Chương 5
Các thư viện hỗ trợ
Entity Framework

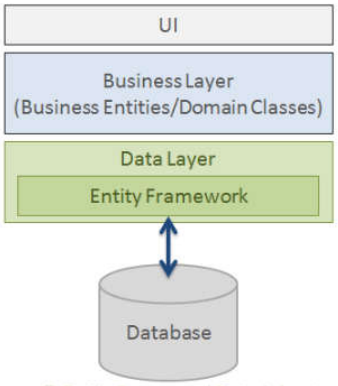
Khoa Công nghệ thông tin – Trường Đại học Đà Lạt

Agenda

- Entity Framework Basics
- Entity Framework Code-First Approach
- Inheritance Strategy in Entity Framework
- N-layer architecture
- Repository and Unit of Work Pattern
- Demo

What is Entity Framework?

- “Entity Framework is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write”
- It enables developers to work with data using objects of domain specific classes without focusing on the underlying database tables and columns where this data is stored



UI

Business Layer
(Business Entities/Domain Classes)

Data Layer

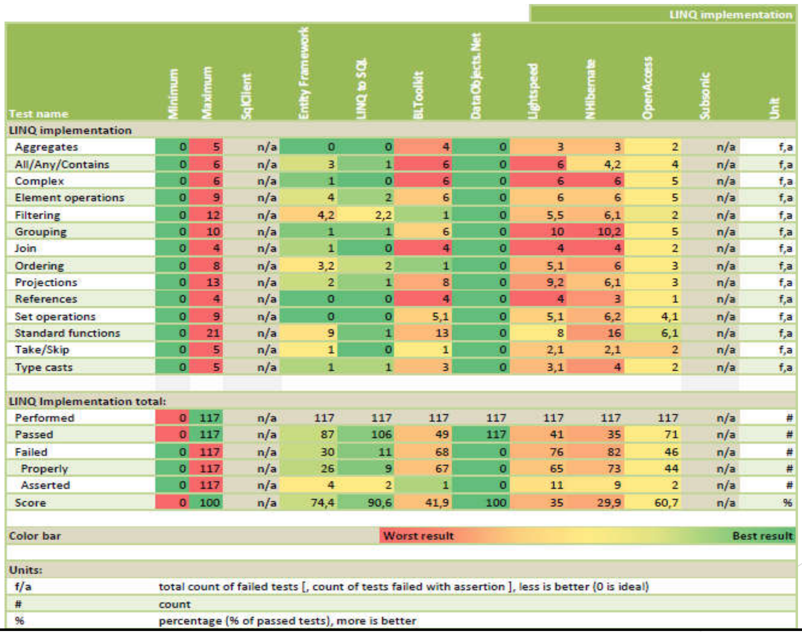
Entity Framework

Database

ORM Framework

.NET	Java	PHP
EF6, EF Core	Enterprise JavaBeans Entity Beans	RedBeanPHP
NHibernate	Java Data Objects	Doctrine ORM
BLToolkit	Castor	Eloquent ORM
Linq2Db	TopLink	Cycle ORM
Dapper	Hibernate	Solr
DbConnector	Spring DAO	Cycle ORM

.NET ORM Framework Performance Comparision



LINQ implementation

Test name	Minimum	Maximum	SqlClient	Entity Framework	LINQ to SQL	BLToolkit	DataObjects.Net	Lightspeed	NHibernate	OpenAccess	Subsonic	Unit
LINQ implementation												
Aggregates	0	5	n/a	0	0	4	0	3	3	2	n/a	f,a
All/Any/Contains	0	6	n/a	3	1	6	0	6	4,2	4	n/a	f,a
Complex	0	6	n/a	1	0	6	0	6	6	5	n/a	f,a
Element operations	0	9	n/a	4	2	6	0	6	6	5	n/a	f,a
Filtering	0	12	n/a	4,2	2,2	1	0	5,5	6,1	2	n/a	f,a
Grouping	0	10	n/a	1	1	6	0	10	10,2	5	n/a	f,a
Join	0	4	n/a	1	0	4	0	4	4	2	n/a	f,a
Ordering	0	8	n/a	3,2	2	1	0	5,1	6	3	n/a	f,a
Projections	0	13	n/a	2	1	8	0	9,2	6,1	3	n/a	f,a
References	0	4	n/a	0	0	4	0	4	3	1	n/a	f,a
Set operations	0	9	n/a	0	0	5,1	0	5,1	6,2	4,1	n/a	f,a
Standard functions	0	21	n/a	9	1	13	0	8	16	6,1	n/a	f,a
Take/Skip	0	5	n/a	1	0	1	0	2,1	2,1	2	n/a	f,a
Type casts	0	5	n/a	1	1	3	0	3,1	4	2	n/a	f,a
LINQ Implementation total:												
Performed	0	117	n/a	117	117	117	117	117	117	117	n/a	#
Passed	0	117	n/a	87	106	49	117	41	35	71	n/a	#
Failed	0	117	n/a	30	11	68	0	76	82	46	n/a	#
Properly	0	117	n/a	26	9	67	0	65	73	44	n/a	#
Asserted	0	117	n/a	4	2	1	0	11	9	2	n/a	#
Score	0	100	n/a	74,4	90,6	41,9	100	35	29,9	60,7	n/a	%

Color bar: Worst result … Best result

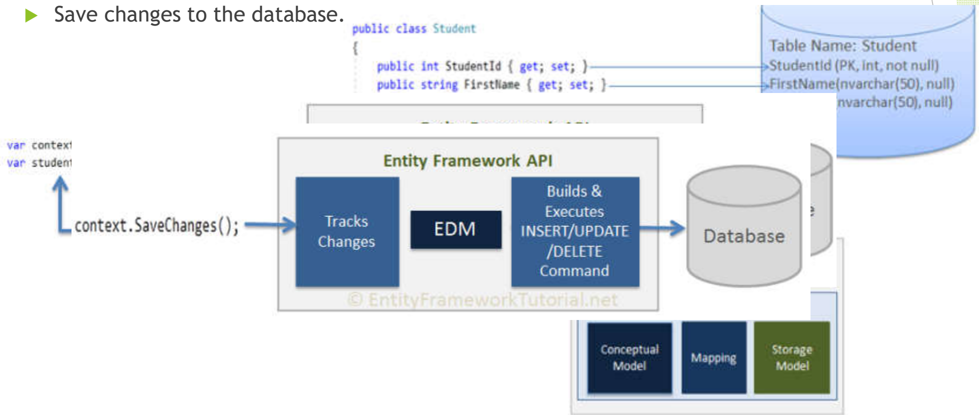
Units:

f/a	total count of failed tests [, count of tests failed with assertion], less is better (0 is ideal)
#	count
%	percentage (% of passed tests), more is better

http://www.ormeter.net/

How EF works?

- Entity Framework API (EF6 & EF Core) includes the ability to:
  - Map domain (entity) classes to the database schema
  - Translate & execute LINQ queries to SQL
  - Track changes occurred on entities during their lifetime
  - Save changes to the database.



```
public class Student
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
```

```
var context
var student
context.SaveChanges();
```

Table Name: Student
StudentId (PK, int, not null)
FirstName(nvarchar(50), null)
nvarchar(50), null)
Entity Framework API
Tracks Changes
EDM
Builds & Executes INSERT/UPDATE /DELETE Command
Database
© EntityFrameworkTutorial.net
Conceptual Model
Mapping
Storage Model

EF Architecture

Query Language: write queries against the object model

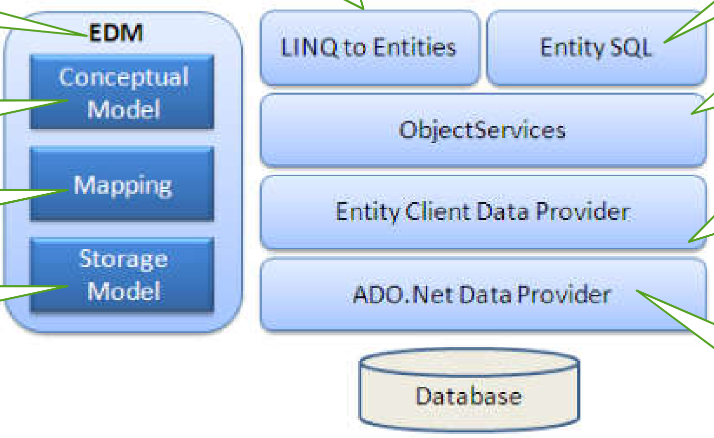
Another query language

Entity Data Model

Model classes and their relationships

How to map

Database design model



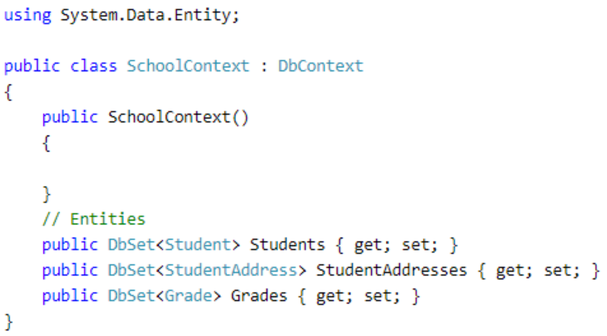
Accessing data from the database and returning back

Convert LINQ-to-Entities or Entity SQL queries into a SQL query

Communicates with the database using standard ADO.Net

Context Class in Entity Framework

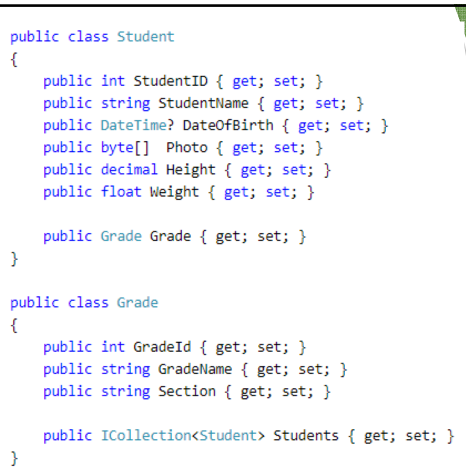
- The context class is a most important class while working with EF 6 or EF Core
- It represent a session with the underlying database using which you can perform CRUD (Create, Read, Update, Delete) operations.
- The context class is used to:
  - Query or save data to the database
  - Configure domain classes, database related mappings, change tracking settings, caching, transaction etc.



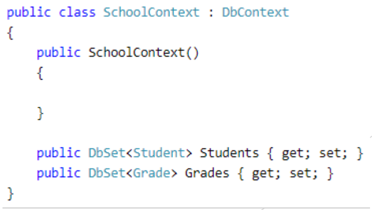
```
using System.Data.Entity;

public class SchoolContext : DbContext
{
    public SchoolContext()
    {

    }
    // Entities
    public DbSet<Student> Students { get; set; }
    public DbSet<StudentAddress> StudentAddresses { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```

Entity

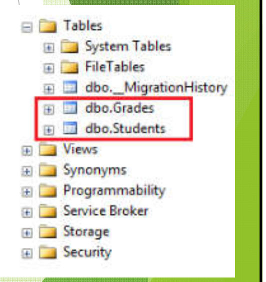
- An entity in Entity Framework is a class that maps to a database table.
- This class must be included as a DBSet<TEntity> type property in the DB Context class.
- EF API maps each entity to a table and each property of an entity to a column in the database.



