PERSISTENCE LAYER DESIGN

Seminar 5

Goals

- Learn how to design the persistence layer by applying design patterns and layers separation principles
- Use of Entity Framework to implement a persistence layer

Design Patterns for Persistence

- When designing the persistence layer we are interested in
  - Abstract the implementation details
  - Enable an eventual change of persistence technology
- In our previous sessions we have discussed two appropriate patterns
  - DAO pattern
  - Repository + UoW pattern

Persistence and Data Access Layer

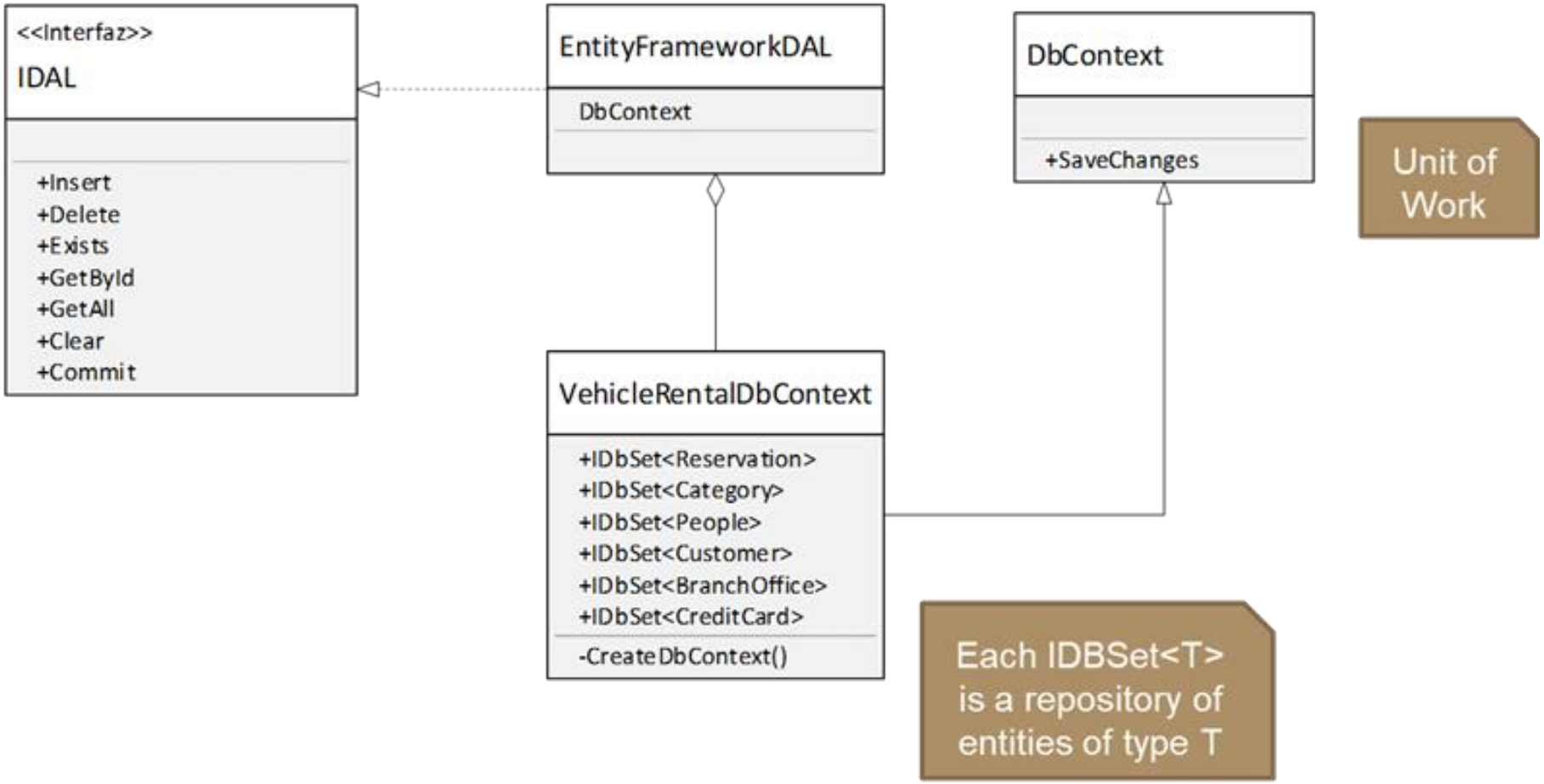
- **Persistence** is all the infrastructure to store/retrieve data to/from some storage (database, file, …)

- **Data Access Layer** (**DAL**) is the part of the persistence providing services to other layers(in our closed 3-layered architecture the DAL provides services to the business logic layer)

Persistence layer design

- In the laboratories we will develop a small-size desktop app with a simplified architecture but using standard design patterns and following good development practices.
  - Our persistence mechanism will be an ORM framework , **Entity Framework,** that implements the patterns **Repository + UoW** for accessing data.
  - To avoid the coupling between the data Access layer and the concrete implementation of the persistence we will use an adapter (a generic interface with all the needed persistence services) IDAL

