Polsko-Japońska Wyższa Szkoła Technik Komputerowych

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Marcin Niegowski
Nr albumu 3245

Generyczne mapowanie obiektowo-relacyjne z wykorzystaniem dedykowanego oprogramowania

Praca magisterska
napisana pod kierunkiem
Dr inż. Mariusza Trzaski

Warszawa, wrzesień 2009

Streszczenie

Praca obejmuje zagadnienia mapowania relacyjno-obiektowego. Skupia się na problemach związanych z połączeniem programowania zorientowanego obiektowo z językiem zapytań SQL. Dodatkowo autor pracy zwraca uwagę na zautomatyzowanie zadań programistycznych, w tym wypadku dostępu do danych, poprzez generowanie kodu źródłowego.

Autor przedstawia prototyp aplikacji mapującej w postaci wtyczki do środowiska programistycznego – Visual Studio 2008 – generującej kod warstwy dostępu do danych utrwalonych w bazie danych MS SQL. Programiści mogą w ten sposób zautomatyzować proces wytwarzania aplikacji oraz mają dostęp do obiektowego języka zapytań, jakim jest technologia LINQ.

Spis treści

1. Wstęp

Mapowanie obiektowo-relacyjne to technika pozwalająca konwertować dane z niekompatybilnych systemów. Niezgodność impedancji języków programowania zorientowanych obiektowo z językiem zapytań SQL prowadzi do wzrastającej popularności tego typu rozwiązań. Programiści przy użyciu semantyki i paradygmatów właściwych dla obiektowego języka programowania mogą uzyskiwać dane przechowywane w postaci relacyjnej i przeprowadzać na nich operacje.

Rozwiązania takie znacząco wpływają na czytelność kodu aplikacji, a co za tym idzie, także na łatwość zarządzania nim i rozwijania go.

1.1. Cel Pracy

Autor za cel postawił sobie zebranie i przedstawienie zasad, którymi powinny odznaczać się aplikacje mapujące dane w postaci relacyjnej na obiekty dostępne w nowoczesnych językach programowania.

Pośrednim rezultatem pracy jest opracowanie wtyczki do środowiska programistycznego Visual Studio 2008 realizującej następujące założenia:

- Wygenerowanie klas mapujących dane relacyjne
- Automatyzacja procesu tworzenia warstwy dostępu do danych przechowywanych w relacyjnej bazie danych MS SQL 2005
- Możliwość tworzenia zapytań w sposób zgodny z zasadami programowania obiektowego.

Opracowana wtyczka pozwala programistom zautomatyzować powtarzalny proces przygotowania warstwy dostępu do danych, a także uniknąć mieszania kodu SQL w kodzie aplikacji.

1.2. Przyjęte rozwiązania

Proces tworzenia prototypu został oparty na środowisku programistycznym Visual Studio 2008. Aplikację wykonano przy użyciu języka C# oraz, na potrzeby uzyskania meta-danych relacyjnej bazy danych, języka zapytań SQL. Autor wykorzystał także biblioteki wchodzące w skład platformy .NET w wersji 3.5.

Do zaprojektowania aplikacji wykorzystano wzorzec Active Record.

1.3. Osiągnięte rezultaty

Rezultatem pracy jest określenie najważniejszych założeń i reguł tworzenia aplikacji mapujących oraz wskazanie problemów wiążących się z funkcjonowaniem dwóch systemów: relacyjnego oraz obiektowego.

Wynikiem pracy jest także przygotowanie aplikacji generującej kod źródłowy warstwy dostępu do danych uwzględniającej mapowanie relacyjno-obiektowe w postaci wtyczki do środowiska programistycznego Visual Studio 2008.

1.4. Organizacja pracy

Praca rozpoczyna się od zaprezentowania i omówienia rozwiązań dostępnych na rynku. Przedstawione zostały wady i zalety istniejących aplikacji mapujących.

W rozdziale trzecim wymieniono narzędzia wykorzystane przy konstrukcji prototypu, w tym także wykorzystane biblioteki dostępne w .NET Framework 3.5 oraz technologię LINQ.

W kolejnym rozdziale omawia się założenia funkcjonalne, które powinny być spełnione przez aplikacje mapujące. Co więcej, przedstawia się elementy relacyjnych typów danych i możliwości ich odwzorowania w obiektowych językach programowania. Kolejnym opracowanym zagadnieniem jest automatyzacja procesów wytwarzania oprogramowania i generowania kodu.

Następnie autor pracy prezentuje rozwiązania użyte w prototypie oraz charakteryzuje problemy, które pojawiły się w trakcie tworzenia aplikacji.

Praca kończy się podsumowaniem zalet i wad omawianych wcześniej rozwiązań, a także prezentuje możliwości dalszego rozwoju przygotowanej aplikacji.

2. Mapowanie relacyjno-obiektowe – stan sztuki

Dzisiejsze aplikacje zarówno sieciowe – dostępne przez przeglądarkę www – jak i aplikacje desktopowe, coraz częściej są zależne od zewnętrznego źródła danych. W większości przypadków dane te przechowywane są w relacyjnych bazach danych, w celu zapewnienia im trwałości. W związku z tym zadaniem programisty jest połączenie dwóch różnych modeli operowania na danych – obiektowego i relacyjnego. Co za tym idzie, konieczne staje się mieszanie języka zapytań z językiem programowania, co powoduje niezgodność impedancji (Definicja 1) i jest sprzeczne z zasadami programowania zorientowanego obiektowo.

Definicja 1
Niezgodność impedancji[1] *jest to zespół niekorzystnych zjawisk towarzyszących formalnemu połączeniu języka zapytań (w szczególności SQL) z uniwersalnym językiem programowania takim jak C, Cobol lub PL/I.* □

W sztuce programowania objawia się ona w szczególności w zakresie:

- **składni –** konieczność używania dwóch stylów językowych oraz gramatyk w procesie tworzenia oprogramowania
- **systemu typów –** język zapytań operuje na innych typach (np. relacje) niż język programowania; dodatkowo nowoczesne języki programowania posiadają statyczną kontrolę typów w trakcie kompilacji, podczas gdy język SQL takiej kontroli nie posiada
- **semantyki i paradygmatów języków –** język zapytań oraz język programowania posiadają różne koncepcje semantyki; podczas gdy język zapytań oparty jest na stylu deklaracyjnym, w językach programowania stosowany jest styl imperatywny
- **pragmatyki użycia –** język zapytań nie wymaga od programisty wiedzy dotyczącej organizacji, implementacji danych lub struktur je przechowujących

natomiast języki programowania wymagają, aby struktury danych były odpowiednio oprogramowane

- **faz i mechanizmów wiązania –** w językach programowania występuje wczesne wiązanie typów, czyli podczas kompilacji natomiast języki zapytań są interpretowane (późne wiązanie typów),
- **traktowania cechy trwałości danych –** języki zapytań operują tylko na danych przechowywanych fizycznie na dysku natomiast języki programowania przetwarzają dane nietrwałe znajdujące się w pamięci operacyjnej.

Z niezgodności impedancji wynika konieczność tworzenia warstwy dostępu do danych, pośredniczącej w komunikacji języka zapytań oraz języka programowania. Zaletami wynikającymi z wprowadzenia takiego rozwiązania są: warstwowość tworzonej aplikacji oraz mapowanie danych na system obiektowy.

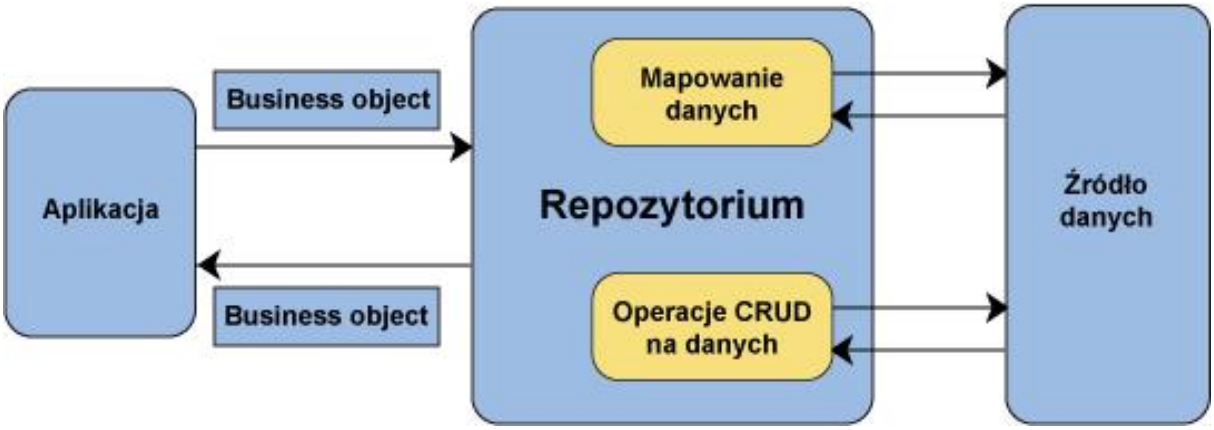
W celu ograniczenia błędów popełnianych podczas programowania warstwy dostępu, powstał szereg rozwiązań, zarówno komercyjnych, jak i zintegrowanych z platformą .NET, które realizują założenia mapowania obiektowo-relacyjnego.

2.1. Wzorzec Repository

Wzorzec Repository ściśle łączy się z ideą Domain Driven Design, która opiera się na wykorzystaniu domeny problemowej. Analiza i modelowanie zgodne z Domain Driven Design zakłada, że nacisk powinien zostać położony na zagadnienia biznesowe nie na technologię. Dlatego w ramach DDD wyróżnia się domain objects, zwane też business objects. W domenie problemowej, którą jest zamówienie w sklepie, przykładem business object będzie obiekt **Zamówienie** (Rysunek 1), który zawiera także informacje o produkcie oraz kliencie. Podczas gdy technicznie (na poziomie bazy danych) są to trzy różne obiekty: **Klient, Produkt, Zamówienie.**

Rysunek 1 – Przykład business object

Do obsługi business objects służą specjalne repozytoria, których zadaniem jest nie tylko stworzenie odpowiedniej struktury, zdolnej przechowywać tak modelowane dane, lecz także przygotowanie mechanizmu do operowania na nich. Architekturę rozwiązań opartych na repozytorium prezentuje Rysunek 2.



Rysunek 2 – Architektura rozwiązań opartych o repozytorium

Repozytorium pośredniczy pomiędzy domeną problemową a warstwą mapowania tworząc swoistą kolekcję obiektów domeny w pamięci [2]. Dzięki użyciu repozytoriów, programiści mogą bezpośrednio korzystać z danych zamodelowanych w sposób odpowiadający procesom biznesowym bez konieczności tworzenia dodatkowych struktur danych.

Należy pamiętać, że repozytorium nie jest warstwą dostępu do danych. Udostępnia ono obiekty często składające się z kilku podmiotów, natomiast DAL (Data Access Layer) zazwyczaj zwraca zbiór rekordów. Repozytoria wyróżnia także możliwość połączenia z różnymi źródłami danych. Teoretycznie jest zatem możliwe, że pojedynczy business object będzie składał się z dwóch rodzajów informacji:

- pobranych z bazy danych
- dostępnych np. w pliku XML lub otrzymanych przez WebService.

Innym wzorcem używanym przy mapowaniu danych jest Active Record, który został opisany w dalszej części pracy w rozdziale pt. Narzędzia użyte w pracy.

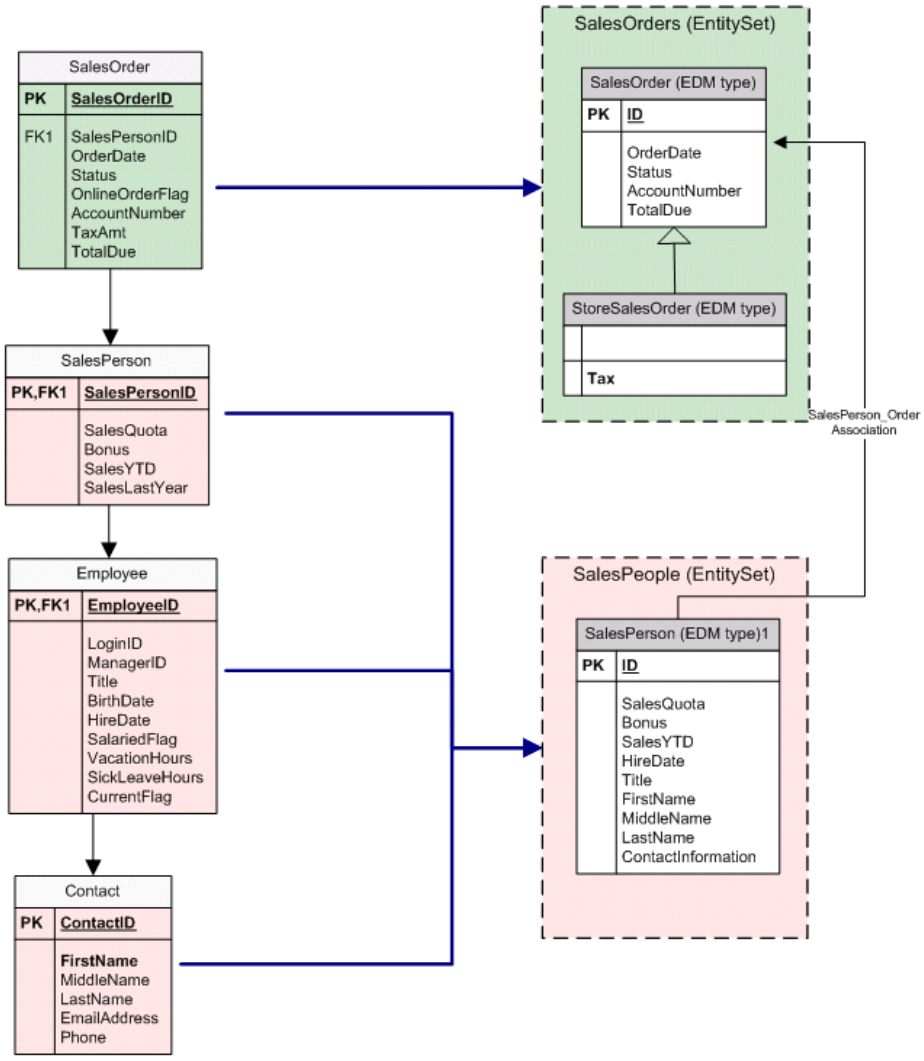
2.2. ADO.Net Entity Framework

Pierwsza wersja Entity Framework pojawiła się wraz z platformą .NET 3.5 dostarczaną z Visual Studio 2008. Jest to implementacja idei Domain Driven Design oraz repozytorium. Głównym założeniem jest istnienie trzech modeli danych: fizycznego, logicznego oraz koncepcyjnego. Model fizyczny odnosi się do specyficznej platformy przechowywania informacji w pamięci trwałej np. sposobu zapisu danych na dyskach twardych przez MS SQL. Najważniejsze dla programistów są jednak elementy Entity Framework:

- model koncepcyjny omawiający encje i relacje w ramach systemu, który jest analizowany
- model logiczny, czyli model koncepcyjny znormalizowany dla relacyjnych baz danych, opisany tabelami i relacjami między nimi.

W procesie tworzenia oprogramowania często pomija się model koncepcyjny, który jest stosowany podczas zbierania wymagań i opisu zależności w systemach. Zespoły programistyczne od razu przystępują do modelowania zagadnienia w sposób relacyjny. Entity Framework zakłada, że model koncepcyjny jest najbardziej zbliżony do rzeczywistych wymagań stawianych wobec systemów informatycznych. Z tego powodu został wprowadzony tzw. Entity Data Model (EDM), który jest specyfikacją mapowania pomiędzy modelem koncepcyjnym a modelem logicznym (Rysunek 3). W ten sposób

aplikacja może współdziałać z modelem koncepcyjnym, natomiast EDM konwertuje dane do postaci, w której zostaną zapisane w relacyjnej bazie danych.



Rysunek 3 – Model logiczny (po lewej) i odpowiadający mu model koncepcyjny (po prawej)[4]

2.2.1. Entity Data Model

Budowa Entity Data Model jest trójwarstwowa (Rysunek 4). Każda warstwa odpowiada za inny model. Podczas projektowania aplikacji opartej na Entity Framework, należy utworzyć 3 pliki opisujące EDM:

- plik definiujący model koncepcyjny – .csdl
- plik definiujący model logiczny – .ssdl
- plik mapujący, definiujący zależności między nimi – .mdl

Rysunek 4 – Warstwy składowe Entity data Model

Wszystkie pliki są oparte na standardzie XML i w ten sposób opisują warstwę, za którą są odpowiedzialne.

Entity Framework bazuje na modelu koncepcyjnym, dlatego projektowanie należy zacząć od przygotowania pliku .csdl (Conceptual schema definition language). Przykładowy opis w postaci kodu XML przedstawia Przykład 1.

```
<?xml version="1.0" encoding="utf-8"?>

<Schema xmlns="http://schemas.microsoft.com/ado/2006/04/edm"

        Namespace="MyCompany.LOBSchema" Alias="Self">
```

```
    <EntityType Name="Customer">
        <Key>
            <PropertyRef Name="CustomerId" />
        </Key>
        <Property Name="CustomerId" Type="Guid" Nullable="false" />
        <!-- Other properties-->
    </EntityType>
    <EntityType Name="Order">
        <Key>
            <PropertyRef Name="OrderId" />
        </Key>
        <Property Name="OrderId" Type="Guid" Nullable="false" />
        <!-- Other properties-->
    </EntityType>
    <Association Name="Customer_Order">
        <End Role="Customer" Type="Self.Customer" Multiplicity="0..1"
/>
        <End Role="Order" Type="Self.Order" Multiplicity="*" />
    </Association>

    <EntityContainer Name="LOBSchemaData">
        <EntitySet Name="Customers" EntityType="Self.Customer" />
        <EntitySet Name="Orders" EntityType="Self.Order" />
        <AssociationSet Name="Customer_Orders"
Association="Self.Customer_Order">
            <End Role="Customer" EntitySet="Customers" />
            <End Role="Order" EntitySet="Orders" />
        </AssociationSet>
```

```
    </EntityContainer>
</Schema>
```

Przykład 1 – Plik CSDL[4]

Dwa podstawowe znaczniki opisujące model koncepcyjny to:

- EntityType – służy do opisania modelu danych
- Association – służy do opisu zależności między modelami.

W Entity Framework należy także opisać, w jaki sposób dane mają być przechowywane. Służą do tego pliki SSDL (store schema definition language), w których znajduje się charakterystyka struktury bazy danych. Opis, tak jak w przypadku plików CDSL, składa się ze znaczników z tymże dotyczy istniejących tabel i kolumn oraz relacji między nimi.

Ostatni plik, który należy przygotować na potrzeby Entity Framework, zawiera opis mapowania modelu koncepcyjnego na model logiczny. Przyporządkowuje on tabele i kolumny (opisane w pliku SSDL) typom danych (wskazanym w pliku CSDL). Fragment pliku MSL (mapping specification language) pokazuje, w jaki sposób należy przygotować takie mapowanie (Przykład 2).

```
<?xml version="1.0" encoding="utf-8"?>
<Mapping Space="C-S" xmlns="urn:schemas-microsoft-
com:windows:storage:mapping:CS">
  <EntityContainerMapping StorageEntityContainer="Production"
CdmEntityContainer="AdventureWorksContext">
    <EntitySetMapping Name="AWBuildVersion"
StoreEntitySet="AWBuildVersion"
TypeName="AdventureWorks.AWBuildVersion">
      <ScalarProperty Name="SystemInformationID"
ColumnName="SystemInformationID" />
      <ScalarProperty Name="Database_Version" ColumnName="Database
Version" />
```

```
      <ScalarProperty Name="VersionDate" ColumnName="VersionDate" />
      <ScalarProperty Name="ModifiedDate" ColumnName="ModifiedDate"
/>
    </EntitySetMapping>
...
<AssociationSetMapping Name="FK_StoreContact_Store_CustomerID"
TypeName="AdventureWorks.FK_StoreContact_Store_CustomerID"
StoreEntitySet="StoreContact">
      <EndProperty Name="Store">
        <ScalarProperty Name="CustomerID" ColumnName="CustomerID" />
      </EndProperty>
      <EndProperty Name="StoreContact">
        <ScalarProperty Name="CustomerID" ColumnName="CustomerID" />
        <ScalarProperty Name="ContactID" ColumnName="ContactID" />
      </EndProperty>
    </AssociationSetMapping>
  </EntityContainerMapping>
</Mapping>
```

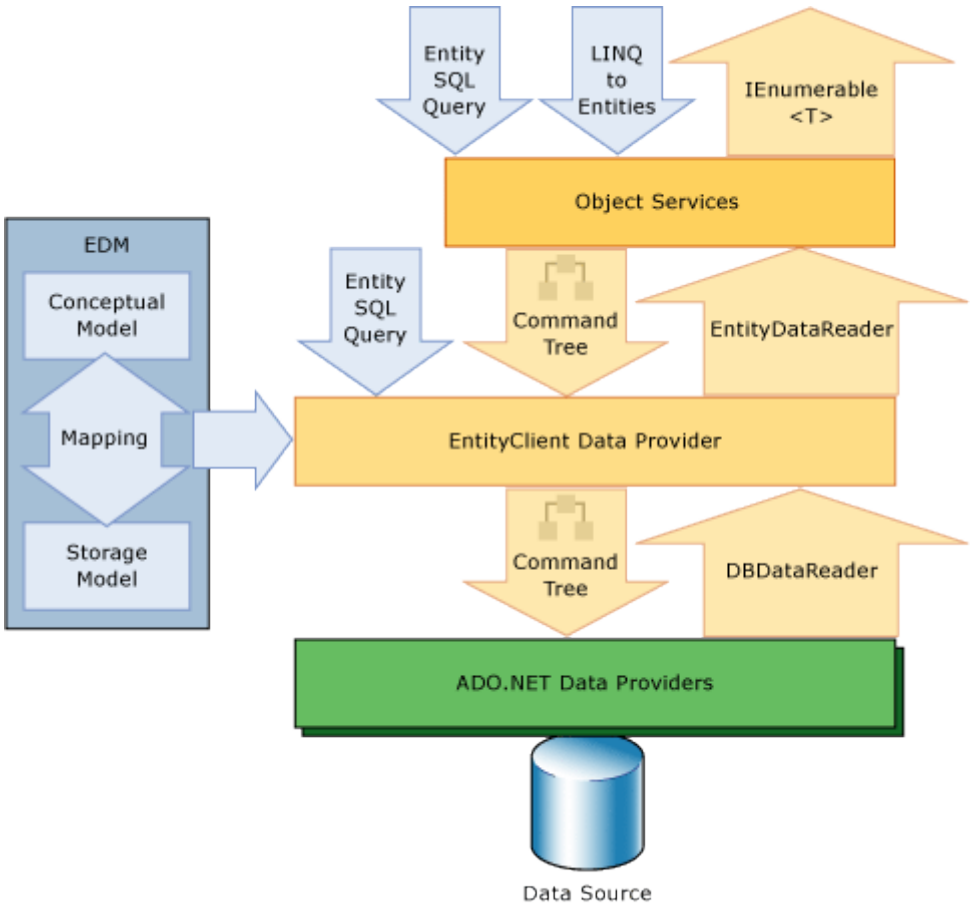
Przykład 2 – plik MSL[4]

2.2.2. Entity Client Data Provider

Aplikacje działające w oparciu o Entity Framework korzystają z modelu koncepcyjnego. To założenie powoduje, że należy zmienić sposób odczytu i zapisu danych. W tym celu wprowadzony został Entity Client odpowiedzialny za zarządzanie połączeniami, tłumaczenie zapytań na język specyficzny dla źródła danych, oraz dostarczenie danych. Rysunek 5 przedstawia architekturę dostępu do danych w Entity Framework. Przy tłumaczeniu zapytań Entity Client ściśle opiera się na opisach modeli danych zawartych w Entity Data Model.

Zostały też wprowadzone dwa sposoby tworzenia zapytań na modelu koncepcyjnym:

- Entity SQL Query – dialekt języka SQL służący do tworzenia zapytań opartych na koncepcyjnym modelu danych; Entity SQL uwzględnia konstrukcje zawarte w EDM, takie jak relacje oraz dziedziczenie
- LINQ to Entities – część technologii LINQ (Language-Integrated Query), która jest opisana w dalszej części pracy w rozdziale o użytych narzędziach.



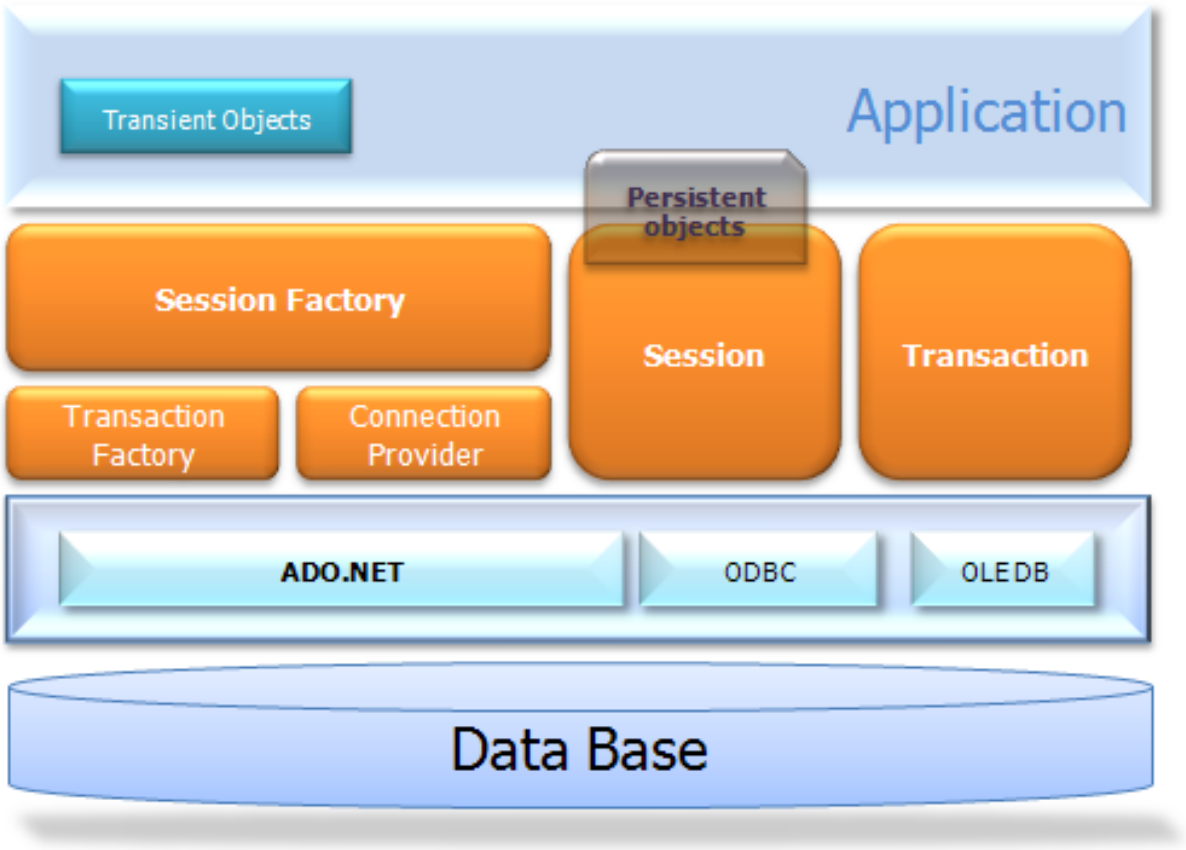
Rysunek 5 – Architektura Entity Framework

2.3. NHibernate

NHibernate jest to przeniesienie rozwiązania dostępnego dla języka Java na platformę .NET. Projekt jest rozwijany jako wolne oprogramowanie udostępniane na

licencji LGPL (Lesser GNU Public License). [5] Architektura NHibernate (Rysunek 6) jest elastyczna, przez co rozwiązanie to jest często wybierane przez programistów.

Głównym zadaniem NHibernate jest mapowanie klas języka C# na tabele bazy danych, w tym także mapowanie typów CLR (Common Language Runtime) na typy bazy danych. Jednocześnie oprogramowanie to dostarcza metod, służących do operowania na danych – zapisu, odczytu, edycji, usuwania.



Rysunek 6 – Architektura NHibernate

NHibernate jest bardzo popularnym rozwiązaniem ze względu na najbardziej rozbudowane API. Jednak przez nadmierne dodawanie funkcjonalności dokumentacja rozwiązań jest niepełna. Z tego powodu nauka tej technologii jest czasochłonna.

Do poprawnej pracy wymagane jest utworzenie mapowań wskazujących odpowiednio, w jaki sposób obiekty mają być przechowywane w relacyjnej bazie danych. Dostępne są 3 metody mapowania.

2.3.1. Mapowanie z użyciem pliku XML

Mapowanie z użyciem XML jest pierwotnym i podstawowym sposobem. Polega na utworzeniu pliku wskazującego klasa po klasie, w jaki sposób będzie odbywać się mapowanie (Przykład 3).

```
<?xml version=”1.0” encoding=”utf-8” ?>
<hibernate-mapping xmlns=”urn:nhibernate-mapping-2.0”>
    <class name=”Nhibernate.Examples.QuickStart.User,
Nhibernate.Examples” table=”users”>
        <id name=”Id” column=”LogonId” type=”String” length=”20”>
            <generator class=”assigned” />
        </id>
        <property name=”UserName” column=”Name” type=”String”
length=”40”/>
        <property name=”Password” type=”String” length=”20”/>
        <property name=”EmailAddress” type=”String” length=”40”/>
        <property name=”LastLogon” type=”DateTime”/>
    </class>
</hibernate-mapping>
```

Przykład 3 – Plik mapujący dla NHibernate[5]

Rozwiązanie to może generować trudności związane z:

- utworzeniem poprawnej konfiguracji
- koniecznością poznania języka XML i znaczników wykorzystywanych do mapowania
- refeaktoryzacją
- dublowaniem opisu obiektów (XML i klasy w C#).

2.3.2. Mapowanie przy użyciu atrybutów

Użycie atrybutów rozwiązuje problem tylko w pewnym stopniu. W rzeczywistości to z atrybutów generowane są pliki XML. Jeśli w ten sposób zefiniuje się mapowanie można uniknąć dublowania opisu obiektów. Metoda ta jest krytykowana za niepełną dokumentację oraz fakt, że jest to jedynie przepisanie pliku XML na atrybuty. Sposób użycia atrybutów do opisu mapowania w NHibernate prezentuje Przykład 4.

```
[Class(NameType = typeof(Dog))]
public class Dog
{
        private Guid id;
        private string strName;

        [Id(0,
        Column = "ID",
        Name = "ID",
        TypeType = typeof(Guid),
        UnsavedValue = "(00000000-0000-0000-0000-000000000000)"
        )]
        [Generator(1, Class = "guid.comb")]
        public virtual Guid ID
        {
```

```
                get { return id; }
                private set { id = value; }
        }

        [Property(0, Column = "Name",
        Name = "Name",
        TypeType = typeof(string),
        Length = 50,
        NotNull = false
        )]
        public virtual string Name
        {
                get { return strName; }
                set { strName = value; }
        }
}
```

Przykład 4 – Wykorzystanie atrybutów do opisu mapowania

2.3.3. Mapowanie przy użyciu Fluent NHibernate

Fluent NHibernate to projekt, który także pozwala na pominięcie tworzenia pliku XML przy opisie mapowania. Rozwiązanie to jest o tyle interesujące, o ile eliminuje praktycznie wszystkie problemy związane z tworzeniem plików konfiguracyjnych. Inaczej niż w przypadku użycia atrybutów, opis mapowania jest oddzielony od opisu obiektu, przy czym nadal używa języka C#. Rozwiązanie to jest także bardziej odporne na błędy (sprawdzanie podczas kompilacji) oraz łatwiejsze do zarządzania, ponieważ umożliwia wykorzystanie narzędzi dostępnych w IDE. Przykład 5 prezentuje sposób opisu mapowań przy wykorzystaniu Fluent NHibernate.

```
public class StateMap : ClassMap<State> {
	public StateMap() {
		Id(x => x.StateId);
		Map(x => x.Name);
	}
}

public class SupplierMap : ClassMap<Supplier> {
	public SupplierMap() {
		Id(x => x.SupplierId);
		Map(x => x.Name);
		HasManyToMany<State>(x => x.StatesServiced)
		.AsBag()
		.WithTableName("Supplier_StatesServiced")
		.WithParentKeyColumn("SupplierId")
		.WithChildKeyColumn("StateId");
	}
}
```

Przykład 5 – Mapowanie przy użyciu Fluent NHibernate

2.4. Castle ActiveRecord

Castle ActiveRecord jest implementacją wzorca ActiveRecord na platformę .NET. Projekt ten jest oparty na Hibernate jednak założeniem jest pozbycie się uciążliwej konfiguracji w pliku XML. Zamiast tego twórcy proponują oparcie mapowania na atrybutach (Przykład 6). Ponieważ Castle ActiveRecord jest tylko podzbiorem funkcjonalności Hibernate, nie udało się wyeliminować niedogodności związanych z zastosowaniem tego rozwiązania. Z tego powodu podstawowa wiedza dotycząca technologii bazowej jest potrzebna, aby sprawnie korzystać z Castle ActiveRecord. Projekt jest ciagle rozwijany, a twórcy zaznaczają, że w obecnym stadium ich rozwiązanie nie jest w stanie poprawnie mapować złożonych baz danych i w tym wypadku zalecają użycie NHibernate.

```
[ActiveRecord]
public class Category : ActiveRecordBase
{
    private int id;
    private string name;
    private Category parent;
    private IList<Category> subcategories = new List<Category>();

    [PrimaryKey]
    public int Id
    {
        get { return id; }
        set { id = value; }
    }

    [Property]
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    [BelongsTo("parent_id")]
    public Category Parent
    {
        get { return parent; }
        set { parent = value; }
    }

    [HasMany]
    public IList<Category> SubCategories
    {
```

```
        get { return subcategories; }
        set { subcategories = value; }
    }
}
```

Przykład 6 – Wykorzystanie atrybutów Castle ActiveRecord[7]

2.5. SubSonic

SubSonic jest darmowym rozwiązaniem, które jest intensywnie rozwijane. W trzeciej wersji projektu dostępne są dwa podejścia mapowania obiektowo-relacyjnego umożliwiające różny stopień kontroli.

2.5.1. SimpleRepository

SimpleRepository jest kierowane do osób, którym struktura bazy danych jest obojętna. SubSonic potrafi bowiem stworzyć schemat bazy danych na podstawie klas jednak nie jest on optymalny. Eliminuje to problem niezgodności impedancji pozwalając programiście na pisanie kodu klas bez konieczności dokonywania zmian w schemacie bazy danych. Jednocześnie obiekty w ten sposób otrzymane są wolne od klas bazowych i mogą istnieć samodzielnie.

Tabele bazy danych są tworzone w trakcie uruchamiania pisanej aplikacji. Konstruktor SimpleRepository posiada enumerator ***SimpleRepositoryOptions*** definiujący, czy podczas łączenia z bazą danych porównywać i tworzyć tabele, czy nie. Jeżeli jest ustawiony na ***RunMigrations***, to przy każdej operacji na tabeli bazy danych będzie ona porównywana z klasą obiektu i odpowiednio modyfikowana.

2.5.2. ActiveRecord

ActiveRecord jest implementacją wzorca o tej samej nazwie, w którym każda instancja jest odpowiednikiem jednego wiersza w bazie danych, czyli każda klasa aplikacji odpowiada tabeli w bazie. ActiveRecord w SubSonic daje większą kontrolę

nad schematem bazy danych. Nie potrafi automatycznie generować schematu, projektant zatem musi zadbać zarówno o model relacyjny, jak i utworzyć odpowiadające mu klasy w aplikacji.

2.6. Wady istniejących rozwiązań

Autor pracy przedstawił najnowsze (Entity Framework) oraz najbardziej popularne (NHibernate) rozwiązania istniejące na rynku. Obecnie dostępne są liczne aplikacje mapujące dane z relacyjnych baz na obiekty dla platformy .NET. Jednak wszystkie zdają się działać w podobny sposób, w związku z tym powielają wady znajdujące się w NHibernate.

Każde rozwiązanie wydaje się skomplikowane, zwłaszcza idea Entity Framework wprowadzająca model koncepcyjny, co dodatkowo utrudnia projektowanie. Programiści podczas tworzenia aplikacji mapujących relacyjne bazy danych na obiekty zgodne z zasadami programowania obiektowego, starają się połączyć dwa różne sposoby opisu zagadnień biznesowych. Propozycja Microsoftu zdaje się dodawać jeszcze jeden model, z którym programiści będą musieli sobie radzić, co może komplikować tworzenie aplikacji.

Większość rozwiązań nie eliminuje problemu niezgodności impedancji. W przypadku domyślnego użycia NHibernate i Entity Framework projektant musi utworzyć dodatkowe pliki XML opisujące zagadnienia biznesowe. Co prawda, nie jest to tak formalne i bezpośrednie łączenie dwóch języków jak mieszanie kodu SQL z kodem np. C# jednak pośrednio wymaga od projektanta poznania składni i znaczników stosowanych w plikach konfiguracyjnych właściwych dla danego rozwiązania.

Praktycznie wszystkie technologie zmuszają programistów do zapoznania się z dodatkowym opisem niezbędnym do działania ich aplikacji. Jednocześnie wprowadza to kolejną warstwę, w której może wystąpić błąd.

3. Narzędzia użyte w pracy

3.1. Wzorzec Active Record

Active Record polega na odwzorowaniu tabel baz danych na klasy, które są ściśle przyporządkowane strukturze rekordów. Każdy obiekt takiej klasy (tzw. Active Record) jest odpowiedzialny za zapisywanie informacji.

Struktura danych wzorca Active Record powinna dokładnie odpowiadać strukturze bazy. Z tego powodu zakłada się, że każde pole w klasie odnosi się do kolumny w źródle danych. Typowymi funkcjonalnościami, które znajdują się w klasie Active Record są:

- tworzenie instancji na podstawie danych otrzymanych z zapytania SQL
- tworzenie nowej instancji i późniejsze jej zapisanie
- statyczne metody wyszukiwania
- uaktualnianie oraz dodawanie danych do bazy
- ustawianie oraz odczytywanie poszczególnych pól
- umieszczanie części logiki biznesowej.

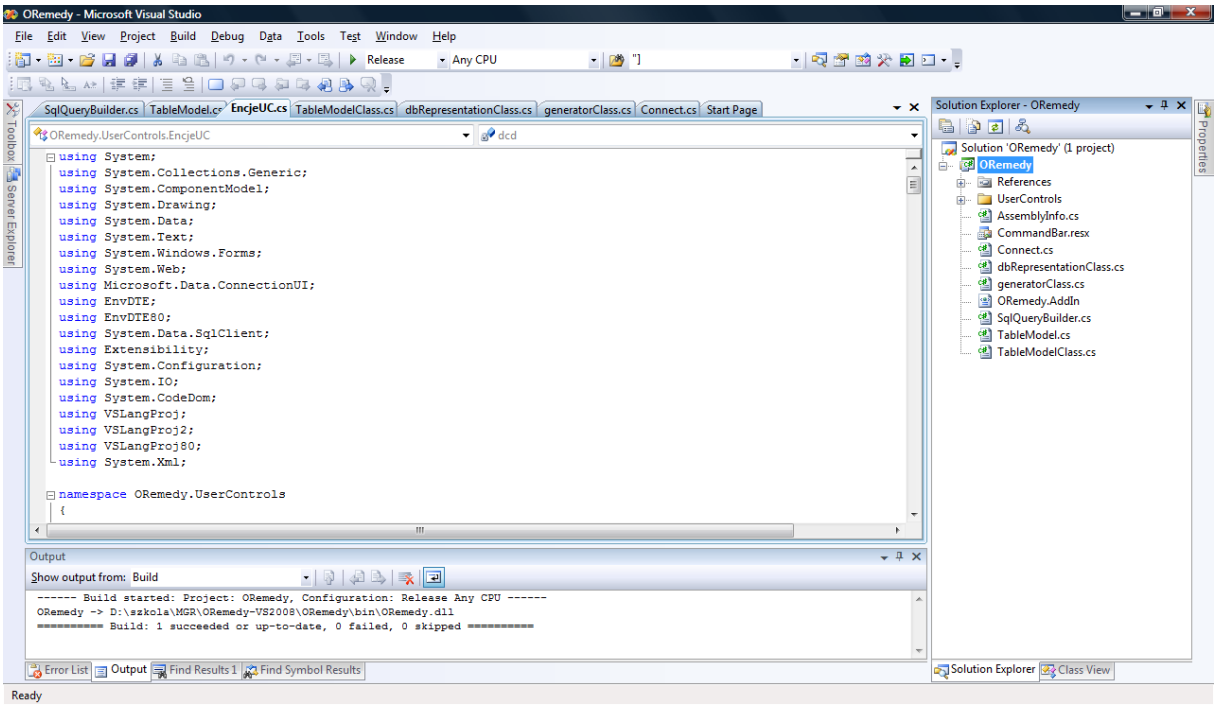
Często też przyjmuje się, że w przypadku powiązanych tabel, powinny być zwracane obiekty a nie tylko dane w postaci klucza obcego.

3.2. Visual Studio 2008

Visual Studio (Rysunek 7) to środowisko programistyczne służące do budowania aplikacji w oparciu o technologie firmy Microsoft, w szczególności:

- aplikacje dedykowane dla rodziny systemów Windows – technologie Windows Forms oraz WPF (Windows Presentation Fundation)
- pakietu biurowego Office
- aplikacje mobilne – Windows CE i Windows Mobile
- aplikacje internetowe – ASP.Net oraz Silverlight.

Visual Studio zawiera cenione przez programistów środowisko do debugowania tworzonych aplikacji. Dodatkowo posiada rozbudowany system podpowiedzi kodu tzw. IntelliSense, który pozwala na wydajniejszą pracę programistów. Prócz tych udogodnień wersja 9.0 (Visual Studio 2008) wprowadza platformę do tworzenia testów jednostkowych oraz zaawansowany system kontroli wersji i pracy grupowej (Visual Studio Team System).



Rysunek 7 – Visual Studio 2008

Visual Studio posiada duże możliwości rozbudowy. Programiści mogą tworzyć:

- makra – w przypadku czynności powtarzalnych, takich jak zapisywanie dokumentów
- wtyczki (add-ins), kreatorzy (wizards) – do rozbudowy IDE o nowe funkcjonalności
- aplikacje oparte na Visual Studio SDK – dające największe możliwości rozbudowy; służą do tworzenia zintegrowanych narzędzi lub dodawania obsługi innych języków programowania

3.3. C# i .Net Framework 3.5

Wraz z Visual Studio 2008 zaprezentowane zostały nowe wersje frameworka .NET – 3.5 – oraz języka C# – 3.0.

.NET Framework to zestaw bibliotek wspomagających tworzenie oprogramowania i ułatwiających rozwiązanie wielu typowych problemów programistycznych. W ramach bibliotek znajdują się m.in. biblioteki do tworzenia interfejsów użytkownika, dostępu do danych, połączeń bazodanowych, kryptografii oraz połączeń sieciowych.

Głównym elementem jest środowisko uruchomieniowe Common Language Runtime (CLR) wspomagające wykonywanie aplikacji. Do najważniejszych zadań CLR należą:

- zarządzanie pamięcią – poprzez Garbage Collector, który korzystając ze stosu obiektów usuwa te, do których nie ma żadnych referencji,
- zarządzanie wyjątkami
- weryfikację bezpieczeństwa kodu.

Dzięki CTS (Common Type System) udało się osiągnąć niezależność platformy od języka programowania. CTS definiuje możliwe typy danych oraz sposób radzenia sobie z nimi przez CLR. Ważnym aspektem jest także kompatybilność ze starszymi technologiami. Możliwość pisania zarządzanego i niezarządzanego kodu pozwala programistom w dalszym ciągu na użycie komponentów COM.

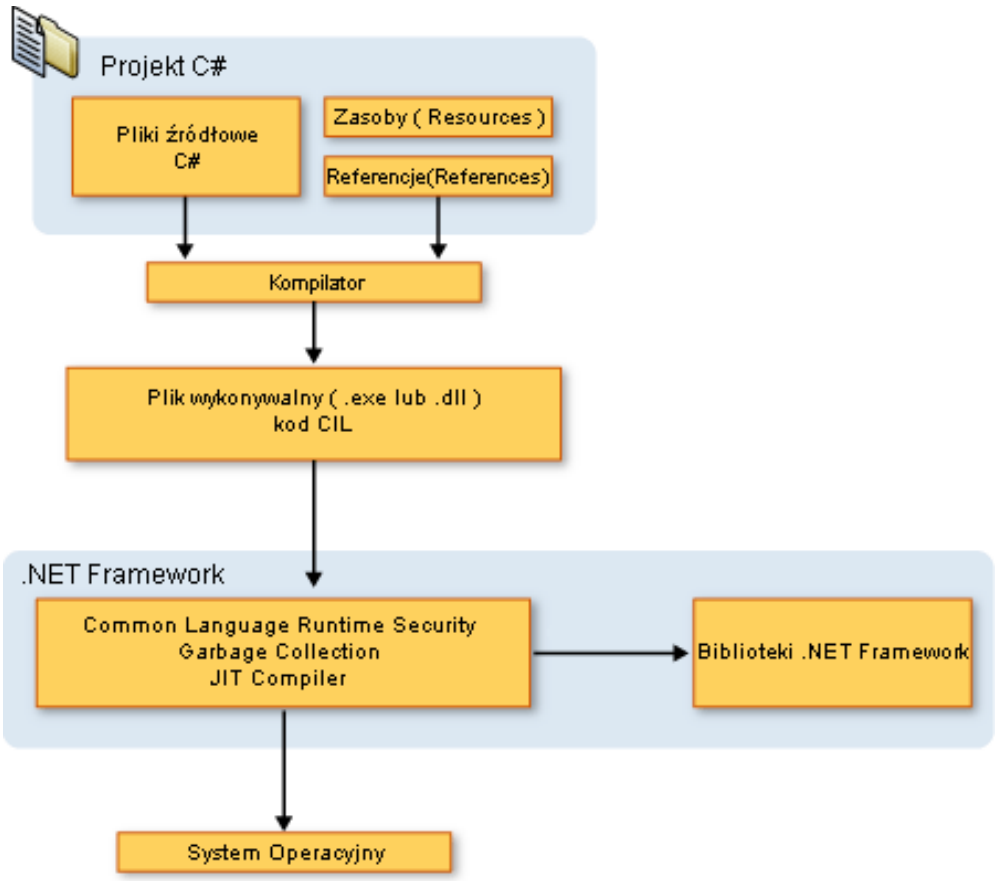
Pomimo prac nad implementacją .NET Framework dla innych systemów operacyjnych, na obecnie jest on w pełni dostępny jedynie dla rodziny Windows.

C# jest nowoczesnym, obiektowym językiem programowania, którego składnia została oparta na składni języka C++. Występuje tu silna kontrola typów, a także zachowana jest hierarchia klas, której nadrzędnym elementem jest *Object*. W C# niedozwolone jest wielodziedziczenie, w zamian klasy mogą implementować wiele interfejsów.

Niektóre z właściwości wprowadzonych do C# 3.0:

- pośrednie deklarowanie lokalnych zmiennych, gdzie typ zostaje określony na podstawie przypisanej wartości
- deklarowanie metod rozszerzających istniejące typy
- wyrażenia Lambda
- brak wielodziedziczenia, w zamian klasy mogą implementować wiele interfejsów.

Kod C# przed uruchomieniem jest kompilowany do kodu pośredniego tzw. CIL (Common Intermediate Language). Efektem tej kompilacji jest plik wykonywalny .exe lub .dll, który może funkcjonować w środowisku uruchomieniowym .NET Framework, czyli CLR. W trakcie wykonywania aplikacji kod CIL, jeśli spełnia wymogi bezpieczeństwa, jest kompilowany do postaci natywnej dla systemu operacyjnego. Zależności te prezentuje Rysunek 8.



Rysunek 8 – zależności uruchomienia kodu C#

3.4. LINQ

LINQ (Language Integrated Query) to technologia natywnych zapytań dla języków platformy .NET. Pozwala ona odpytywać różne źródła danych: od tablic i enumeratorów, poprzez XML do relacyjnych baz danych. Konieczne jest przedstawienie danych jako obiektów, jeśli natomiast nie są natywnie przechowywane w tej formie, należy je odpowiednio zmapować.[3]

LINQ jako natywna technologia pozwala programistom na wykorzystanie sprawdzania składni podczas kompilacji, kontroli typów jak również IntelliSense, co poprawia proces tworzenia oprogramowania.

Budowanie zapytań umożliwia odpowiednie API, którego operatory prezentuje Tabela 1.

Operator	**Opis**
Where	Operator ograniczeń `Where(s1 => s1.Length == 5)`
Select/SelectMany	Operator wyboru elementów np. `Select(s => s.ToUpper())`
Take/Skip/ TakeWhile/SkipWhile	Operatory wyboru / pominięcia n pierwszych elementów
Join/GroupJoin	Operator łączenia w oparciu o klucze
Concat	Operator konkatenacji
OrderBy/ThenBy/OrderByDescending/ ThenByDescending	Operatory sortowania np. `OrderBy(s => s.Length).ThenBy(s => s)`
Reverse	Operator sortowania odwracający porządek sekwencji

GroupBy	Operator grupowania
Distinct	Operator usuwający duplikaty w zbiorze
Union/Intersect	Operatory zwracające przecięcie lub sumę zbiorów
Except	Operator zwracający różnicę zbiorów
AsEnumerable	Operator konwersji do kolekcji typu **IEnumerable<T>**
ToArray/ToList	Operator konwersji do typu **array** lub **List<T>**
ToDictionary/ToLookup	Operatory konwersji do kolekcji typu **Dictionary<K,T>** lub **Lookup<K,T>**
OfType/Cast	Operator konwersji do kolekcji typu **IEnumerable<T>** elementów o wskazanym typie
SequenceEqual	Operator sprawdzający równość par elementów dwóch sekwencji
First/FirstOrDefault/Last/LastOrDefault/Single/ SingleOrDefault	Operator zwracający odpowiednio pierwszy/ostatni/pojedynczy element
ElementAt/ElementAtOrDefault	Operatory zwracające element na wskazanej pozycji
DefaultIfEmpty	Operator zastępujący pustą sekwencję, sekwencją domyślną
Range	Operator generujący zakres liczb
Repeat	Operator generujący wielokrotne wystąpienia podanej wartości
Empty	Operator zwracający pustą sekwencję

Any/All	Kwantyfikator sprawdzający czy jakikolwiek/wszystkie elementy spełniają warunek
Contains	Kwantyfikator sprawdzający obecność element
Count/LongCount	Operatory agregujące zwracające liczność
Sum/Min/Max/Average	Operator sumy/najmniejszego/ największego/średniej elementów

Tabela 1 – Operatory LINQ

3.4.1. LINQ to SQL

Przy pomocy LINQ można także tworzyć zapytania do relacyjnych baz danych. Służy do tego technologia LINQ to SQL. Jest to infrastruktura odpowiedzialna za zarządzanie relacyjnymi danymi. Definiuje ona przede wszystkim dwa główne atrybuty: **[Table]** oraz **[Column]** wskazujące, jakie typy i właściwości odpowiadają zewnętrznym danym. Pierwszy z nich może być przypisany do klasy i w ten sposób wskazuje tabelę SQL, z którą CLR musi połączyć dany typ. Drugi natomiast służy do przypisania polom lub atrybutom klas odpowiadających im kolumn w bazie danych. Sposób przypisania prezentuje Przykład 5**.**

```
[Table(Name="Customers")]
public class Customer
{
   [Column(IsPrimaryKey=true)]
   public string CustomerID;
   [Column]
   public string City;
}
```

Przykład 7 – Wykorzystanie atrybutów [Table] i [Column]

Atrybut **[Table]** posiada właściwość ***Name***, która jednoznacznie określa tabelę w bazie danych. W przypadku niepodania tej właściwości CLR automatycznie założy, że nazwa klasy odnosi się do nazwy tabeli. Definiując poszczególne pola lub właściwości klas przy pomocy atrybutu **[Column]** można dokładnie określić przechowywane dane rozszerzając atrybut o dodatkowe właściwości. Wszystkie właściwości atrybutu **[Column]** prezentuje Tabela 2.

Właściwość	**Typ**	**Opis**
Name	String	Nazwa kolumny w tabeli. Jeśli nie będzie podana, nazwa właściwości lub pola klasy zostanie użyta jako domyślna
Storage	String	W przypadku oznaczenia właściwości klasy atrybutem **[Column]** właściwość ***Storage*** wskazuje na pole, które przechowuje wartość w obiekcie
DBType	String	Określa typ kolumny w bazie danych. Jako wartość występuje dokładnie ta sama wartość, która służy do określenia kolumn w systemie bazy danych np. `DBType="int NOT NULL IDENTITY"`
IsPrimaryKey	Bool	Ustawiony na **true** jeśli opisywane pole reprezentuje kolumnę będącą kluczem głównym lub jego składową
IsDbGenerated	Boolean	Oznacza, że wartość kolumny reprezentowanej przez dane pole lub właściwość klasy jest automatycznie generowana (zazwyczaj auto-inkrementacja). W przypadku gdy dany element klasy odpowiada kolumnie z kluczem głównym oraz jej wartość jest generowana należy ustawić właściwość ***DBType*** aby posiadała modyfikator **IDENTITY**
IsVersion	Boolean	Identyfikuje kolumnę reprezentowaną przez element klasy jako timestamp lub jako numer wersji. Wersje są zwiększane o 1 natomiast timestamp'y są modyfikowane

		przez bazę danych po zapisaniu krotki
UpdateCheck	UpdateCheck	Określa w jaki sposób radzić sobie ze współbieżnością. LINQ to SQL zakłada, że w przypadku gdy aplikacja chce modyfikować daną a od czasu pobrania ich uległy one zmianie w bazie danych, modyfikacje należy odrzucić. W przypadku braku oznaczenia jakiegokolwiek elementu klasy właściwością ***IsVersion = true*** sprawdzanie odbywa się poprzez porównanie wartości z aktualnego stanu bazy danych z danymi posiadanymi przez aplikację. Kontrola pojedynczych kolumn odbywa się poprzez nadanie właściwości ***UpdateCheck*** wartości: • **Always –** zawsze sprawdza daną kolumnę • **Never –** nigdy nie sprawdza kolumny • **WhenChanged –** sprawdza kolumnę tylko w przypadku gdy wartość została zmieniona przez aplikację
IsDiscriminator	Boolean	W przypadku dziedziczenia klasy właściwość ta określa pole, po którym następuje determinacja typu w hierarchii dziedziczenia
Expression	String	Nie ma wpływu na zapytania LINQ to SQL. Wykorzystywane tylko przy użyciu metody ***CreateDatabase*** aby określić wyrażenie jakie ma być używane do wyliczania kolumny
CanBeNull	Boolean	Określa czy wartość może być ***null*** . przede wszystkim należy użyć tej własności aby jednoznacznie określić, że wartość nie może być ***null***.
AutoSync	AutoSync	Możliwe wartości to **OnInsert**, **Always**, and **Never**. Definiuje czy kolumna jest automatycznie synchronizowana przez bazę danych po dodaniu lub

		edycji danych np. `AutoSync=AutoSync.OnInsert`

Tabela 2 – Właściwości atrybutu Column

Tabele w relacyjnych bazach danych często są połączone z innymi przy pomocy kluczy: głównego i obcego. Takie przyporządkowania są opisane przez obiekty za pomocą kolekcji referencji. Aby odwzorować zależności między tabelami w obiektach obsługiwanych przez LINQ to SQL, technologia ta wprowadza atrybut **Association**. Można go stosować dla właściwości klasy odpowiedzialnej za zachowanie relacji w obiektach. Tak opisana asocjacja odpowiada relacji klucz główny – klucz obcy uzyskiwanej przez połączenie kolumn pomiędzy tabelami w relacyjnej bazie danych. W celu dokładnego opisania relacji atrybut **Association** posiada dodatkowe własności przedstawione i omówione w Tabeli 3.

Właściwość	**Typ**	**Opis**
Name	String	Nazwa relacji. Używana do rozróżnienia wielu relacji w ramach pojedynczej klasy reprezentującej encję. W związku z tym po obu „końcach" relacji musi mieć taką samą nazwę.
Storage	String	Nazwa pola przechowującego informację o relacji. Zaleca się aby asocjacje opisywane były za pomocą właściwości klas z dodatkowym polem klasy zdefiniowanym we właściwości ***Storage***. W takim przypadku informuje on LINQ to SQL w jaki sposób pominąć właściwość klasy i operować bezpośrednio na odpowiadającym polu. Jeśli nie będzie podany LINQ to SQL odczytuje i zapisuje wartości używając właściwości klasy odpowiadającej za relację.
ThisKey	String	Lista oddzielonych przecinkiem nazw jednej lub wielu właściwości klasy, które reprezentują klucz identyfikujący dany „koniec" relacji. Jeśli nie zostanie podany kluczem są

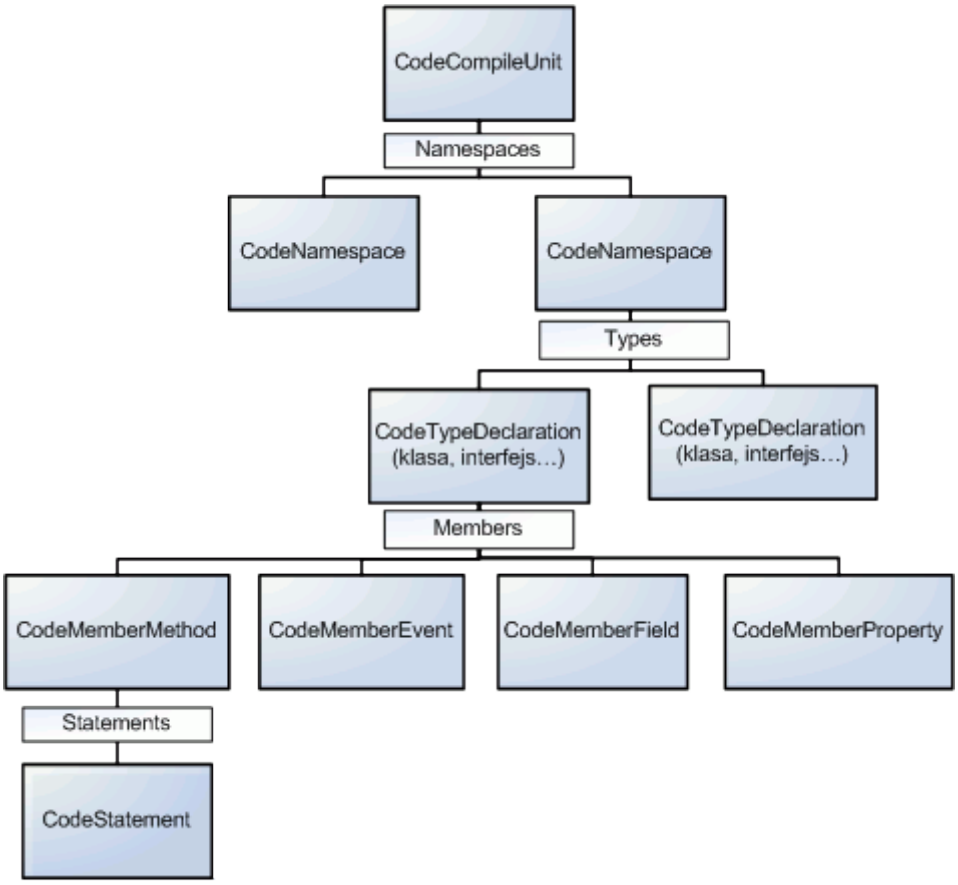
		właściwości tworzące klucz główny.
OtherKey	String	Lista oddzielonych przecinkiem nazw jednej lub wielu właściwości klasy, które reprezentują klucz identyfikujący drugi „koniec” relacji. Jeśli nie zostanie podany kluczem są właściwości tworzące klucz główny klasy będącej w relacji.
IsUnique	Boolean	**True** dla relacji 1:1 (jeden do jeden). Z powodów problemów w zarządzaniu takimi relacjami przez bazy danych właściwość ta jest bardzo rzadko stosowana. Często stosuje się relacje 1:n (jeden do wielu) nawet jeśli odwzorowanie ma być 1:1.
IsForeignKey	Boolean	Określa czy pole odnosi się do kolumny w bazie danych, która jest kluczem obcym. Jeśli tak to wartość powinna być **True**
DeleteRule	String	Definiuje zachowanie podczas usuwania elementów w relacji np. ***CASCADE*** usuwa kaskadowo elementy. Ustawienie ***null*** nie definiuje dodatkowego zachowania.

Tabela 3 – Właściwości atrybutu Association

3.5. CodeDOM

CodeDOM (Code Document Object Model) jest mechanizmem zawartym w .NET Framework, który umożliwia tworzenie aplikacji zdolnych do generowania kodu źródłowego w trakcie uruchomienia. Powstaje na bazie odpowiedniego modelu reprezentującego kolejne elementy, takie jak klasa, przestrzeń nazw, pola i metody itd. Model taki znajduje się w strukturze zwanej CodeDOM Graph, która przechowuje połączenia pomiędzy poszczególnymi elementami kodu. CodeDOM Graph ma strukturę drzewa zawierającego kontenery (Rysunek 9), z któryvh njważniejszy to **CodeCompileUnit**. Każdy kolejny element kodu źródłowego musi zostać dołączony do drzewa.

Możliwości mechanizmu nie ograniczają się tylko do generowania kodu. Istnieje także możliwość kompilacji kodu źródłowego.



Rysunek 9 – Graf kontenerów CodeDOM[6]

CodeDOM domyślnie posiada obsługę języków dostępnych w SDK:

- C#,
- Visual Basic,
- C++,
- J#,
- JScript.

Aby rozszerzyć możliwości o kolejne języki należy zaimplementować interfejs odpowiedzialny za generowanie kodu (**ICodeGenerator**) oraz interfejs odpowiedzialny za kompilacje kodu (**ICodeCompiler**), a także dostarczyć odpowiedni **CodeDomProvider** rozszerzający mechanizm o nowy język.

3.6. Visual Studio Automation

Pomimo że Visual Studio jest środowiskiem programistycznym cenionym za ilość dostępnych możliwości, posiada dodatkowe mechanizmy rozbudowy. Odpowiednie biblioteki, dające dostęp do praktycznie każdej części IDE (Integrated Development Environment) są udostępniane za darmo wraz z SDK. W zależności od efektu, który programista chce uzyskać, wyróżniono trzy poziomy automatyzacji Visual Studio:

- makra
- wtyczki (add-ins), kreatorzy (wizards)
- aplikacje oparte o Visual Studio SDK.

Makra to najprostsza metoda rozbudowy środowiska. Służą głównie do automatycznego wykonywania zadań np. wpisywania powtarzalnego kodu. Tworzenie ich polega na „nagraniu" kolejno wykonywanych czynności, a następnie odtwarzaniu ich. Do tworzenia makr jest przygotowane środowisko programistyczne (Macros IDE), które ułatwia pisanie funkcjonalności. Makra można kodować jedynie w języku Visual Basic. Ze względu na to, że są najprostszą formą rozbudowy IDE, posiadają pewne ograniczenia. W związku z tym, mogą być stosowane tylko jeśli nie będą zależne od danych wejściowych lub będą one proste.

Wtyczki i kreatorzy są aplikacjami, które integrują się z IDE i w ten sposób mogą być wywoływane przez użytkownika. Mają większe możliwości niż makra, ale jednocześnie tworzenie ich jest bardziej skomplikowane. Są to kompilowane obiekty COM (Component Object Model), co powoduje, że ich dystrybucja jest trudniejsza (konieczność przygotowania instalatorów), ale jednocześnie chroniona jest własność intelektualna.

Możliwości, które daje korzystanie z wtyczek to m.in.:

- dostęp do funkcjonalności narzędzia poprzez menu
- możliwość tworzenia opcji do przygotowywanych narzędzi
- możliwość przygotowania okien w tym także okien narzędziowych.

Kreatorzy z kolei są aplikacjami, które prowadzą użytkownika przez proces tworzenia programów lub kończenia zadań programistycznych. Najczęściej są wykorzystywani do tworzenia szablonów aplikacji jak np. kreator aplikacji instalacyjnej (przez polecenia **Nowy projekt** i **Nowy plik**).

Najbardziej zaawansowaną częścią rozbudowy Visual Studio są aplikacje korzystające z SDK. Mają one największe możliwości kontroli IDE i zazwyczaj są stosowane przy tworzeniu nowych edytorów lub dodawaniu nowych języków programowania.

4. Mapowanie relacyjno-obiektowe

Nowoczesne aplikacje w większości wypadków są zależne od zewnętrznych źródeł danych (najczęściej relacyjnych baz danych). W związku z taką architekturą, warstwa ta jest podatna na błędy. Odpowiedzią na to jest oprogramowanie ułatwiające dostęp do danych przechowywanych w formie relacyjnej oraz ich transformację na obiekty dostępne w językach programowania.

4.1. Pobieranie meta-danych

Większość rozwiązań oferujących funkcjonalność mapowania relacyjno-obiektowego opiera się na konfiguracji dostarczanej przez użytkownika. Powoduje to powstanie kolejnej warstwy aplikacji, w której mogą wystąpić błędy.

Uniknięcie tworzenia rozbudowanych plików konfiguracyjnych umożliwia stosowanie podejścia „konwencja ponad konfiguracją". Oznacza to, że użytkownik zobowiązany jest do przestrzegania pewnych reguł (np. sposobu nazewnictwa), jednak w zamian dostaje zautomatyzowane rozwiązanie, niewymagające tworzenia dodatkowych plików.

Dzięki stosowaniu konwencji możliwe jest pobranie i poprawna analiza meta-danych, czyli informacji opisujących budowę tabel, kolumn i zależności między nimi. Wiedza taka pozwala na budowanie generycznych rozwiązań bez potrzeby tworzenia i opierania się na konfiguracji dostarczanej przez użytkownika. Analiza opisu bazy danych dostarcza informacji o budowie klasy i zależnościach między obiektami w oprogramowaniu mapującym schemat relacyjny na obiektowy.

4.2. Generowanie kodu dostępu do danych

Aplikacje mapujące nie powinny ograniczać się tylko do określenia zależności pomiędzy tabelami w bazach danych a obiektami w językach programowania. Aby można było w pełni korzystać z takich rozwiązań należy także automatycznie utworzyć odpowiadające encjom klasy.

Posiadając odpowiednie meta-dane aplikacja w łatwy sposób może wygenerować odpowiednie klasy. Zautomatyzowanie tego procesu umożliwia wyeliminowanie błędów popełnianych przez programistów.

Automatycznie generowany kod jest przydatny dla programistów projektujących aplikację, jednak zdarzają się sytuacje, w których ze względu na przyjętą architekturę, należy rozbudować utworzone klasy. Aplikacja powinna zawierać mechanizmy umożliwiające modyfikacje bez ingerencji w automatycznie generowany kod. Należy pamiętać także o możliwości ponownego generowania klas np. po modyfikacji schematu bazy danych bez utraty funkcjonalności, o którą klasy zostały rozbudowane.

Ważne jest, aby aplikacja mapująca nie tworzyła osobnego bytu. W tak rozbudowanym środowisku programistycznym jak Visual Studio programiści przyzwyczajeni są do narzędzi zintegrowanych z IDE. Przy możliwościach rozbudowy Visual Studio należy postarać się, aby funkcjonalność była dostępna za pomocą kilku kliknięć wewnątrz środowiska.

4.3. Mapowanie encji

Wzorzec Active Record jest jednym z najprostszych wzorców opisujących dostęp do relacyjnych danych. Zakłada on, że poszczególne tabele bazy danych są mapowane jeden do jednego na klasy w obiektowym języku programowania. Dzięki takim założeniom możliwe jest wygenerowanie klas na podstawie otrzymanych meta-danych. Daje to także programiście swobodę w operowaniu na uzyskanych informacjach. Przy zachowaniu zależności i relacji zaprojektowanych w bazie programista korzysta z obiektowości w języku programowania.

Aplikacje takie powinny też w sposób czytelny radzić sobie z encjami znajdujących się we wzajemnych relacjach. W takich wypadkach zamiast uzyskiwać klucz obcy do powiązanych danych z innej tabeli, programiści powinni mieć możliwość operowania bezpośrednio na odpowiadających im obiektach.

4.4. Mapowanie relacji

Informacje w relacyjnych bazach danych są od siebie zależne, co powinno być odpowiednio zobrazowane także w sposób obiektowy. Można to osiągnąć poprzez przechowywanie kluczy obcych do poszczególnych obiektów, jednak aby usprawnić pracę z danymi nowoczesne aplikacje mapujące powinny móc operować bezpośrednio na powiązanych obiektach.

4.4.1. Relacje jeden-do-jeden oraz jeden-do-wiele

W bazach danych relacje jeden-do-jeden są zazwyczaj reprezentowane w postaci jeden-do-wiele, dlatego rozwiązania automatycznie analizujące meta-dane o relacjach utożsamiają te dwa rodzaje zależności.

Relacje takie w językach programowania są odwzorowywane za pomocą referencji do obiektów. W przypadku relacji jeden-do-jeden podobnie jak w bazach danych programiści muszą zapewnić odpowiednie ograniczenia.

4.4.2. Relacje wiele-do-wiele

Relacja wiele-do-wiele jest trudniejsza do odwzorowania. O ile w środowisku obiektowym jest to możliwe, dzięki kolekcjom mogących przechowywać zależne obiekty, o tyle w relacyjnych bazach danych potrzebna jest dodatkowa tabela. Przy użyciu wzorca Active Record tabela ta musi także być odwzorowana, co prowadzi do pewnych niezgodności. W tym wypadku powstają dwie relacje jeden-do-wiele, a programista zamiast odwoływać się bezpośrednio do obiektów zależnych, uzyskuje do nich dostęp pośrednio przez dodatkową klasę. Takie podejście nie jest w pełni obiektowe dlatego generowane klasy powinny posiadać mechanizmy obsługujące

relacje wiele-do-wiele, w których odwołania do tabeli asocjacyjnej będą ukryte przed programistą.

Problematyczne staje się także usuwanie takiej relacji. O ile w przypadku jeden-do-wiele relacja jest usuwana wraz z pozbyciem się obiektu lub przypisaniem referencji do innego obiektu, o tyle specyfika zależności wiele-do-wiele polega na tym, że stan relacji jest zapisywany w dodatkowej tabeli. Autorzy projektujący rozwiązania mapujące muszą z góry określić, czy w przypadku wywołań:

```
obiekt1.obiekt2.remove();
```

należy usunąć obiekt2, czy usunąć relację łączącą oba obiekty.

4.5. Mapowanie typów danych

Każda informacja, zarówno w bazie, jak i w obiektach przechowywana jest przez odpowiednią strukturę. Bazy danych oraz języki programowania mają zaimplementowane własne typy, które różnią się między sobą. Poprawna konwersja umożliwia płynne przenoszenie informacji z pamięci (obiektów) na nośnik fizyczny (do bazy danych).

Praca aplikacji mapujących zależy także od dobrego identyfikowania i mapowania typów danych. Tabela 4 pokazuje typy danych dostępne w MS SQL 2005 oraz odpowiadające im struktury w CLR wykorzystywane w aplikacjach opartych na .NET Framework.

Typy danych w MS SQL	**Typy danych w .NET CLR**
bit, tinyint, smallint, int, bigint	Int16, Uint16, Int32, Uint32, Int64, Uint64
Bit	Boolean
decimal, numeric, smallmoney, money	Decimal
real, float	Single, Double

char, varchar, text, nchar, nvarchar, ntext	String
datetime, smalldatetime	DateTime
uniqueidentifier	Guid
timestamp	Byte[] (Byte() in Visual Basic), Binary
binary, varbinary	Byte[] (Byte() in Visual Basic), Binary

Tabela 4 – Typy danych w MS SQL i ich odpowiedniki w .NET CLR

4.6. Logika związana z wzorcem Active Record

Wzorzec Active Record definiuje część funkcjonalności, która powinna zostać dostarczona wraz z klasami odnoszącymi się do tabel. Aby spełnić te wymagania, aplikacje bazujące na tym modelu powinny zawierać następujące metody:

- read – do uzyskiwania instancji na podstawie pobranych danych SQL
- konstruktor – umożliwiający tworzenie nowych obiektów w kodzie
- save – do zapisywania obiektów w bazie danych lub ich edycji
- publiczne właściwości – do łatwego odczytu i ustawiania poszczególnych wartości.

5. Prototyp

5.1. Założenia

Autor opracował prototyp aplikacji mapującej dostępnej w postaci wtyczki dla środowiska Visual Studio 2008. Prototyp generuje w języku C# klasy mapujące encje i relacje, zgodnie z zaleceniami dotyczącymi wzorca Active Record.

5.2. Wtyczka do Visual Studio

5.2.1. Budowa wtyczki

Wtyczki do Visual Studio są skompilowanymi aplikacjami integrującymi się z IDE. Aby utworzyć wtyczkę, potrzebna jest implementacja głównej klasy ***Connect*** rozszerzonej o interfejsy ***IDEExtensibility2*** z przestrzeni nazw ***Extensibility*** oraz ***IDTCommandTarget*** z przestrzeni ***EnvDTE*** (Rysunek 10).

Interfejs ***IDEExtensibility2*** zawiera metody służące do komunikacji z projektowaną wtyczką. Visual Studio wywołuje je za każdym razem gdy nastąpi dotyczące jej zdarzenie. Najważniejszą spośród nich jest metoda ***onConnection*** używana, gdy Add-in jest ładowany. Można w niej umieścić dodatkową funkcjonalność np. tworzenie opcji i konfiguracji. Przygotowana wtyczka jest inicjalizowana wraz z instancją Visual Studio w tym przypadku metoda ***onConnection*** została wykorzystana do umieszczenia opcji „*ORemedy*” w zakładce *Tools* w pasku poleceń (Przykład 8).

```
public void OnConnection(object application, ext_ConnectMode
connectMode, object addInInst, ref Array custom)
      {
            _applicationObject = (DTE2)application;
            _addInInstance = (AddIn)addInInst;
            if(connectMode == ext_ConnectMode.ext_cm_UISetup)
            {
                  object []contextGUIDS = new object[] { };
```

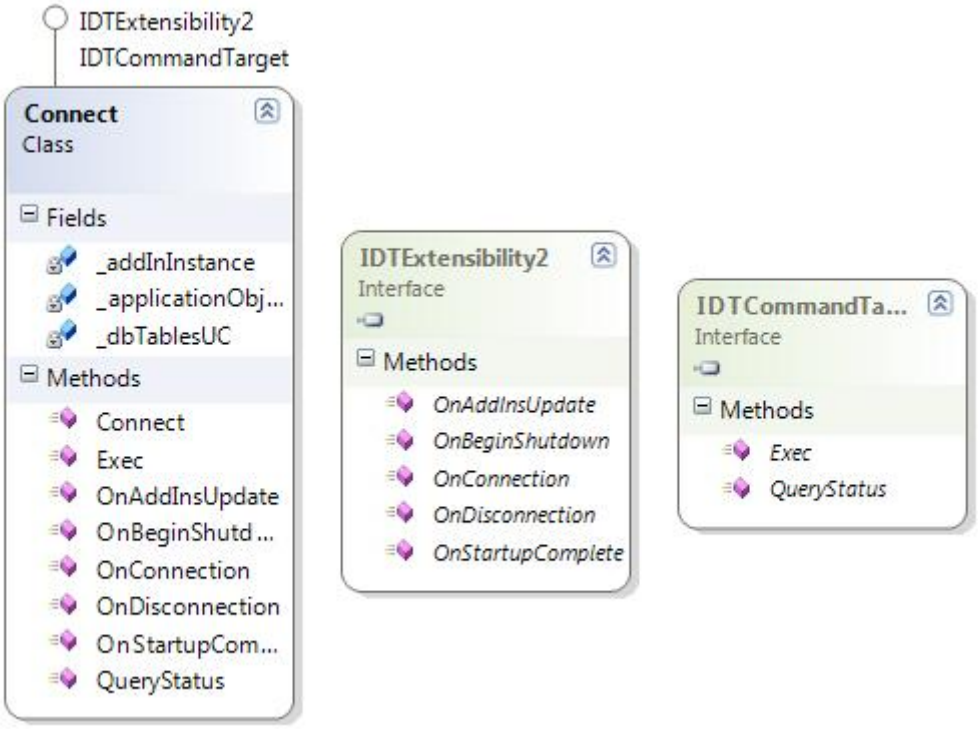
```
                    Commands2 commands =
(Commands2)_applicationObject.Commands;
                    string toolsMenuName;

                    try
                    {
                          ResourceManager resourceManager = new
ResourceManager("ORemedy.CommandBar",
Assembly.GetExecutingAssembly());
                          CultureInfo cultureInfo = new
System.Globalization.CultureInfo(_applicationObject.LocaleID);
                          string resourceName =
String.Concat(cultureInfo.TwoLetterISOLanguageName, "Tools");
                          toolsMenuName =
resourceManager.GetString(resourceName);
                    }
                    catch
                    {
                          toolsMenuName = "Tools";
                    }
                    Microsoft.VisualStudio.CommandBars.CommandBar
menuBarCommandBar =
((Microsoft.VisualStudio.CommandBars.CommandBars)_applicationObject.Co
mmandBars)["MenuBar"];
                    CommandBarControl toolsControl =
menuBarCommandBar.Controls[toolsMenuName];
                    CommandBarPopup toolsPopup =
(CommandBarPopup)toolsControl;
                    try
                    {
                          Command command =
commands.AddNamedCommand2(_addInInstance, "ORemedy", "ORemedy",
"Executes the command for ORemedy", true, 59, ref contextGUIDS,
(int)vsCommandStatus.vsCommandStatusSupported+(int)vsCommandStatus.vsC
ommandStatusEnabled, (int)vsCommandStyle.vsCommandStylePictAndText,
vsCommandControlType.vsCommandControlTypeButton);

                    if((command != null) && (toolsPopup != null))
                    {
```

```
                        command.AddControl(toolsPopup.CommandBar, 1);
                    }
                }
                catch(System.ArgumentException)
                {}
            }
}
```

Przykład 8 – Implementacja metody *onConnect*

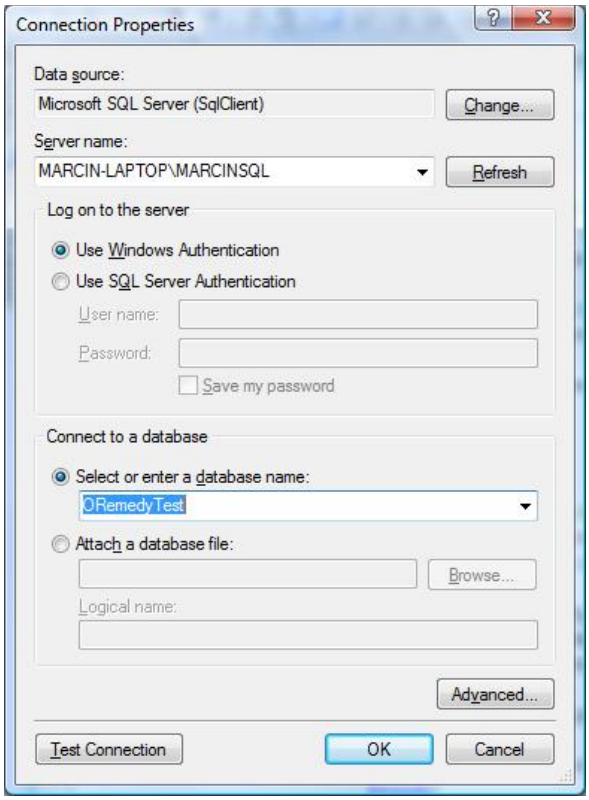


Rysunek 10 – Klasa Connect i rozszerzające ją interfejsy

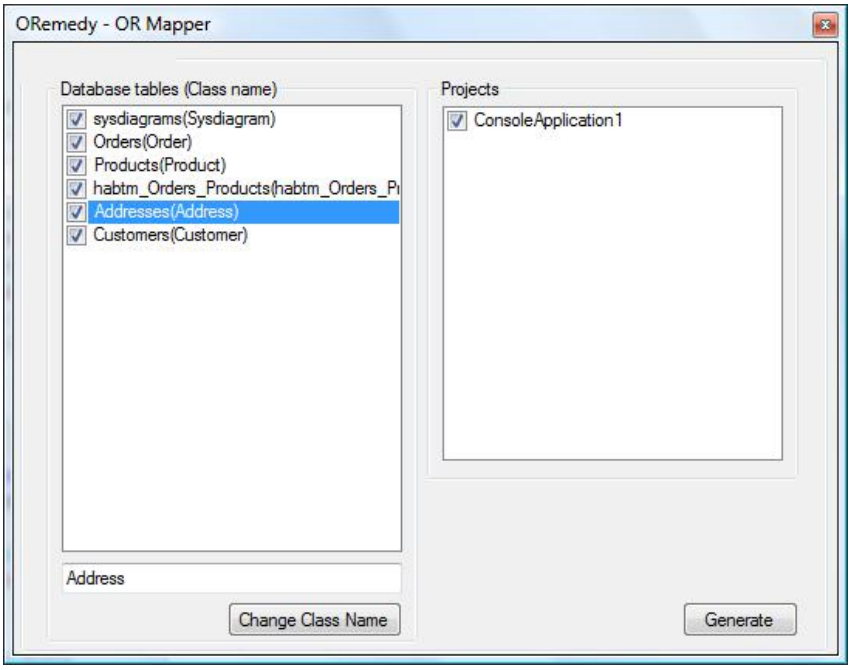
Aby możliwe było wywołanie odpowiednich funkcjonalności wtyczki po wybraniu opcji z menu górnego, niezbędna jest implementacja metody ***Exec*** z interfejsu ***IDTCommandTarget***. Służy on do tworzenia „nazwanych poleceń", które przypisane są do odpowiednich opcji. Uruchomienie ich powoduje wywołanie metody ***Exec***, w której znajduje się obsługa polecenia. W prototypie metoda ta uruchamia interfejs graficzny wtyczki, jeżeli otwarty jest co najmniej jeden projekt.

5.2.2. Interfejs graficzny

Interfejs graficzny wtyczki pozwala użytkownikowi na kontrolowanie aplikacji mapującej. Po wybraniu opcji *ORemedy* w menu górnym otwierany jest ekran wyboru źródła danych (Rysunek 11). Programista ma możliwość wyboru tabel, które chce odwzorować oraz projektów (wybór ograniczony do projektów otwartych), do których mają być dołączone automatycznie wygenerowane pliki klas (Rysunek 12). Aplikacja konwertuje angielskie nazwy tabel w liczbie mnogiej na liczbę pojedyńczą, w której występuje nazwa klasy. Istnieje możliwość ręcznego ustawienia jej, w polu poniżej listy dostępnych tabel.



Rysunek 11 – Interfejs graficzny (wybór bazy danych)



Rysunek 12 – Interfejs graficzny (wybór tabel i projektów)

W celu wyświetlenia okna z interfejsem użytkownika należy przygotować odpowiednią kontrolkę, która nie musi się znajdować wewnątrz projektu tworzonej wtyczki. Może być to oddzielne assembly dołączone do projektu. Kolekcją wszystkich okien środowiska zarządza interfejs ***Windows2*** znajdujący się w przestrzeni nazw ***EnvDTE80***. Przy pomocy metody ***CreateToolWindow2***, która jako parametry przyjmuje między innymi referencję do assembly oraz ścieżkę do klasy zawierającej graficzną kontrolkę użytkownika, tworzony jest nowy obiekt ***Window2*** i dodawany do kolekcji (Przykład 9).

```
string ctrlProgID, guidStr;
EnvDTE80.Windows2 toolWins;
object objTemp = null;

ctrlProgID = "ORemedy.UserControls.EncjeUC";
guidStr = "{2C73C576-6153-a42d-82FE-9D54F4B6AD09}";

System.Reflection.Assembly asm =
      System.Reflection.Assembly.GetExecutingAssembly();
```

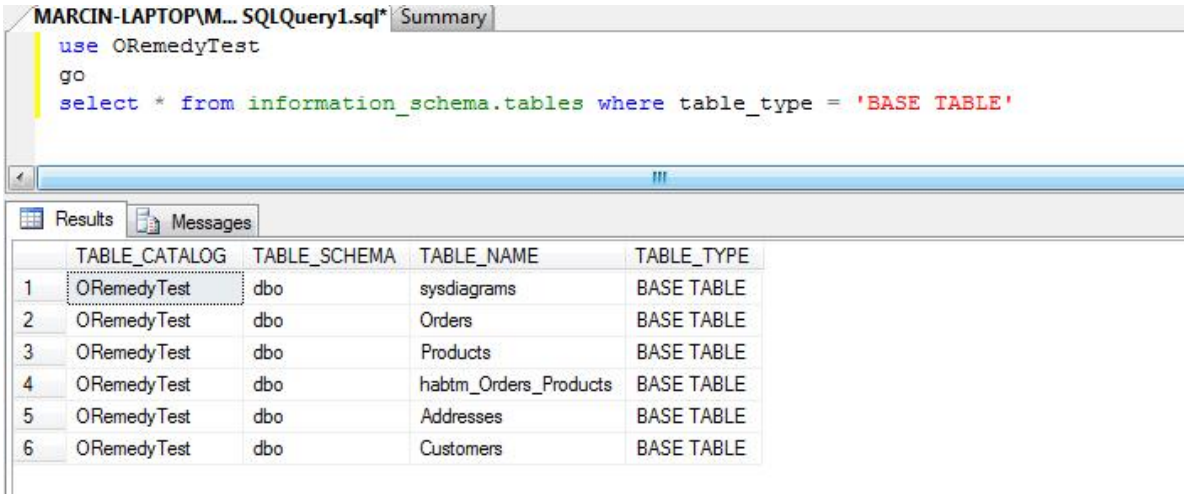
```
toolWins = (Windows2)_applicationObject.Windows;
_dbTablesUC = toolWins.CreateToolWindow2(
      _addInInstance,
      asm.Location,
      ctrlProgID,
      "ORemedy - OR Mapper",
      guidStr, ref objTemp);
```

Przykład 9 – Fragment metody *Exec* tworzący okno interfejsu użytkownika

5.3. Mapowanie bazy danych

Aby aplikacja mogła poprawnie zbudować i wygenerować klasy niezbędne jest zebranie meta-danych o wybranej bazie danych. W tym celu prototyp wykorzystuje tabelę **INFORMATION_SCHEMA.**

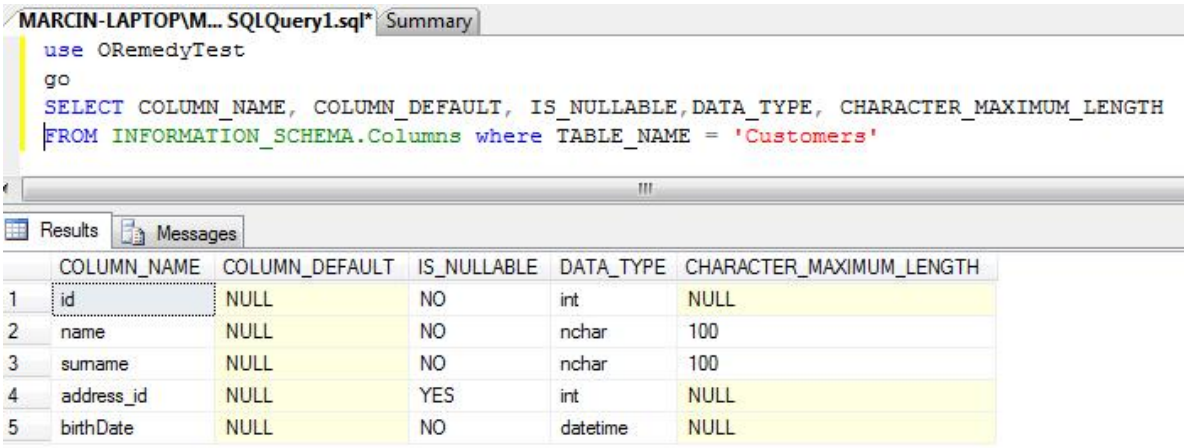
Pierwszym etapem po wybraniu źródła danych jest pobranie informacji o tabelach, które będą odwzorowane w postaci klas. Przykładowe zapytanie SQL wraz z wynikiem prezentuje Rysunek 13.



	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE
1	ORemedyTest	dbo	sysdiagrams	BASE TABLE
2	ORemedyTest	dbo	Orders	BASE TABLE
3	ORemedyTest	dbo	Products	BASE TABLE
4	ORemedyTest	dbo	habtm_Orders_Products	BASE TABLE
5	ORemedyTest	dbo	Addresses	BASE TABLE
6	ORemedyTest	dbo	Customers	BASE TABLE

Rysunek 13 – zapytanie SQL pobierające informacje o tabelach bazy danych i jego wynik

Dla każdej wybranej przez użytkownika tabeli pobierane są szczegółowe metadane. Wszystkie zebrane informacje przechowywane są w obiektach ***TableModel***. Dane o poszczególnych kolumnach bazy danych będą odwzorowane na pola wygenerowanej klasy. Lista pól oraz informacje o domyślnej wartości, możliwości wstawienia ***null***, relacyjnym typie danych i długości pola przechowywane są w kolekcji ***Hashtable***. Rysunek 14 prezentuje otrzymane dane dla Tabeli *Customers* przykładowej bazy danych.



	COLUMN_NAME	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	CHARACTER_MAXIMUM_LENGTH
1	id	NULL	NO	int	NULL
2	name	NULL	NO	nchar	100
3	surname	NULL	NO	nchar	100
4	address_id	NULL	YES	int	NULL
5	birthDate	NULL	NO	datetime	NULL

Rysunek 14 – Informacje o kolumnach należących do tabeli *Customers*

Przechowywanie relacji zachodzących pomiędzy tabelami odbywa się za pomocą dwóch list – jednej dla relacji jeden-do-wiele i drugiej dla relacji wiele-do-wiele. Obie listy przechowują kolekcje ***Hashtable*** z informacjami dotyczącymi zależności. Rozróżnienie relacji wiele-do-wiele odbywa się na podstawie przedrostka ***habtm_*** (od pierwszych liter potocznego określenia tego typu relacji jako ***has and belongs to many***) w nazwie tabeli.

Dla każdej encji wykonywane jest zapytanie zwracające relacje jeden-do-wiele (Przykład 10).

```
Select a.table_name as [FK_TABLE],
      a.Column_name as [FK_Column],
      c.TABLE_name [PK_TABLE],
      c.column_name AS [PK_Column],
      d.DATA_TYPE as [DATA_TYPE]
from INFORMATION_SCHEMA.Columns d
      join INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE a
            on a.column_name = d.column_name
      join INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS b
            on a.Constraint_Schema=b.Constraint_Schema and
            a.Constraint_Name=b.Constraint_Name
      join INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE c
            on c.Constraint_Schema=b.Unique_Constraint_Schema and
            c.Constraint_Name=b.Unique_Constraint_Name
where a.table_name = 'Orders' or c.table_name = 'Orders'
```

Przykład 10 – Zapytanie SQL zwracające dane o relacji jeden-do-wiele

W przypadku wykrycia relacji z tabelą o nazwie z przedrostkiem ***habtm_***, pobierane są dane dotyczące relacji wiele-do-wiele i zapisywane są w drugiej liście. Otrzymane informacje dla tabeli ***Orders*** na temat relacji jeden-do-wiele (z encją ***Customers***) oraz relacji wiele-do-wiele (z encją ***Products***) prezentuje Rysunek 15.

Results | Messages

	FK_TABLE	FK_Column	PK_TABLE	PK_Column	DATA_TYPE
1	Orders	person_id	Customers	id	int
2	habtm_Orders_Products	order_id	Orders	id	int

	FK_TABLE	FK_Column	PK_TABLE	PK_Column	DATA_TYPE
1	habtm_Orders_Products	order_id	Orders	id	int
2	habtm_Orders_Products	product_id	Products	id	int

Rysunek 15 – Informacje na temat relacji uzyskiwane przez zapytanie SQL

5.4. Generowanie kodu klas mapujących

Zebranie meta-danych pozwala na wygenerowanie klas odpowiadających encjom w bazie danych. Podstawą do utworzenia plików jest graf odpowiadający za ich strukturę. Nadrzędnym kontenerem jest obiekt ***CodeCompileUnit*** znajdujący się w przestrzeni nazw ***System.CodeDOM.*** Do niego dodawane są kolejne obiekty odwzorowujące odpowiednie elementy pliku. Są to:

- obiekt reprezentujący deklarację przestrzeni nazw dodawany do właściwości ***Namespaces*** obiektu ***CodeCompileUnit*** – ***CodeNamespace***
- obiekty reprezentujące dyrektywę importu zewnętrznych przestrzeni nazw – ***CodeNamespaceImport*** (dodawane do kolekcji ***Imports*** obiektu ***CodeNamespace***)
- obiekt reprezentujący deklarację klasy – ***CodeTypeDeclaration***
- obiekty pola klasy – ***CodeMemberField***
- obiekty konstruktora klasy – ***CodeConstructor***
- obiekty właściwości klasy – ***CodeMemberProperty***
- obiekty reprezentujące metody klasy – ***CodeMemberMethod***

Obiekty te stanowią dodatkowe kontenery przechowywujące definicje poszczególnych elementów. Zbudowane są one w oparciu o graf, a dołączanie kolejnych części pliku odbywa się poprzez tworzenie liści. Przykład 11 pokazuje sposób budowania metody w klasie w oparciu o mechanizm ***CodeDOM*** oraz dołączania jej do kontenera ***CodeTypeDeclaration.***

```
private void generateReadPageMethod(CodeTypeDeclaration dcl)
{
      CodeMemberMethod readPage = new CodeMemberMethod();
      readPage.Attributes = (readPage.Attributes &
            ~MemberAttributes.AccessMask) | MemberAttributes.Public;
      readPage.Attributes = (readPage.Attributes &
            ~MemberAttributes.ScopeMask) | MemberAttributes.Static;
      readPage.Name = "readPage";
```

```
        readPage.ReturnType =
              new CodeTypeReference("List<"+model.ClassName+">");
        readPage.Parameters.Add(
              new CodeParameterDeclarationExpression(typeof(int),
                    "pageNumber"));
        readPage.Parameters.Add(
        new CodeParameterDeclarationExpression(typeof(int),
              recordNumber"));
        readPage.Parameters.Add(
              new CodeParameterDeclarationExpression(
                    new CodeTypeReference(model.DbName), "db"));
        readPage.Statements.Add(
              new CodeVariableDeclarationStatement(
                    new CodeTypeReference("var"), "query",
                    new CodeSnippetExpression(@"(from c in db." +
                    model.TableName + " select
                    c).Skip((pageNumber-1)*recordNumber).Take
                    (recordNumber)")));
        readPage.Statements.Add(
              new CodeMethodReturnStatement(
                    new CodeMethodInvokeExpression(
                          new CodeVariableReferenceExpression("query"),
                          "ToList<" + model.ClassName + ">",
                          new CodeExpression[] { }
                    )
              ));
        dcl.Members.Add(readPage);
}
```

Przykład 11- Budowanie metody za pomocą mechanizmu *CodeDOM*

W celu wygenerowania pliku języka C# należy utworzyć obiekt abstrakcyjnej klasy ***CodeDomProvider.*** Implementacje tej klasy zawierają interfejsy generatoratorów i kompilatorów kodu. .NET Framework posiada implementacje swoich podstawowych języków w tym C#. W celu utworzenia obiektu odpowiedzialnego za generowanie kodu w tym właśnie języku trzeba użyć metody ***CreateProvider(„CSharp”)***. Uruchomienie przygotowanego generatora następuje poprzez wywołanie

GenerateCodeFromCompileUnit. Jako parametry przyjmuje ona obiekt ***CodeCompileUnit*** (zawierający graf programu), obiekt ***TextWriter*** (służący do tworzenia i zapisu plików na dysku) oraz obiekt ***CodeGeneratorOptions*** umożliwiający sterowanie generatorem. W projekcie została utworzona statyczna klasa generująca kod (Przykład 12). Została ona wyposażona w mechanizm umożliwiający w przyszłości rozbudowę aplikacji o kolejne języki dostępne bezpośrednio w Visual Studio (Visual Basic, JScript).

```
public static class GeneratorClass
{
      public static string GenerateCode(CodeCompileUnit compileunit,
String clsName, String src, String language)
      {
            string sourceFile;
            CodeDomProvider provider = GetCurrentProvider(language);
            if (provider.FileExtension[0] == '.')
            {
                  sourceFile = src + clsName + provider.FileExtension;
            }
            else
            {
                  sourceFile = src + clsName + "." +
provider.FileExtension;
            }

            IndentedTextWriter tw = new IndentedTextWriter(
            new StreamWriter(sourceFile, false), "    ");
            CodeGeneratorOptions opt = new CodeGeneratorOptions();

            provider.GenerateCodeFromCompileUnit(compileunit, tw, opt);

            tw.Close();
            return sourceFile;
      }
      private static CodeDomProvider GetCurrentProvider(string
language)
      {
```

```
            CodeDomProvider provider;
            switch (language)
            {
                  case "CSharp":
                        provider =
CodeDomProvider.CreateProvider("CSharp");
                        break;
                  case "Visual Basic":
                        provider =
CodeDomProvider.CreateProvider("VisualBasic");
                        break;
                  case "JScript":
                        provider =
CodeDomProvider.CreateProvider("JScript");
                        break;
                  default:
                        provider =
CodeDomProvider.CreateProvider("CSharp");
                        break;
            }
            return provider;
      }
}
```

Przykład 12- Statyczna klasa wykorzystywana do generowania kodu

Aby umożliwić programistom wygodną rozbudowę tworzonych przez nich aplikacji, wszystkie generowane klasy oznaczone są jako ***partial***. Konstrukcja ta pozwala na umieszczanie ciała klasy w wielu miejscach. Wraz z plikami generowanymi z meta-danych tworzone są puste definicje mapujących klas, które nie są nadpisywane jeśli już istnieją. Umożliwia to naniesienie poprawek w bazie danych i ponowne utworzenie klas mapujących bez utraty fragmentów rozszerzających implementacje wygenerowane automatycznie.

Oprócz kodu tworzonego według zaleceń wzorca Active Record w przypadku relacji wiele-do-wiele tworzone jest dodatkowo jedno pole (oraz odpowiadająca mu właściwość) i dwie metody. Funkcjonalność ta pozwala obsługiwać ten typ zależności

między encjami w obiektowy sposób niż poprzez osobne tworzenie obiektu tabeli asocjacyjnej. Dodatkowe pole jest kolekcją obiektów encji zależnej. W akcesorze ***get*** korespondującej właściwości ustawiane jest odpowiednie źródło. Dwie dodatkowe metody odpowiedzialne są za dodawanie i usuwanie obiektów z kolekcji. Przykład 13 zawiera kod dodatkowych elementów.

```
private EntitySet<Product> _Products;
public EntitySet<Product> Products
{
      get
      {
            if ((this._Products == null))
            {
                  this._Products = new EntitySet<Product>(OnProductAdd,
OnProductRemove);
                  this._Products.SetSource(this.habtm_Orders_Products.S
elect(c => c.Products));
            }
            return this._Products;
      }
      set
      {
            this._Products.Assign(value);
      }
}
private void OnProductsAdd(Products entity)
{
      this.habtm_Orders_Products.Add(new habtm_Orders_Products {
Orders = this, Products = entity });
}
private void OnProductsRemove(Products entity)
{
      habtm_Orders_Products entityToRem =
this.habtm_Orders_Products.FirstOrDefault(c => c.order_id == this.id
&& c.product_id == entity.id);
      this.habtm_Orders_Products.Remove(entityToRem);
}
```

Przykład 13 – Kod poprawiający obsługę relacji wiele-do-wiele

Dzięki temu zabiegowi możliwe jest dodawanie i usuwanie obiektów w relacji wiele-do-wiele za pomocą metod ***Add*** i ***Remove*** czyli z pominięciem obiektów klasy reprezentującej tabelę pośrednią.

5.5. Przykład wykorzystania

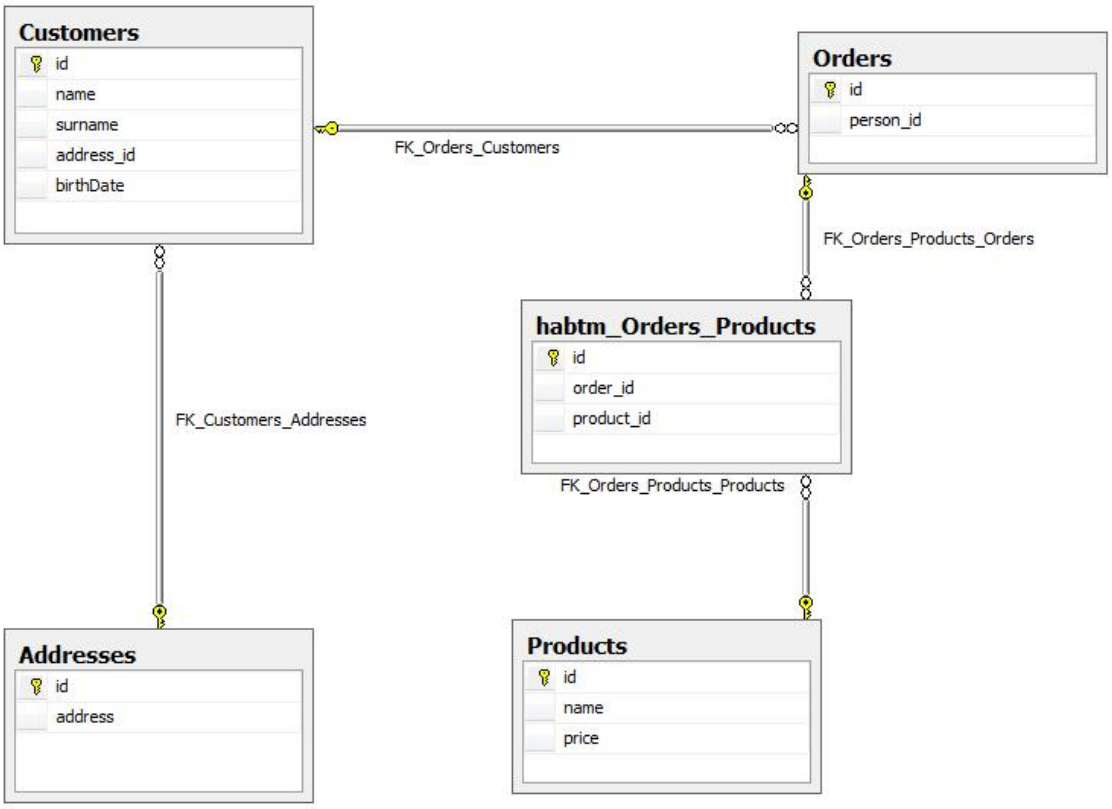
Wykorzystanie prototypu zostanie zademonstrowane na przykładzie tworzenia aplikacji konsoli Windows wykorzystującej bazę danych MS SQL 2005.

5.5.1. Schemat bazy danych

Do przykładu zostanie wykorzystana baza danych zawierająca relację:

- jeden-do-wiele: pomiędzy tabelami ***Customers i Addresses*** oraz ***Customers i Orders***
- wiele-do-wiele: pomiędzy tabelami ***Orders i Products*** z wykorzystaniem ***habtm_Orders_Products*** jako tabeli asocjacyjnej.

Schemat bazy danych przykładowej aplikacji prezentuje Rysunek 16.



Rysunek 16 – Schemat przykładowej bazy danych

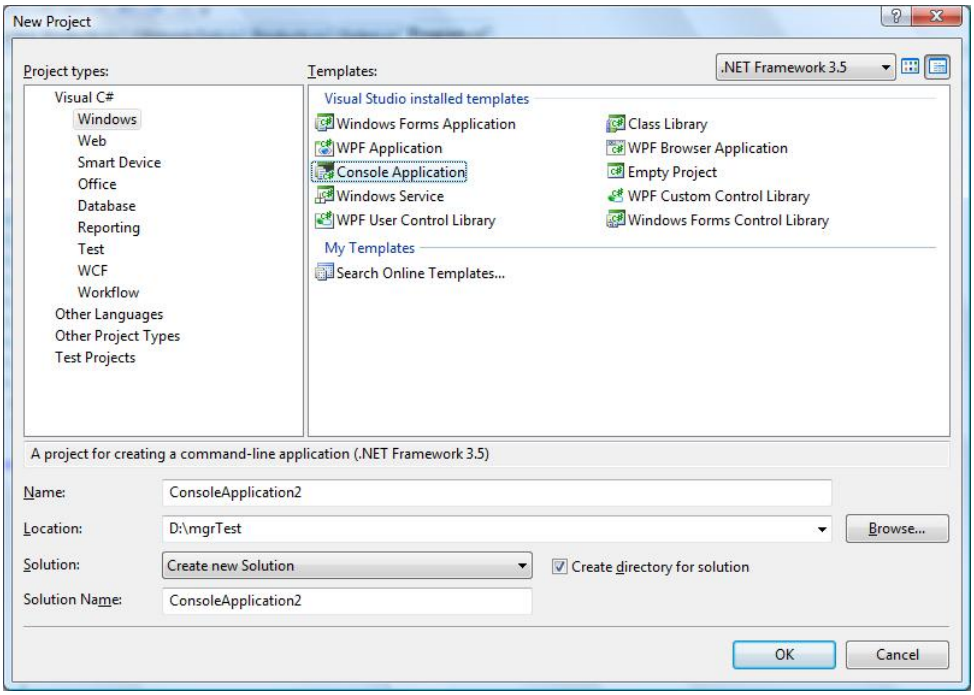
5.5.2. Aplikacja konsoli windows – generowanie klas

Po uruchomieniu Viual Studio 2008 należy utworzyć nowy projekt aplikacji konsoli Windows (Rysunek 17). Kreator przygotuje odpowiednie pliki i ustawi niezbędne referencje do takiego projektu. Aby uruchomić wtyczkę i wygenerować odpowiednie pliki, należy w górnym menu w zakładce ***Tools (Narzędzia)*** wybrać opcję ***ORemedy***. Uruchomiony zostanie interfejs graficzny (Rysunek 11 i Rysunek 12) a nastepnie dla wybranych tabel aplikacji zostaną wygenerowane klasy mapujące encje. Widok projektu po wygenerowaniu plików prezentuje Rysunek 18.

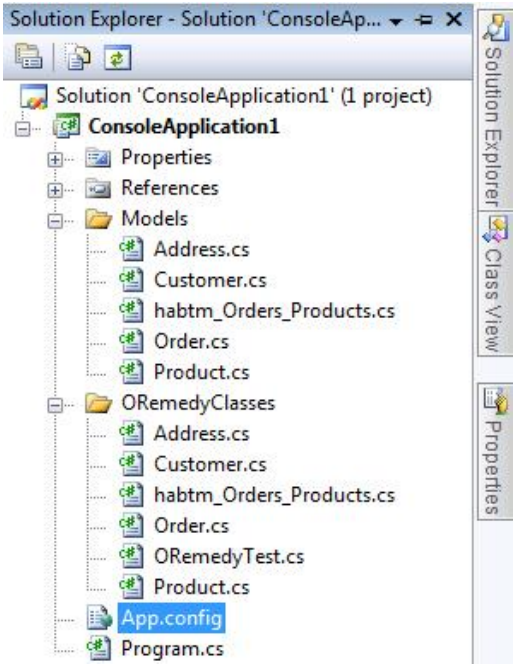
W projekcie utworzone zostały 2 foldery:

- ORemedyClasses – w którym znajdują się wygenerowane klasy odpowiedzialne za mapowanie encji (kod plików znajduje się w Dodatku A do pracy)

- Models – w którym znajdują się klasy umożliwiające rozbudowę wygenerowanych modeli



Rysunek 17 – Ekran wyboru nowego projektu w Visual Studio 2008



Rysunek 18 – Widok projektu po wygenerowaniu klas mapujących

Wraz z plikami mapującymi została wygenerowana klasa ***ORemedyTest.cs*** dziedzicząca po ***DataContext*** znajdującym się w przestrzeni nazw ***System.Data.Linq.*** Jest on odpowiedzialny głównie za tworzenie połączeń z bazą danych i śledzenie zmian w mapowanych obiektach. Klasa ta dokonuje także tłumaczenia zapytań LINQ na poprawne zapytania SQL.

Wygenerowane klasy są automatycznie opisane za pomocą dyrektyw LINQ. Każde pole utworzonych klas posiada odpowiadającą mu właściwość, która jest także opisana zgodnie z wymaganiami LINQ za pomocą odpowiednich atrybutów (Przykład 14).

```
[Column(Storage="_id", IsPrimaryKey=true, IsDbGenerated=true,
DbType="int NOT NULL IDENTITY")]
public Int32 id {
      get {
            return this._id;
      }
      set {
            this._id = value;
      }
}
```

Przykład 14 – Właściwość opisana za pomocą atrybutów LINQ

Właściwości odpowiadające za opis relacji są przechowywane przez odpowiedni typ danych:

- ***EntitySet<TEntity>*** – dla „*końca*” relacji oznaczającego wiele
- ***EntityRef<TEntity>*** – dla „*końca*” relacji oznaczającego jeden

gdzie ***TEntity*** oznacza klasę.

Wszystkie pola typu ***EntitySet<TEntity>*** musza być zaincjalizowane w kontruktorze wraz z akcjami opowiedzialnymi za dodanie i usuwanie elementów z kolekcji (Przykład 15).

```
public Customer() {
      this._Addresses = default(EntityRef<Address>);
      this._Orders = new EntitySet<Order>(
            new Action<Order>(this.attachOrder),
            new Action<Order>(this.detachOrder));
}

private void attachOrder(Order ent) {

      ent.Customers = this;
}
private void detachOrder (Order ent) {
      ent.Customers = null;
}
```

Przykład 15 – Przykład konstruktora i akcji odpowiedzialnych za dodawanie i usuwanie elementów kolekcji *EntitySet*

5.5.3. Praca z obiektami

Utworzenie obiektu odbywa się poprzez wywołanie jednego z dwóch konstruktorów. Tak utworzoną instancję po wypełnieniu danymi należy zapisać za pomocą wygenerowanej metody ***save.*** Jako parametry przymuje ona obiekt ***DataContext*** oraz flagę informującą czy obiekt jest nowy czy będzie to modyfikacja danych. Przykład 16 ilustruje tworzenie obiektów ***Customer*** oraz przypisanie im utworzonych obiektów ***Address*** (Rysunek 19).

```
var db = new ORemedyTest();
List<Address> ad = new List<Address>();
for (int i = 0; i < 46; i++)
{
      Address a = new Address();
      a.address = "ul.Jasińska " + i + ", warszawa";
      ad.Add(a);
}
for (int i = 0; i < 46; i++)
{
      Customer c = new Customer();
      c.name = "Jan" + i;
```

```
        c.surname = "Kowalski" + i;
        c.birthDate = new DateTime(1900 + i, 10, 07);
        c.Addresses = ad[i];
        cu.Add(c);
        c.save(db, true);
}
```

Przykład 16 – Tworzenie i zapisywanie obiektów

	id	name	surname	address_id	birthDate
1	3012	Jan0	Kowalski0	3012	1900-10-07 00:00:00.000
2	3013	Jan1	Kowalski1	3013	1901-10-07 00:00:00.000
3	3014	Jan2	Kowalski2	3014	1902-10-07 00:00:00.000
4	3015	Jan3	Kowalski3	3015	1903-10-07 00:00:00.000
5	3016	Jan4	Kowalski4	3016	1904-10-07 00:00:00.000
6	3017	Jan5	Kowalski5	3017	1905-10-07 00:00:00.000
7	3018	Jan6	Kowalski6	3018	1906-10-07 00:00:00.000
8	3019	Jan7	Kowalski7	3019	1907-10-07 00:00:00.000

	id	address
1	3012	ul.Jasińska 0, warszawa
2	3013	ul.Jasińska 1, warszawa
3	3014	ul.Jasińska 2, warszawa
4	3015	ul.Jasińska 3, warszawa
5	3016	ul.Jasińska 4, warszawa
6	3017	ul.Jasińska 5, warszawa
7	3018	ul.Jasińska 6, warszawa
8	3019	ul.Jasińska 7, warszawa

Rysunek 19 – Część danych zapisanych w bazie po utworzeniu i zapisaniu obiektów

Do usuwania obiektów z bazy służy generowana metoda ***delete***, która jako parametr przyjmuje obiekt ***DataContext***. Przykład 17 pokazuje, w jaki sposób pobierany jest obiekt za pomocą zapytań LINQ, a następnie usuwany jest przy pomocy tej właśnie metody.

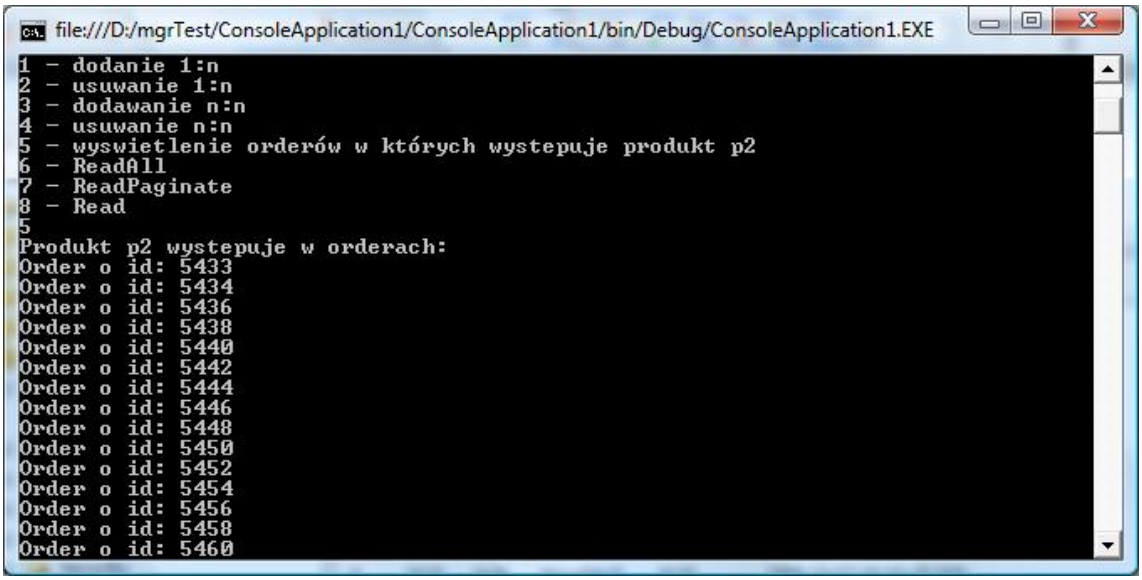
```
var db = new ORemedyTest();
var k = from cust in db.Customers where
cust.Addresses.address.Equals("ul.Jasińska 0, warszawa") select cust;
foreach (Customer s in k)
{
```

```
        Console.WriteLine(String.Format("Usuwam:
Imię:{0},Nazwisko:{1},Urodz.:{2},addres:{3}",s.name,s.surname,s.birthD
ate,s.Addresses.address));
        s.Addresses.Customers.Remove(s);
        s.delete(db);
}
```

Przykład 17 – Usuwanie danych o obiekcie z bazy danych

W przypadku relacji wiele-do-wiele dzięki zastosowaniu odpowiednich mechanizmów możliwe jest ukrycie tabeli pośredniczącej podczas korzystania z obiektów zależnych. Przykład 18 pokazuje sposób wyświetlenia wszystkich zamówień, w których wystepuje produkt o nazwie „*p2*”.

```
var db = new ORemedyTest();
Product p = db.Products.Single(prd => prd.name.Equals("p2"));
Console.WriteLine("Produkt p2 wystepuje w orderach:");
foreach(Order o in p.Orders)
        Console.WriteLine(String.Format("Order o id: {0}", o.id));
```

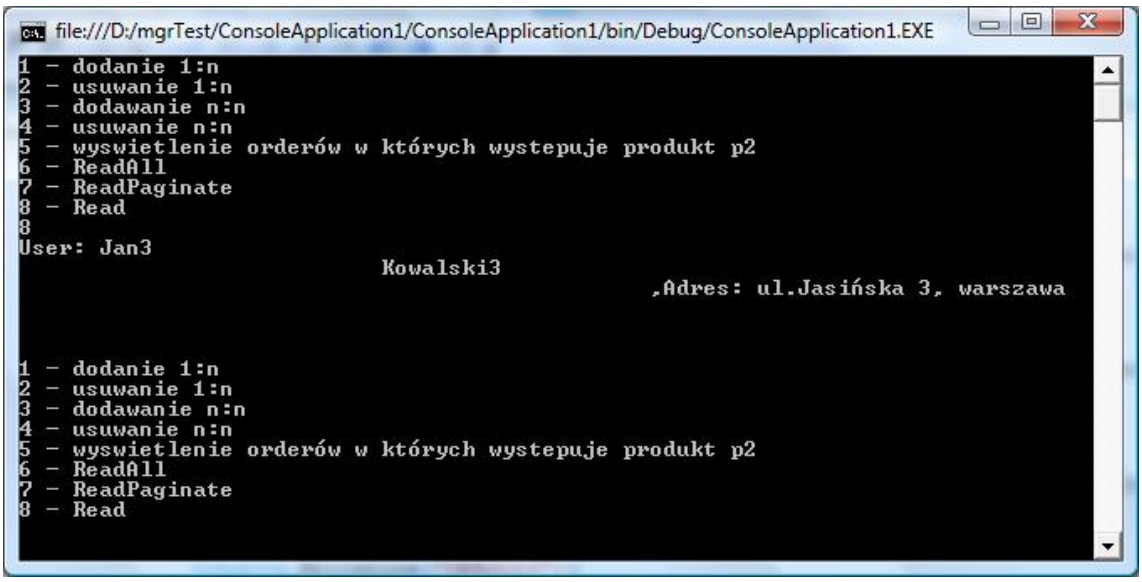


Przykład 18 – Iteracja po obiektach relcaji wiele-do-wiele

5.5.4. Dodatkowe metody

Oprócz metod do zapisu obiektów do bazy – ***save*** – oraz usuwania – ***delete*** – w klasach wygenerowane są także metody umożliwiające pobieranie danych. Jedną z nich jest ***read*** służąca do odczytania pojedyńczej krotki z bazy danych. Jako parametr przyjmuje ona id rekordu, który ma być odczytany oraz obiekt ***DataContext*** (Przykład 19).

```
var db = new ORemedyTest();
Customer cr = Customer.read(3015,db);
Console.WriteLine(String.Format("User: {0} {1},Adres: {2}", cr.name,
cr.surname, cr.Addresses.address));
```

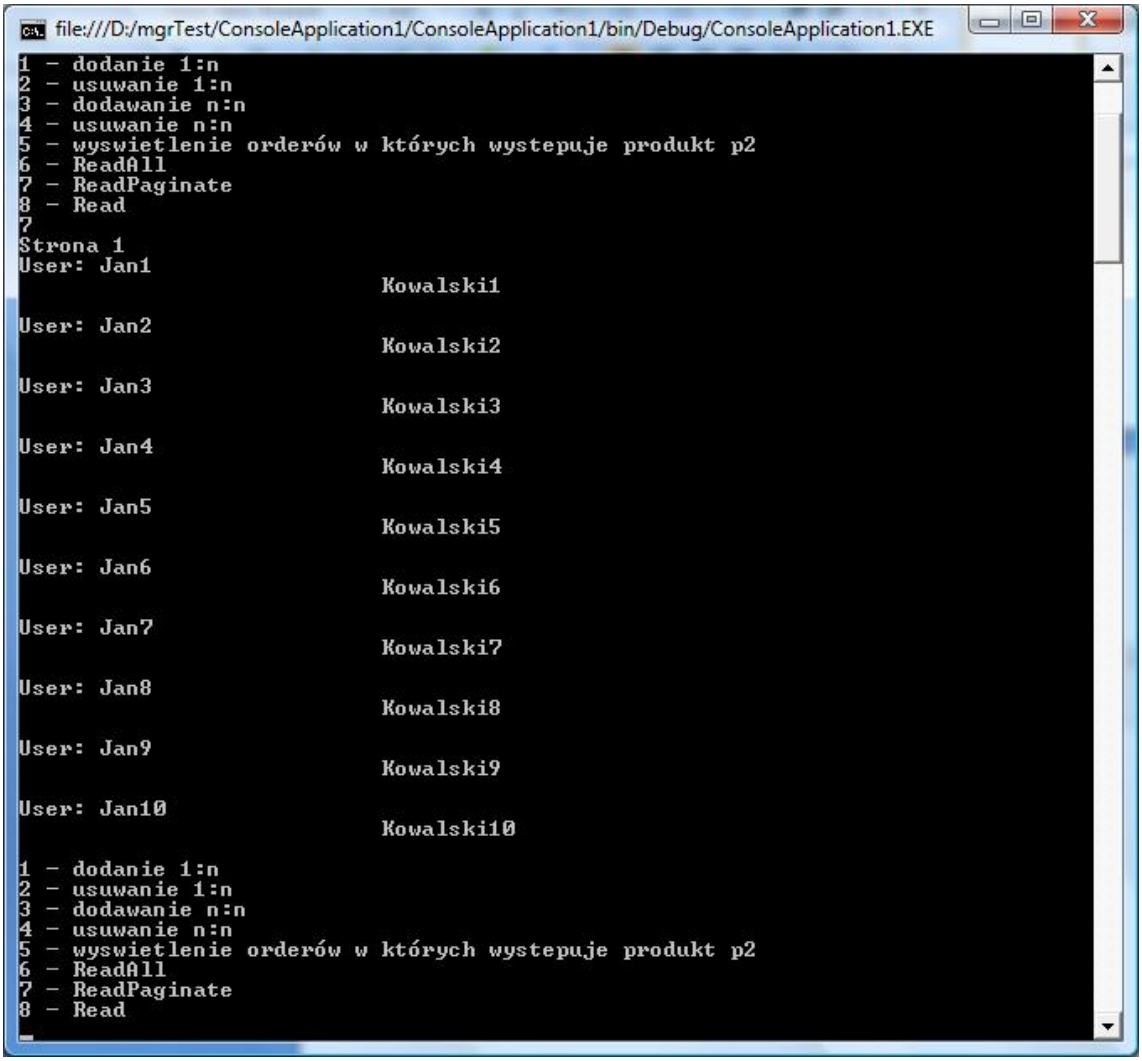


Przykład 19 – Wykorzystanie metody *read*

Istnieją też statyczne metody pozwalające odczytać wszystkie rekordy oraz prosty mechanizm paginacji (Przykład 20). Obie zwracają silnie typowaną listę obiektów. W przypadku metody używanej do stronicowania jako parametry należy podać numer strony oraz ilość rekordów, które zostaną odczytane z bazy danych.

```
var db = new ORemedyTest();
Console.WriteLine("Strona 1");
List<Customer> l2 = Customer.readPage(1,10,db);
foreach (Customer c in l2)
```

```
        Console.WriteLine(String.Format("User: {0} {1}", c.name,
c.surname));
```



Przykład 20 – Wykorzystanie metody *readPage*

6. Zalety i wady przyjętych rozwiązań

Prezentowane rozwiązanie jest dedykowane do tworzenia oprogramowania w oparciu o bazy danych o nieskomplikowanym schemacie. Duża liczba encji i relacji może utrudnić orientację w strukturze klas. Przy czym wykorzystanie natywnych technologii firmy Microsoft pozwala założyć, że aplikacje tworzone przy użyciu tego rozwiązania, nie będą odbiegać pod względem wydajności od aplikacji konfigurowanych ręcznie.

Niewątpliwą zaletą omawianej aplikacji do mapowania obiektowo-relacyjnego jest brak konfiguracji i automatyzacja generowania kodu. Pozwala to nie tylko zaoszczędzić czas, lecz także zmniejsza podatność oprogramowania na błędy. Podejście *„konwencja ponad konfiguracją”* eliminuje element konfiguracji, w którym często programiści popełniają błędy, które zazwyczaj trudno zlokalizować, zwłaszcza gdy konfiguracja nie jest objęta działaniem programów debugujących.

Dodatkowym atutem rozwiązania jest generowanie kodu poprawiającego dodawanie i usuwanie obiektów w relacji wiele-do-wiele. Podejście, które jest prezentowane w pracy, pozwala zarządzać relacją w stylu obiektowym bez potrzeby używania obiektów odwzorowujących tabelę pośredniczącą.

7. Proponowane plany rozwoju

7.1. Inżynieria odwrotna

Naturalnym kierunkiem rozwoju aplikacji mapującej jest dodanie automatycznej obsługi i śledzenia zmian w bazie danych. Możliwość generowania lub modyfikacji schematów baz danych na podstawie schematu klas byłaby pomocna i przyczyniłaby się do automatyzacji projektowania aplikacji informatycznych, co znacznie ułatwiłoby adeptom programowania poznanie środowiska i tworzenia zorientowanych obiektowo aplikacji bez potrzeby nauki SQL i systemu relacyjnego.

7.2. Generowanie szablonów i rozwój metod dostępu do danych

Autor pracy uważa, że kolejnym krokiem zwiększającym użyteczność prezentowanego rozwiązania byłoby dodanie podstawowych szablonów aplikacji. Baza szablonów mogłaby być otwarta, co umożliwiłoby powstanie zaawansowanych kontrolek do Visual Studio współpracujących z aplikacją mapującą. Wraz z rozwojem niezbędne byłoby poszerzenie bazy metod operujących na danych a generowanych przez aplikację jak na przykład zaawansowanej metody do paginacji danych.

7.3. Obsługa innych języków

Choć cześciowo aplikacja została przygotowana do obsługi innych języków niż C# jest to obszar, w którym także można rozwijać prototyp. Możliwości Visual Studio są bardzo szerokie również w zakresie rozszerzania o inne języki programowania. Aplikacja jako zintegrowana ze środowiskiem mogłaby pełnić funkcję mapera dla każdego zainstalowanego języka.

7.4. Opensource

Przykłady aplikacji dostępnych w ramach tak zwanego wolnego oprogramowania pokazują, że ruch ten niejednokrotnie przyczynia się do rozbudowy funkcjonalności oprogramowania. Autor pracy uważa, że szeroka społeczność zebrana wokół *Opensource* istotnie przyczyniłaby się do rozwoju aplikacji.

8. Podsumowanie

Aplikacje mapujące obiektowo-relacyjnie nabierają coraz większego znaczenia w procesie wytwarzania oprogramowaia. Używanie ich staje się regułą, gdyż poprawiają czytelność kodu, a także umożliwiają pracę w systemie obiektowym bez potrzeby mieszania go z relacyjnym. Dziedzina ta jest rozwijana i nie posiada jeszcze w pełni określonych standardów. Dostępne są narzędzia zaawansowane ale przez to skomplikowane. Autor pracy uważa, że rowój tej dziedziny powinien być nakierowany na automatyczną analizę struktur baz danych i automatyzację procesów. Takie podejście eliminuje problem tworzenia skomplikowanych konfiguracji.

Zaprezentowany prototyp aplikacji mapującej realizuje założenia stawiane przez wzorzec Active Record dotyczące mapowania relacyjnych baz danych. Dodatkowo kod jest generowany automatycznie, co niewątpliwie jest zaletą tego oprogramowania. Pominięta została skomplikowana konfiguracja, a analiza schematu bazy danych odbywa się automatycznie. Dzięki temu jest to oprogramowanie ułatwiające zapoznanie się z zagadnieniem programowania w oparciu o bazy danych.

9. Bibliografia

[1]. Kazimierz Subieta **Słownik terminów z zakresu obiektowości**.

Akademicka Oficyna Wydawnicza PLJ, Warszawa 1999

Wersja elektroniczna dostępna pod adresem:

http://www.ipipan.waw.pl/~subieta/artykuly/slownik_obiektowosci/hasla_slownika.htm

[2]. Martin Fowler **Patterns of enterprise application architecture**

Pearson Education, Inc, 2004

[3]. Fabio Claudio Ferracchiati **LINQ for Visual C# 2008**

Apres, 2008

[4]. Dokumentacja techniczna **Microsoft Developer Network**

http://msdn.microsoft.com

[5]. Dokumentacja techniczna **Hibernate**

http://www.hibernate.org

[6]. Radosław Zawartko **Skrypty we własnej aplikacji – część 2**

artykuł na portalu codeguru.pl: http://codeguru.pl/article-491.aspx

[7]. Dokumentacja projektu **Castle ActiveRecord**

http://www.castleproject.org/activerecord/index.html

10. Spis Rzeczy

10.1. Spis Rysunków

10.2. Spis Tabel

10.3. Spis Przykładów

11. Dodatki

11.1. Dodatek A – Kod C# klas mapujących

11.1.1. Address.cs

```
//------------------------------------------------------------------------
----------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.4200
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------
----------

namespace ConsoleApplication1.ORemedy.ORemedyTest {
    using System;
    using System.Data;
    using System.Configuration;
    using System.Data.SqlClient;
    using System.Collections.Generic;
    using System.Linq;
    using System.Data.Linq;
    using System.Data.Linq.Mapping;


    [Table(Name = "Addresses")]
    public partial class Address {

        private Int32 _id;

        private String _address;

        private EntitySet<Customer> _Customers;

        #region Constructors declarations
        public Address() {
            this._Customers = new EntitySet<Customer>(new
Action<Customer>(this.attachCustomer), new
Action<Customer>(this.detachCustomer));
        }

        public Address(Int32 id, String address) {
            this._id = id;
            this._address = address;
            this._Customers = new EntitySet<Customer>(new
Action<Customer>(this.attachCustomer), new
Action<Customer>(this.detachCustomer));
```

```
        }
        #endregion

        [Column(Storage="_id", IsPrimaryKey=true, IsDbGenerated=true,
DbType="int NOT NULL IDENTITY")]
        public Int32 id {
            get {
                return this._id;
            }
            set {
                this._id = value;
            }
        }

        [Column(Storage="_address", DbType="nchar(255) NOT NULL ",
CanBeNull=false )]
        public String address {
            get {
                return this._address;
            }
            set {
                this._address = value;
            }
        }

        [Association(Name="FK_Customers_Addresses",
Storage="_Customers", ThisKey="id", OtherKey="address_id")]
        public EntitySet<Customer> Customers {
            get {
                return this._Customers;
            }
            set {
                this._Customers.Assign(value);
            }
        }

        private void attachCustomer(Customer ent) {
            ent.Addresses = this;
        }

        private void detachCustomer(Customer ent) {
            ent.Addresses = null;
        }

        public void save(ORemedyTest db, bool isNewObject) {
            if (isNewObject) {
                db.Addresses.InsertOnSubmit(this);
            }
            db.SubmitChanges();
        }

        public static Address read(int id, ORemedyTest db) {
            return db.Addresses.Single(c => c.id == id);
        }

        public static List<Address> readPage(int pageNumber, int
recordNumber, ORemedyTest db) {
            var query = (from c in db.Addresses select
c).Skip((pageNumber-1)*recordNumber).Take(recordNumber);
```

```
            return query.ToList<Address>();
        }

        public static List<Address> readAll(ORemedyTest db) {
            var query = from c in db.Addresses select c;
            return query.ToList<Address>();
        }

        public void delete(ORemedyTest db) {
            db.Addresses.DeleteOnSubmit(this);
            db.SubmitChanges();
        }
    }
}
```

11.1.2. Customer.cs

```
//------------------------------------------------------------------------
----------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.4200
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------
----------

namespace ConsoleApplication1.ORemedy.ORemedyTest {
    using System;
    using System.Data;
    using System.Configuration;
    using System.Data.SqlClient;
    using System.Collections.Generic;
    using System.Linq;
    using System.Data.Linq;
    using System.Data.Linq.Mapping;


    [Table(Name = "Customers")]
    public partial class Customer {

        private Int32 _address_id;

        private String _name;

        private String _surname;

        private DateTime _birthDate;

        private Int32 _id;

        private EntityRef<Address> _Addresses;
```

```
        private EntitySet<Order> _Orders;

        #region Constructors declarations
        public Customer() {
            this._Addresses = default(EntityRef<Address>);
            this._Orders = new EntitySet<Order>(new
Action<Order>(this.attachOrder), new Action<Order>(this.detachOrder));
        }

        public Customer(Int32 address_id, String name, String surname,
DateTime birthdate, Int32 id) {
            this._address_id = address_id;
            this._name = name;
            this._surname = surname;
            this._birthDate = birthdate;
            this._id = id;
            this._Addresses = default(EntityRef<Address>);
            this._Orders = new EntitySet<Order>(new
Action<Order>(this.attachOrder), new Action<Order>(this.detachOrder));
        }
        #endregion

        [Column(Storage="_address_id", DbType="int", CanBeNull=true )]
        public Int32 address_id {
            get {
                return this._address_id;
            }
            set {
                this._address_id = value;
            }
        }

        [Column(Storage="_name", DbType="nchar(100) NOT NULL ",
CanBeNull=false )]
        public String name {
            get {
                return this._name;
            }
            set {
                this._name = value;
            }
        }

        [Column(Storage="_surname", DbType="nchar(100) NOT NULL ",
CanBeNull=false )]
        public String surname {
            get {
                return this._surname;
            }
            set {
                this._surname = value;
            }
        }

        [Column(Storage="_birthDate", DbType="datetime NOT NULL ",
CanBeNull=false )]
        public DateTime birthDate {
            get {
                return this._birthDate;
```

```
            }
            set {
                this._birthDate = value;
            }
        }

        [Column(Storage="_id", IsPrimaryKey=true, IsDbGenerated=true,
DbType="int NOT NULL IDENTITY")]
        public Int32 id {
            get {
                return this._id;
            }
            set {
                this._id = value;
            }
        }

        [Association(Name="FK_Customers_Addresses",
Storage="_Addresses", ThisKey="address_id", OtherKey="id",
IsForeignKey=true)]
        public Address Addresses {
            get {
                return this._Addresses.Entity;
            }
            set {
                Address tmp = this._Addresses.Entity;
                if (((tmp != value)
                            ||
(this._Addresses.HasLoadedOrAssignedValue == false))) {
                    if ((tmp != null)) {
                        this._Addresses.Entity = null;
                        tmp.Customers.Remove(this);
                    }
                    this._Addresses.Entity = value;
                    if ((value != null)) {
                        value.Customers.Add(this);
                        this._address_id = value.id;
                    }
                    else {
                        this._address_id = default(Int32);
                    }
                }
            }
        }

        [Association(Name="FK_Orders_Customers", Storage="_Orders",
ThisKey="id", OtherKey="person_id")]
        public EntitySet<Order> Orders {
            get {
                return this._Orders;
            }
            set {
                this._Orders.Assign(value);
            }
        }

        private void attachOrder(Order ent) {
            ent.Customers = this;
        }
```

```
        private void detachOrder(Order ent) {
            ent.Customers = null;
        }

        public void save(ORemedyTest db, bool isNewObject) {
            if (isNewObject) {
                db.Customers.InsertOnSubmit(this);
            }
            db.SubmitChanges();
        }

        public static Customer read(int id, ORemedyTest db) {
            return db.Customers.Single(c => c.id == id);
        }

        public static List<Customer> readPage(int pageNumber, int
recordNumber, ORemedyTest db) {
            var query = (from c in db.Customers select
c).Skip((pageNumber-1)*recordNumber).Take(recordNumber);
            return query.ToList<Customer>();
        }

        public static List<Customer> readAll(ORemedyTest db) {
            var query = from c in db.Customers select c;
            return query.ToList<Customer>();
        }

        public void delete(ORemedyTest db) {
            db.Customers.DeleteOnSubmit(this);
            db.SubmitChanges();
        }
    }
```

11.1.3. habtm_Orders_Products.cs

```
//------------------------------------------------------------------------
----------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.4200
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------
----------

namespace ConsoleApplication1.ORemedy.ORemedyTest {
    using System;
    using System.Data;
    using System.Configuration;
    using System.Data.SqlClient;
    using System.Collections.Generic;
    using System.Linq;
```

```
    using System.Data.Linq;
    using System.Data.Linq.Mapping;


    [Table(Name = "habtm_Orders_Products")]
    public partial class habtm_Orders_Products {

        private Int32 _order_id;

        private Int32 _product_id;

        private Int32 _id;

        private EntityRef<Order> _Orders;

        private EntityRef<Product> _Products;

        #region Constructors declarations
        public habtm_Orders_Products() {
            this._Orders = default(EntityRef<Order>);
            this._Products = default(EntityRef<Product>);
        }

        public habtm_Orders_Products(Int32 order_id, Int32 product_id,
Int32 id) {
            this._order_id = order_id;
            this._product_id = product_id;
            this._id = id;
            this._Orders = default(EntityRef<Order>);
            this._Products = default(EntityRef<Product>);
        }
        #endregion

        [Column(Storage="_order_id", DbType="int NOT NULL ",
CanBeNull=false )]
        public Int32 order_id {
            get {
                return this._order_id;
            }
            set {
                this._order_id = value;
            }
        }

        [Column(Storage="_product_id", DbType="int NOT NULL ",
CanBeNull=false )]
        public Int32 product_id {
            get {
                return this._product_id;
            }
            set {
                this._product_id = value;
            }
        }

        [Column(Storage="_id", IsPrimaryKey=true, IsDbGenerated=true,
DbType="int NOT NULL IDENTITY")]
        public Int32 id {
            get {
```

```
                return this._id;
            }
            set {
                this._id = value;
            }
        }

        [Association(Name="FK_habtm_Orders_Products_Orders",
Storage="_Orders", ThisKey="order_id", OtherKey="id",
IsForeignKey=true, DeleteOnNull = true)]
        public Order Orders {
            get {
                return this._Orders.Entity;
            }
            set {
                Order tmp = this._Orders.Entity;
                if (((tmp != value)
                            || (this._Orders.HasLoadedOrAssignedValue
== false))) {
                    if ((tmp != null)) {
                        this._Orders.Entity = null;
                        tmp.habtm_Orders_Products.Remove(this);
                    }
                    this._Orders.Entity = value;
                    if ((value != null)) {
                        value.habtm_Orders_Products.Add(this);
                        this._order_id = value.id;
                    }
                    else {
                        this._order_id = default(Int32);
                    }
                }
            }
        }

        [Association(Name="FK_habtm_Orders_Products_Products",
Storage="_Products", ThisKey="product_id", OtherKey="id",
IsForeignKey=true, DeleteOnNull = true)]
        public Product Products {
            get {
                return this._Products.Entity;
            }
            set {
                Product tmp = this._Products.Entity;
                if (((tmp != value)
                            ||
(this._Products.HasLoadedOrAssignedValue == false))) {
                    if ((tmp != null)) {
                        this._Products.Entity = null;
                        tmp.habtm_Orders_Products.Remove(this);
                    }
                    this._Products.Entity = value;
                    if ((value != null)) {
                        value.habtm_Orders_Products.Add(this);
                        this._product_id = value.id;
                    }
                    else {
                        this._product_id = default(Int32);
                    }
```

```
                }
            }
        }

        public void save(ORemedyTest db, bool isNewObject) {
            if (isNewObject) {
                db.habtm_Orders_Products.InsertOnSubmit(this);
            }
            db.SubmitChanges();
        }

        public static habtm_Orders_Products read(int id, ORemedyTest
db) {
            return db.habtm_Orders_Products.Single(c => c.id == id);
        }

        public static List<habtm_Orders_Products> readPage(int
pageNumber, int recordNumber, ORemedyTest db) {
            var query = (from c in db.habtm_Orders_Products select
c).Skip((pageNumber-1)*recordNumber).Take(recordNumber);
            return query.ToList<habtm_Orders_Products>();
        }

        public static List<habtm_Orders_Products> readAll(ORemedyTest
db) {
            var query = from c in db.habtm_Orders_Products select c;
            return query.ToList<habtm_Orders_Products>();
        }

        public void delete(ORemedyTest db) {
            db.habtm_Orders_Products.DeleteOnSubmit(this);
            db.SubmitChanges();
        }
    }
}
```

11.1.4. Order.cs

```
//------------------------------------------------------------------------
----------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.4200
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------
----------

namespace ConsoleApplication1.ORemedy.ORemedyTest {
    using System;
    using System.Data;
    using System.Configuration;
    using System.Data.SqlClient;
```

```
    using System.Collections.Generic;
    using System.Linq;
    using System.Data.Linq;
    using System.Data.Linq.Mapping;


    [Table(Name = "Orders")]
    public partial class Order {

        private Int32 _person_id;

        private Int32 _id;

        private EntityRef<Customer> _Customers;

        private EntitySet<habtm_Orders_Products>
_habtm_Orders_Products;

        private EntitySet<Product> _Products;

        #region Constructors declarations
        public Order() {
            this._Customers = default(EntityRef<Customer>);
            this._habtm_Orders_Products = new
EntitySet<habtm_Orders_Products>(new
Action<habtm_Orders_Products>(this.attachhabtm_Orders_Products), new
Action<habtm_Orders_Products>(this.detachhabtm_Orders_Products));
        }

        public Order(Int32 person_id, Int32 id) {
            this._person_id = person_id;
            this._id = id;
            this._Customers = default(EntityRef<Customer>);
            this._habtm_Orders_Products = new
EntitySet<habtm_Orders_Products>(new
Action<habtm_Orders_Products>(this.attachhabtm_Orders_Products), new
Action<habtm_Orders_Products>(this.detachhabtm_Orders_Products));
        }
        #endregion

        [Column(Storage="_person_id", DbType="int NOT NULL ",
CanBeNull=false )]
        public Int32 person_id {
            get {
                return this._person_id;
            }
            set {
                this._person_id = value;
            }
        }

        [Column(Storage="_id", IsPrimaryKey=true, IsDbGenerated=true,
DbType="int NOT NULL IDENTITY")]
        public Int32 id {
            get {
                return this._id;
            }
            set {
                this._id = value;
```

```
            }
        }

        [Association(Name="FK_Orders_Customers", Storage="_Customers",
ThisKey="person_id", OtherKey="id", IsForeignKey=true)]
        public Customer Customers {
            get {
                return this._Customers.Entity;
            }
            set {
                Customer tmp = this._Customers.Entity;
                if (((tmp != value)
                            ||
(this._Customers.HasLoadedOrAssignedValue == false))) {
                    if ((tmp != null)) {
                        this._Customers.Entity = null;
                        tmp.Orders.Remove(this);
                    }
                    this._Customers.Entity = value;
                    if ((value != null)) {
                        value.Orders.Add(this);
                        this._person_id = value.id;
                    }
                    else {
                        this._person_id = default(Int32);
                    }
                }
            }
        }

        [Association(Name="FK_habtm_Orders_Products_Products",
Storage="_habtm_Orders_Products", ThisKey="id",
OtherKey="product_id")]
        public EntitySet<habtm_Orders_Products> habtm_Orders_Products
{
            get {
                return this._habtm_Orders_Products;
            }
            set {
                this._habtm_Orders_Products.Assign(value);
            }
        }

        public EntitySet<Product> Products {
            get {
                if ((this._Products == null)) {
                    this._Products = new
EntitySet<Product>(OnProductAdd, OnProductRemove);

this._Products.SetSource(this.habtm_Orders_Products.Select(c =>
c.Products));
                }
                return this._Products;
            }
            set {
                this._Products.Assign(value);
            }
        }
```

```
        private void attachhabtm_Orders_Products(habtm_Orders_Products
ent) {
            ent.Orders = this;
        }

        private void detachhabtm_Orders_Products(habtm_Orders_Products
ent) {
            ent.Orders = null;
        }

        public void save(ORemedyTest db, bool isNewObject) {
            if (isNewObject) {
                db.Orders.InsertOnSubmit(this);
            }
            db.SubmitChanges();
        }

        public static Order read(int id, ORemedyTest db) {
            return db.Orders.Single(c => c.id == id);
        }

        public static List<Order> readPage(int pageNumber, int
recordNumber, ORemedyTest db) {
            var query = (from c in db.Orders select
c).Skip((pageNumber-1)*recordNumber).Take(recordNumber);
            return query.ToList<Order>();
        }

        public static List<Order> readAll(ORemedyTest db) {
            var query = from c in db.Orders select c;
            return query.ToList<Order>();
        }

        public void delete(ORemedyTest db) {
            db.Orders.DeleteOnSubmit(this);
            db.SubmitChanges();
        }

        private void OnProductAdd(Product entity) {
            this.habtm_Orders_Products.Add(new habtm_Orders_Products {
Orders = this, Products = entity });
        }

        private void OnProductRemove(Product entity) {
            habtm_Orders_Products entityToRem =
this.habtm_Orders_Products.FirstOrDefault(c => c.order_id == this.id
&& c.product_id == entity.id);
            this.habtm_Orders_Products.Remove(entityToRem);
        }
    }
}
```

11.1.5. ORemedyTest.cs

```
//------------------------------------------------------------------------
----------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.4200
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------
----------

namespace ConsoleApplication1.ORemedy.ORemedyTest {
    using System;
    using System.Data;
    using System.Configuration;
    using System.Data.SqlClient;
    using System.Collections.Generic;
    using System.Linq;
    using System.Data.Linq;
    using System.Data.Linq.Mapping;


    public partial class ORemedyTest : System.Data.Linq.DataContext {

        public ORemedyTest() :
                base(@"Data Source=MARCIN-LAPTOP\MARCINSQL;Initial
Catalog=ORemedyTest;Integrated Security=True") {
        }

        public System.Data.Linq.Table<Order> Orders {
            get {
                return this.GetTable<Order>();
            }
        }

        public System.Data.Linq.Table<Product> Products {
            get {
                return this.GetTable<Product>();
            }
        }

        public System.Data.Linq.Table<habtm_Orders_Products>
habtm_Orders_Products {
            get {
                return this.GetTable<habtm_Orders_Products>();
            }
        }

        public System.Data.Linq.Table<Address> Addresses {
            get {
                return this.GetTable<Address>();
            }
        }
```

```
        public System.Data.Linq.Table<Customer> Customers {
            get {
                return this.GetTable<Customer>();
            }
        }
    }
}
```

11.1.6. Product.cs

```
//------------------------------------------------------------------------
----------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.4200
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------
----------

namespace ConsoleApplication1.ORemedy.ORemedyTest {
    using System;
    using System.Data;
    using System.Configuration;
    using System.Data.SqlClient;
    using System.Collections.Generic;
    using System.Linq;
    using System.Data.Linq;
    using System.Data.Linq.Mapping;


    [Table(Name = "Products")]
    public partial class Product {

        private String _name;

        private Int32 _price;

        private Int32 _id;

        private EntitySet<habtm_Orders_Products>
_habtm_Orders_Products;

        private EntitySet<Order> _Orders;

        #region Constructors declarations
        public Product() {
            this._habtm_Orders_Products = new
EntitySet<habtm_Orders_Products>(new
Action<habtm_Orders_Products>(this.attachhabtm_Orders_Products), new
Action<habtm_Orders_Products>(this.detachhabtm_Orders_Products));
        }
```

```
        public Product(String name, Int32 price, Int32 id) {
            this._name = name;
            this._price = price;
            this._id = id;
            this._habtm_Orders_Products = new
EntitySet<habtm_Orders_Products>(new
Action<habtm_Orders_Products>(this.attachhabtm_Orders_Products), new
Action<habtm_Orders_Products>(this.detachhabtm_Orders_Products));
        }
        #endregion

        [Column(Storage="_name", DbType="nchar(10) NOT NULL ",
CanBeNull=false )]
        public String name {
            get {
                return this._name;
            }
            set {
                this._name = value;
            }
        }

        [Column(Storage="_price", DbType="int NOT NULL ",
CanBeNull=false )]
        public Int32 price {
            get {
                return this._price;
            }
            set {
                this._price = value;
            }
        }

        [Column(Storage="_id", IsPrimaryKey=true, IsDbGenerated=true,
DbType="int NOT NULL IDENTITY")]
        public Int32 id {
            get {
                return this._id;
            }
            set {
                this._id = value;
            }
        }

        [Association(Name="FK_habtm_Orders_Products_Orders",
Storage="_habtm_Orders_Products", ThisKey="id", OtherKey="order_id")]
        public EntitySet<habtm_Orders_Products> habtm_Orders_Products
{
            get {
                return this._habtm_Orders_Products;
            }
            set {
                this._habtm_Orders_Products.Assign(value);
            }
        }

        public EntitySet<Order> Orders {
            get {
                if ((this._Orders == null)) {
```

```
                    this._Orders = new EntitySet<Order>(OnOrderAdd,
OnOrderRemove);

this._Orders.SetSource(this.habtm_Orders_Products.Select(c =>
c.Orders));
                }
                return this._Orders;
            }
            set {
                this._Orders.Assign(value);
            }
        }

        private void attachhabtm_Orders_Products(habtm_Orders_Products
ent) {
            ent.Products = this;
        }

        private void detachhabtm_Orders_Products(habtm_Orders_Products
ent) {
            ent.Products = null;
        }

        public void save(ORemedyTest db, bool isNewObject) {
            if (isNewObject) {
                db.Products.InsertOnSubmit(this);
            }
            db.SubmitChanges();
        }

        public static Product read(int id, ORemedyTest db) {
            return db.Products.Single(c => c.id == id);
        }

        public static List<Product> readPage(int pageNumber, int
recordNumber, ORemedyTest db) {
            var query = (from c in db.Products select
c).Skip((pageNumber-1)*recordNumber).Take(recordNumber);
            return query.ToList<Product>();
        }

        public static List<Product> readAll(ORemedyTest db) {
            var query = from c in db.Products select c;
            return query.ToList<Product>();
        }

        public void delete(ORemedyTest db) {
            db.Products.DeleteOnSubmit(this);
            db.SubmitChanges();
        }

        private void OnOrderAdd(Order entity) {
            this.habtm_Orders_Products.Add(new habtm_Orders_Products {
Products = this, Orders = entity });
        }

        private void OnOrderRemove(Order entity) {
```

```
            habtm_Orders_Products entityToRem =
this.habtm_Orders_Products.FirstOrDefault(c => c.product_id == this.id
&& c.order_id == entity.id);
            this.habtm_Orders_Products.Remove(entityToRem);
        }
    }
}
```