```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Students { get; set; }
}
```

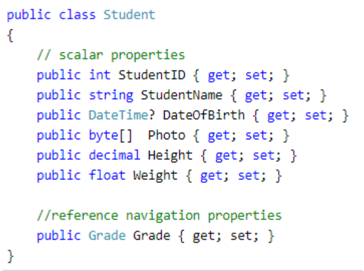


```
public class SchoolContext : DbContext
{
    public SchoolContext()
    {

    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```

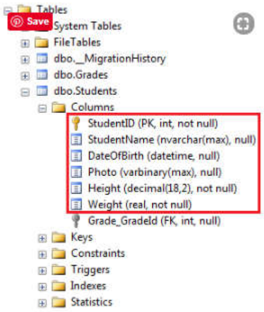


Entity Properties

- Two types of properties: Scalar Properties and Navigation Properties.
- Scalar properties:
  - The primitive type properties
  - Each scalar property maps to a column in the database table which stores an actual data

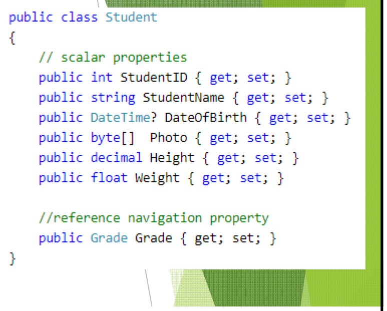


```
public class Student
{
    // scalar properties
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    //reference navigation properties
    public Grade Grade { get; set; }
}
```

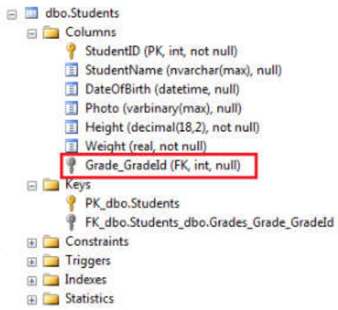

Tables
Save
System Tables
FileTables
dbo._MigrationHistory
dbo.Grades
dbo.Students
Columns
StudentID (PK, int, not null)
StudentName (nvarchar(max), null)
DateOfBirth (datetime, null)
Photo (varbinary(max), null)
Height (decimal(18,2), not null)
Weight (real, not null)
Grade_GradeId (FK, int, null)
Keys
Constraints
Triggers
Indexes
Statistics

Navigation Property

- The navigation property represents a relationship to another entity
- Two types of navigation properties: Reference Navigation and Collection Navigation
- Reference Navigation
  - A property of another entity type
  - Points to a single entity and represents multiplicity of one (1) in the entity relationships
  - EF API will create a ForeignKey column in the table for the navigation properties that points to a PrimaryKey of another table in the database
- Collection Navigation



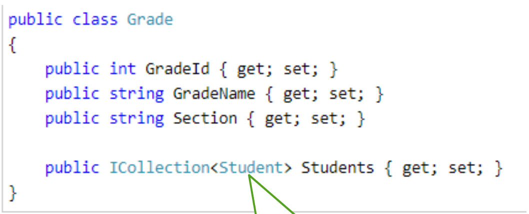
```
public class Student
{
    // scalar properties
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    //reference navigation property
    public Grade Grade { get; set; }
}
```



```
⊟ dbo.Students
   ⊟ Columns
        StudentID (PK, int, not null)
        StudentName (nvarchar(max), null)
        DateOfBirth (datetime, null)
        Photo (varbinary(max), null)
        Height (decimal(18,2), not null)
        Weight (real, not null)
        Grade_GradeId (FK, int, null)
   ⊟ Keys
        PK_dbo.Students
        FK_dbo.Students_dbo.Grades_Grade_GradeId
   ⊞ Constraints
   ⊞ Triggers
   ⊞ Indexes
   ⊞ Statistics
```

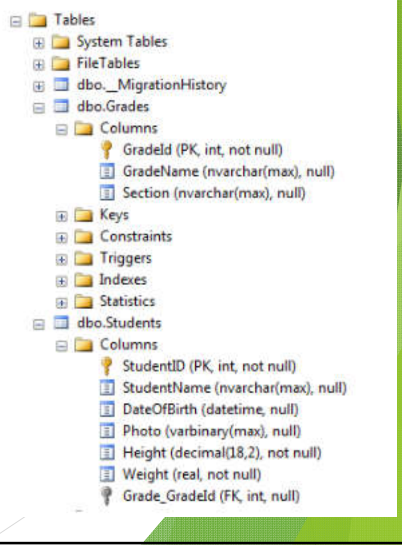
Collection Navigation

- A property of generic collection of an entity type
- It represents multiplicity of many (*).
- EF API does not create any column for the collection navigation property in the related table of an entity, but it creates a column in the table of an entity of generic collection.

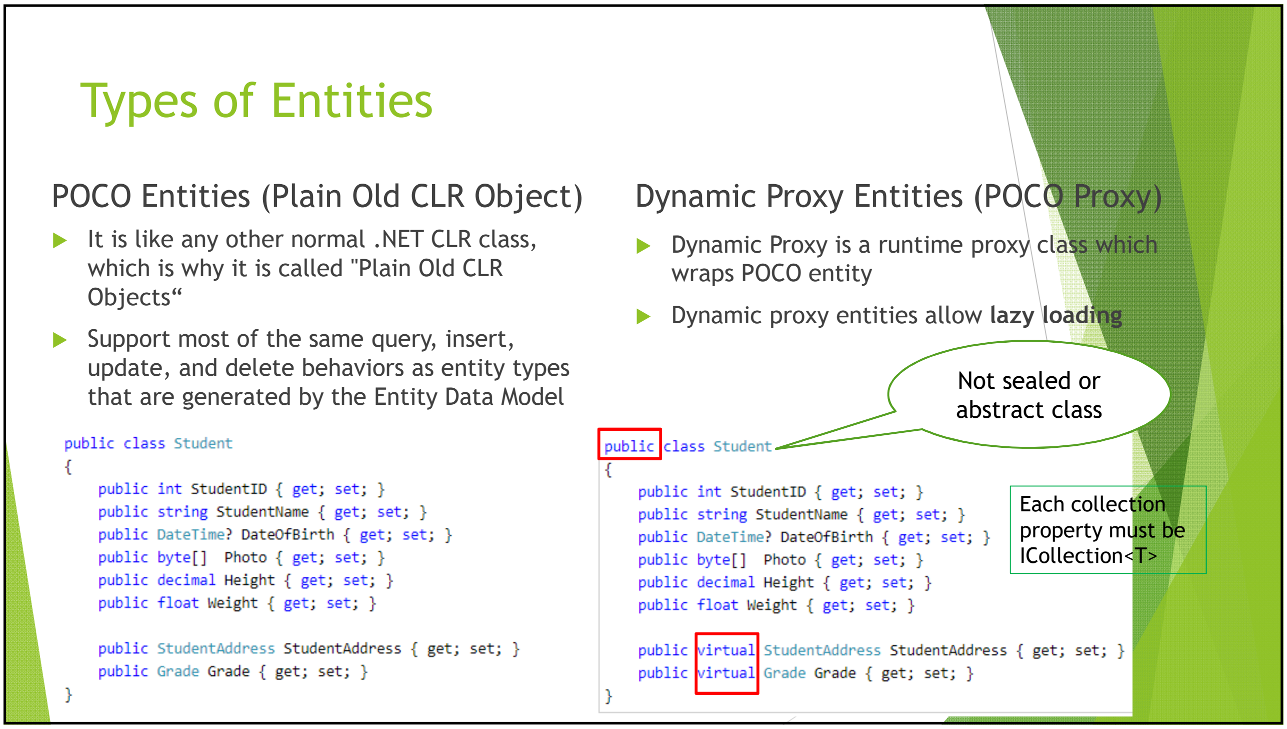


```
public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Students { get; set; }
}
```

generic type



```
⊟ Tables
   ⊞ System Tables
   ⊞ FileTables
   ⊞ dbo.__MigrationHistory
   ⊟ dbo.Grades
      ⊟ Columns
           GradeId (PK, int, not null)
           GradeName (nvarchar(max), null)
           Section (nvarchar(max), null)
      ⊞ Keys
      ⊞ Constraints
      ⊞ Triggers
      ⊞ Indexes
      ⊞ Statistics
   ⊟ dbo.Students
      ⊟ Columns
           StudentID (PK, int, not null)
           StudentName (nvarchar(max), null)
           DateOfBirth (datetime, null)
           Photo (varbinary(max), null)
           Height (decimal(18,2), not null)
           Weight (real, not null)
           Grade_GradeId (FK, int, null)
```



Types of Entities

POCO Entities (Plain Old CLR Object)

- It is like any other normal .NET CLR class, which is why it is called "Plain Old CLR Objects“
- Support most of the same query, insert, update, and delete behaviors as entity types that are generated by the Entity Data Model

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public StudentAddress StudentAddress { get; set; }
    public Grade Grade { get; set; }
}
```

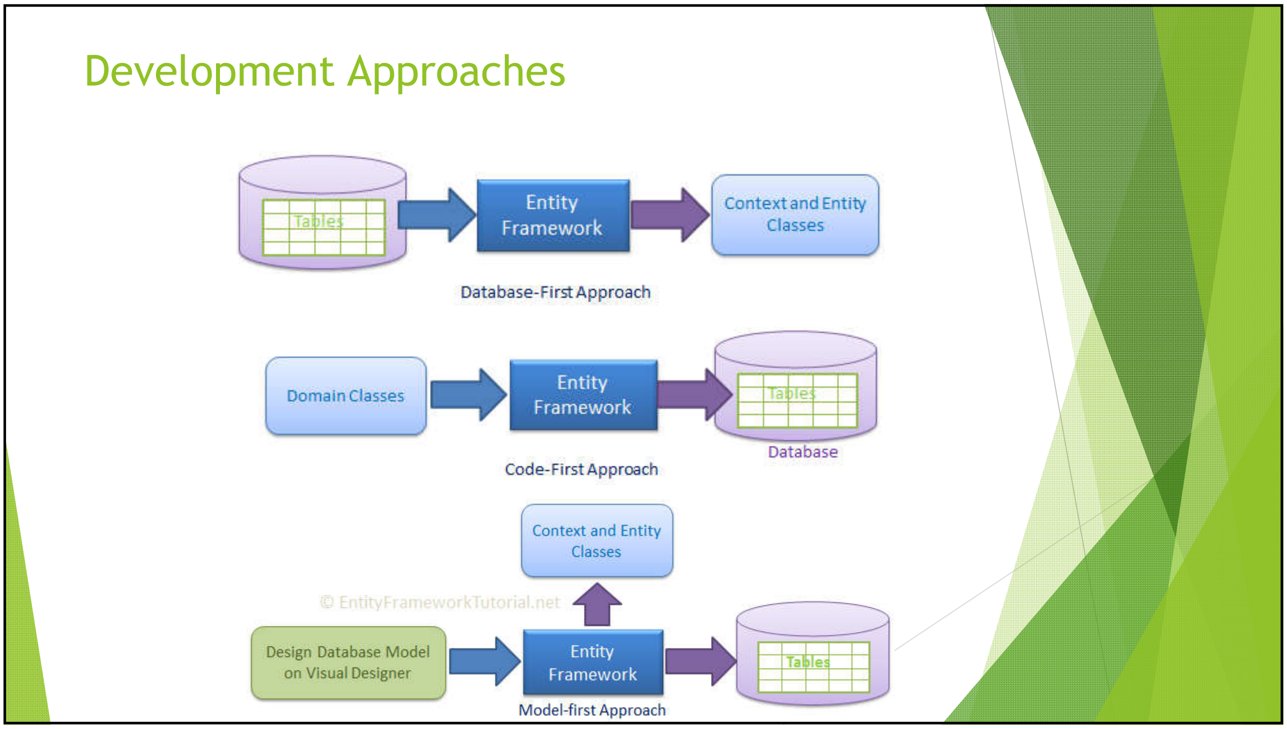
Dynamic Proxy Entities (POCO Proxy)

- Dynamic Proxy is a runtime proxy class which wraps POCO entity
- Dynamic proxy entities allow **lazy loading**

Not sealed or abstract class

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public virtual StudentAddress StudentAddress { get; set; }
    public virtual Grade Grade { get; set; }
}
```

Each collection property must be ICollection<T>



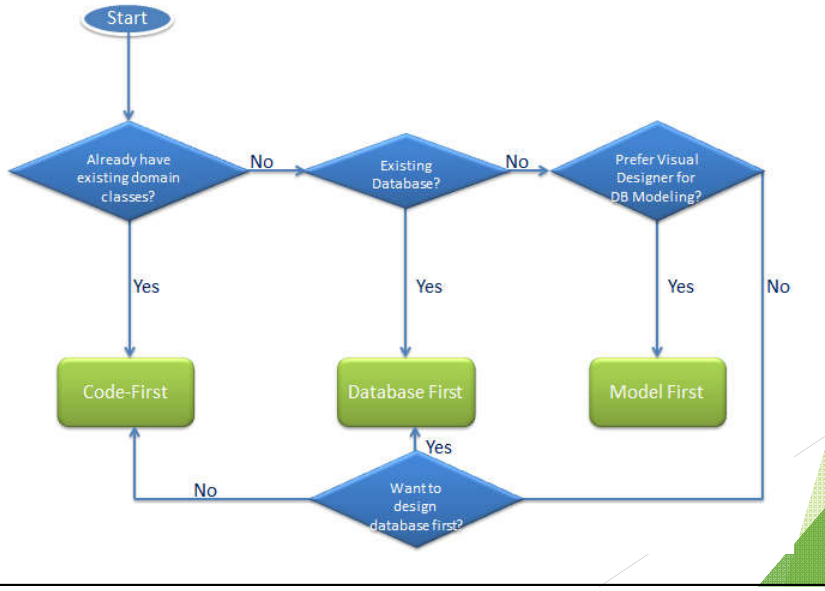
Development Approaches

Tables
Entity Framework
Context and Entity Classes
Database-First Approach

Domain Classes
Entity Framework
Tables
Database
Code-First Approach

Context and Entity Classes
© EntityFrameworkTutorial.net
Design Database Model on Visual Designer
Entity Framework
Tables
Model-first Approach

Development Approaches



Start

Already have existing domain classes?