Design


<<Interfaz>>
IDAL
+Insert
+Delete
+Exists
+GetById
+GetAll
+Clear
+Commit
EntityFrameworkDAL
DbContext
DbContext
+SaveChanges
Unit of Work
VehicleRentalDbContext
+IDbSet<Reservation>
+IDbSet<Category>
+IDbSet<People>
+IDbSet<Customer>
+IDbSet<BranchOffice>
+IDbSet<CreditCard>
-CreateDbContext()
Each IDBSet<T> is a repository of entities of type T

Interface IDAL

```
public interface IDAL
    {
        void Insert<T>(T entity) where T : class;
        void Delete<T>(T entity) where T : class;
        IEnumerable<T> GetAll<T>() where T : class;
        T GetById<T>(IComparable id) where T : class;
        bool Exists<T>(IComparable id) where T : class;
        void Clear<T>() where T : class;
        void Commit();
        IEnumerable<T> GetWhere<T>(Expression<Func<T,
bool>> predicate) where T : class;
    }
```

DAL with Entity Framework

```
public class EntityFrameworkDAL : IDAL
{
  private readonly DbContext dbContext;

  public DAL_EF(DbContext dbContext)
  {
    this.dbContext = dbContext;
  }

  public void Insert<T>(T entity) where T : class
  {
    dbContext.Set<T>().Add(entity);
  }

  public void Delete<T>(T entity) where T : class
  {
    dbContext.Set<T>().Remove(entity);
  }

  public IEnumerable<T> GetAll<T>() where T : class
  {
    return dbContext.Set<T>();
  }

  public T GetById<T>(IComparable id) where T : class
  {
    return dbContext.Set<T>().Find(id);
  }
```

DAL with Entity Framework

```
  public bool Exists<T>(IComparable id) where T : class
  {
    return dbContext.Set<T>().Find(id) != null;
  }

  public void Clear<T>() where T : class
  {
    dbContext.Set<T>().RemoveRange(dbContext.Set<T>());
  }

  public void Commit()
  {
    dbContext.SaveChanges();
  }

  public IEnumerable<T> GetWhere<T>(Expression<Func<T, bool>> predicate) where T : class
  {
    return dbContext.Set<T>().Where(predicate).AsEnumerable();
  }

}
```

Example DbContext

```
public class VehicleRentalDbContext : DbContext
{
  public IDbSet<BranchOffice> BranchOffices { get; set; }
  public IDbSet<Reservation> Reservations { get; set; }
  public IDbSet<Category> Categories { get; set; }
  public IDbSet<Person> People { get; set; }
  public IDbSet<Customer> Customers { get; set; }
  public IDbSet<CreditCard> CreditCards { get; set; }

  public VehicleRentalDbContext() : base("Name=VehicleRentalDbConnection") //connection string name
  {
     /*
     See DbContext.Configuration documentation
     */
     Configuration.LazyLoadingEnabled = true;
     Configuration.ProxyCreationEnabled = true;
}

…
```

Example DbContext

```
  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
    // Primary keys with non conventional name
    modelBuilder.Entity<Person>().HasKey(p => p.Dni);
    modelBuilder.Entity<Customer>().HasKey(c => c.Dni);
    modelBuilder.Entity<CreditCard>().HasKey(c => c.Digits);

     // Classes with more than one relationship
     modelBuilder.Entity<Reservation>().HasRequired(r => r.PickUpOffice).WithMany(o =>
o.PickUpReservations).WillCascadeOnDelete(false);
     modelBuilder.Entity<Reservation>().HasRequired(r => r.ReturnOffice).WithMany(o =>
o.ReturnReservations).WillCascadeOnDelete(false);
  }

  static VehicleRentalDbContext()
  {
    //Database.SetInitializer<VehicleRentalDbContext>(new CreateDatabaseIfNotExists<VehicleRentalDbContext>());
    Database.SetInitializer<VehicleRentalDbContext>(new
DropCreateDatabaseIfModelChanges<VehicleRentalDbContext>());
    //Database.SetInitializer<VehicleRentalDbContext>(new DropCreateDatabaseAlways<VehicleRentalDbContext>());
    //Database.SetInitializer<VehicleRentalDbContext>(new VehicleRentalDbInitializer());
    //Database.SetInitializer(new NullDatabaseInitializer<VehicleRentalDbContext>());
  }
}
```

Configuration of ORM with Fluent API

Database Initialization

Database Initialization

- **CreateDatabaseIfNotExists:** This is **default** initializer. As the name suggests, it will create the database if none exists as per the configuration. However, if you change the model class and then run the application with this initializer, then it will throw an exception.
- **DropCreateDatabaseIfModelChanges:** This initializer drops an existing database and creates a new database, if your model classes (entity classes) have been changed. So you don't have to worry about maintaining your database schema, when your model classes change.
- **DropCreateDatabaseAlways:** As the name suggests, this initializer drops an existing database every time you run the application, irrespective of whether your model classes have changed or not. This will be useful, when you want fresh database, every time you run the application, like while you are developing the application.
- **Custom DB Initializer:** You can also create your own custom initializer, if any of the above doesn't satisfy your requirements or you want to do some other process that initializes the database using the above initializer.

References

- Martin Fowler (2002). Patterns of Enterprise Application Architecture
- Scott Millet (2015) Patterns, Principles, and Practices of Domain-Driven Design

Online Material

- Repository Pattern en MSDN:
- Entity Framework Fluent API - Configuring and Mapping Properties and Types
- Entity Framework Fluent API - Relationships