employee)          {              /\* Validation and error handling omitted \*/              \_employeeRepository.Add(employee);          }            public void UpdateEmploee(Employee employee)          {              /\* Validation and error handling omitted \*/              \_employeeRepository.Update(employee);          }            public void RemoveEmployee(Employee employee)          {              /\* Validation and error handling omitted \*/              \_employeeRepository.Remove(employee);          }      }  } |

### Client

A client application consuming the sever side code will only need references to the business layer and the entity classes defined in the Mm.DomainModel project. Below is a simple C# console application to test the functionality provided by the business layer. It’s important to note that there are no references or dependencies to EF in this application. In fact you could replace the EF-based DAL with another one using raw T-SQL commands to communicate with the database without affecting the client side code. The only thing in the console application that hints that EF may be involved is the connection string that was generated in the DAL project when the EDM was created and has to be added to the application’s configuration file (App.config). Connection strings used by EF contain information about the required model, the mapping files between the model and the database and how to connect to the database using the underlying data provider.

14. To be able to test the functionality of the business layer and the DAL, create a new console application and add references to the Mm.BusinessLayer project and the Mm.DomainModel project.

|  |
| --- |
| using Mm.BusinessLayer;  using Mm.DomainModel;  using System;  using System.Collections.Generic;    namespace Mm.ConsoleClientApplication  {      class Program      {          static void Main(string[] args)          {              IBusinessLayer businessLayer = new BuinessLayer();                /\* Create some departments and insert them to the database through the business layer \*/              Department it = new Department() { Name = "IT" };              Department sales = new Department() { Name = "Sales" };              Department marketing = new Department() { Name = "Marketing" };              businessLayer.AddDepartment(it, sales, marketing);                /\* Get a list of departments from the database through the business layer \*/              Console.WriteLine("Existing departments:");              IList<Department> departments = businessLayer.GetAllDepartments();              foreach (Department department in departments)                  Console.WriteLine(string.Format("{0} - {1}", department.DepartmentId, department.Name));                  /\* Add a new employee and assign it to a department \*/              Employee employee = new Employee()              {                  FirstName = "Magnus",                  LastName = "Montin",                  DepartmentId = it.DepartmentId              };              businessLayer.AddEmployee(employee);                /\* Get a single department by name \*/              it = businessLayer.GetDepartmentByName("IT");              if (it != null)              {                  Console.WriteLine(string.Format("Employees at the {0} department:", it.Name));                  foreach (Employee e in it.Employees)                      Console.WriteLine(string.Format("{0}, {1}", e.LastName, e.FirstName));              };                /\* Update an existing department \*/              it.Name = "IT Department";              businessLayer.UpdateDepartment(it);                /\* Remove employee \*/              it.Employees.Clear();              businessLayer.RemoveEmployee(employee);                /\* Remove departments\*/              businessLayer.RemoveDepartment(it, sales, marketing);                Console.ReadLine();          }      }  } |



### Persisting disconnected graphs

While avoiding the complexity of accepting graphs of objects to be persisted at once makes life easier for server side developers, it potentially makes the client component more complex. As you may have noticed by looking at the code for the business layer above, you are also likely to end up with a large number of operations exposed from the server. If you do want your business layer to be able to handle graphs of objects to passed in and be persisted correctly, you need a way of determining what changes were made to the passed in entity objects in order for you to set their states correctly.

For example, consider a scenario when you get a Department object representing a graph with related Employee objects. If all entities in the graph are new, i.e. are not yet in the database, you can simply call the DbSet.Add method to set the state of all entities in the graph to Added and call the SaveChanges to persist the changes. If the root entity, the Department in this case, is new and all related Employee objects are unchanged and already existing in the database you can use the DbSet.Entry method to change the state of the root only. If the root entity is modified and some related items have also been changed, you would first use the DbSet.Entry method to set the state of the root entity to Modified. This will attach the entire graph to the context and set the state of the related objects to Unchanged. You will then need to identify the related entities that have been changed and set the state of these to Modified too. Finally, you may have a graph with entities of varying states including added ones. The best thing here is to use the DbSet.Add method to set the states of the related entities that were truly added to Added and then use the DbSet.Entry method to set the correct state of the other ones.

So how do you know the state of an entity when it comes from a disconnected source and how do you make your business layer able to persist a graph with a variety of objects with a variety of states? The key here is to have the entity objects track their own state by explicitly setting the state on the client side before passing them to the business layer. This can be accomplished by letting all entity classes implement an interface with a state property. Below is a sample interface and an enum defining the possible states.

|  |
| --- |
| namespace Mm.DomainModel  {      public interface IEntity      {          EntityState EntityState { get; set; }      }        public enum EntityState      {          Unchanged,          Added,          Modified,          Deleted      }  } |
| /\* Entity classes implementing IEntity \*/  public partial class Department : IEntity  {      public Department()      {          this.Employees = new HashSet<Employee>();      }        public int DepartmentId { get; set; }      public string Name { get; set; }        public virtual ICollection<Employee> Employees { get; set; }        public EntityState EntityState { get; set; }  }    public partial class Employee : IEntity  {      public int EmployeeId { get; set; }      public int DepartmentId { get; set; }      public string FirstName { get; set; }      public string LastName { get; set; }      public string Email { get; set; }        public virtual Department Department { get; set; }        public EntityState EntityState { get; set; }  } |

With this solution, the business layer will know the state of each entity in a passed in graph assuming the states have been set correctly in the client application. The repository will need a helper method to convert the custom EntityState value to a System.Data.EntityState enumeration value. The below static method can be added to the GenericDataRepository<T> class in the DAL to takes care of this.

|  |
| --- |
| protected static System.Data.EntityState GetEntityState(Mm.DomainModel.EntityState entityState)  {      switch (entityState)      {          case DomainModel.EntityState.Unchanged:              return System.Data.EntityState.Unchanged;          case DomainModel.EntityState.Added:              return System.Data.EntityState.Added;          case DomainModel.EntityState.Modified:              return System.Data.EntityState.Modified;          case DomainModel.EntityState.Deleted:              return System.Data.EntityState.Deleted;          default:              return System.Data.EntityState.Detached;      }  } |

Next, you need to specify a constraint on the IGenericDataRepository<T> interface and the GenericDataRepository<T> class to ensure that the type parameter T implements the IEntity interface and then make some modifications to the CUD methods in the repository as per below. Note that the Update method will actually be able to do all the work now as it basically only sets the System.Data.EntityState of an entity based on the value of the custom enum property.

|  |
| --- |
| public interface IGenericDataRepository<T> where T : class, IEntity { ... } |
| public virtual void Add(params T[] items)  {      Update(items);  }    public virtual void Update(params T[] items)  {      using (var context = new Entities())      {          DbSet<T> dbSet = context.Set<T>();          foreach (T item in items)          {              dbSet.Add(item);              foreach (DbEntityEntry<IEntity> entry in context.ChangeTracker.Entries<IEntity>())              {                  IEntity entity = entry.Entity;                  entry.State = GetEntityState(entity.EntityState);              }          }          context.SaveChanges();      }  }    public virtual void Remove(params T[] items)  {      Update(items);  } |

Also note they key to all this working is that the client application must set the correct state of an entity as the repository will be totally dependent on this. Finally, below is some client side code that shows how to set the state of entities and passing a graph of objects to the business layer.

|  |
| --- |
| using Mm.BusinessLayer;  using Mm.DomainModel;  using System;  using System.Collections.Generic;    namespace Mm.ConsoleClientApplication  {      class Program      {          static void Main(string[] args)          {              IBusinessLayer businessLayer = new BuinessLayer();                /\* Create a department graph with two related employee objects \*/              Department it = new Department() { Name = "IT" };              it.Employees = new List<Employee>              {                  new Employee { FirstName="Donald", LastName="Duck", EntityState=EntityState.Added },                  new Employee { FirstName="Mickey", LastName="Mouse", EntityState=EntityState.Added }              };              it.EntityState = EntityState.Added;              businessLayer.AddDepartment(it);                /\*Add another employee to the IT department \*/              Employee employee = new Employee()              {                  FirstName = "Robin",                  LastName = "Hood",                  DepartmentId = it.DepartmentId,                  EntityState = EntityState.Added              };              /\* and change the name of the department \*/              it.Name = "Information Technology Department";              it.EntityState = EntityState.Modified;              foreach (Employee emp in it.Employees)                  emp.EntityState = EntityState.Unchanged;              it.Employees.Add(employee);              businessLayer.UpdateDepartment(it);                /\* Verify changes by quering for the updated department \*/              it = businessLayer.GetDepartmentByName("Information Technology Department");              if (it != null)              {                  Console.WriteLine(string.Format("Employees at the {0} department:", it.Name));                  foreach (Employee e in it.Employees)                      Console.WriteLine(string.Format("{0}, {1}", e.LastName, e.FirstName));              };                /\* Delete all entities \*/              it.EntityState = EntityState.Deleted;              foreach (Employee e in it.Employees)                  e.EntityState = EntityState.Deleted;              businessLayer.RemoveDepartment(it);                Console.ReadLine();          }      }  } |