Existing Database?

Prefer Visual Designer for DB Modeling?

No

No

Yes

Yes

Yes

No

Code-First

Database First

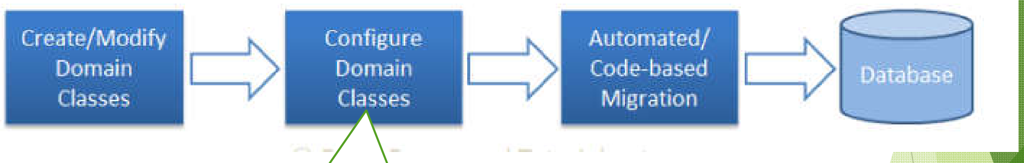
Model First

Yes

No

Want to design database first?

Code-first Appoach



Create/Modify Domain Classes

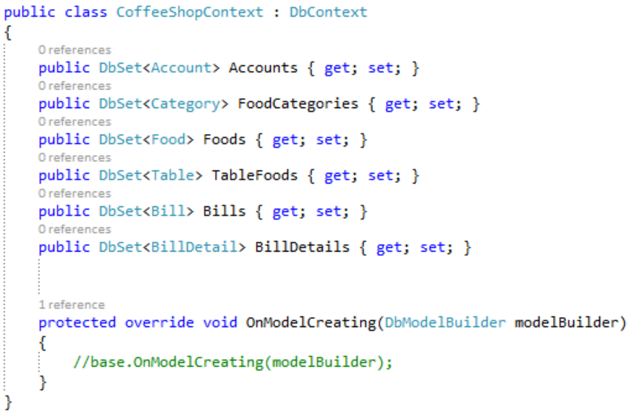
Configure Domain Classes

Automated/ Code-based Migration

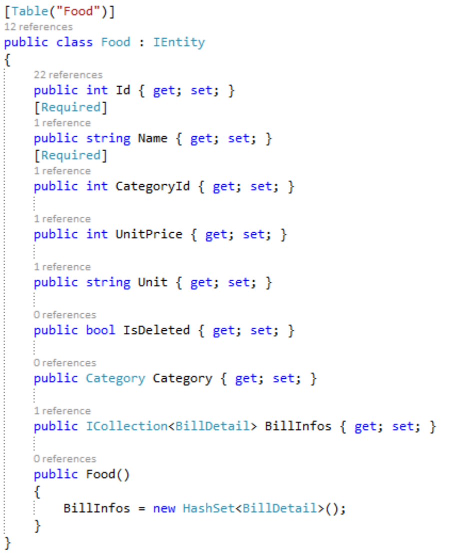
Database

using Fluent-API or data annotation attributes

Coffee Shop Management Application

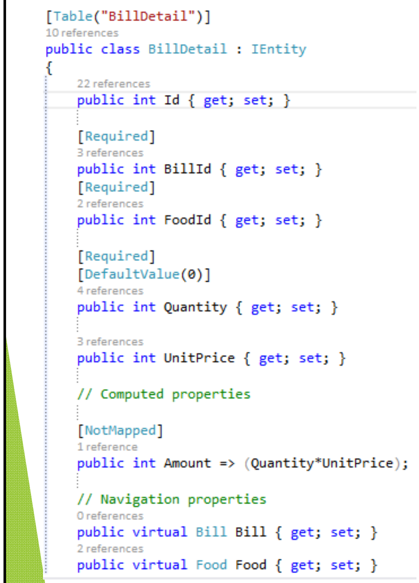


```
public class CoffeeShopContext : DbContext
{
    0 references
    public DbSet<Account> Accounts { get; set; }
    0 references
    public DbSet<Category> FoodCategories { get; set; }
    0 references
    public DbSet<Food> Foods { get; set; }
    0 references
    public DbSet<Table> TableFoods { get; set; }
    0 references
    public DbSet<Bill> Bills { get; set; }
    0 references
    public DbSet<BillDetail> BillDetails { get; set; }

    1 reference
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //base.OnModelCreating(modelBuilder);
    }
}
```

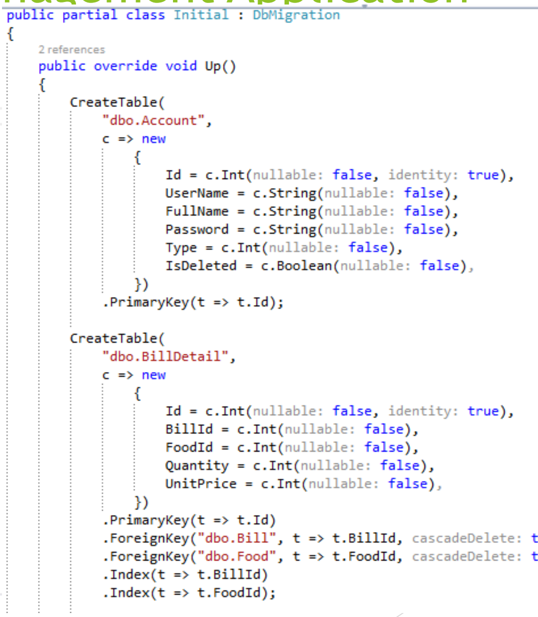


```
[Table("Food")]
12 references
public class Food : IEntity
{
    22 references
    public int Id { get; set; }
    [Required]
    1 reference
    public string Name { get; set; }
    [Required]
    1 reference
    public int CategoryId { get; set; }

    1 reference
    public int UnitPrice { get; set; }

    1 reference
    public string Unit { get; set; }

    0 references
    public bool IsDeleted { get; set; }

    0 references
    public Category Category { get; set; }

    1 reference
    public ICollection<BillDetail> BillInfos { get; set; }

    0 references
    public Food()
    {
        BillInfos = new HashSet<BillDetail>();
    }
}
```

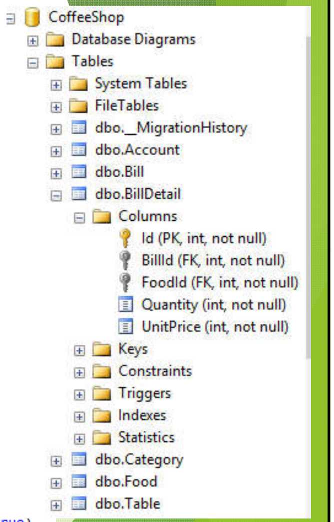
Coffee Shop Management Application



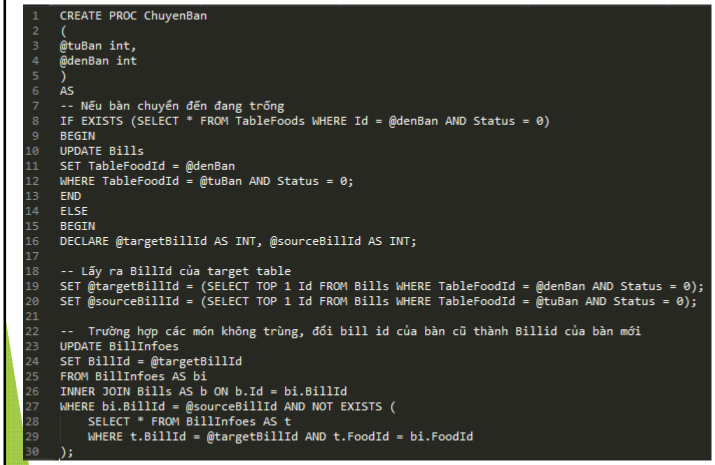
```
[Table("BillDetail")]
10 references
public class BillDetail : IEntity
{
    22 references
    public int Id { get; set; }

    [Required]
    3 references
    public int BillId { get; set; }
    [Required]
    2 references
    public int FoodId { get; set; }

    [Required]
    [DefaultValue(0)]
    4 references
    public int Quantity { get; set; }

    3 references
    public int UnitPrice { get; set; }

    // Computed properties

    [NotMapped]
    1 reference
    public int Amount => (Quantity*UnitPrice);

    // Navigation properties
    0 references
    public virtual Bill Bill { get; set; }
    2 references
    public virtual Food Food { get; set; }
```



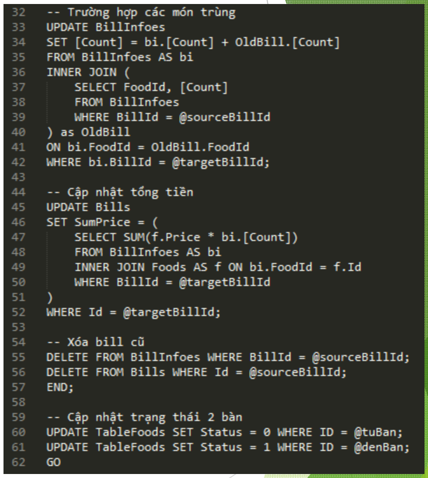
```
public partial class Initial : DbMigration
{
    2 references
    public override void Up()
    {
        CreateTable(
            "dbo.Account",
            c => new
                {
                    Id = c.Int(nullable: false, identity: true),
                    UserName = c.String(nullable: false),
                    FullName = c.String(nullable: false),
                    Password = c.String(nullable: false),
                    Type = c.Int(nullable: false),
                    IsDeleted = c.Boolean(nullable: false),
                })
            .PrimaryKey(t => t.Id);

        CreateTable(
            "dbo.BillDetail",
            c => new
                {
                    Id = c.Int(nullable: false, identity: true),
                    BillId = c.Int(nullable: false),
                    FoodId = c.Int(nullable: false),
                    Quantity = c.Int(nullable: false),
                    UnitPrice = c.Int(nullable: false),
                })
            .PrimaryKey(t => t.Id)
            .ForeignKey("dbo.Bill", t => t.BillId, cascadeDelete: true)
            .ForeignKey("dbo.Food", t => t.FoodId, cascadeDelete: true)
            .Index(t => t.BillId)
            .Index(t => t.FoodId);
```



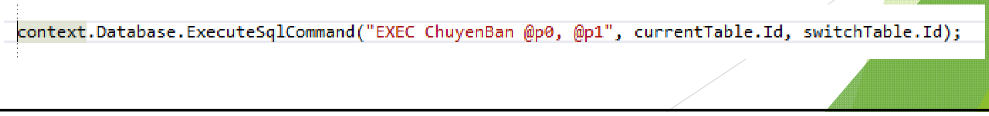
Coffee Shop Management Application



```sql
CREATE PROC ChuyenBan
(
@tuBan int,
@denBan int
)
AS
-- Nếu bàn chuyển đến đang trống
IF EXISTS (SELECT * FROM TableFoods WHERE Id = @denBan AND Status = 0)
BEGIN
UPDATE Bills
SET TableFoodId = @denBan
WHERE TableFoodId = @tuBan AND Status = 0;
END
ELSE
BEGIN
DECLARE @targetBillId AS INT, @sourceBillId AS INT;

-- Lấy ra BillId của target table
SET @targetBillId = (SELECT TOP 1 Id FROM Bills WHERE TableFoodId = @denBan AND Status = 0);
SET @sourceBillId = (SELECT TOP 1 Id FROM Bills WHERE TableFoodId = @tuBan AND Status = 0);

--  Trường hợp các món không trùng, đổi bill id của bàn cũ thành Billid của bàn mới
UPDATE BillInfoes
SET BillId = @targetBillId
FROM BillInfoes AS bi
INNER JOIN Bills AS b ON b.Id = bi.BillId
WHERE bi.BillId = @sourceBillId AND NOT EXISTS (
    SELECT * FROM BillInfoes AS t
    WHERE t.BillId = @targetBillId AND t.FoodId = bi.FoodId
);
```

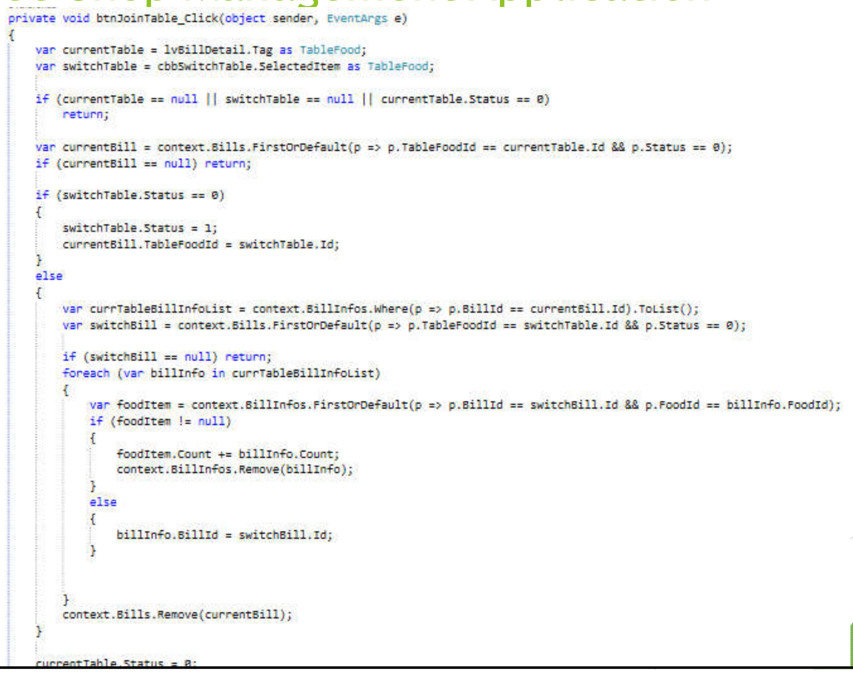


```sql
-- Trường hợp các món trùng
UPDATE BillInfoes
SET [Count] = bi.[Count] + OldBill.[Count]
FROM BillInfoes AS bi
INNER JOIN (
    SELECT FoodId, [Count]
    FROM BillInfoes
    WHERE BillId = @sourceBillId
) as OldBill
ON bi.FoodId = OldBill.FoodId
WHERE bi.BillId = @targetBillId;

-- Cập nhật tổng tiền
UPDATE Bills
SET SumPrice = (
    SELECT SUM(f.Price * bi.[Count])
    FROM BillInfoes AS bi
    INNER JOIN Foods AS f ON bi.FoodId = f.Id
    WHERE BillId = @targetBillId
)
WHERE Id = @targetBillId;

-- Xóa bill cũ
DELETE FROM BillInfoes WHERE BillId = @sourceBillId;
DELETE FROM Bills WHERE Id = @sourceBillId;
END;

-- Cập nhật trạng thái 2 bàn
UPDATE TableFoods SET Status = 0 WHERE ID = @tuBan;
UPDATE TableFoods SET Status = 1 WHERE ID = @denBan;
GO
```



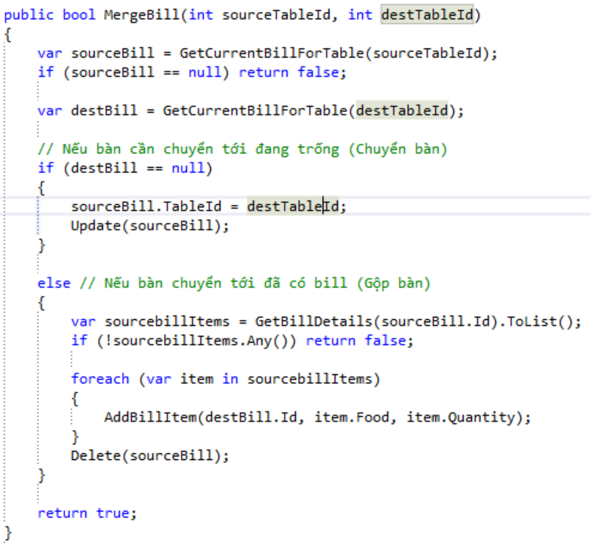
```csharp
context.Database.ExecuteSqlCommand("EXEC ChuyenBan @p0, @p1", currentTable.Id, switchTable.Id);
```

Coffee Shop Management Application

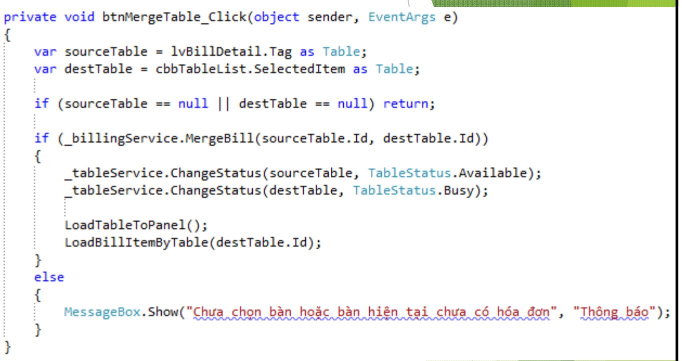


```csharp
private void btnJoinTable_Click(object sender, EventArgs e)
{
    var currentTable = lvBillDetail.Tag as TableFood;
    var switchTable = cbbSwitchTable.SelectedItem as TableFood;

    if (currentTable == null || switchTable == null || currentTable.Status == 0)
        return;

    var currentBill = context.Bills.FirstOrDefault(p => p.TableFoodId == currentTable.Id && p.Status == 0);
    if (currentBill == null) return;

    if (switchTable.Status == 0)
    {
        switchTable.Status = 1;
        currentBill.TableFoodId = switchTable.Id;
    }
    else
    {
        var currTableBillInfoList = context.BillInfos.Where(p => p.BillId == currentBill.Id).ToList();
        var switchBill = context.Bills.FirstOrDefault(p => p.TableFoodId == switchTable.Id && p.Status == 0);

        if (switchBill == null) return;
        foreach (var billInfo in currTableBillInfoList)
        {
            var foodItem = context.BillInfos.FirstOrDefault(p => p.BillId == switchBill.Id && p.FoodId == billInfo.FoodId);
            if (foodItem != null)
            {
                foodItem.Count += billInfo.Count;
                context.BillInfos.Remove(billInfo);
            }
            else
            {
                billInfo.BillId = switchBill.Id;
            }


        }
        context.Bills.Remove(currentBill);
    }

    currentTable.Status = 0;
```

Coffee Shop Management Application



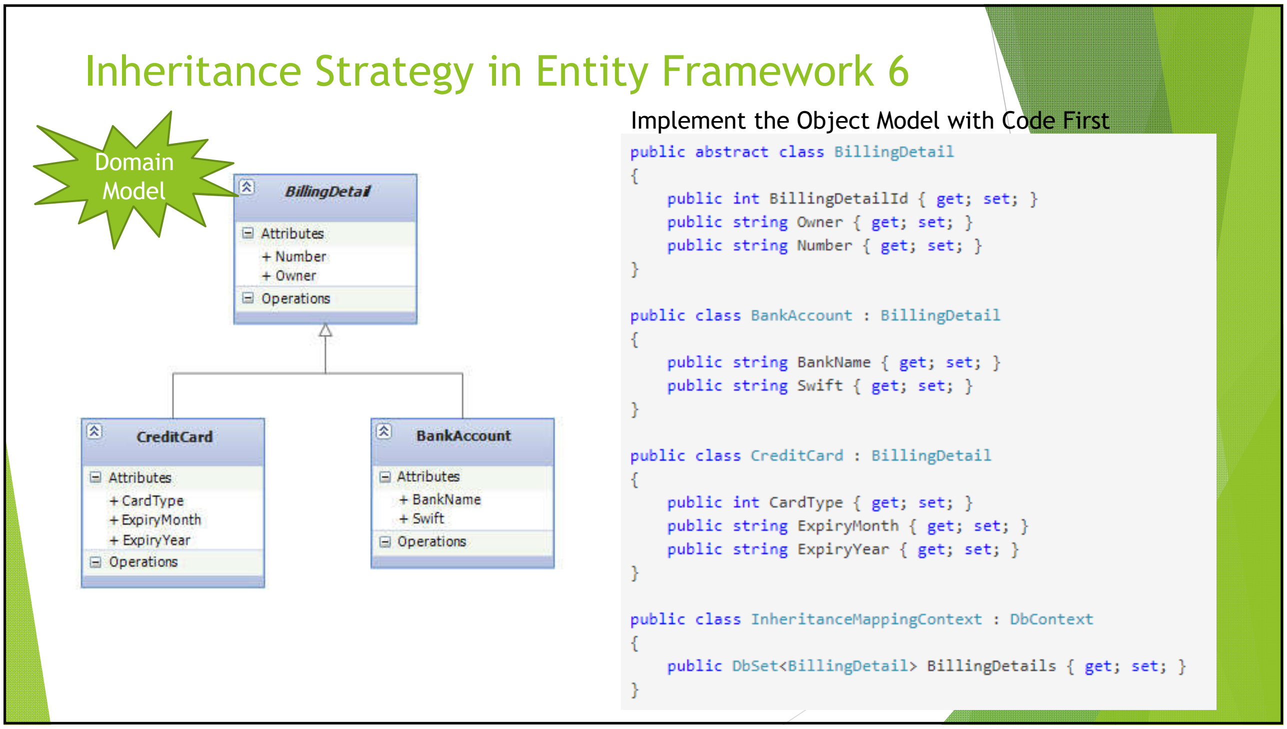
```
public bool MergeBill(int sourceTableId, int destTableId)
{
    var sourceBill = GetCurrentBillForTable(sourceTableId);
    if (sourceBill == null) return false;

    var destBill = GetCurrentBillForTable(destTableId);

    // Nếu bàn cần chuyển tới đang trống (Chuyển bàn)
    if (destBill == null)
    {
        sourceBill.TableId = destTableId;
        Update(sourceBill);
    }

    else // Nếu bàn chuyển tới đã có bill (Gộp bàn)
    {
        var sourcebillItems = GetBillDetails(sourceBill.Id).ToList();
        if (!sourcebillItems.Any()) return false;

        foreach (var item in sourcebillItems)
        {
            AddBillItem(destBill.Id, item.Food, item.Quantity);
        }
        Delete(sourceBill);
    }

    return true;
}
```



```
private void btnMergeTable_Click(object sender, EventArgs e)
{
    var sourceTable = lvBillDetail.Tag as Table;
    var destTable = cbbTableList.SelectedItem as Table;

    if (sourceTable == null || destTable == null) return;

    if (_billingService.MergeBill(sourceTable.Id, destTable.Id))
    {
        _tableService.ChangeStatus(sourceTable, TableStatus.Available);
        _tableService.ChangeStatus(destTable, TableStatus.Busy);

        LoadTableToPanel();
        LoadBillItemByTable(destTable.Id);
    }
    else
    {
        MessageBox.Show("Chưa chọn bàn hoặc bàn hiện tại chưa có hóa đơn", "Thông báo");
    }
}
```

Inheritance Strategy in Entity Framework 6

- Problem:
  - EF creates database tables for each concrete domain class. However, you can design your domain classes using inheritance.
  - Object-oriented techniques include "has a" and "is a" relationships, whereas SQL-based relational model has only a "has a" relationship between tables.
  - SQL database management systems don't support type inheritance
- How would you map object-oriented domain classes with the relational database?



Inheritance Strategy in Entity Framework 6

Domain Model

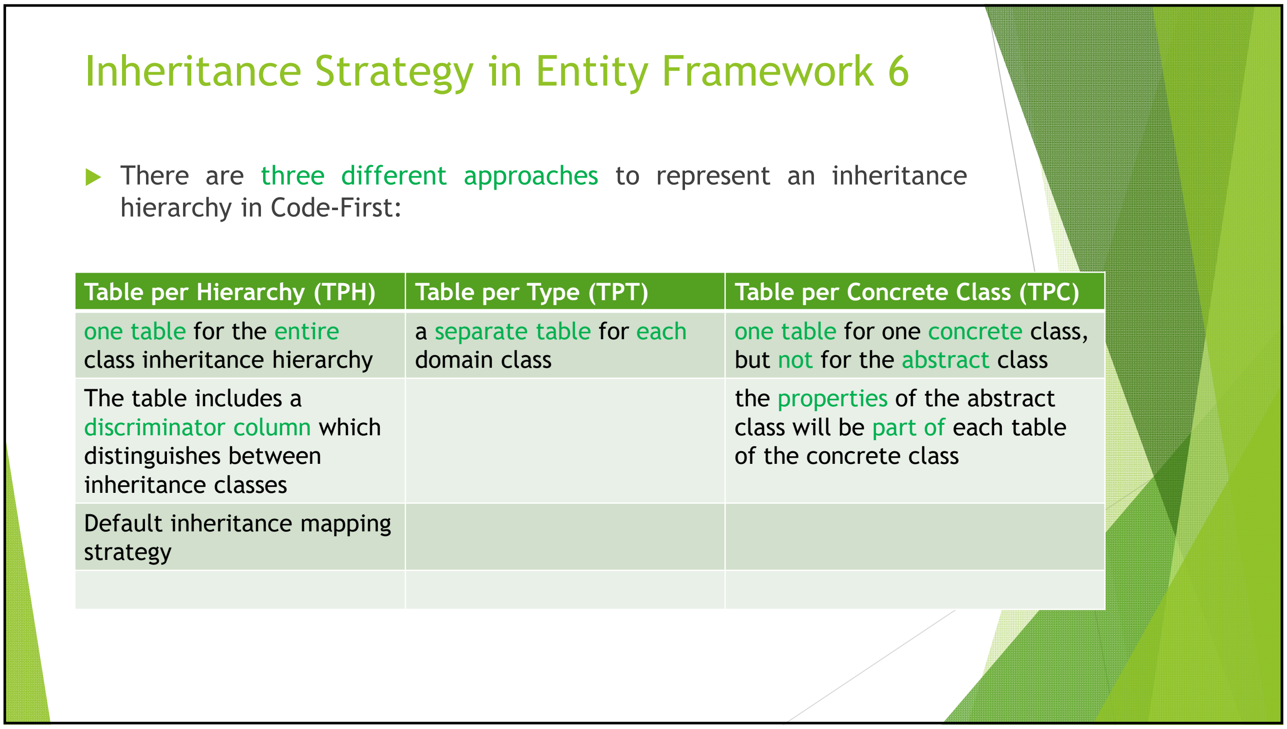
BillingDetail
- Attributes
  - + Number
  - + Owner
- Operations

CreditCard
- Attributes
  - + CardType
  - + ExpiryMonth
  - + ExpiryYear
- Operations

BankAccount
- Attributes
  - + BankName
  - + Swift
- Operations

Implement the Object Model with Code First

```
public abstract class BillingDetail
{
    public int BillingDetailId { get; set; }
    public string Owner { get; set; }
    public string Number { get; set; }
}

public class BankAccount : BillingDetail
{
    public string BankName { get; set; }
    public string Swift { get; set; }
}

public class CreditCard : BillingDetail
{
    public int CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}

public class InheritanceMappingContext : DbContext
{
    public DbSet<BillingDetail> BillingDetails { get; set; }
}
```



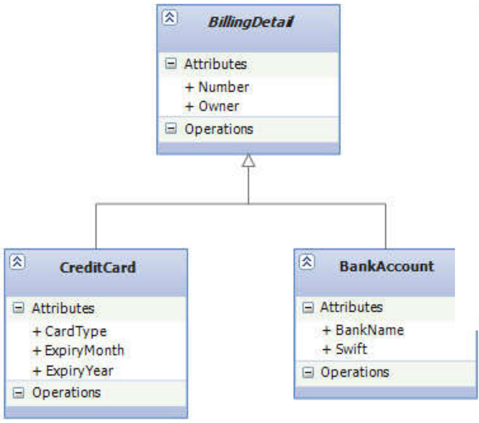
Inheritance Strategy in Entity Framework 6

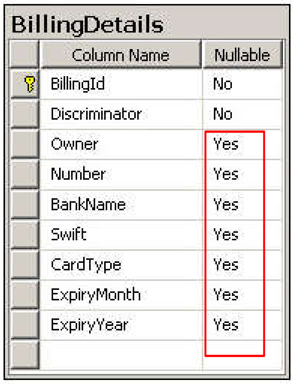
- There are three different approaches to represent an inheritance hierarchy in Code-First:

Table per Hierarchy (TPH)	Table per Type (TPT)	Table per Concrete Class (TPC)
one table for the entire class inheritance hierarchy	a separate table for each domain class	one table for one concrete class, but not for the abstract class
The table includes a discriminator column which distinguishes between inheritance classes		the properties of the abstract class will be part of each table of the concrete class
Default inheritance mapping strategy		

Inheritance Strategy in Entity Framework 6

- Table per Hierarchy (TPH)



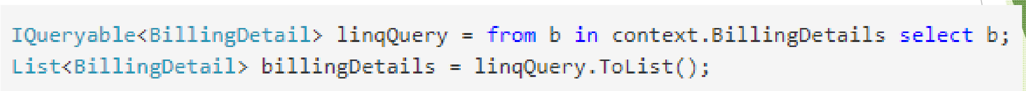


BillingDetails

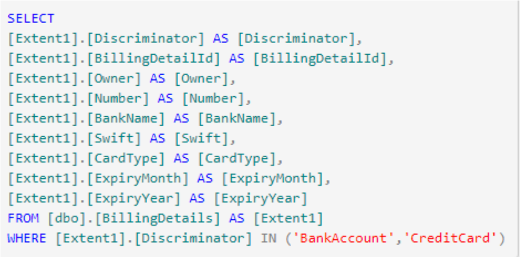
Column Name	Nullable
BillingId	No
Discriminator	No
Owner	Yes
Number	Yes
BankName	Yes
Swift	Yes
CardType	Yes
ExpiryMonth	Yes
ExpiryYear	Yes

Inheritance Strategy in Entity Framework 6

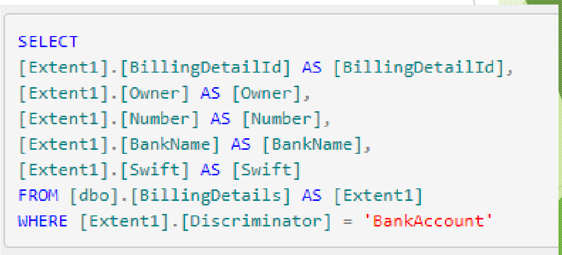
- Table per Hierarchy (TPH)



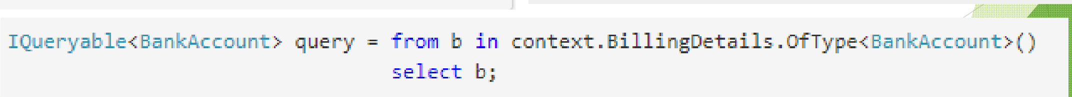
```
IQueryable<BillingDetail> linqQuery = from b in context.BillingDetails select b;
List<BillingDetail> billingDetails = linqQuery.ToList();
```



```
SELECT
[Extent1].[Discriminator] AS [Discriminator],
[Extent1].[BillingDetailId] AS [BillingDetailId],
[Extent1].[Owner] AS [Owner],
[Extent1].[Number] AS [Number],
[Extent1].[BankName] AS [BankName],
[Extent1].[Swift] AS [Swift],
[Extent1].[CardType] AS [CardType],
[Extent1].[ExpiryMonth] AS [ExpiryMonth],
[Extent1].[ExpiryYear] AS [ExpiryYear]
FROM [dbo].[BillingDetails] AS [Extent1]
WHERE [Extent1].[Discriminator] IN ('BankAccount','CreditCard')
```



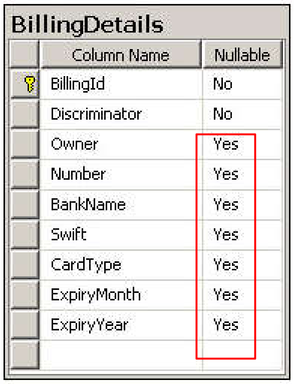
```
SELECT
[Extent1].[BillingDetailId] AS [BillingDetailId],
[Extent1].[Owner] AS [Owner],
[Extent1].[Number] AS [Number],
[Extent1].[BankName] AS [BankName],
[Extent1].[Swift] AS [Swift]
FROM [dbo].[BillingDetails] AS [Extent1]
WHERE [Extent1].[Discriminator] = 'BankAccount'
```



```
IQueryable<BankAccount> query = from b in context.BillingDetails.OfType<BankAccount>()
                                select b;
```

Inheritance Strategy in Entity Framework 6

- Table per Hierarchy (TPH) – Problems:
  - Major problem: If subclasses each define several non-nullable properties, the loss of NOT NULL constraints may be a serious problem from the point of view of data integrity
  - Another important issue is normalization. Denormalization for performance can be misleading, because it sacrifices long-term stability, maintainability, and the integrity of data.

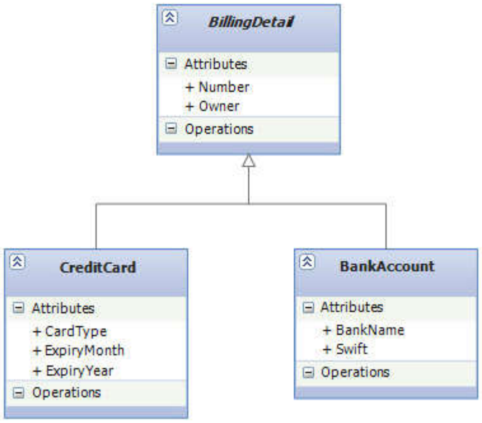


BillingDetails

Column Name	Nullable
BillingId	No
Discriminator	No
Owner	Yes
Number	Yes
BankName	Yes
Swift	Yes
CardType	Yes
ExpiryMonth	Yes
ExpiryYear	Yes

Inheritance Strategy in Entity Framework 6

- Table per Type (TPT)



BillingDetai

Attributes

+ Number

+ Owner

Operations

CreditCard

Attributes

+ CardType

+ ExpiryMonth

+ ExpiryYear

Operations

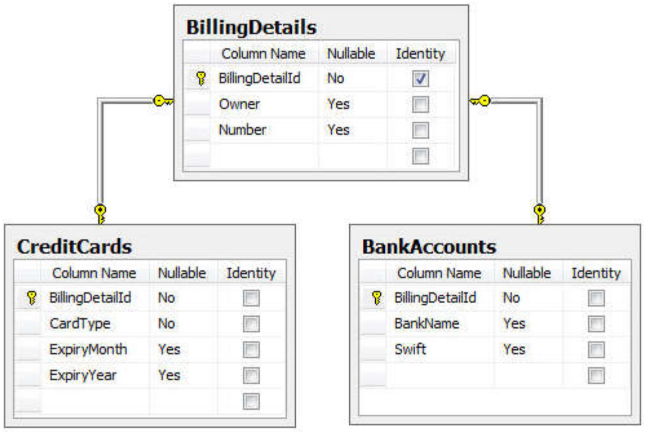
BankAccount

Attributes

+ BankName

+ Swift

Operations



BillingDetails

Column Name	Nullable	Identity
BillingDetailId	No	☑
Owner	Yes	□
Number	Yes	□
		□

CreditCards

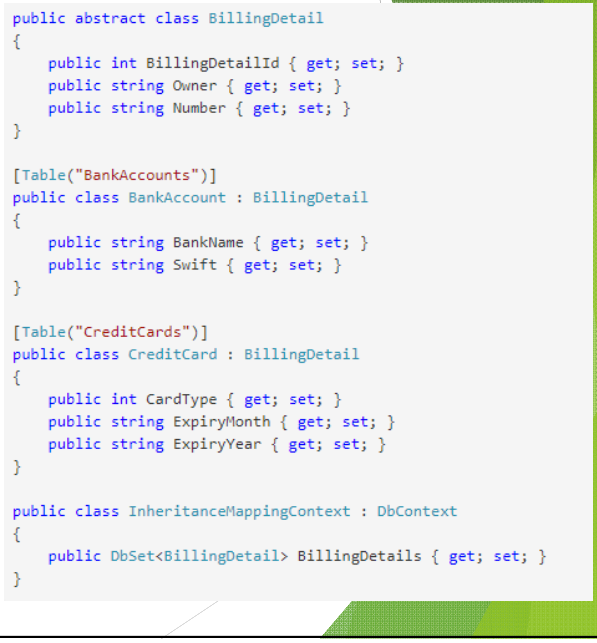
Column Name	Nullable	Identity
BillingDetailId	No	□
CardType	No	□
ExpiryMonth	Yes	□
ExpiryYear	Yes	□
		□

BankAccounts

Column Name	Nullable	Identity
BillingDetailId	No	□
BankName	Yes	□
Swift	Yes	□
		□

Inheritance Strategy in Entity Framework 6

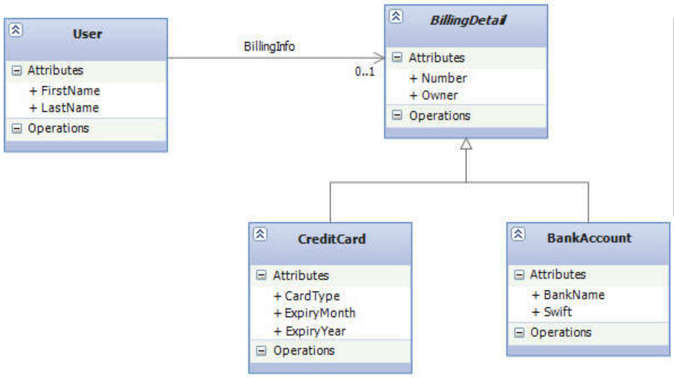
- Table per Type (TPT)
  - Advantages: SQL schema is normalized
    - schema evolution is straightforward
    - Integrity constraint definition are also straightforward

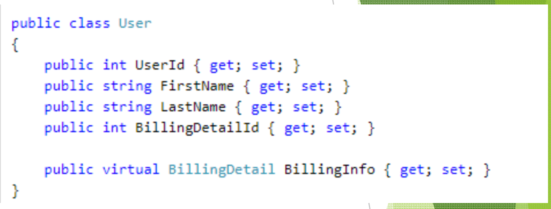


```
public abstract class BillingDetail
{
    public int BillingDetailId { get; set; }
    public string Owner { get; set; }
    public string Number { get; set; }
}

[Table("BankAccounts")]
public class BankAccount : BillingDetail
{
    public string BankName { get; set; }
    public string Swift { get; set; }
}

[Table("CreditCards")]
public class CreditCard : BillingDetail
{
    public int CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}

public class InheritanceMappingContext : DbContext
{
    public DbSet<BillingDetail> BillingDetails { get; set; }
}
```

Inheritance Strategy in Entity Framework 6

- Table per Type (TPT):





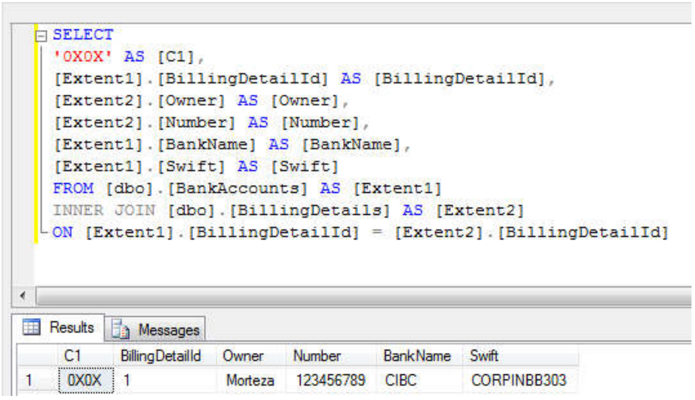
```
public class User
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int BillingDetailId { get; set; }

    public virtual BillingDetail BillingInfo { get; set; }
}
```

- Advantages : the ability to handle polymorphic associations

Inheritance Strategy in Entity Framework 6

- Table per Type (TPT) - Considerations :

```
var query = from b in context.BillingDetails.OfType<BankAccount>() select b;
```



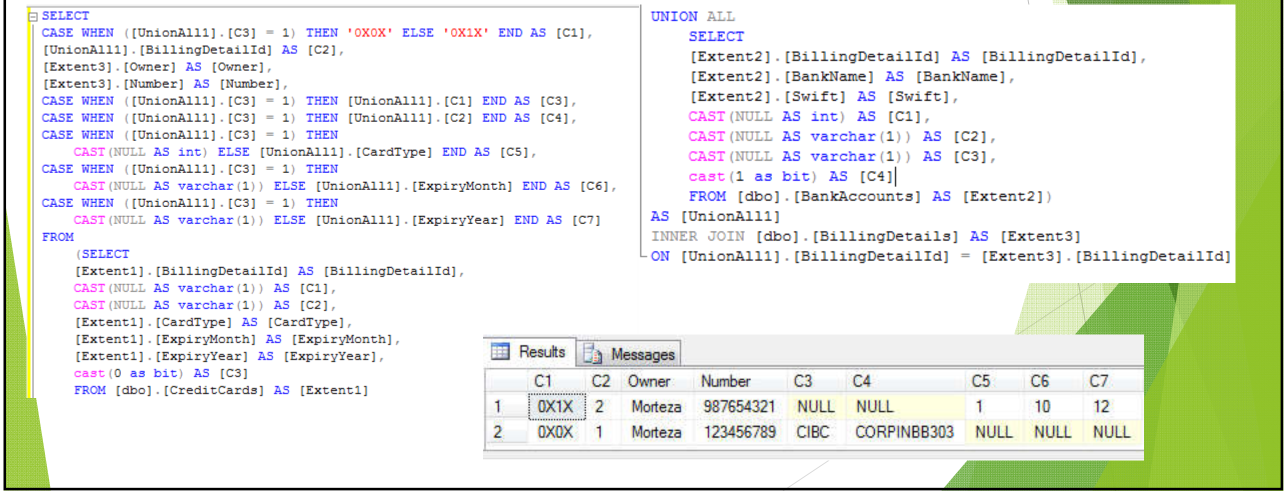
```
SELECT
'0X0X' AS [C1],
[Extent1].[BillingDetailId] AS [BillingDetailId],
[Extent2].[Owner] AS [Owner],
[Extent2].[Number] AS [Number],
[Extent1].[BankName] AS [BankName],
[Extent1].[Swift] AS [Swift]
FROM [dbo].[BankAccounts] AS [Extent1]
INNER JOIN [dbo].[BillingDetails] AS [Extent2]
ON [Extent1].[BillingDetailId] = [Extent2].[BillingDetailId]
```

	C1	BillingDetailId	Owner	Number	BankName	Swift
1	0X0X	1	Morteza	123456789	CIBC	CORPINBB303

Inheritance Strategy in Entity Framework 6

- Table per Type (TPT) - Considerations :

```
var query = from b in context.BillingDetails select b;
```



```
SELECT
CASE WHEN ([UnionAll1].[C3] = 1) THEN '0X0X' ELSE '0X1X' END AS [C1],
[UnionAll1].[BillingDetailId] AS [C2],
[Extent3].[Owner] AS [Owner],
[Extent3].[Number] AS [Number],
CASE WHEN ([UnionAll1].[C3] = 1) THEN [UnionAll1].[C1] END AS [C3],
CASE WHEN ([UnionAll1].[C3] = 1) THEN [UnionAll1].[C2] END AS [C4],
CASE WHEN ([UnionAll1].[C3] = 1) THEN
    CAST(NULL AS int) ELSE [UnionAll1].[CardType] END AS [C5],
CASE WHEN ([UnionAll1].[C3] = 1) THEN
    CAST(NULL AS varchar(1)) ELSE [UnionAll1].[ExpiryMonth] END AS [C6],
CASE WHEN ([UnionAll1].[C3] = 1) THEN
    CAST(NULL AS varchar(1)) ELSE [UnionAll1].[ExpiryYear] END AS [C7]
FROM
    (SELECT
    [Extent1].[BillingDetailId] AS [BillingDetailId],
    CAST(NULL AS varchar(1)) AS [C1],
    CAST(NULL AS varchar(1)) AS [C2],
    [Extent1].[CardType] AS [CardType],
    [Extent1].[ExpiryMonth] AS [ExpiryMonth],
    [Extent1].[ExpiryYear] AS [ExpiryYear],
    cast(0 as bit) AS [C3]
    FROM [dbo].[CreditCards] AS [Extent1]
UNION ALL
    SELECT
    [Extent2].[BillingDetailId] AS [BillingDetailId],
    [Extent2].[BankName] AS [BankName],
    [Extent2].[Swift] AS [Swift],
    CAST(NULL AS int) AS [C1],
    CAST(NULL AS varchar(1)) AS [C2],
    CAST(NULL AS varchar(1)) AS [C3],
    cast(1 as bit) AS [C4]
    FROM [dbo].[BankAccounts] AS [Extent2])
AS [UnionAll1]
INNER JOIN [dbo].[BillingDetails] AS [Extent3]
ON [UnionAll1].[BillingDetailId] = [Extent3].[BillingDetailId]
```

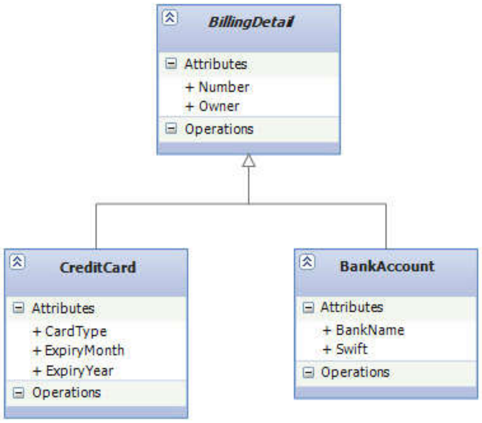
	C1	C2	Owner	Number	C3	C4	C5	C6	C7
1	0X1X	2	Morteza	987654321	NULL	NULL	1	10	12
2	0X0X	1	Morteza	123456789	CIBC	CORPINBB303	NULL	NULL	NULL

Inheritance Strategy in Entity Framework 6

- Table per Type (TPT) - Considerations :
  - Performance can be unacceptable for complex class hierarchies because queries always require a join across many tables
  - This mapping strategy is more difficult to implement by hand. This is an important consideration if you plan to use handwritten SQL in your application

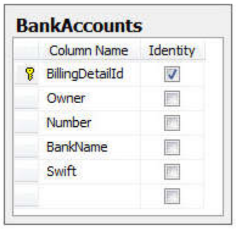
Inheritance Strategy in Entity Framework 6

- Table per Concrete Type (TPC):

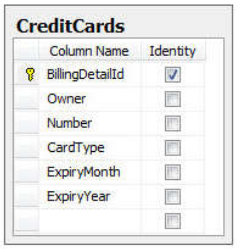

BillingDetail
Attributes
+ Number
+ Owner
Operations

CreditCard
Attributes
+ CardType
+ ExpiryMonth
+ ExpiryYear
Operations

BankAccount
Attributes
+ BankName
+ Swift
Operations


BankAccounts

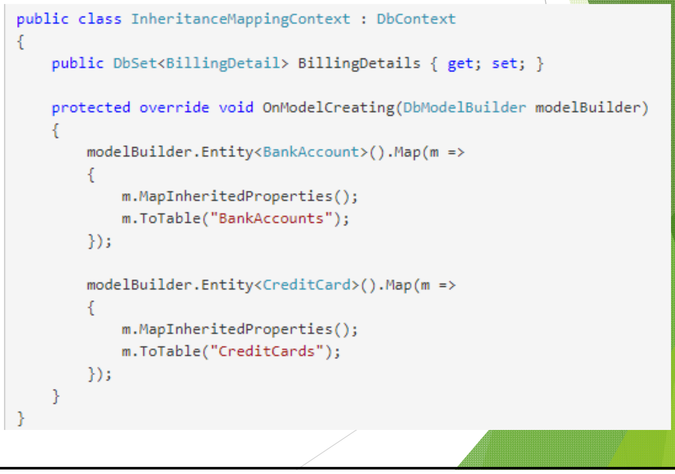
	Column Name	Identity
🔑	BillingDetailId	☑
	Owner	□
	Number	□
	BankName	□
	Swift	□
		□


CreditCards

	Column Name	Identity
🔑	BillingDetailId	☑
	Owner	□
	Number	□
	CardType	□
	ExpiryMonth	□
	ExpiryYear	□
		□

Inheritance Strategy in Entity Framework 6

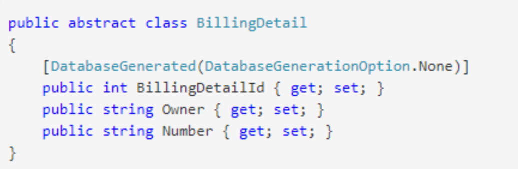
- Table per Concrete Type (TPC):
  - The SQL schema is not aware of the inheritance
  - There is no relationship between the database tables, except for the fact that they share some similar columns.



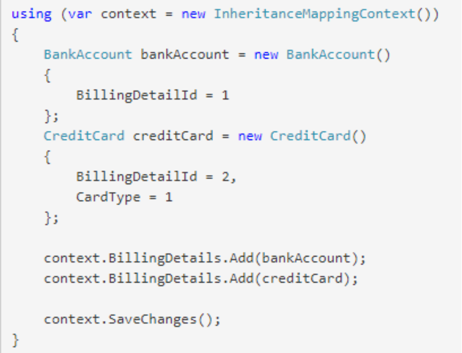
```
public class InheritanceMappingContext : DbContext
{
    public DbSet<BillingDetail> BillingDetails { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<BankAccount>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("BankAccounts");
        });

        modelBuilder.Entity<CreditCard>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("CreditCards");
        });
    }
}
```

Inheritance Strategy in Entity Framework 6

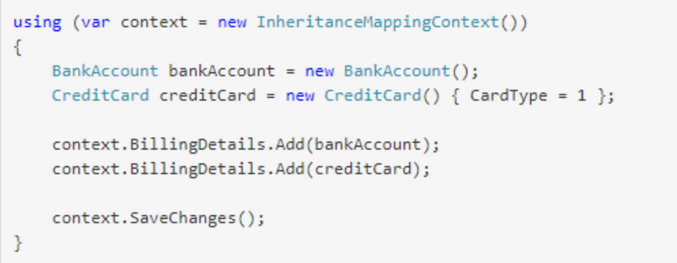
- Table per Concrete Type (TPC) - Problems :
  - Identity problem



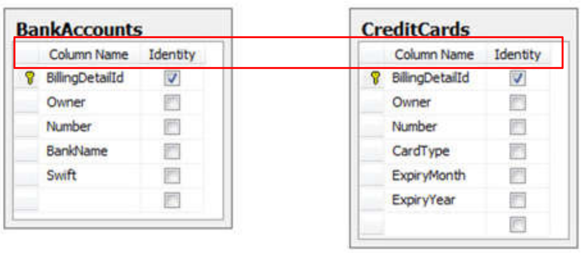
```
public abstract class BillingDetail
{
    [DatabaseGenerated(DatabaseGenerationOption.None)]
    public int BillingDetailId { get; set; }
    public string Owner { get; set; }
    public string Number { get; set; }
}
```



```
using (var context = new InheritanceMappingContext())
{
    BankAccount bankAccount = new BankAccount()
    {
        BillingDetailId = 1
    };
    CreditCard creditCard = new CreditCard()
    {
        BillingDetailId = 2,
        CardType = 1
    };

    context.BillingDetails.Add(bankAccount);
    context.BillingDetails.Add(creditCard);

    context.SaveChanges();
}
```



```
using (var context = new InheritanceMappingContext())
{
    BankAccount bankAccount = new BankAccount();
    CreditCard creditCard = new CreditCard() { CardType = 1 };

    context.BillingDetails.Add(bankAccount);
    context.BillingDetails.Add(creditCard);

    context.SaveChanges();
}
```



BankAccounts

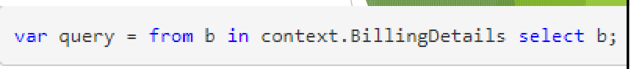
	Column Name	Identity
🔑	BillingDetailId	☑
	Owner	□
	Number	□
	BankName	□
	Swift	□
		□

CreditCards

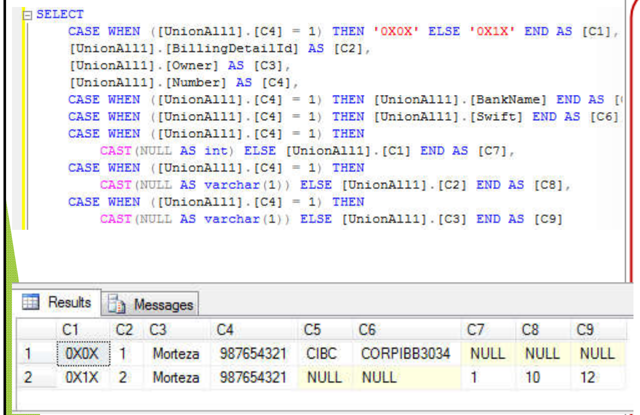
	Column Name	Identity
🔑	BillingDetailId	☑
	Owner	□
	Number	□
	CardType	□
	ExpiryMonth	□
	ExpiryYear	□
		□

Inheritance Strategy in Entity Framework 6

- Table per Concrete Type (TPC) - Problems:
  - Polymorphic Associations with TPC is Problematic
  - Schema Evolution with TPC is Complex

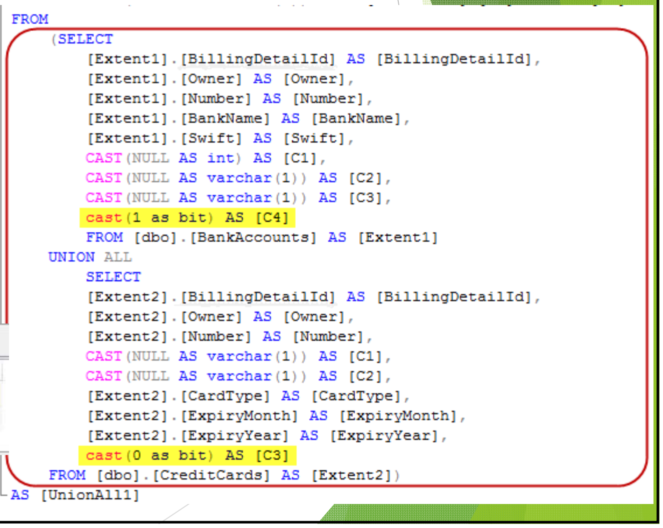


```
var query = from b in context.BillingDetails select b;
```



```
SELECT
    CASE WHEN ([UnionAll1].[C4] = 1) THEN '0X0X' ELSE '0X1X' END AS [C1],
    [UnionAll1].[BillingDetailId] AS [C2],
    [UnionAll1].[Owner] AS [C3],
    [UnionAll1].[Number] AS [C4],
    CASE WHEN ([UnionAll1].[C4] = 1) THEN [UnionAll1].[BankName] END AS [
    CASE WHEN ([UnionAll1].[C4] = 1) THEN [UnionAll1].[Swift] END AS [C6]
    CASE WHEN ([UnionAll1].[C4] = 1) THEN
        CAST(NULL AS int) ELSE [UnionAll1].[C1] END AS [C7],
    CASE WHEN ([UnionAll1].[C4] = 1) THEN
        CAST(NULL AS varchar(1)) ELSE [UnionAll1].[C2] END AS [C8],
    CASE WHEN ([UnionAll1].[C4] = 1) THEN
        CAST(NULL AS varchar(1)) ELSE [UnionAll1].[C3] END AS [C9]
```

	C1	C2	C3	C4	C5	C6	C7	C8	C9
1	0X0X	1	Morteza	987654321	CIBC	CORPIBB3034	NULL	NULL	NULL
2	0X1X	2	Morteza	987654321	NULL	NULL	1	10	12



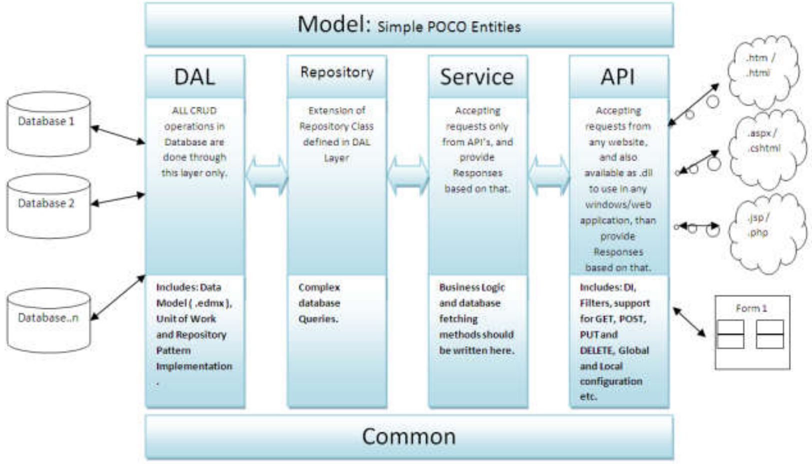
```
FROM
    (SELECT
        [Extent1].[BillingDetailId] AS [BillingDetailId],
        [Extent1].[Owner] AS [Owner],
        [Extent1].[Number] AS [Number],
        [Extent1].[BankName] AS [BankName],
        [Extent1].[Swift] AS [Swift],
        CAST(NULL AS int) AS [C1],
        CAST(NULL AS varchar(1)) AS [C2],
        CAST(NULL AS varchar(1)) AS [C3],
        cast(1 as bit) AS [C4]
        FROM [dbo].[BankAccounts] AS [Extent1]
    UNION ALL
        SELECT
        [Extent2].[BillingDetailId] AS [BillingDetailId],
        [Extent2].[Owner] AS [Owner],
        [Extent2].[Number] AS [Number],
        CAST(NULL AS varchar(1)) AS [C1],
        CAST(NULL AS varchar(1)) AS [C2],
        [Extent2].[CardType] AS [CardType],
        [Extent2].[ExpiryMonth] AS [ExpiryMonth],
        [Extent2].[ExpiryYear] AS [ExpiryYear],
        cast(0 as bit) AS [C3]
    FROM [dbo].[CreditCards] AS [Extent2])
AS [UnionAll1]
```

Inheritance Strategy in Entity Framework 6

- Choosing Strategy Guidelines

Table per Hierarchy (TPH)	Table per Type (TPT)	Table per Concrete Class (TPC)
Require polymorphic associations or queries, and subclasses declare relatively few properties	Require polymorphic associations or queries, and subclasses declare many properties	Don’t require polymorphic associations or queries
Goal is to minimize the number of nullable columns		Use TPC (only) for the top level of class hierarchy, where polymorphism isn’t usually required
For simple problems	For more complex cases	
Default		

N- tier Architecture



Model: Simple POCO Entities

DAL

ALL CRUD operations in Database are done through this layer only.

Includes: Data Model (.edmx), Unit of Work and Repository Pattern Implementation .

Repository

Extension of Repository Class defined in DAL Layer

Complex database Queries.

Service

Accepting requests only from API's, and provide Responses based on that.

Business Logic and database fetching methods should be written here.

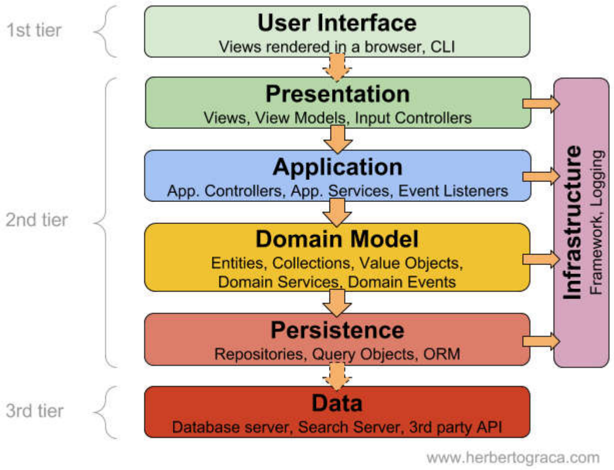
API

Accepting requests from any website, and also available as .dll to use in any windows/web application, than provide Responses based on that.

Includes: DI, Filters, support for GET, POST, PUT and DELETE, Global and Local configuration etc.

Common

N- tier Architecture



1st tier

User Interface
Views rendered in a browser, CLI

2nd tier

Presentation
Views, View Models, Input Controllers

Application
App. Controllers, App. Services, Event Listeners

Domain Model
Entities, Collections, Value Objects, Domain Services, Domain Events

Persistence
Repositories, Query Objects, ORM

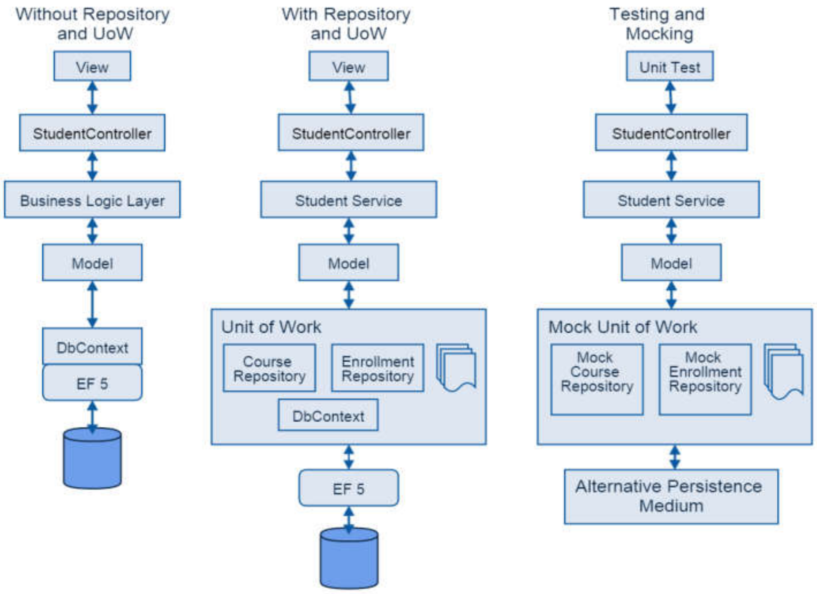
Infrastructure
Framework, Logging

3rd tier

Data
Database server, Search Server, 3rd party API

www.herbertograca.com

The Repository and Unit of Work Patterns



The Repository and Unit of Work Patterns

- The repository and unit of work patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application
    - The Repository mediates between the domain and data mapping layers, acting like an in-memory collection of domain objects. ("Patterns of Enterprise Application Architecture" by Martin Fowler)
- Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD).

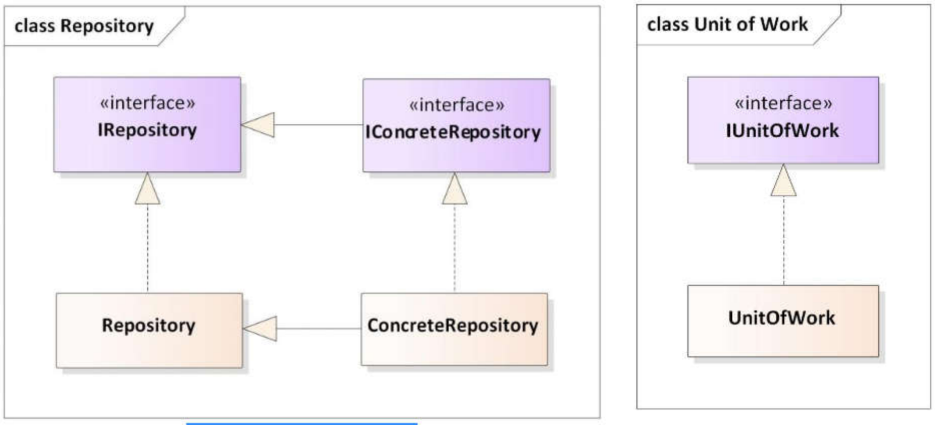
The Repository and Unit of Work Patterns

- Repository Pattern Goals
  - Decouple Business code from data Access. As a result, the persistence Framework can be changed without a great effort
  - Separation of Concerns
  - Minimize duplicate query logic
  - Testability

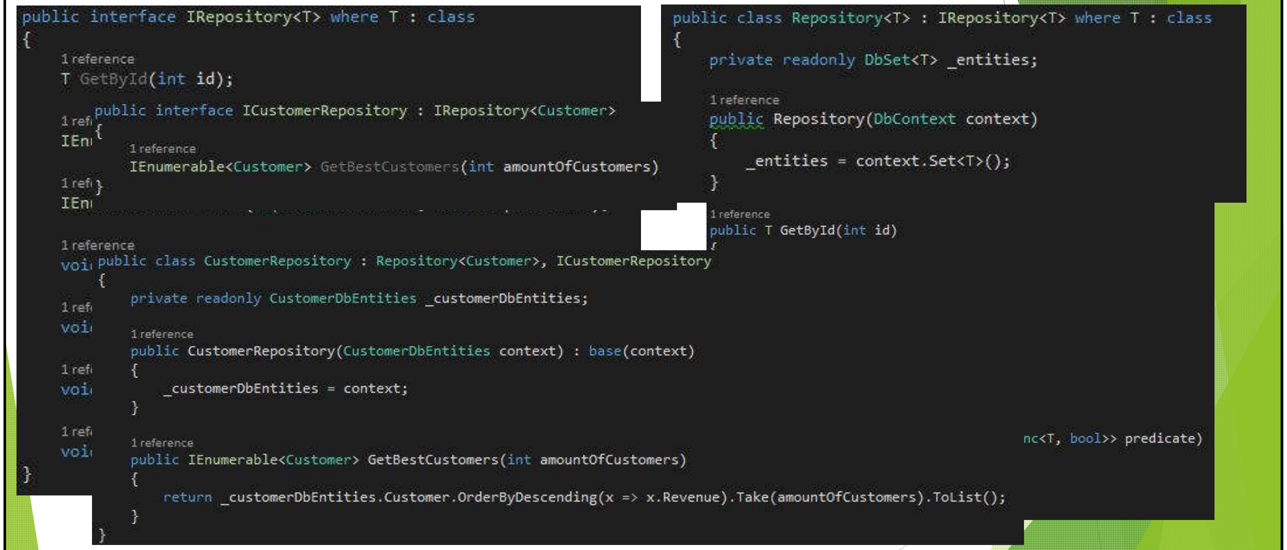
The Repository and Unit of Work Patterns

- Definition Unit of Work
  - Maintains a list of objects affected by a business transaction and coordinates the writing out of changes (“Patterns of Enterprise Application Architecture” by Martin Fowler)
- Consequences of the Unit of Work Pattern
  - Increases the level of abstraction and keep business logic free of data access code
  - Increased maintainability, flexibility and testability
  - More classes and interfaces but less duplicated code
  - The business logic is further away from the data because the repository abstracts the infrastructure. This has the effect that it might be harder to optimize certain operations which are performed against the data source

Repository pattern UML diagram



Implementing Repositories



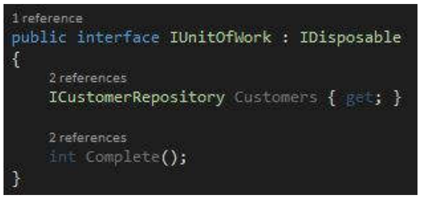
```
public interface IRepository<T> where T : class
{
    1 reference
    T GetById(int id);

    1 ref
    IEn

    1 ref
    IEn

    1 reference
    voi

    1 ref
    voi

    1 ref
    voi

    1 ref
    voi
}
```

```
public interface ICustomerRepository : IRepository<Customer>
{
    1 reference
    IEnumerable<Customer> GetBestCustomers(int amountOfCustomers)
}
```

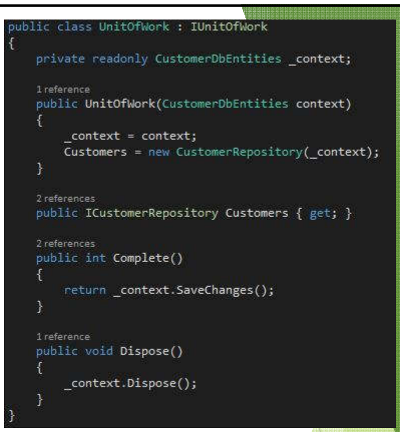
```
public class Repository<T> : IRepository<T> where T : class
{
    private readonly DbSet<T> _entities;

    1 reference
    public Repository(DbContext context)
    {
        _entities = context.Set<T>();
    }

    1 reference
    public T GetById(int id)
    {

                                      nc<T, bool>> predicate)
```

```
public class CustomerRepository : Repository<Customer>, ICustomerRepository
{
    private readonly CustomerDbEntities _customerDbEntities;

    1 reference
    public CustomerRepository(CustomerDbEntities context) : base(context)
    {
        _customerDbEntities = context;
    }

    1 reference
    public IEnumerable<Customer> GetBestCustomers(int amountOfCustomers)
    {
        return _customerDbEntities.Customer.OrderByDescending(x => x.Revenue).Take(amountOfCustomers).ToList();
    }
}
```

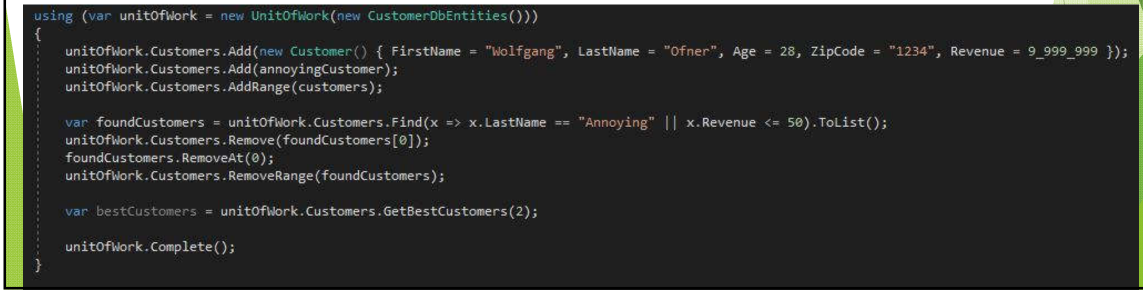
Implementing Unit of Work



```
1 reference
public interface IUnitOfWork : IDisposable
{
    2 references
    ICustomerRepository Customers { get; }

    2 references
    int Complete();
}
```

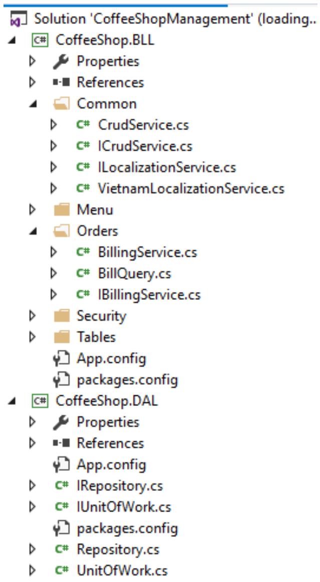


```
public class UnitOfWork : IUnitOfWork
{
    private readonly CustomerDbEntities _context;

    1 reference
    public UnitOfWork(CustomerDbEntities context)
    {
        _context = context;
        Customers = new CustomerRepository(_context);
    }

    2 references
    public ICustomerRepository Customers { get; }

    2 references
    public int Complete()
    {
        return _context.SaveChanges();
    }

    1 reference
    public void Dispose()
    {
        _context.Dispose();
    }
}
```

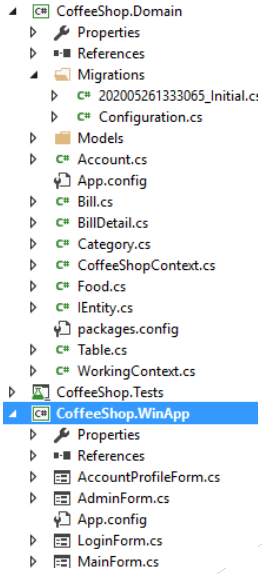


```
using (var unitOfWork = new UnitOfWork(new CustomerDbEntities()))
{
    unitOfWork.Customers.Add(new Customer() { FirstName = "Wolfgang", LastName = "Ofner", Age = 28, ZipCode = "1234", Revenue = 9_999_999 });
    unitOfWork.Customers.Add(annoyingCustomer);
    unitOfWork.Customers.AddRange(customers);

    var foundCustomers = unitOfWork.Customers.Find(x => x.LastName == "Annoying" || x.Revenue <= 50).ToList();
    unitOfWork.Customers.Remove(foundCustomers[0]);
    foundCustomers.RemoveAt(0);
    unitOfWork.Customers.RemoveRange(foundCustomers);

    var bestCustomers = unitOfWork.Customers.GetBestCustomers(2);

    unitOfWork.Complete();
}
```

Demonstration and Discussion



```
Solution 'CoffeeShopManagement' (loading..
  CoffeeShop.BLL
    Properties
    References
    Common
      CrudService.cs
      ICrudService.cs
      ILocalizationService.cs
      VietnamLocalizationService.cs
    Menu
    Orders
      BillingService.cs
      BillQuery.cs
      IBillingService.cs
    Security
    Tables
    App.config
    packages.config
  CoffeeShop.DAL
    Properties
    References
    App.config
    IRepository.cs
    IUnitOfWork.cs
    packages.config
    Repository.cs
    UnitOfWork.cs
```



```
  CoffeeShop.Domain
    Properties
    References
    Migrations
      202005261333065_Initial.c
      Configuration.cs
    Models
    Account.cs
    App.config
    Bill.cs
    BillDetail.cs
    Category.cs
    CoffeeShopContext.cs
    Food.cs
    IEntity.cs
    packages.config
    Table.cs
    WorkingContext.cs
  CoffeeShop.Tests
  CoffeeShop.WinApp
    Properties
    References
    AccountProfileForm.cs
    AdminForm.cs
    App.config
    LoginForm.cs
    MainForm.cs
```

References

- [1] https://www.entityframeworktutorial.net/, from May 30th, 2020
- [2] https://docs.microsoft.com/en-us/ef/ , from May 30th, 2020
- [3] https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/ , from May 30th, 2020
- [4] https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application , from May 30th, 2020