POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Radosław Kowalczyk
Nr albumu 9116

Maper obiektowo-relacyjny dla systemów spadkowych

Praca magisterska napisana
pod kierunkiem:

Dr inż. Mariusz Trzaska

Warszawa, luty 2013

Streszczenie

Praca dotyczy istotnego, praktycznego problemu wspomagania wytwarzania oprogramowania w językach obiektowych z wykorzystaniem relacyjnych baz danych za pomocą narzędzia do mapowania obiektowo-relacyjnego w systemach spadkowych. Zwraca uwagę na problemy wynikające z braku rozwoju technologii wykorzystywanych w systemach, które z różnych przyczyn nie mogą być zastąpione, lecz muszą być rozwijane.

W ramach pracy wykonano prototyp narzędzia pozwalającego mapować relacyjną bazę danych na klasy w obiektowym języku programowania Delphi. Narzędzie automatyzuje proces tworzenia oprogramowania wprowadzając szereg udogodnień związanych z połączeniem z bazą danych oraz bezpieczeństwem ich przetwarzania. Dodatkowo odseparowuje programistę od języka SQL dzięki pomocniczej klasie kryteriów wyszukiwania.

Spis treści

1. Wstęp

Mapowanie obiektowo-relacyjne jest metodą odwzorowania tabel z relacyjnej bazy danych na obiekty aplikacji klienckiej i na odwrót. Większość oprogramowania wytwarzana jest w językach zorientowanych obiektowo, a dane przechowywane są w relacyjnych bazach danych. W przeciwieństwie do obiektowych baz danych, relacyjne cechują się dużą popularnością i szerokim zastosowaniem ze względu na szereg mechanizmów wspierających efektywną pracę oraz duże wsparcie.

Przestarzałe technologie nie udostępniają narzędzi radzących sobie z problemem odwzorowania relacji w paradygmacie obiektowym. Tempo rozwoju systemów wykorzystujących takie technologie często jest nie do zaakceptowania, a zmiana technologii zbyt kosztowna lub po prostu niemożliwa. W takich sytuacjach narzędzie do mapowania obiektowo-relacyjnego może stanowić o dalszym istnieniu projektu.

1.1. Cel pracy

Celem pracy jest przedstawienie problemów występujących przy systemach spadkowych zaimplementowanych w przestarzałych językach zorientowanych obiektowo oraz metody wspomagania rozwoju takich systemów za pomocą mapowania obiektowo-relacyjnego.

Produktem ubocznym pracy jest opracowane narzędzie mapujące dla obiektowego języka programowania Delphi. Narzędzie to jest aplikacją kliencką, pozwalającą:

- Sprawdzić poprawność schematu bazy danych w pliku XML
- Wygenerować kod klas mapujących dane relacyjne
- Definiować kryteria pobierania danych z bazy dzięki pomocniczej klasie *TCriteria* bez konieczności definiowania natywnych zapytań SQL

Opracowane narzędzie pozwala zautomatyzować proces powtarzalnej i żmudnej implementacji tabel na obiekty wprowadzając dobre praktyki programowania i wykorzystując sprawdzone wzorce projektowe.

1.2. Rozwiązanie przyjęte w pracy

Prototyp narzędzia został wykonany w zintegrowanym środowisku programistycznym RAD Studio XE3. Do jego implementacji wykorzystany został język Delphi XE3. Wprowadzono możliwość łącznia się z bazami danych Microsoft SQL Server w wersji 2008 lub wyższej. Kod generowany przez narzędzie działa w aplikacjach opartych na języku Delphi w wersji 2005 lub wyższej.

Projekt aplikacji oparty został na elementach rozwiązań open-source tego typu narzędzi (np. *Propel*) oraz na doświadczeniu autora pracy w wykorzystaniu wzorców projektowych i stosowaniu dobrych praktyk programistycznych.

1.3. Rezultaty pracy

Rezultatem pracy jest identyfikacja problemów związanych z wytwarzaniem oprogramowania w przestarzałych językach programowania zorientowanych obiektowo z wykorzystaniem relacyjnych baz danych oraz metodach radzenia sobie z tymi problemami za pomocą mapowania obiektowo-relacyjnego.

Wynikiem powyższych działań jest prototyp narzędzia do mapowania obiektowo-relacyjnego dla obiektowego języka programowania Delphi oraz bazy danych Microsoft SQL Server.

1.4. Organizacja pracy

Pierwszy rozdział pracy zawiera krótki wstęp do poruszanego problemu mapowania obiektowo-relacyjnego i systemów spadkowych. Rozdział *Mapery w systemach spadkowych – stan sztuki* jest rozwinięciem wstępu – znajduje się tutaj dokładny opis niezgodności impedancji i systemów spadkowych oraz przedstawione są istniejące rozwiązania tego problemu. W rozdziale trzecim zamieszczone są informacje na temat narzędzi i koncepcji użytych w realizacji pracy oraz prototypu aplikacji. Rozdział czwarty opisuje propozycję dla realizacji narzędzia mapującego dla systemów spadkowych, natomiast rozdział piąty zawiera dokładny opis realizacji i korzystania z prototypu mapera. Podsumowanie pracy zawarte jest w ostatnim rozdziale. Pracę zakańczają spis prac cytowanych oraz spis rzeczy umieszczonych w treści pracy.

2. Mapery w systemach spadkowych – stan sztuki

Systemy, które zostały wdrożone dawno temu z wykorzystaniem przestarzałych technologii, działają efektywnie i z różnych przyczyn nie mogą być z tego działania wyłączone nazywamy systemami spadkowymi. Zastąpienie ich nowszymi systemami jest często zbyt kosztowne lub po prostu niemożliwe. Rozwój tego typu oprogramowania przysparza dużych trudności programistom.

Nowoczesne aplikacje bazodanowe wytwarzane w obiektowych językach programowania wykorzystują w większości relacyjne bazy danych. Takie rozwiązanie powoduje problem określany terminem *niezgodności impedancji*. Różnice w strukturze danych, sposobie ich przechowywania oraz możliwościach ich przeszukiwania to główne problemy towarzyszące połączeniu modelu obiektowego z relacyjnym. Jak zaproponowano w [1] problem niezgodności impedancji może zostać rozwiązany poprzez wykorzystanie tego samego modelu dla programowania oraz przechowywania danych. Ze względu na znaczną przewagę w możliwościach programowania języków obiektowych nad językami zapytań naturalnym wydaje się wybór modelu obiektowego. Rozwój obiektowych baz danych oraz języków zapytań znacząco wzrósł w ostatnich latach. Relacyjne bazy danych posiadają rozbudowany i efektywny język zapytań SQL, jednak niewiele obiektowych języków programowania posiada takie rozwiązania – najszerzej stosowane, to:

- LINQ (ang. *Language INtegrated Query*) – język zapytań dla platformy .NET
- SBQL4J (ang. *Structure-Based Query Language*) – język zapytań dla języka Java [2]

Definicja 1

Niezgodność impedancji [3] jest to zespół niekorzystnych cech towarzyszących formalnemu połączeniu modelu relacyjnego z modelem obiektowym. Różnice w strukturach danych, sposobie ich przechowywania oraz koncepcji języków obiektowych i języków zapytań powodują znaczne trudności techniczne w realizacji tego rodzaju połączenia. Niezgodność impedancji powoduje konieczność istnienia dodatkowej warstwy oprogramowania pośredniczącego pomiędzy modelami. □

Główne cechy niezgodności impedancji widoczne są w zakresie:

- Składni – programista zmuszony jest do znajomości i używania dwóch gramatyk językowych

- Systemu typów – różnice między systemami typów języków zapytań oraz programowania, takie jak np. typ relacja występujący w SQL oraz statyczna kontrola typów występująca w większości języków programowania
- Semantyki i paradygmatów językowych – języki zapytań bazują na semantyce o stylu deklaracyjnym, natomiast języki programowania na semantyce o tylu imperatywnym
- Poziomu abstrakcji – język zapytań izoluje programistę od wielu szczegółów organizacji i implementacji danych. Powoduje to dodatkowy nakład pracy, gdy zachodzi konieczność dostępu do danych poprzez standardowe konstrukcje języka programowania
- Faz i mechanizmów wiązania – języki zapytań są interpretowane (późne wiązanie), przez co problematycznym staje się wprowadzenie mocnej kontroli typów, implementacji narzędzi debugujących, itd.
- Przestrzeni nazw i reguł zakresu – język zapytań i język programowania posiadają własne przestrzenie nazw, a ich odwzorowanie wymaga dodatkowych środków syntaktycznych i semantycznych. Dodatkowo język zapytań ignoruje reguły zakresu języka programowania opartego na zasadzie stosu
- Traktowania wartości zerowych – w bazach danych i językach zapytań istnieją środki dla przechowywania i przetwarzania wartości zerowych, w językach programowania takie środki nie istnieją
- Schematów iteracyjnych – iterowanie po elementach zbioru w językach programowania wymaga użycia konstrukcji pętli takich jak *for*, *while*, *foreach* itd., co powoduje konieczność występowania udogodnień takich jak kursory i iteratory. Iteracja w języku zapytań jest hermetycznie zamknięta w semantyce operatorów, takich jak selekcja, widok, czy złączenie
- Traktowania cechy trwałości danych – języki zapytań operują na danych znajdujących się na dysku, podczas gdy języki programowania operują na danych przechowywanych w pamięci operacyjnej. Powoduje to konieczności transmisji danych z dysku do pamięci operacyjnej i w przeciwnym kierunku

Problem ten jest częściowo rozwiązywany poprzez stosowanie mapowania obiektowo-relacyjnego. Mapowanie obiektowo-relacyjne jest metodą konwersji reprezentacji modelu relacyjnego na obiektowy i w drugą stronę. Proces ten jest powtarzalny, dzięki czemu można

go zautomatyzować. Dzięki takiemu podejściu programista może manipulować danymi za pomocą instancji obiektów klas mapujących w wykorzystywanym języku programowania będąc jednocześnie uwolnionym od konieczności żmudnego pisania metod zapisu i wyłuskiwania danych z bazy. Jest to bardzo pożądana własność mapowania obiektowo-relacyjnego, ponieważ pozwala programiście skoncentrować się na implementacji funkcjonalności o wartości biznesowej.

Jeden z mechanizmów mapowania obiektowo-relacyjnego zakłada odwzorowanie struktury bazy danych w pliku konfiguracyjnym, na podstawie którego generowany jest odpowiedni kod bazujący na wzorcach projektowych oraz zapewniający istotne bezpieczeństwo. Podejście to jest bardzo dobrym rozwiązaniem w obiektowych językach programowania z nierozwiniętym, lub nieistniejącym mechanizmem refleksji. Pozwala możliwie bezszwowo usprawnić rozwój projektu.

Główne zalety używania narzędzi ORM (ang. *Object-Relational Mapping*):

- znacząco redukują ilość pracy związanej z oprogramowaniem manipulacji danymi
- uniezależniają programistę od dostawcy serwera bazy danych
- automatyzują obsługę transakcji i puli połączeń
- zwiększają wydajność operacji na bazie
- wiele innych

Stare, obiektowe języki programowania nie posiadają, lub posiadają mało rozwinięte mechanizmy mapowania obiektowo-relacyjnego. Najszerzej stosowanym językiem programowania zaliczającym się do powyższego zbioru jest Delphi. Na dzień dzisiejszy istnieje kilka narzędzi ORM dla tej technologii, jednak żadne z nich nie nadaje się do wykorzystania w systemach napisanych w starszych wersjach tego języka.

2.1. Delphi ORM

Delphi ORM jest najczęściej polecanym narzędziem ORM dla Delphi. Jego głównymi zaletami są:

- Licencja Open Source Apache License 2.0
- Implementuje wzorzec projektowy *Data Mapper*
- Wsparcie dla relacji *jeden-do-wielu*; *jeden-do-jednego*; *wiele-do-wielu*

- Konfiguracja w pliku (mapowanie)
- Wspieranie wielu dostawców serwera bazy danych
- Leniwe ładowanie
- Płynny interfejs dla zapytań
- Generowanie SQL z pliku konfiguracyjnego

Narzędzie to cieszy się dużą popularnością wśród programistów Delphi, dedykowane jest jednak dla nowszych wersji tego języka (XE lub wyższa) ze względu na wykorzystany mechanizm refleksji, który w Delphi pojawił się w wersji 2010, a rozbudowany został w wersji XE.

2.1.1. Wzorzec projektowy Data Mapper

Idea Delphi ORM oparta jest na wzorcu projektowym *Data Mapper*. Wzorzec ten zakłada separację obiektów od bazy danych dzięki pośrednikowi mapującemu operacje związane z pobieraniem i zapisywaniem danych w bazie lub obiekcie. Diagram 1 przedstawia przykładową strukturę wzorca z wykorzystaniem diagramu klas. Diagram 2 przedstawia diagram sekwencji dla przykładowej metody *find()*.

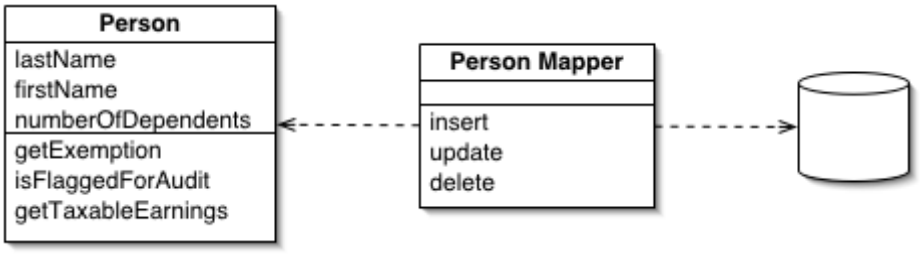


Diagram 1 - diagram architektury wzorca projektowego *Data Mapper Enterprise Design Pattern* [4]

Podstawowe cechy wzorca *Data Mapper*, to:

- Odseparowanie obiektów od bazy danych
- Utrwalenie/pobranie stanów obiektów z/do bazy danych
- Całkowita separacja między modelem domeny a bazą danych
- Obiekty „nie znają” maperów

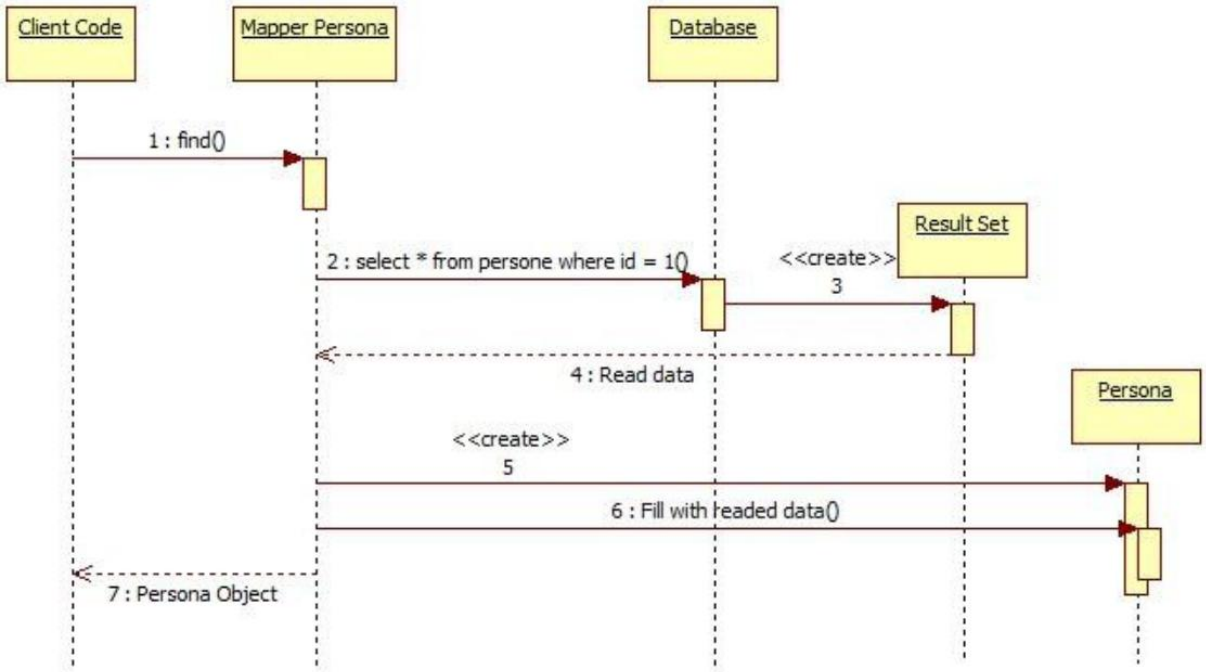


Diagram 2 - przykładowy diagram sekwencji wyszukiwania obiektu *Persona* [5]

2.1.2. Mapowanie poprzez plik konfiguracyjny

Delphi ORM udostępnia możliwość mapowania obiektów w pliku konfiguracyjnym JSON. Przykładowe mapowanie widoczne jest na listingu 1.

Listing 1 - przykładowe mapowanie w pliku JSON

```
"TEmail": {
 "has_one": [],
 "fields": [
  {
   "field_type": "string",
   "field": "ADDRESS",
   "size": 400,
   "name": "Address"
  },{
   "field_type": "integer",
   "field": "ID_PERSON",
   "name": "IdPerson"
  }],
 "package": "BusinessObjects",
 "has_many": [],
 "id": {
  "field_type": "integer",
  "field": "ID",
  "name": "Id"
 },
 "table": "EMAILS",
 "belongs_to": []
},
```

Mimo istniejącej możliwości mapowania obiektowo-relacyjnego z pliku, dokumentacja na ten temat nie istnieje, a samo narzędzie rozwijane jest pod kątem mapowania poprzez atrybuty.

2.1.3. Mapowanie poprzez atrybuty

Atrybuty w Delphi są odpowiednikiem adnotacji w Javie i atrybutów w C# – formą syntaktycznych metadanych w kodzie. Atrybuty dodaje się do klasy lub jej właściwości. Delphi ORM [6] przedstawia następujące atrybuty:

- Entity - określa klasę jako klasę encji, pozwalając na jej utrwalenie. Bez parametru klasa zostanie zapisana do tabeli o nazwie równej nazwie klasy bez przedrostka *T*

Listing 2 - atrybut Entity

```
[Entity]
TPerson = class
public
  property id: integer;
  property Name: String;
  property Address: String;
  property Age: Integer;
end;
```

W innym wypadku klasa zostanie zmapowana do tabeli o nazwie równej określonemu parametrowi

Listing 3 - atrybut Entity z parametrem

```
[Entity('PEOPLE')]
TPerson = class
public
  property id: integer;
  [Column('FULL_NAME', 60)]
  property Name: String;
  property Address: String;
  property Age: integer;
end;
```

- Id – określa właściwość jako klucz główny tabeli

Listing 4 - atrybut Id

```
[Entity]
TPerson = class
public
  [Id]
  property Identifier: integer;
  property Name: String;
  property Address: String;
  property Age: integer;
end;
```

- Column – określa nazwę kolumny w tabeli dla właściwości

Listing 5 - atrybut Column

```
[Entity]
TPerson = class
public
  [Id]
  [Column('Identifier')]
  property MyIdentifier: integer;
  property Name: String;
  property Address: String;
  property Age: integer;
end;
```

- Transient – określa, że właściwość klasy ma nie być utrwalana

Listing 6 - atrybut Transient

```
[Entity]
TPerson = class
private
  FAge: Integer;
  function getIsAdult: boolean;
public
  property id: integer;
  property Name: String;
  property Address: String;
  property Age: integer;
  [Transient]
  property IsAdult: boolean read GetIsAdult;
end;

function TPerson.getIsAdult;
begin
  Result := FAge > 18;
end;
```

- HasMany – określa relację *jeden-do-wielu* oraz *wiele-do-jednego*. Atrybut ten przyjmuje dwa parametry – nazwę klasy oraz właściwości do których odnosi się relacja

Listing 7 - atrybut HasMany

```
[Entity]
TPerson = class
public
  property id: integer;
  property Name: String;
  property Address: String;
  property Age: integer;
  [HasMany(TPhone, 'OwnerID')]
  property Phones: TObjectList<TPhone>;
end;
[Entity('PHONES')]
TPhone = class
protected
  [Column('PARENT_PERSON_ID')]
  property OwnerID: integer;
public
  property id: integer;
  property Number: String;
end;
```

2.1.4. Przykłady zastosowania Delphi ORM

Poniżej przedstawione zostaną listingi przykładowego zastosowania narzędzia Delphi ORM, w tym: tworzenie, zapis, usuwanie oraz tworzenie na podstawie kryteriów.

Listing 8 – Tworzenie, aktualizacja i usuwanie obiektów

```
var
  person: TPerson; oid: Integer;
begin
  person := Session.Load<TPerson>(1);
  try
    person.FirstName := 'Daniele';
    person.LastName := 'Teti';
    Session.Update(person);
    Session.Delete(person);
  finally
    person.Free;
  end;
end;
```

Listing 8 przedstawia metody pobierania, aktualizacji i usuwania danych z bazy używając do tego celu klasy mapującej oraz obiektu sesji.

Listing 9 - Tworzenie obiektów na podstawie kryteriów

```
var
  criteria: TdormCriteria;
  people: TdormCollection;
begin
  criteria := TdormCriteria.
    NewCriteria('FirstName',
      TdormCompareOperator.Equal, 'Daniele').
    Add('LastName',
      TdormCompareOperator.Different, 'Smith');
  people := Session.List<TPerson>(criteria);
  //operacje na people
end;
```

Przykład z listingu 9 prezentuje zastosowanie pomocniczej klasy kryteriów do tworzenia obiektów.

2.2. TMS Aurelius

TMS Aurelius jest najbardziej zaawansowanym narzędziem ORM dla Delphi dostępnym na rynku. W przeciwieństwie do Delphi ORM jest płatny, ale oferuje znacznie bardziej rozbudowane funkcjonalności [7]:

- Zapis, aktualizacja i ładowanie encji obiektów w sposób zorientowany obiektowo
- Wsparcie dla wielu dostawców serwera baz danych (MS SQL Server, Firebird, MySQL, DB2, Interbase, Oracle, …)
- Wsparcie dla wielu komponentów dostępu do serwera bazy danych (dbExpress, AnyDac, SQLDirect, ADO, IBX, …)
- Rozwiązanie wieloplatformowe – Win32, Win64, Mac OS X, VCL, FireMonkey
- Tworzenie kryteriów w sposób podobny do obiektowego języka zapytań LINQ
- Przezroczystość – wykorzystanie jednego kodu dla wielu serwerów baz danych
- Narzędzie TMS Data Modeler pozwala na utworzenie bazy danych na podstawie klas, lub utworzenie klas na podstawie bazy danych
- Leniwe ładowanie

- Cachowanie obiektów
- Wsparcie dla typów Nullable
- Logowanie zapytań SQL

TMS Aurelius, podobnie jak Delphi ORM, opiera się na mechanizmie refleksji. Główną siłą tego podejścia jest możliwość mapowania klas w samym kodzie za pomocą atrybutów.

2.2.1. Mapowanie poprzez atrybuty

Mapowanie obiektowo-relacyjne w TMS Aurelius realizowane jest za pomocą atrybutów. Z takim podejściem można definiować mapowanie bezpośrednio podczas implementacji klas, a podczas przeglądania kodu można z łatwością dowiedzieć się jak klasa jest mapowana do bazy danych.

Atrybuty dodaje się do klasy, jej pól, lub właściwości:

Listing 10 - Przykładowe atrybuty klasy

```
[Table('Customer')]
TMyCustomer = class
private
  [Column('Customer_Name')]
  FCustomerName: String;
end;
```

Najważniejsze atrybuty mapowania:

- Entity – określa klasę jako klasę encji, pozwalając na jej utrwalenie

Listing 11 - atrybut Entity

```
[Entity]
TCustomer = class(TObject)
```

- Id – określa unikalny identyfikator obiektu. Każdy obiekt musi być unikalnie identyfikowany przez Aureliusa, aby mógł być zapisywany i zarządzany.

Listing 12 - atrybut Id

```
[Id('FId', TIdGenerator.IdentityOrSequence)]
TCustomer = class(TObject)
private
  [Column('CUSTOMER_ID')]
  FId: Integer;
```

- Table – określa tabelę w bazie danych w której zostanie utrwalony obiekt

Listing 13 - atrybut Table

```
[Table('Customers')]
TCustomer = class(TObject)
end;

[Table('Orders', 'dbo')]
TOrder = class(TObject)
end;
```

- Column – określa kolumnę tabeli w której zostanie utrwalone pole/właściwość

Listing 14 - atrybut Column

```
[Column('MEDIA_NAME', [TColumnProp.Required], 100)]
property MediaName: string read FMediaName write FMediaName;

[Column('DURATION', [])]
property Duration: Nullable<integer> read FDuration write FDuration;
```

- Sequence – określa algorytm używany do generowania identyfikatorów

Listing 15 - atrybut Sequence

```
[Sequence('SEQ_MEDIA_FILES')]
[Id('FId', TIdGenerator.IdentityOrSequence)]
TMediaFile = class
```

- UniqueKey – określa unikalny indeks dla tabeli

Listing 16 - atrybut UniqueKey

```
[UniqueKey('INVOICE_TYPE, INVOICENO')]
TTC_Invoice = class
```

- Enumeration – określa jak mają być zapisywane typy wyliczeniowe

Listing 17 - atrybut Enumeration

```
[Enumeration(TEnumMappingType.emChar, 'M,F')]
TSex = (tsMale, tsFemale);
```

Atrybuty mapowania asocjacji:

- Association – określa relację *wiele-do-jednego*

Listing 18 - atrybut Association

```
[Association([], CascadeTypeAll)]
[JoinColumn('ID_SONG_FORMAT', [])]
property SongFormat: TSongFormat read FSongFormat write FSongFormat;

[Association([TAssociationProp.Lazy], [TCascadeType.SaveUpdate])]
[JoinColumn('ID_ARTIST', [])]
FArtist: Proxy<TArtist>;
```

- JoinColumn – określa kolumnę jako klucz obcy

Listing 19 - atrybut JoinColumn

```
[Association([TAssociationProp.Lazy], [])]
[JoinColumn('ID_ARTIST', [])]
FArtist: Proxy<TArtist>;
```

2.2.2. Automapowanie

Mapowanie obiektowo-relacyjne w TMS Aurelius możliwe jest również bez konieczności definiowania atrybutów mapowania. W tym celu należy dodać do klasy atrybut *Automapping*. Duża część mapowania jest realizowana przez Aurelius na podstawie metadanych klasy. Przykładowo nazwa tabeli, jeżeli nie jest bezpośrednio określona przez atrybut *Table*, zostanie określona jako nazwa klasy bez przedrostka *T*.

2.2.3. Manipulacja obiektami

Manipulowanie obiektami realizowane jest za pomocą klasy menadżera obiektów *TObjectManager*. Aby skorzystać z menadżera obiektów należy go utworzyć przekazując obiekt połączenia z bazą danych jako atrybut konstruktora klasy:

Listing 20 – tworzenie instancji menadżera obiektów

```
Manager := TObjectManager.Create(MyConnection);
try
  // perform operations with objects
finally
  Manager.Free;
end;
```

Najistotniejsze metody klasy *TObjectManager*, to:

- Save – używana do zapisu nowych encji w bazie danych

Listing 21 – zapis nowej encji za pomocą menadżera obiektów

```
Customer := TCustomer.Create;
Customer.Name := 'TMS Software';
Manager.Save(Customer);
```

- Update – używana do aktualizacji istniejącej encji tabeli w bazie danych

Listing 22 – aktualizacja istniejącej encji za pomocą menadżera obiektów

```
Customer := TCustomer.Create;
Customer.Id := 10;
Customer.Email := 'customer@company.com';
Manager.Update(Customer);
```

- SaveOrUpdate – używana do zapisu nowej, lub aktualizacji istniejącej encji w tabeli w bazie danych na podstawie identyfikatora klasy

Listing 23 – zapis lub aktualizacja encji za pomocą menadżera obiektów

```
Customer.LastName := 'Smith';
Manager.SaveOrUpdate(Customer);
```

- Find – używana do pozyskiwania obiektów na podstawie ich identyfikatorów

Listing 24 – wyszukiwanie obiektów na podstawie identyfikatora

```
Customer := Manager.Find<TCustomer>(CustomerId);
```

- Remove – używana do usuwania encji z tabeli

Listing 25 – usuwanie obiektów za pomocą menadżera obiektów

```
CustomerToRemove := Manager.Find<TCustomer>(CustomerId);
Manager.Remove(CustomerToRemove);
```

- CreateCriteria – używana do pozyskania zbioru encji z tabeli na podstawie kryteriów wyszukiwania

Listing 26 – tworzenie kryteriów wyszukiwania za pomocą menadżera obiektów

```
Results := Manager.CreateCriteria<TTC_Customer>
  .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
  .List;
```

2.2.4. Kryteria wyszukiwania

TMS Aurelius wprowadza pomocniczą klasę *TCriteria* do definicji kryteriów wyszukiwania obiektów w bazie danych. Za jej pomocą można określać filtrowanie, sortowanie, widoki, itd. Klasa *TCriteria* realizuje ideę płynnego interfejsu, tzn. większość metod klasy zwraca w rezultacie instancję samej siebie, dzięki czemu możliwe jest eleganckie i czytelne definiowanie kryteriów. Zamiast pisać:

Listing 27 – tworzenie kryteriów bez użycia płynnego interfejsu

```
var
  Result: TObjectList<TCustomer>;
  Criteria: TCriteria<TCustomer>;
  Filter: TCustomCriteria;
begin
  Criteria := Manager1.CreateCriteria<TCustomer>;
  Filter := TExpression.Eq('Name', 'Mia Rosenbaum');
  Criteria.Add(Filter);
  Results := Criteria.List;
```

Można zdefiniować kryteria w następujący sposób:

Listing 28 – tworzenie kryteriów z wykorzystaniem płynnego interfejsu

```
var
  Result: TObjectList<TCustomer>;
begin
  Results := Manager1.CreateCriteria<Customer>
    .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
    .List;
```

Główną funkcjonalnością klasy *TCriteria* jest filtrowanie rezultatów. Aby przefiltrować wyniki należy wywołać metodę *Add()* przekazując jako parametr instancję obiektu *TCustomCriterion*. Do tworzenia obiektu tej klasy wykorzystywana jest klasa pomocnicza *TExpression* realizująca wzorzec projektowy *Factory*. Udostępnia ona większość wymaganych wyrażeń SQL (równe; większe od; większe równe; like; isNull; itd.)

Listing 29 – filtrowanie wyników za pomocą dodatkowych kryteriów

```
Results := Manager1.CreateCriteria<Customer>
  .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
  .List;
```

2.3. ECO

ECO (ang. Enterprise Core Objects) jest narzędziem wytwarzanym przez firmę Capable Objects, dostosowanym do *Domain-Driven Design*.

Definicja 2

Domain-Driven Design (DDD) [8] jest to podejście do tworzenia oprogramowania zorientowane na definiowanie obiektów i komponentów systemu oraz ich zachowania w sposób wiernie odzwierciedlający rzeczywistość. Dopiero po utworzeniu modelu konceptualnego przez ekspertów dziedzinowych nieznających się na projektowaniu architektury systemów informatycznych następuje techniczna realizacja systemu. □

Zadaniem ECO [9] jest zwiększenie produktywności poprzez wykorzystanie diagramów UML w definiowaniu klas konceptualnych (diagram klas) oraz kontroli zachowania (diagram stanów). Udostępnia takie usługi jak transakcje, funkcjonalność cofnij/wróć, czy łatwe łączenie z interfejsem użytkownika. Jedną z jego głównych zalet jest brak konieczności definiowania modelu bazy danych, dzięki mapowaniu obiektowo-relacyjnemu. Model relacyjnej bazy danych nie jest bezpośrednio znany programiście a mapowanie realizowane jest transparentnie w wygenerowanym kodzie. Niemniej jednak zaleta ta jest wielką wadą w perspektywie wykorzystania ECO w systemach spadkowych, gdzie bazy danych są już zamodelowane i wypełnione danymi. Transformacja bazy lub przeniesienie danych na nowy model wygenerowany przez narzędzie może być niezwykle trudne lub niewykonalne. Kolejnym problemem w kontekście systemów spadkowych w technologii Delphi jest brak rozwoju dla tego języka w wersjach większych od 4.

Podejście DDD ma wiele zalet, przede wszystkim pozwala dobrze określić dziedzinę problemu przez osoby z nią związane, dzięki czemu realizowany system jest dobrze rozumiany zarówno przez klienta jak i programistów, a architektura jest przejrzysta i łatwiejsza do zrozumienia. ECO wykorzystuje to podejście, ale można odnieść wrażenie, że twórcy chcą stworzyć narzędzie, które wyeliminuje konieczność zatrudniania informatyków.

Pewna część wytwarzania oprogramowania realizowana jest przez klienta, ale istnieje potrzeba ingerencji programistów przy bardziej złożonych funkcjonalnościach systemu. Na chwilę obecną takie rozwiązanie wydaje się tylko mrzonką.

2.4. Wady istniejących rozwiązań

Przedstawione rozwiązania realizują ideę mapowania obiektowo-relacyjnego, w dużej mierze rozwiązują problem niezgodności impedancji, jednak bazują one na mechanizmie refleksji, który wprowadzony został dopiero w nowszych wersjach języka Delphi, lub w przypadku ECO, nie oferują możliwości wygenerowania kodu dla istniejącego modelu bazy danych. Wykorzystanie ich w systemach spadkowych staje się przez to niemożliwe. Delphi ORM wprowadza wprawdzie możliwość definiowania bazy danych w osobnym pliku konfiguracyjnym JSON, jednak brakuje dokumentacji na ten temat, a samo narzędzie dedykowane jest dla wersji języka XE lub wyższej (głownie testowane na wersji XE2).

Do celów rozwoju systemów spadkowych dzięki narzędziu ORM konieczne jest uniwersalne podejście, które pozwoli na wygenerowanie kodu działającego w starszej technologii, realizującego przy tym funkcjonalności istniejących rozwiązań opartych na mechanizmie refleksji.

3. Narzędzia i koncepcje użyte w pracy

Niniejszy rozdział zawiera opis technologii, wzorców i narzędzi wykorzystanych w realizacji założeń mapera obiektowo-relacyjnego dla systemów spadkowych oraz jego prototypu.

3.1. Wzorce projektowe GRASP

Tworząc architekturę klas mapujących wzięto pod uwagę wykorzystanie wzorców projektowych GRASP (ang. *General Responsibility Assignment Software Patterns*) w projektach korzystających z mapera. Struktura wygenerowanego kodu jest przejrzysta i łatwa do zrozumienia dla programistów z doświadczeniem w tworzeniu aplikacji bazodanowych w paradygmacie obiektowym, dzięki czemu projektowanie aplikacji zgodnie z zasadami wzorców GRASP jest intuicyjne i łatwe w realizacji.

Definicja 3

Wzorzec projektowy [10] jest uniwersalnym, sprawdzonym w praktyce rozwiązaniem często pojawiających się, powtarzalnych problemów projektowych. Jest opisem rozwiązania a nie jego implementacją. □

Lista wzorców wchodzących w zbiór wzorców GRASP, to:

- Twórca
- Ekspert
- Niskie Sprzężenie
- Wysoka Spójność
- Kontroler
- Polimorfizm
- Czysty wymysł
- Pośrednictwo
- Ochrona Zmienności

Są one podstawą dla dobrej analizy i projektowania obiektowego. Poniżej opisane są najistotniejsze z nich [11].

3.1.1. Twórca

Jedną z najczęściej wykonywanych czynności w systemach obiektowych jest tworzenie obiektów. Do wykonania większości operacji systemowych wymagane jest utworzenie kilku, lub kilkunastu instancji różnych klas. Powstaje więc pytanie: którym klasom przydzielić odpowiedzialność za ich tworzenie? Dobry wybór poprawi hermetyzację, czytelność oraz możliwość ponownego wykorzystania kodu.

Wzorzec Twórca wprowadza cztery warunki, na podstawie których można zidentyfikować najlepszego kandydata na rolę twórcy:

- *B* zawiera *A* lub agreguje *A* w sposób złożony (kompozycja)
- *B* zapamiętuje *A*
- *B* bezpośrednio używa *A*
- *B* posiada dane inicjalizacyjne dla *A*, które zostaną przekazane do *A* po jego utworzeniu

Im więcej z powyższych warunków spełnia dana klasa, tym lepiej. Po identyfikacji mówi się, że *B jest twórcą obiektów A*

3.1.2. Ekspert

Podczas projektowania obiektowego należy określić, która klasa będzie odpowiedzialna za wykonanie danej operacji systemowej. Operacje często potrzebują informacji rozproszonych w wielu klasach programistycznych. Wzorzec ekspert podpowiada, by klasa realizująca operację znała możliwie jak najwięcej informacji – bezpośrednio, lub poprzez asocjacje. Dzięki takiemu podejściu projekt jest bardziej spójny, a ilość powiązań między klasami ograniczona jest do minimum. Są to pożądane cechy projektu, o czym traktują wzorce *niskie sprzężenie* oraz *wysoka spójność*.

3.1.3. Niskie Sprzężenie

Zasada niskiego sprzężenia polega na zminimalizowaniu ilości powiązań między obiektami w celu zmniejszenia liczby zależności i zasięgu zmian oraz zwiększenia możliwości ponownego użycia kodu.

Definicja 4

Stopień sprzężenia [11] (ang. coupling) to miara siły, z jaką pewien obiekt jest połączony z innymi elementami lub od nich zależny. Element z niskim (słabym) stopniem sprzężenia nie zależy od zbyt wielu elementów. □

Klasy z dużym (silnym) stopniem sprzężenia prowadzą do następujących problemów:

- Lokalne zmiany wymuszone przez zmiany w powiązanych klasach
- Kod trudny do zrozumienia w izolacji (bez odniesienia do innych klas)
- Kod trudny do ponownego wykorzystania – wymagane jest załączenie wielu powiązanych klas

3.1.4. Wysoka Spójność

Zasada wysokiej spójności nakazuje projektowanie klas, które mają jasny cel, są zrozumiałe i łatwe w utrzymaniu, przy okazji zmniejszając ich stopień sprzężenia. Klasy o niskiej spójności charakteryzują się dużą ilością kodu przez zbyt duży nakład niepowiązanych ze sobą zadań. Prowadzi to do następujących problemów:

- Trudno zrozumieć ich kod i cel
- Trudno je ponownie wykorzystać
- Trudno je utrzymywać i rozwijać
- Są delikatne i podatne na wpływ zmian

3.1.5. Polimorficzne wołanie metod

Polimorfizm jest jedną z najważniejszych cech obiektowości. Jego użycie znajduje zastosowanie w sytuacjach, kiedy wybór ścieżki postępowania zależy od typu (klasy). Początkujący programiści często stosują w takich sytuacjach wyrażenia warunkowe *if-else,* przez co kod „pęcznieje" i staje się wrażliwy na zmiany (wzorzec *ochrona zmienności*). Użycie operacji polimorficznych pozwala przydzielić zobowiązania związane z danym zachowaniem typom, dla których to zachowanie jest różne.

3.2. Delphi

Delphi jest językiem programowania wysokiego poziomu realizującym paradygmat programowania obiektowego. Jego składnia opiera się na składni języka Object Pascal, z którego powstał. Język posiada mocną kontrolę typów, pozwala na dziedziczenie z jednej klasy i implementację wielu interfejsów. Każda bazowa klasa dziedziczy z metaklasy *TObject*.

Autorem Delphi jest firma Borland, która rozwijała język w latach 1995 – 2008. W 2006 roku sekcja firmy Borland związana z rozwojem narzędzi dla deweloperów została przetransferowana do firmy zależnej o nazwie CodeGear, która została w całości wykupiona przez firmę Embarcadero w roku 2008. W roku 2001, przy okazji premiery Delphi w wersji 6, w dokumentacji po raz pierwszy użyto terminu *Delphi Language*. W przeciwieństwie do języka Pascal, który powstał w celach edukacyjnych, Delphi jest produktem komercyjnym, którego celem było połączenie prostoty i przejrzystości języka Pascal z łatwym i wygodnym tworzeniem aplikacji w zintegrowanym środowisku programistycznym. Kod Delphi kompilowany jest do natywnego kodu x86 lub zarządzalnego kodu .NET [12]

Najistotniejsze cechy języka Delphi:

- Komponenty wspomagające obsługę relacyjnych systemów bazodanowych
- Obsługa standardowych mechanizmów windowsowych, np. COM/ActiveX
- Duży zestaw komponentów graficznych i nie tylko
- Budowa wizualnej części aplikacji za pomocą techniki *drag and drop*
- Szybki, efektywny kompilator

Tabela 1 przedstawia historię wydań języka Delphi.

Wersja	Rok	Nazwa wydania	Nazwa kodowa
1.0	1995	Borland Delphi	Delphi 1
2.0	1996	Borland Delphi 2	Delphi 2
3.0	1997	Borland Delphi 3	Delphi 3
4.0	1998	Inprise Delphi 4	Delphi 4
5.0	1999	Borland Delphi 5	Delphi 5
6.0	2001	Borland Delphi 6	Delphi 6
7.0	2002	Borland Delphi 7	Delphi 7

8.0	2003	Borland Delphi 8	Delphi 8
9.0	2004	Borland Delphi 2005	Delphi 9
10.0	2005	Borland Delphi 2006	Delphi 10
11.0	2007	Borland Delphi 2007	Delphi 11
12.0	2008	Embarcadero Delphi 2009	Tiburón
14.0	2009	Embarcadero Delphi 2010	Weaver
-	2010	Embarcadero Delphi XE	Fulcrum
-	2011	Embarcadero Delphi XE2	Pulsar
-	2012	Embarcadero Delphi XE3	Delphi XE3

Tabela 1- historia wydań Delphi

Poniżej przedstawione zostały podstawowe pojęcia paradygmatu obiektowego w ujęciu Delphi [13].

- Klasa – określa strukturę obiektów, danych i operacji, jakie można wykonać na takich obiektach
- Obiekt – część programu komputerowego wykonująca określone zadanie
- Metoda – *procedura* lub *funkcja* będąca składnikiem klasy
- Funkcja – blok kodu wykonujący jakąś czynność i zwracający wynik pod swoją nazwą
- Procedura – blok kodu wykonujący jakąś czynność i niezwracający wyniku
- Procedura obsługi zdarzenia – fragment kodu, który wywoływany jest w wyniku zajścia jakiegoś zdarzenia
- Zdarzenie – zachodzi w wyniku interakcji *komponentu* z użytkownikiem lub systemem
- Komponenty – są to części, z których budowane są programy
- Moduł – jest to plik tekstowy, który może być kompilowany do programu wykonywalnego

Aplikacje w Delphi pisane są w modułach – plikach tekstowych z rozszerzeniem *pas*. Moduły mają określoną strukturę – każdy moduł składa się z kilku sekcji:

- *Unit* – nazwa modułu, wpisywana w pierwszej linijce pliku – musi być taka sama jak nazwa pliku z modułem

- *Interface* – sekcja zawierająca nazwy wszystkich elementów modułu (wewnętrznych oraz zewnętrznych)
- *Uses* – sekcja zawierająca nazwy modułów dodatkowych
- *Type* – sekcja zawierająca nazwy użytych w module obiektów i metod oraz definicje własnych struktur i klas
- *Var* – sekcja zawierająca nazwy użytych w module zmiennych
- *Implementation* – zawiera implementację zdefiniowanych metod w sekcji *interface*. Mogą się w niej znaleźć również sekcje *uses* oraz *var*.

Listing 30 - przykładowy moduł w Delphi

```
unit MasterThesis;

interface

uses
  Math;

type
  TMasterThesis = class
  private
    fAttribute: String;
  public
    procedure setAttribute(val: String);
    function getAttribute(): String;
  end;

var
  MyMasterThesis: TMasterThesis;

implementation

{ TMasterThesis }

function TMasterThesis.getAttribute: String;
begin
  Result := fAttribute;
end;

procedure TMasterThesis.setAttribute(val: String);
begin
  fAttribute := val;
end;

end.
```

Moduł przedstawiony na listingu 30 zawiera deklarację i implementację klasy *TMasterThesis*. Nazwa modułu to *MasterThesis* i wykorzystuje on zewnętrzny moduł *Math*. W sekcji *type* zadeklarowane są dwie metody, których implementacja znajduje się w sekcji *implementation*. Sekcja *var* zawiera deklarację zmiennej o typie *TMasterThesis* – taka deklaracja spowoduje utworzenie obiektu podczas uruchomienia aplikacji, który widoczny będzie wszędzie, gdzie dołączony zostanie moduł *MasterThesis*. Cały moduł zakończony jest słowem kluczowym *end* z kropką na końcu – jest to identyfikator końca modułu.

Warto zwrócić uwagę na kilka elementarnych różnic w składni Delphi w porównaniu z innymi językami obiektowymi:

- Bloki kodu otoczone są parą słów kluczowych *begin* (początek bloku) oraz *end* ze średnikiem na końcu (koniec bloku)
- Metody zadeklarowane są z pomocą różnych słów kluczowych – *function* definiuje metodę zwracającą jakiś wynik, *procedurę* definiuje metodę niezwracającą wyniku
- Przypisanie wartości do zmiennej wykonywane jest za pomocą operatora := (dwukropek i znak równości)
- Wartość zwracana przez funkcję *getAttribute* przypisywana jest do zarezerwowanej przez Delphi zmiennej *Result*, która jest wynikiem metody
- Operatory logiczne:
    - Równy - =
    - Różny od - <>
    - Większy/równy - >=
    - Mniejszy/równy - <=
    - Większy - >
    - Mniejszy - <
    - Zaprzeczenie – *NOT*

Elementy GUI (ang. *Graphic User Interface*) w Delphi są zwykłymi modułami, z odpowiednim wpisem w sekcji *implementation* odwołującym się do pliku z zapisanymi właściwościami graficznymi obiektu. Plik ten posiada rozszerzenie *dfm* i tworzony jest

automatycznie przez zintegrowane środowisko programistyczne. Listing 31 przedstawia przykładowy kod pliku *dfm*.

Listing 31 - przykładowy kod pliku *dfm*

```
object Form4: TForm4
  Left = 0
  Top = 0
  Caption = 'Form4'
  ClientHeight = 300
  ClientWidth = 635
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Panel1: TPanel
    Left = 0
    Top = 0
    Width = 635
    Height = 300
    Align = alClient
    BevelOuter = bvNone
    TabOrder = 0
    ExplicitLeft = 208
    ExplicitTop = 104
    ExplicitWidth = 185
    ExplicitHeight = 41
  end
end
```

3.3. RAD Studio XE3

RAD Studio XE3 to zintegrowane środowisko programistyczne (IDE) dla języka Delphi XE3 realizujące założenia szybkiego tworzenia aplikacji (RAD).

Definicja 5

Zintegrowane środowisko programistyczne (ang. *Integrated Development Environment*, IDE) jest to aplikacja, lub zespół aplikacji (środowisko) służących do tworzenia, modyfikowania, testowania i konserwacji oprogramowania. □

RAD jest rozwinięciem koncepcji IDE. Programiści korzystający z RAD Studio XE3 mogą pracować wydajniej dzięki takim funkcjonalnościom, jak:

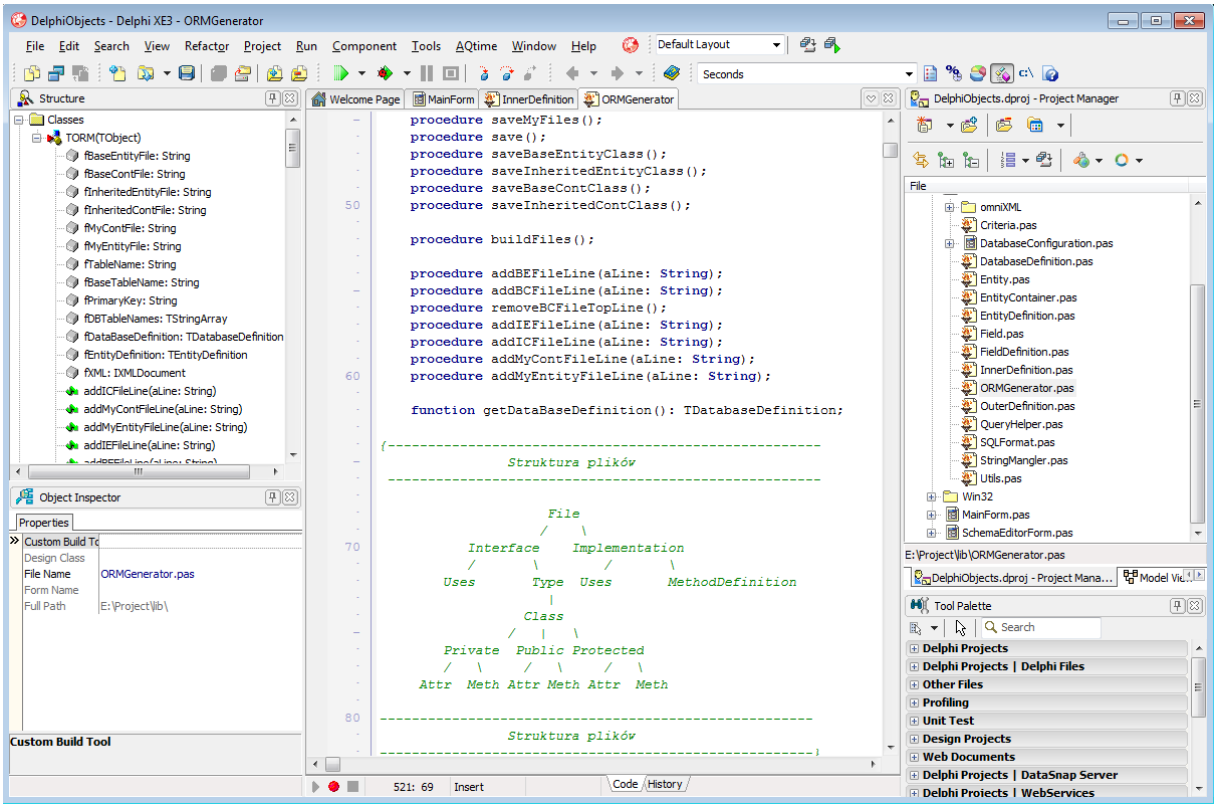
- Środowisko do debugowania tworzonych aplikacji
- System szybkiej podpowiedzi kodu
- Platforma tworzenia testów jednostkowych
- