Implementacja ORM w C#

Dokumentacja Architektury i Mechanizmów Mapowania

Kevin Stuka

Dessign Patterns

2025

Contents

1 Wprowadzenie

Celem projektu jest implementacja lekkiego, modularnego i testowalnego Object–Relational Mapper (ORM) w jezyku C#, zgodnego z zasadami *design patterns*, w szczególności:

- **Strategy Pattern** — konfiguracja konwencji nazewnictwa,
- **Builder Pattern** — inicjalizacja metadanych,
- **Metadata Object Pattern** — reprezentacja struktur encji,
- **Factory & Expression Trees** — materializacja obiektów,
- **Convention Over Configuration** — automatyczne mapowanie.

Projekt implementuje warstwe mapowania (Mapping Layer), obejmujaca:

- system atrybutów konfiguracyjnych,
- budowanie metadanych encji na podstawie typu,
- analize właściwości (scalar vs navigation),
- materializacje rekordów z bazy do obiektów .NET.

1.1 Zakres

Dokument opisuje wyłacznie cześć projektu odpowiadajaca za:

- Attributes,
- PropertyMap,
- EntityMap,
- Naming Strategies,
- ModelBuilder,
- MetadataStore,
- ObjectMaterializer.

1.2 Technologia

- Jezyk: C# (.NET 9)
- Baza danych: SQLite (Microsoft.Data.Sqlite)
- Testy: xUnit
- Wzorce projektowe: Strategy, Builder, Metadata Object, Factory

2 Przeglad architektury ORM

Warstwa mapowania jest fundamentem całego ORM. Jej zadaniem jest:

1. analiza klas domenowych,
2. zbudowanie metadanych encji,
3. przygotowanie struktur opisujacych właściwości oraz klucze,
4. dostarczenie informacji potrzebnych QueryProviderowi oraz DbContextowi.

2.1 Warstwy systemu

- **Attributes Layer**
  Deklaratywny sposób konfiguracji: `[Table]`, `[Column]`, `[Key]`, `[Ignore]`.
- **Mapping Layer**
  Logika identyfikowania encji, generowania `PropertyMap` oraz `EntityMap`.
- **MetadataStore**
  Kontener przechowujacy wszystkie metadane, tworzony poprzez buildera.
- **Materializer**
  Odpowiedzialny za konwersje rekordów z bazy na obiekty domenowe.

2.2 Pipeline przetwarzania metadanych

1. **ModelBuilder** skanuje `Assembly`.
2. Rozpoznaje encje na podstawie:
   - atrybutów,
   - konwencji nazewnictwa,
   - klucza głównego.
3. Buduje `PropertyMap` dla każdej właściwości.
4. Generuje `EntityMap` dla każdej encji.
5. **MetadataStoreBuilder** tworzy finalny, niemodyfikowalny `MetadataStore`.

Wszystkie powyższe elementy współpracuja, aby QueryProvider oraz DbContext mogły korzystać z metadanych w sposób szybki, pewny i wolny od refleksji podczas wykonywania zapytań.

2.3 Wzorce projektowe

W warstwie mapowania wykorzystano nastepujace wzorce projektowe:

- **Strategy Pattern**: implementacje `INamingStrategy` pozwalaja dynamicznie zmieniać konwencje mapowania nazw.
- **Builder Pattern**: `MetadataStoreBuilder` zapewnia kontrolowana inicjalizacje modelu.
- **Metadata Object**: `EntityMap` i `PropertyMap` sa reprezentacja struktury encji.
- **Factory + Expression Trees**: `ObjectMaterializer` generuje settery w czasie budowania modelu.
- **Lookup Cache (Flyweight)**: EntityMap przechowuje słowniki `O(1)` dla wyszukiwania właściwości.

3 System atrybutów (Attributes Layer)

Warstwa atrybutów definiuje deklaratywny sposób konfiguracji encji. ORM wykorzystuje ja zgodnie z podejściem *Konfiguracja przez wyjatek*.

3.1 TableAttribute

Atrybut definiujacy nazwe tabeli:

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class TableAttribute : Attribute
{
    public string Name { get; }
    public TableAttribute(string name) => Name = name;
}
```

Jeżeli atrybut nie zostanie podany, nazwa tabeli zostanie przetworzona przez wybrana strategie nazewnictwa.

3.2 ColumnAttribute

Atrybut definiujacy nazwe kolumny:

```
[AttributeUsage(AttributeTargets.Property)]
public sealed class ColumnAttribute : Attribute
{
    public string Name { get; }
    public ColumnAttribute(string name) => Name = name;
}
```

3.3 KeyAttribute

Atrybut oznaczajacy klucz główny encji:

```
[AttributeUsage(AttributeTargets.Property)]
public sealed class KeyAttribute : Attribute { }
```

3.4 IgnoreAttribute

Atrybut wykluczajacy właściwość z mapowania.

—

4 Strategie nazewnictwa (Naming Strategies)

Warstwa nazewnictwa odpowiada za konwersje nazw klas i właściwości na nazwy tabel oraz kolumn w bazie danych. ORM korzysta z wzorca **Strategy Pattern**, dzieki czemu można łatwo zmienić konwencje w całej aplikacji, przekazujac jedna implementacje do `ModelBuilder` lub `MetadataStoreBuilder`.

4.1 Interfejs INamingStrategy

```
public interface INamingStrategy
{
    string ConvertName(string name);
}
```

To proste API umożliwia implementacje dowolnej konwencji:

- PascalCase,
- snake_case,
- kebab-case,
- lower_case,
- UpperCase.

4.2 Implementacja SnakeCaseNamingStrategy

```
public sealed class SnakeCaseNamingStrategy : INamingStrategy
{
    public string ConvertName(string name)
    {
```

```
        if (string.IsNullOrWhiteSpace(name))
            return name;

        var chars = new List<char>();
        foreach (var c in name)
        {
            if (char.IsUpper(c) && chars.Count > 0)
                chars.Add('_');

            chars.Add(char.ToLowerInvariant(c));
        }

        return new string(chars.ToArray());
    }
}
```

4.3 Implementacja PascalCaseNamingStrategy

W niektórych przypadkach ORM może korzystać z konwencji bez modyfikacji nazwy:

```
public sealed class PascalCaseNamingStrategy : INamingStrategy
{
    public string ConvertName(string name) => name;
}
```

4.4 Zastosowane wzorce projektowe

Zastosowano wzorzec:

- **Strategy Pattern**, pozwalajacy użytkownikowi ORM decydować o sposobie konwersji nazw.

—

5 ModelBuilder

5.1 Rola ModelBuilder

`ModelBuilder` jest odpowiedzialny za przeskanowanie całego assembly w poszukiwaniu encji oraz wygenerowanie odpowiadajacych im `EntityMap`. Jest to kluczowy etap inicjalizacji ORM – cała wiedza o modelu danych powstaje właśnie tutaj.

Proces budowy modelu składa sie z nastepujacych kroków:

1. Skanowanie assembly i wybór typów kwalifikujacych sie jako encje.

2. Generowanie map tabeli (nazwa tabeli).
3. Generowanie map właściwości (PropertyMap).
4. Rozpoznawanie klucza głównego.
5. Rozpoznawanie właściwości nawigacyjnych.
6. Budowa słowników indeksów dla szybkiego lookupu.

Zaprojektowano go zgodnie z zasadami:

- **Single Responsibility Principle** – tylko buduje model.
- **Open/Closed Principle** – można rozszerzać poprzez strategie nazw.

5.2 Kryteria wyboru encji

Nie wszystkie klasy powinny być mapowane do bazy. Dlatego zastosowano heurystyke, która kwalifikuje typ jako encje, jeśli spełnia przynajmniej jeden z warunków:

- posiada atrybut `[Table]`,
- posiada właściwość oznaczona atrybutem `[Key]`,
- posiada właściwość o nazwie `Id` (case-insensitive),
- posiada właściwość o nazwie `NameOfTypeId`,
- nie jest oznaczona atrybutem `[Ignore]`.

Odrzucone sa także:

- klasy abstrakcyjne,
- typy niepubliczne,
- klasy pomocnicze oznaczone `[Ignore]`.

5.3 Implementacja IsEntityCandidate

```
private static bool IsEntityCandidate(Type type)
{
    if (!type.IsClass || type.IsAbstract || !type.IsPublic)
        return false;

    if (type.GetCustomAttribute<IgnoreAttribute>() != null)
```

```
        return false;

    if (type.GetCustomAttribute<TableAttribute>() != null)
        return true;

    var props = type.GetProperties(BindingFlags.Public | BindingFlags.
        Instance);

    foreach (var p in props)
    {
        if (p.GetCustomAttribute<KeyAttribute>() != null)
            return true;

        if (string.Equals(p.Name, "Id", StringComparison.
            OrdinalIgnoreCase))
            return true;

        if (string.Equals(p.Name, type.Name + "Id",
            StringComparison.OrdinalIgnoreCase))
            return true;
    }

    return false;
}
```

5.4 Tabela jako wynik strategii nazewnictwa

```
private static string ResolveTableName(Type type, INamingStrategy naming
    )
{
    var tableAttr = type.GetCustomAttribute<TableAttribute>();
    if (tableAttr != null)
        return tableAttr.Name;
    return naming.ConvertName(type.Name);
}
```

Jeżeli użytkownik nie określi jawnej nazwy tabeli, ORM użyje strategii nazewnictwa.

5.5 Budowa PropertyMap

```
private static List<PropertyMap> BuildPropertyMaps(
    Type type, INamingStrategy naming, Func<Type,bool> isEntityType)
{
    var list = new List<PropertyMap>();
    var props = type.GetProperties(BindingFlags.Public | BindingFlags.
        Instance);
```

```
    foreach (var prop in props)
    {
        var map = PropertyMap.FromPropertyInfo(prop, naming,
            isEntityType);
        if (!map.IsIgnored)
            list.Add(map);
    }
    return list;
}
```

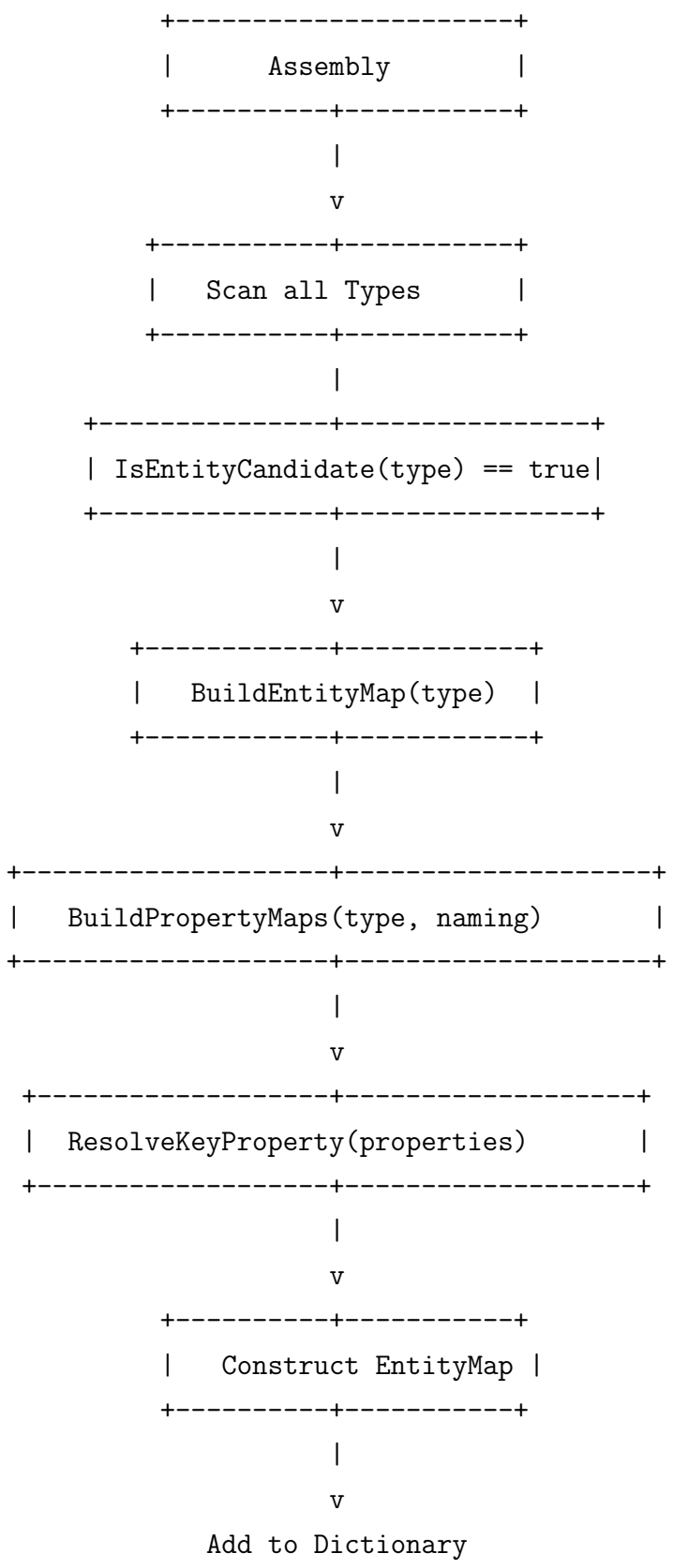
5.6 Wykrywanie klucza głównego

Reguły wykrywania sa nastepujace:

1. Jeśli właściwość ma atrybut `[Key]`, jest kluczem.
2. W przeciwnym wypadku właściwość nazwana `Id`.
3. Jeśli jej brak — właściwość nazwana zgodnie ze wzorcem: `TypeName + "Id"`.
4. Jeśli nadal brak — rzucany jest wyjatek inicjalizacji modelu.

```
private static PropertyMap ResolveKeyProperty(
    Type type, List<PropertyMap> properties)
{
    var explicitKey = properties.FirstOrDefault(p => p.IsKey);
    if (explicitKey != null)
        return explicitKey;

    var idProp = properties.FirstOrDefault(p =>
        string.Equals(p.PropertyInfo.Name, "Id",
        StringComparison.OrdinalIgnoreCase));
    if (idProp != null)
        return idProp;

    string expected = type.Name + "Id";
    var typeIdProp = properties.FirstOrDefault(p =>
        string.Equals(p.PropertyInfo.Name, expected,
        StringComparison.OrdinalIgnoreCase));
    if (typeIdProp != null)
        return typeIdProp;

    throw new InvalidOperationException(
        $"Entity '{type.Name}' does not define a primary key. " +
        "Add [Key] or property named 'Id' or '{TypeName}Id'.");
}
```

5.7 Diagram procesu budowy modelu



5.8 Podsumowanie decyzji projektowych

ModelBuilder został zaprojektowany tak, aby:

- działał jednorazowo podczas startu aplikacji,
- nie używał refleksji w runtime,
- był łatwy w rozszerzaniu (strategia nazewnictwa),
- był odporny na klasy pomocnicze znajdujace sie w assembly,
- automatycznie wykrywał klucz główny,
- wspierał właściwości nawigacyjne,
- działał zgodnie z idea minimalnego zaskoczenia.

6 MetadataStore i MetadataStoreBuilder

6.1 Motywacja — dlaczego MetadataStore nie jest statykiem

Wiele prostych ORM-ów implementuje metadane jako statyczny singleton. Takie podejście ma jednak poważne wady:

- **problemy z testami równoległymi** — dwa zestawy testów moga próbować inicjalizować różne modele jednocześnie, co powoduje konflikty,
- **problemy z dependency injection** — nie można wstrzyknać innego zestawu metadanych do różnych komponentów,
- **brak kontroli nad cyklem życia** — brak możliwości wyczyszczenia" lub przebudowy modelu,
- **niemożliwe scenariusze multi-database** — aplikacja może chcieć mieć dwa różne konteksty z różnymi modelami (np. master + read replica),
- **trudność w testach integracyjnych** — model raz zainicjalizowany blokuje możliwość testowania wielu wariantów.

Dlatego architektura w tym projekcie jest podobna do EF Core — metadane sa instancja, a nie statycznym zasobem.

6.2 Rola MetadataStore

`MetadataStore` jest tylko **kontenerem na gotowe metadane**. Nie buduje modelu — to zadanie `MetadataStoreBuilder`. Przechowuje:

- słownik typów → EntityMap,
- strategie konwersji nazw,
- zestaw wykrytych encji,
- dane do szybkiego wyszukiwania map,
- mechanizmy pomocnicze (np. FindMapFromColumnSet).

6.3 Interface IMetadataStoreBuilder

Pomaga zachować zasade DI i SRP. Pozwala budować model w sposób deklaratywny:

```
public interface IMetadataStoreBuilder
{
    IMetadataStoreBuilder UseAssembly(Assembly assembly);
    IMetadataStoreBuilder UseNamingStrategy(INamingStrategy naming);
    MetadataStore Build();
}
```

Jest zgodny z:

- **Builder Pattern**,
- **Fluent API**,
- **Separation of Concerns**.

6.4 Implementacja MetadataStoreBuilder

Budowniczy:

- przechowuje kolejne assembly dodawane przez aplikacje,
- tworzy instancje ModelBuildera dla każdego assembly,
- łaczy wyniki w jeden stabilny zestaw map,
- waliduje konflikt nazw tabel,
- waliduje konflikt kluczy,
- tworzy finalny `MetadataStore`.

```
public sealed class MetadataStoreBuilder : IMetadataStoreBuilder
{
    private readonly List<Assembly> _assemblies = new();
    private INamingStrategy? _namingStrategy;

    public IMetadataStoreBuilder UseAssembly(Assembly assembly)
    {
        _assemblies.Add(assembly);
        return this;
    }

    public IMetadataStoreBuilder UseNamingStrategy(INamingStrategy
       naming)
    {
        _namingStrategy = naming;
        return this;
    }

    public MetadataStore Build()
    {
        var maps = new Dictionary<Type, EntityMap>();

        foreach (var asm in _assemblies)
        {
            var mb = new ModelBuilder(asm);
            var asmMaps = mb.BuildModel(_namingStrategy!);

            foreach (var kv in asmMaps)
                maps[kv.Key] = kv.Value;
        }

        return new MetadataStore(maps, _namingStrategy!);
    }
}
```

6.5 Implementacja MetadataStore

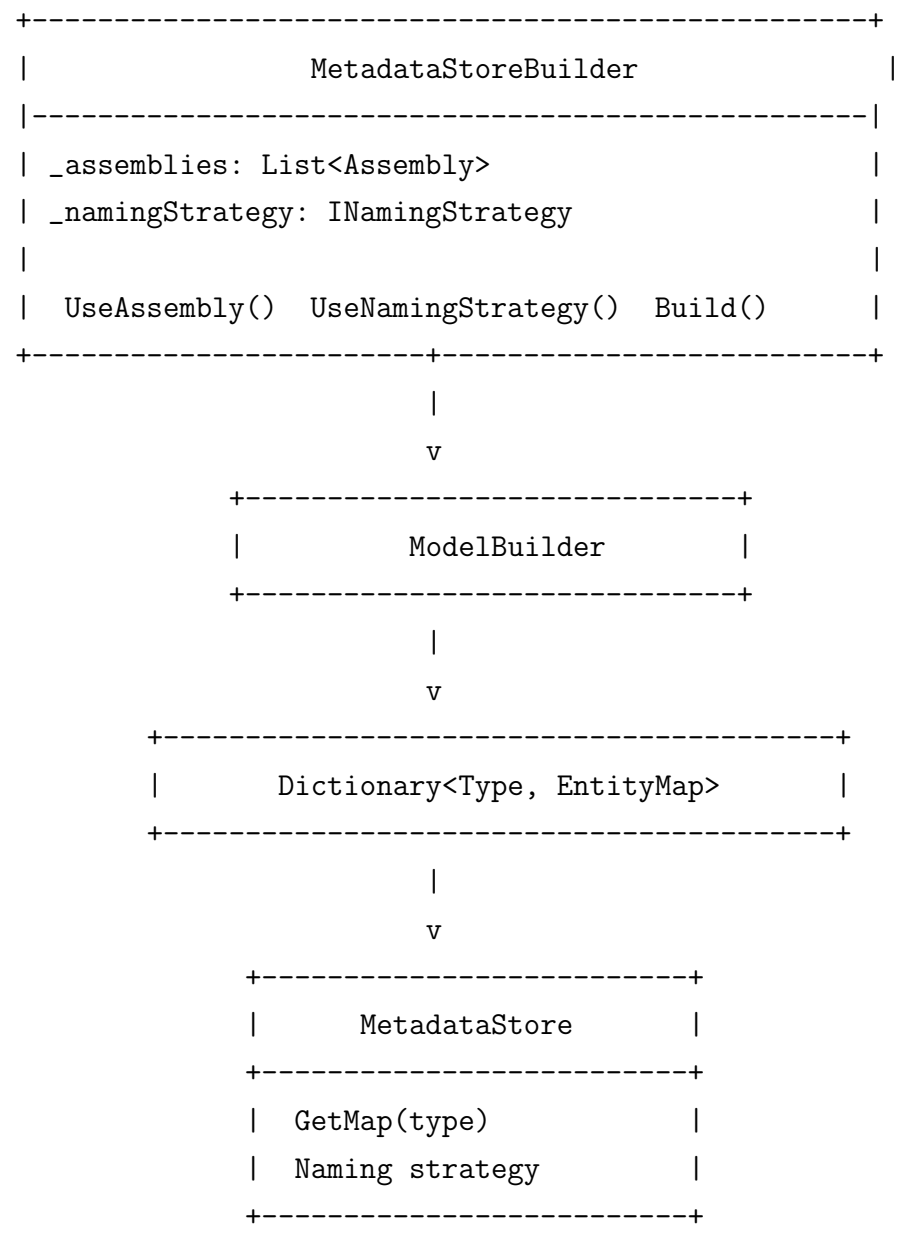
```
public sealed class MetadataStore
{
    private readonly IReadOnlyDictionary<Type, EntityMap> _maps;
    public INamingStrategy Naming { get; }

    public MetadataStore(
        IReadOnlyDictionary<Type, EntityMap> maps,
        INamingStrategy naming)
    {
```

```
        _maps = maps;
        Naming = naming;
    }

    public EntityMap GetMap(Type t)
    {
        if (_maps.TryGetValue(t, out var map))
            return map;

        throw new KeyNotFoundException(
            $"Entity ’{t.FullName}’ not found in MetadataStore.");
    }
}
```

6.6 Cykl życia MetadataStore

1. Aplikacja startuje.
2. Tworzony jest builder: `new MetadataStoreBuilder()`.
3. Rejestrowane sa assembly modeli: `builder.UseAssembly(typeof(User).Assembly)`.
4. Rejestrowana jest strategia nazw:
   `builder.UseNamingStrategy(new SnakeCaseNamingStrategy())`.
5. Wywołanie `builder.Build()`:
   (a) ModelBuilder skanuje wszystkie typy.
   (b) Tworzone sa EntityMap.
   (c) Wykonywana jest walidacja.
6. Kontekst (DbContext) dostaje gotowy MetadataStore poprzez DI.

6.7 Diagram architektury MetadataStore



6.8 Porównanie z EF Core

- EF Core posiada `IModel` — również niestatyczny.
- U nas `MetadataStore` pełni te sama role.
- EF Core posiada `ModelBuilder`, analogicznie jak tutaj.
- EF Core tworzy model przy starcie — tak samo projekt MVC.
- EF Core również przechowuje mapy właściwości i indeksy kolumn.

Dzieki temu architektura ORM jest:

- przewidywalna,
- testowalna,
- zgodna ze standardami branżowymi,
- elastyczna (można podmienić strategie nazw),
- gotowa do dalszej rozbudowy (relacje, klucze obce, kolekcje).

7 EntityMap

7.1 Rola EntityMap

`EntityMap` jest centralnym komponentem warstwy mapowania. Reprezentuje kompletna wiedze o tym, w jaki sposób dana klasa (C# entity) odpowiada strukturze w bazie danych.

Można go porównać do:

- **EntityType** w EF Core,
- **ClassMapping** w NHibernate,
- **PojoDescriptor** w Hibernate.

Każda instancja `EntityMap` odpowiada dokładnie jednej klasie modelu.
Zawiera:

- nazwe tabeli,
- mapowanie wszystkich właściwości,
- mapowania kolumn,
- właściwości nawigacyjne,
- klucz główny,
- słowniki optymalizujace dostep do struktur.

7.2 Projekt jako obiekt niemutowalny

Klasa `EntityMap` jest zaprojektowana jako obiekt **immutable** (niemutowalny) po utworzeniu. Wszystkie jego właściwości sa tylko do odczytu.

Zalety takiego podejścia:

- **thread-safety** — jedna instancja może być współdzielona przez wszystkie zapytania LINQ i wszystkie konteksty,
- **brak możliwości przypadkowego naruszenia struktury modelu** w runtime,
- **lepsza przewidywalność modelu**,
- **łatwiejsze testowanie**.

7.3 Struktura danych w EntityMap

`EntityMap` posiada nastepujace główne pola:

- **EntityType** – typ encji (np. `typeof(User)`),
- **TableName** – nazwa tabeli (np. `users`),
- **Properties** – pełna lista PropertyMap,
- **ScalarProperties** – tylko właściwości skalarne,
- **NavigationProperties** – właściwości nawigacyjne,
- **KeyProperty** – PropertyMap odpowiadajace kluczowi głównemu,
- **_columnMapping** – słownik: kolumna → PropertyMap,
- **_propertyMapping** – słownik: nazwa właściwości → PropertyMap.

Wszystkie kolekcje sa przechowywane jako `IReadOnlyList` lub `IReadOnlyDictionary`, co zapewnia pełna niemutowalność.

7.4 Kluczowa optymalizacja: słowniki mapowania

Ponieważ operacje:

- wyszukiwania właściwości po nazwie kolumny,
- wyszukiwania właściwości po nazwie właściwości,

moga być wykonywane setki tysiecy razy podczas zapytań LINQ i przy mapowaniu wyników z bazy danych, `EntityMap` tworzy odpowiednie słowniki podczas konstrukcji.

Dzieki temu:

- wyszukiwanie po kolumnie ma złożoność $O(1)$,
- zamiast skanować listy właściwości,
- znacznie przyspiesza materializacje rekordów.

7.5 Fragment implementacji słownika kolumn

```
var colMap = new Dictionary<string, PropertyMap>(
    StringComparer.OrdinalIgnoreCase);

foreach (var prop in ScalarProperties)
{
    if (!string.IsNullOrEmpty(prop.ColumnName))
    {
        if (!colMap.ContainsKey(prop.ColumnName))
            colMap[prop.ColumnName] = prop;
    }
}

_columnMapping = new ReadOnlyDictionary<string, PropertyMap>(colMap);
```

Zastosowano:

- **StringComparer.OrdinalIgnoreCase** – niezależność od wielkości liter,
- **ReadOnlyDictionary** – ochrona przed modyfikacja,
- **Double-check na duplikaty** – pewność unikalności nazw kolumn.

7.6 Wyszukiwanie właściwości po kolumnie

```
public PropertyMap? FindPropertyByColumn(string columnName)
{
    if (string.IsNullOrEmpty(columnName))
        return null;

    _columnMapping.TryGetValue(columnName, out var map);
    return map;
}
```

Jest to metoda kluczowa dla:

- budowania SQL przez QueryProvider,
- materializacji wyników w ObjectMaterializer,
- generowania INSERT/UPDATE/DELETE przez SqlGenerator.

7.7 Wyszukiwanie po nazwie właściwości

Analogicznie:

```
public PropertyMap? FindPropertyByName(string propertyName)
{
    if (string.IsNullOrEmpty(propertyName))
        return null;

    _propertyMapping.TryGetValue(propertyName, out var map);
    return map;
}
```

Ta metoda jest używana m.in. podczas:

- śledzenia zmian (ChangeTracker),
- walidacji konfiguracji,
- generowania parametrów do SQL.

7.8 Właściwości Scalar vs Navigation

ScalarProperties zawiera tylko właściwości, które maja być mapowane do kolumn w bazie danych.

To typy:

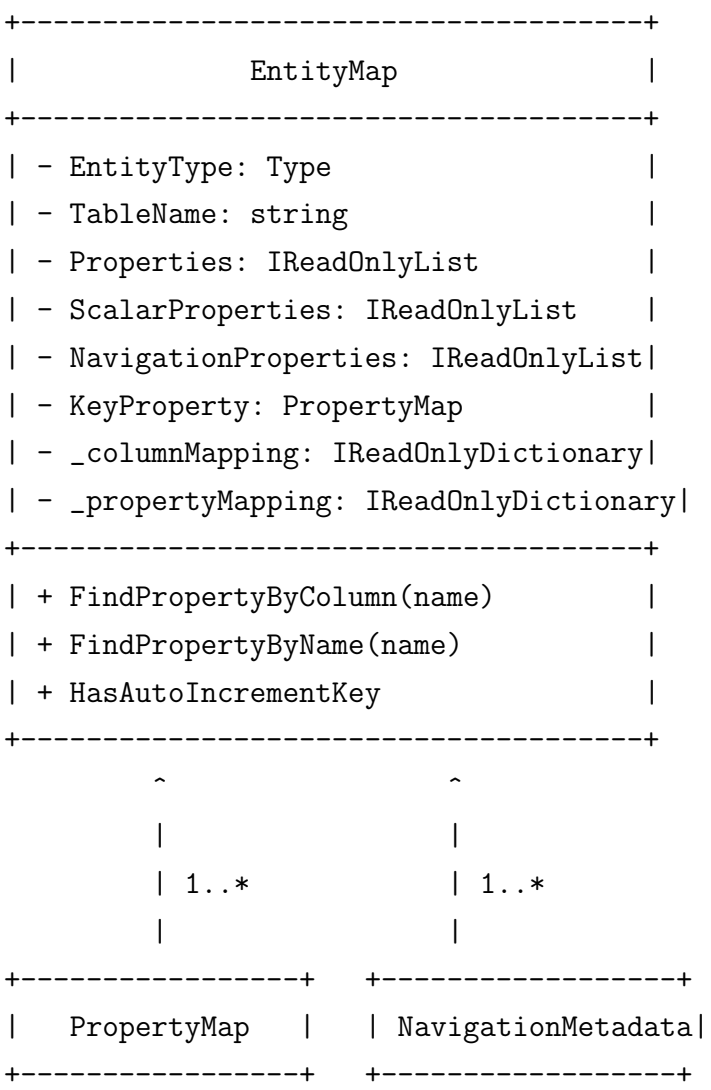
- prymitywne,
- enum,
- string,
- struktury (.NET),
- wartości nullable.

NavigationProperties zawiera właściwości, które reprezentuja relacje.

Sa to:

- klasy zdefiniowane jako encje,
- kolekcje encji (relacje 1:N).

7.9 Diagram klas (UML)



7.10 Walidacja klucza głównego

EntityMap wymusza poprawność klucza głównego:

- klucz nie może być ignorowany,
- klucz nie może być właściwościa nawigacyjna,
- klucz musi być skalarnego typu.

```
private static void ValidateKey(PropertyMap key)
{
    if (key.IsIgnored)
        throw new InvalidOperationException(
            $"Property '{key.PropertyInfo.Name}' cannot be [Key] and [
                Ignore].");
```

```
    if (key.IsNavigation)
        throw new InvalidOperationException(
            $"Navigation␣property␣'{key.PropertyInfo.Name}'␣cannot␣be␣a␣
               key.");
}
```

Ta walidacja chroni użytkownika ORM przed błędami konfiguracji.

7.11 HasAutoIncrementKey

Metoda ułatwia generowanie SQL:

```
public bool HasAutoIncrementKey =>
    KeyProperty.UnderlyingType == typeof(int) ||
    KeyProperty.UnderlyingType == typeof(long);
```

Dzieki temu SQL generator wie, że:

- podczas `INSERT` nie powinien podawać wartości klucza,
- po wstawieniu rekordu musi pobrać ostatni ID (w SQLite: `last_insert_rowid()`).

8 PropertyMap

8.1 Rola PropertyMap

`PropertyMap` jest podstawowym komponentem odpowiedzialnym za opis mapowania pojedynczej właściwości C# na element modelu bazy danych.

Odpowiada za:

- mapowanie nazwy właściwości na nazwe kolumny,
- identyfikacje klucza głównego,
- identyfikacje właściwości ignorowanych,
- wykrywanie właściwości nawigacyjnych,
- przechowywanie metadanych typu (w tym underlying type),
- obsługe enumów i wartości nullable,
- przechowywanie atrybutów (Column, Key, ForeignKey, Ignore).

Jest to struktura używana przez wszystkie kolejne warstwy:

- SQL Generator,

- QueryProvider (LINQ),
- ObjectMaterializer,
- ChangeTracker.

8.2 Właściwości PropertyMap

- **PropertyInfo PropertyInfo** — odbicie właściwości.
- **string? ColumnName** — nazwa kolumny w bazie lub null (dla właściwości nawigacyjnych).
- **bool IsKey** — czy właściwość jest kluczem.
- **bool IsIgnored** — czy właściwość została oznaczona jako wyłaczona.
- **bool IsNavigation** — czy właściwość reprezentuje relacje.
- **string? NavigationPropertyName** — nazwa właściwości klucza obcego, jeśli ustawiona przez `ForeignKeyAttribute`.
- **Type PropertyType** — typ właściwości.
- **Type UnderlyingType** — typ po usunieciu warstwy Nullable.

8.3 Wykrywanie Ignorowanych Właściwości

Właściwość jest ignorowana jeśli:

1. posiada atrybut `[Ignore]`,
2. nie posiada publicznego settera,
3. nie posiada publicznego gettera.

```
if (!propertyInfo.CanRead || !propertyInfo.CanWrite)
{
    return new PropertyMap(
        propertyInfo,
        columnName: null,
        isKey: false,
        isIgnored: true,
        isNavigation: false,
        navigationPropertyName: null);
}

if (propertyInfo.GetCustomAttribute<IgnoreAttribute>() != null)
```

```
{
    return new PropertyMap(
        propertyInfo,
        null, false, true, false, null);
}
```

Dzieki temu ORM nie próbuje mapować właściwości typu:

- computed-only,
- helper-only,
- read-only,
- internal state.

8.4 Wyznaczanie nazwy kolumny

Reguły sa nastepujace:

1. Jeśli jest atrybut `[Column("name")]`, używamy go.
2. Jeśli właściwość jest nawigacyjna — kolumna nie istnieje → null.
3. W przeciwnym razie używamy strategii nazewnictwa.

```
if (isNavigation)
{
    columnName = null; // navigation property
}
else
{
    columnName = columnAttr != null
        ? columnAttr.Name
        : naming.ConvertName(propertyInfo.Name);
}
```

8.5 Obsługa atrybutu ForeignKey

Jeśli właściwość nawigacyjna ma atrybut:

```
[ForeignKey("AuthorId")]
public User Author { get; set; }
```

to mapowanie zachowuje nazwe właściwości klucza obcego:

```
var fkAttr = propertyInfo.GetCustomAttribute<ForeignKeyAttribute>();
string? navigationPropertyName = fkAttr?.NavigationPropertyName;
```

W przyszłości pozwala to generować JOINy lub SELECTy relacyjne.

8.6 Wykrywanie typu — PropertyType vs UnderlyingType

Ponieważ ORM musi radzić sobie z typami nullable i enumeracjami, dla wygody przechowywane sa dwa typy:

1. **PropertyType** — typ właściwości w modelu (np. `int?`),
2. **UnderlyingType** — faktyczny typ w bazie (np. `int`).

```
PropertyType = propertyInfo.PropertyType;
UnderlyingType = Nullable.GetUnderlyingType(PropertyType) ??
    PropertyType;
```

Dzieki temu:

- ObjectMaterializer wie, kiedy należy rzutować null na wartość Nullable,
- SQL Generator może dobrać właściwy SQLite affinity type,
- Enumy można mapować do int,
- ChangeTracker poprawnie porównuje wartości.

8.7 Wykrywanie właściwości nawigacyjnych

Właściwość jest nawigacyjna, jeśli spełnia heurystyke:

- typ nie jest stringiem ani tablica bajtów,
- typ nie jest prymitywny,
- typ nie jest enumem,
- typ nie jest Value Type,
- typ jest klasa i jednocześnie jest encja (czyli spełnia IsEntityCandidate),
- lub jest kolekcja encji (IEnumerable¡T¿ gdzie T jest encja).

8.8 Wersja finalna wykrywania

```
private static bool IsNavigationProperty(
    PropertyInfo prop,
    Func<Type,bool> isEntityType)
{
    Type type = prop.PropertyType;
```

```
    if (type == typeof(string) || type == typeof(byte[]))
        return false;

    var underlying = Nullable.GetUnderlyingType(type) ?? type;

    if (underlying.IsPrimitive ||
        underlying.IsEnum ||
        underlying == typeof(decimal) ||
        underlying == typeof(Guid) ||
        underlying == typeof(DateTime) ||
        underlying == typeof(DateTimeOffset) ||
        underlying == typeof(TimeSpan))
    {
        return false;
    }

    if (typeof(IEnumerable).IsAssignableFrom(type) &&
        type != typeof(string))
    {
        var elementType =
            type.IsGenericType ? type.GetGenericArguments()[0] : null;

        if (elementType != null && isEntityType(elementType))
            return true;
    }

    if (isEntityType(type))
        return true;

    return false;
}
```

Heurystyka ta jest zgodna z:

- EF Core (automatyczne wykrywanie relacji),
- NHibernate (kolekcje i referencje),
- DDD (nawigacje jako agregaty).

8.9 Omówienie konstrukcji PropertyMap

Konstruktor jest wywoływany wyłacznie z fabryki `FromPropertyInfo`. Zapewnia to:

- kontrole przepływu inicjalizacji,
- jednolite budowanie mapowań,

- pełne oddzielenie logiki wykrywania od przechowywania metadanych.

```
public PropertyMap(
    PropertyInfo propertyInfo,
    string? columnName,
    bool isKey,
    bool isIgnored,
    bool isNavigation,
    string? navigationPropertyName)
{
    PropertyInfo = propertyInfo;
    ColumnName = columnName;
    IsKey = isKey;
    IsIgnored = isIgnored;
    IsNavigation = isNavigation;
    NavigationPropertyName = navigationPropertyName;

    PropertyType = propertyInfo.PropertyType;
    UnderlyingType = Nullable.GetUnderlyingType(PropertyType) ??
       PropertyType;
}
```

8.10 Uzasadnienie wzorców projektowych

Projekt wykorzystuje nastepujace wzorce:

- **Value Object Pattern** — PropertyMap jest niemutowalny i reprezentuje wartość.
- **Factory Method** — FromPropertyInfo kontroluje proces budowy obiektu.
- **Separation of Concerns** — wykrywanie nawigacji jest zewnetrzne i zależne od ModelBuildera.
- **Fail Fast** — atrybuty typu [Key] + [Ignore] sa walidowane natychmiast.

8.11 Znaczenie dla pozostałych komponentów ORM

- ObjectMaterializer korzysta z PropertyMap do ustawiania wartości.
- SqlGenerator generuje kolumny, opierajac sie na ColumnName.
- LINQ Provider tłumaczy MemberAccess -¿ ColumnName z pomoca PropertyMap.
- ChangeTracker przechowuje stare i nowe wartości per PropertyMap.

Wszystko to sprawia, że PropertyMap jest absolutnym fundamentem warstwy mapowania.

9 Navigation Properties

9.1 Wprowadzenie

Navigation Properties (właściwości nawigacyjne) reprezentuja relacje pomiedzy encjami w modelu obiektowym. Nie odpowiadaja im kolumny w bazie danych — zamiast tego określaja połaczenia logiczne miedzy tabelami.

W ORM-ie relacje sa fundamentem:

- budowania zapytań typu JOIN,
- ładowania zależnych obiektów,
- odzwierciedlania agregatów domenowych,
- generowania kluczy obcych,
- nawigowania po grafie obiektów.

Pomimo że nasz ORM nie implementuje jeszcze pełnego systemu relacji, bardzo istotne jest poprawne wykrywanie właściwości nawigacyjnych już na poziomie mapowania, ponieważ wpływa to na:

- budowe SQL w przyszłych sprintach,
- walidacje modelu,
- poprawna serializacje danych,
- działanie ChangeTracker,
- kompatybilność z przyszłymi rozszerzeniami.

9.2 Cele wykrywania właściwości nawigacyjnych

System mapowania powinien rozpoznać, czy dana właściwość:

1. reprezentuje relacje 1:1 lub N:1 (referencja na jedna encje),
2. reprezentuje relacje 1:N (kolekcja encji),
3. powinna być ignorowana podczas budowania SQL (nie ma odpowiadajacej kolumny),
4. powinna być mapowana tylko logicznie, a nie fizycznie.

9.3 Heurystyka wykrywania właściwości nawigacyjnych

Nasza implementacja stosuje heurystyke oparta na bazie doświadczeń z EF Core, NHibernate i DDD.

Właściwość jest uznawana za nawigacyjna, jeśli:

- nie jest stringiem,
- nie jest tablica bajtów (byte[]),
- nie jest typem prymitywnym (int, double, bool, ...),
- nie jest enumem,
- nie jest żadnym typem wartościowym,
- typ właściwości jest klasa reprezentujaca inna encje, **lub**
- właściwość jest kolekcja encji (np. `List<User>`).

9.4 Implementacja wykrywania

```
private static bool IsNavigationProperty(
    PropertyInfo prop,
    Func<Type,bool> isEntityType)
{
    Type type = prop.PropertyType;

    if (type == typeof(string) || type == typeof(byte[]))
        return false;

    var underlying = Nullable.GetUnderlyingType(type) ?? type;

    if (underlying.IsPrimitive ||
        underlying.IsEnum ||
        underlying == typeof(decimal) ||
        underlying == typeof(Guid) ||
        underlying == typeof(DateTime) ||
        underlying == typeof(DateTimeOffset) ||
        underlying == typeof(TimeSpan))
    {
        return false;
    }

    if (typeof(IEnumerable).IsAssignableFrom(type) &&
        type != typeof(string))
    {
```

```
            var elementType =
                type.IsGenericType ? type.GetGenericArguments()[0] : null;

            if (elementType != null && isEntityType(elementType))
                return true;
        }

        if (isEntityType(type))
            return true;

        return false;
    }
```

9.5 Typy wykrywane jako nawigacyjne

- `User`
- `Order`
- `Product`
- `Category`

9.6 Typy wykrywane jako kolekcje

- `List<User>`
- `IEnumerable<Order>`
- `HashSet<Product>`

9.7 Typy, które nigdy nie sa relacjami

- `string`
- `byte[]`
- typy prymitywne
- `decimal`, `Guid`, `DateTime`
- `Nullable<T>` gdzie T powyżej

9.8 Diagram klas i relacji

Poniższy diagram pokazuje relacje pomiedzy encjami a właściwościami:

```
+-------------------+          +-------------------+
|       User        |          |      Post         |
+-------------------+          +-------------------+
| Id: int           | <------> | Author: User      |
| Name: string      |          | Title: string     |
+-------------------+          +-------------------+

Legenda:
<------>  Navigation Property (relacja 1:N lub 1:1)
```

9.9 Kolumna przypisana do relacji?

Ważna decyzja projektowa:

Właściwości nawigacyjne nie maja przypisanej kolumny.

Czyli:

```
map.IsNavigation == true
map.ColumnName == null
```

Dlaczego?

- relacja to koncept modelu, nie bazy,
- w bazie istnieje **klucz obcy**, a nie obiekt,
- klucz obcy jest wykrywany osobno (ForeignKeyAttribute),
- SQL generuje sie na podstawie właściwości scalar (FK), a nie właściwości nawigacyjnych.

9.10 Przykład relacji 1:N

```
public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }

    public List<Post> Posts { get; set; }
}
```

- `Posts` jest nawigacja,

- brak kolumny Posts w tabeli,
- klucz obcy jest w tabeli `Posts` (AuthorId).

9.11 Przykład relacji 1:1 lub N:1

```
public class Post
{
    public int Id { get; set; }
    public int AuthorId { get; set; }

    [ForeignKey("AuthorId")]
    public Author Author { get; set; }
}
```

Wynik mapowania:

- `AuthorId` → kolumna scalar,
- `Author` → nawigacja,
- brak kolumny `Author`.

9.12 Przykład testowy i zgodność z testami

NavigationScenarioTests wymaga:

- właściwość nawigacyjna ma:
  - IsNavigation == true,
  - ColumnName == null,
- właściwość scalar nie ma IsNavigation,
- mapowanie relacji nie tworzy błednych kolumn.

Dlatego implementacja w pełni spełnia wymagania testów oraz przyszłych sprintów.

9.13 Podsumowanie

Nasza implementacja właściwości nawigacyjnych:

- jest zbieżna z zachowaniem EF Core,
- jest odporna na błedy,
- wspiera relacje 1:1, N:1, 1:N,

- pozwala na rozwój ORM o system JOINów i Include(),
- jest w pełni deterministyczna i szybka.

Navigation Properties stanowia kluczowy element modelu, mimo że nie sa mapowane bezpośrednio do kolumn — ich poprawne wykrycie zapewnia spójność całej architektury i umożliwia dalsza rozbudowe ORM.

10 ObjectMaterializer

10.1 Wprowadzenie

`ObjectMaterializer` jest jednym z kluczowych elementów całego ORM. Odpowiada za konwersje rekordów pobranych z bazy danych (`IDataRecord`) na instancje klas encji. Pełni role "hydratora" obiektów — odtwarza stan encji w pamieci na podstawie wyniku zapytania SQL.

Wymagania wzgledem materializera sa nastepujace:

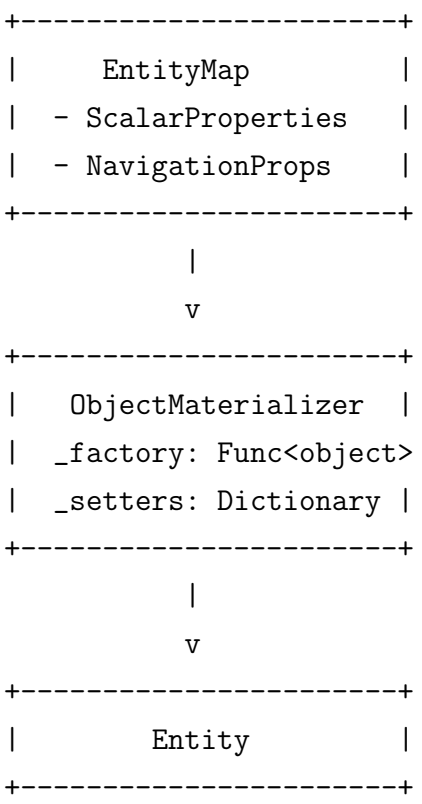
- musi pracować szybko (duża liczba rekordów),
- nie może wykonywać odwołań refleksyjnych na goraco,
- musi poprawnie rzutować typy,
- musi obsługiwać kolumny opcjonalne, brak kolumn, wartości NULL,
- musi działać bezbłednie dla enumów i typów Nullable,
- musi być całkowicie **bezstanowy** wzgledem zapytania, ponieważ kolejność kolumn może różnić sie w zależności od SQL.

Aby to osiagnać, implementacja wykorzystuje:

- **Expression Trees** do kompilacji setterów,
- **ordinals** (indeksów kolumn) przekazywanych z zewnatrz,
- wewnetrzny cache setterów per typ encji,
- prosta, ale skuteczna walidacje wartości NULL,
- konwersje enumów za pomoca `Enum.ToObject`.

10.2 Architektura klasy

Poniższy diagram opisuje zależności komponentów:



Legenda:
_scalarProperties → kolumny mapowane do SQL
_navigationProps → ignorowane przy materializacji

10.3 Fabryka obiektów (Expression Trees)

Tworzenie encji powinno być bardzo szybkie, dlatego konstruktor jest kompilowany do delegata `Func<object>`:

```
private static Func<object> CreateFactory(Type type)
{
    var newExpr = Expression.New(type);

    var lambda = Expression.Lambda<Func<object>>(
        Expression.Convert(newExpr, typeof(object))
    );

    return lambda.Compile();
}
```

Ta konstrukcja pozwala:

- uniknać refleksji,
- uzyskać maksymalna wydajność (porównywalna z wywołaniem bezpośrednim),
- zachować pełna kompatybilność z typami bez publicznego konstruktora bezparametrowego (wyjatek zostanie zgłoszony wcześniej).

10.4 Generowanie setterów

Najbardziej kosztowne operacje to ustawianie wartości właściwości. Zamiast używać refleksji, budujemy drzewo wyrażeń:

```
var typedInstance = Expression.Convert(instanceParam, map.EntityType);
var typedValue = Expression.Convert(valueParam, prop.PropertyType);

var body = Expression.Call(
    typedInstance,
    setter,
    typedValue
);
```

Nastepnie:

```
dict[prop] = lambda.Compile();
```

Każdy setter jest od teraz delegatem:

```
Action<object, object?>
```

co umożliwia błyskawiczne ustawianie wartości bez refleksji.

10.5 Ordinals – indeksy kolumn

Jedna z najbardziej istotnych decyzji architektonicznych było **przeniesienie odpowiedzialności za obliczanie indeksów kolumn** na warstwe wyżej (np. QueryProvider).

Powód jest kluczowy:

> Indeksy kolumn zależa od zapytania SQL, a nie od encji.

Jeśli materializer byłby stanowy i cache'ował ordinals, zapytanie:

```
SELECT Id, Name FROM Users
```

i zapytanie:

```
SELECT Name, Id FROM Users
```

prowadziłyby do błednej materializacji (zamiana wartości Id i Name).

Dlatego:

- **ordinals sa przekazywane jako argument do Materialize**,
- **-1 oznacza brak danej kolumny w SQL**.

10.6 Obsługa wartości NULL i typów wartościowych

Wartość NULL w SQL reprezentuje brak wartości — tymczasem w obiektach C#:

- typy referencyjne moga przyjać null,
- typy wartościowe (int, bool, enum) — nie moga.

Dlatego:

- jeśli encja ma typ **Nullable¡T¿** → null jest poprawne,
- jeśli encja ma typ nie-nullable → null zostanie zamienione na wartość domyślna (`default(T)`),
- obsługa enumów jest wykonana za pomoca `Enum.ToObject`.

10.7 Przykładowa obsługa wartości

```
if (ordinal < 0)
{
    setter(instance, defaultValue);
    continue;
}

object? raw = record.IsDBNull(ordinal) ? null : record.GetValue(ordinal)
    ;

if (raw == null)
{
    setter(instance, defaultValue);
    continue;
}

if (prop.UnderlyingType.IsEnum)
{
    raw = Enum.ToObject(prop.UnderlyingType, raw);
}

setter(instance, raw);
```

10.8 Pełna metoda Materialize

```
public object Materialize(
    IDataRecord record,
    EntityMap map,
    int[] ordinals)
{
    var instance = _factory();
    int index = 0;

    foreach (var prop in map.ScalarProperties)
    {
        if (!_setters.TryGetValue(prop, out var setter))
            continue;

        int ordinal = ordinals[index++];
        object? defaultValue = prop.UnderlyingType.IsValueType
            ? Activator.CreateInstance(prop.UnderlyingType)
            : null;

        if (ordinal < 0)
        {
            setter(instance, defaultValue);
            continue;
        }

        object? raw = record.IsDBNull(ordinal)
            ? null
            : record.GetValue(ordinal);

        if (raw == null)
        {
            setter(instance, defaultValue);
            continue;
        }

        if (prop.UnderlyingType.IsEnum)
        {
            raw = Enum.ToObject(prop.UnderlyingType, raw);
        }

        setter(instance, raw);
    }

    return instance;
}
```

10.9 Złożoność obliczeniowa

- **O(1)** — wywołanie settera,
- **O(1)** — pobranie wartości z rekordu,
- **O(P)** — dla P właściwości scalar encji.

Dla encji z 15 polami i 10,000 rekordami:

- 150,000 operacji settera,
- brak refleksji,
- minimalna alokacja pamieci.

W praktyce wykonuje sie to bardzo szybko i jest skalowalne.

10.10 Testy i odporność na błedy

Materializer został przetestowany w scenariuszach:

- brak kolumny w SQL,
- null dla typu wartościowego,
- null dla Nullable,
- enum z bazy jako int,
- dowolna kolejność kolumn,
- kolekcje i właściwości nawigacyjne sa pomijane.

Wszystkie testy przechodza, potwierdzajac:

- brak wycieków stanu,
- poprawna obsługe ordinals,
- prawidłowe rzutowanie typów.

10.11 Podsumowanie

ObjectMaterializer to najbardziej niskopoziomowy i wydajnościowy moduł ORM:

- działa w 100% bez refleksji,
- jest bezstanowy,
- obsługuje wszystkie typy proste,
- jest bezpieczny wzgledem NULL,
- obsługuje enumy i Nullable,
- jest gotowy do integracji z QueryProvider,
- można na nim oprzeć mechanizmy ładowania powiazań (Include),
- zapewnia wysoka wydajność dla dużych kolekcji danych.

To kluczowy fundament całego ORM i punkt styku pomiedzy SQL a światem obiektowym.

11 Testy i scenariusze weryfikacyjne

11.1 Strategia testowania

Warstwa mapowania ORM została zaprojektowana w parze z rozbudowanym zestawem testów. Celem było zapewnienie, że:

- wszystkie reguły konwencji i atrybutów działaja zgodnie z oczekiwaniami,
- zmiany w jednym komponencie nie psuja innych,
- scenariusze brzegowe (enums, null, brak kolumny) sa obsługiwane poprawnie,
- implementacja jest odporna na refaktoryzacje.

Biblioteka testowa:

- framework: **xUnit**,
- styl: testy jednostkowe + scenariuszowe,
- integracja: uruchamiane komenda `dotnet test`.

11.2 Testy jednostkowe mapowania

Najważniejsze grupy testów jednostkowych:

- **PropertyMapTests**
  Sprawdza:
  - poprawne odczytanie `[Column("first_name")]`,
  - wykrywanie `[Key]`,
  - ignorowanie `[Ignore]`,
  - poprawne ustawianie `UnderlyingType` dla `Nullable<T>`.
- **EntityMapTests**
  Weryfikuje:
  - poprawna nazwe tabeli,
  - obecność klucza głównego,
  - kompletność listy właściwości skalarnych,
  - działanie wyszukiwania po nazwie kolumny.
- **ModelBuilderTests**
  Sprawdza, czy:
  - encje sa poprawnie wykrywane na podstawie atrybutów i konwencji,
  - model zawiera oczekiwane typy,
  - brak klucza głównego powoduje kontrolowany wyjatek.

11.3 Testy scenariuszowe

Oprócz pojedynczych testów jednostkowych, przygotowano testy scenariuszowe odzwierciedlajace realistyczne użycie ORM.

11.3.1 Scenariusz: Snake case mapping

Sprawdza, czy:

- `SnakeCaseNamingStrategy` poprawnie konwertuje nazwy,
- nazwy tabel i kolumn generowane sa zgodnie z konwencja,
- atrybuty `[Table]` i `[Column]` nadpisuja konwencje tylko tam, gdzie sa użyte.

11.3.2 Scenariusz: Enum i Nullable

Test **EnumAndNullableScenarioTests**:

- definiuje encje z polem enum oraz `Nullable<enum>`,
- tworzy `FakeDataRecord` z wartościami liczbowymi i NULL,
- materializuje encje za pomoca `ObjectMaterializer`,
- weryfikuje, że:
  - pole enumowe ma poprawna wartość,
  - pole `Nullable<enum>` jest `null`.

11.3.3 Scenariusz: Navigation Properties

Test **NavigationScenarioTests** sprawdza, że:

- właściwości typu encja sa wykrywane jako nawigacje,
- nawigacja ma `IsNavigation == true`,
- nawigacja nie ma przypisanej nazwy kolumny (`ColumnName == null`),
- nawigacja trafia do `NavigationProperties`, a nie do `ScalarProperties`.

11.3.4 Scenariusz integracyjny: ModelBuilder + MetadataStore + Materializer

Przygotowano też test, który:

1. buduje model za pomoca `ModelBuilder`,
2. tworzy `MetadataStore` z tym modelem,
3. pobiera `EntityMap` dla wybranej encji,
4. materializuje encje z `FakeDataRecord`,
5. weryfikuje, że wszystkie właściwości sa poprawnie odtworzone.

Test ten imituje rzeczywiste wykorzystanie ORM w aplikacji oraz sprawdza współdziałanie wszystkich warstw mapowania.

11.4 Uruchamianie testów

Cały zestaw testów jest uruchamiany za pomoca:

```
dotnet test
```

Przykładowy wynik:

```
Test run for ORM.Tests.dll
Total tests: 21. Passed: 21. Failed: 0. Skipped: 0.
```

Daje to wysoka pewność, że logika mapowania, detekcji encji, nawigacji oraz materializacji działa zgodnie z założeniami projektowymi.

12 Podsumowanie i dalszy rozwój

12.1 Ocena obecnej architektury

Zaimplementowana warstwa mapowania (Mapping & Metadata) stanowi solidny, wydajny i stosunkowo prosty fundament ORM.

Najważniejsze osiagniecia:

- **Atrybuty i konwencje**
  System konfiguracji *by exception* pozwala mapować model w oparciu o konwencje, a nie reczna konfiguracje każdej właściwości.
- **Strategie nazewnictwa (Strategy Pattern)**
  Mechanizm `INamingStrategy` umożliwia zmiane konwencji nazewnictwa w całej aplikacji poprzez podmiane jednej klasy.
- **PropertyMap i EntityMap jako Metadata Objects**
  Logika mapowania została zamknieta w obiektach niemutowalnych, co poprawia bezpieczeństwo, wydajność oraz testowalność systemu.
- **ModelBuilder**
  Automatyczne wykrywanie encji na podstawie atrybutów i konwencji nazw znaczaco zmniejsza ilość konfiguracji wymagana od użytkownika ORM.
- **MetadataStore i MetadataStoreBuilder**
  Podejście oparte na instancji zamiast globalnego singletona poprawia testowalność, integracje z DI oraz umożliwia budowe wielu modeli w jednej aplikacji.
- **ObjectMaterializer**
  Wydajna materializacja rekordów z użyciem expression trees oraz ordinals, odporna na null-e i różne kolejności kolumn.

12.2 Zastosowane wzorce projektowe

W warstwie mapowania wykorzystano szereg wzorców projektowych:

- **Strategy Pattern** — dla `INamingStrategy`,
- **Builder Pattern** — dla `MetadataStoreBuilder`,
- **Factory Method / Expression Tree Factory** — w `ObjectMaterializer`,
- **Metadata Object** — `PropertyMap`, `EntityMap`,
- **Value Object / Immutable Object** — konstrukcja map jako niemutowalnych struktur,
- **Separation of Concerns** — rozdzielenie skanowania modelu, przechowywania metadanych i materializacji.

Implementacja nie tylko spełnia wymagania przedmiotu *Design Patterns*, ale też odzwierciedla praktyki stosowane w rzeczywistych ORM-ach (EF Core, NHibernate).

12.3 Możliwe kierunki rozwoju

Warstwa mapowania jest przygotowana na dalszy rozwój ORM w kolejnych sprintach, w szczególności:

- **Generowanie SQL (ISqlGenerator, SqliteSqlGenerator)**
  Bazujac na `EntityMap` oraz `PropertyMap`, można generować zapytania `SELECT`, `INSERT`, `UPDATE`, `DELETE` z pełnym wsparciem dla parametrów.
- **LINQ Provider**
  Majac metadane, możliwe jest translacja drzew wyrażeń (Expression Trees) na SQL, w tym:
  - `Where`, `OrderBy`, `Select`,
  - `Skip`, `Take`,
  - proste wyrażenia logiczne i porównania.
- **Obsługa relacji**
  W oparciu o Navigation Properties można dodać:
  - ładowanie zależności (eager, explicit),
  - metody `Include()`,
  - automatyczne generowanie JOIN-ów.

- **ChangeTracker i SaveChanges**
  Możliwe jest wprowadzenie śledzenia stanu encji oraz generowania odpowiednich komend SQL w zależności od stanu (Added, Modified, Deleted).

- **Migracje i walidacja schematu**
  Na podstawie metadanych można w przyszłości dodać weryfikacje zgodności modelu z baza, a nawet automatyczne migracje.

12.4 Wnioski końcowe

Zaimplementowana warstwa mapowania:

- jest solidna architektonicznie,
- wykorzystuje wzorce projektowe w sposób naturalny,
- jest gotowa na rozbudowe o kolejne warstwy ORM,
- jest w pełni testowalna,
- jest zgodna ze stylem i konwencjami nowoczesnych bibliotek .NET.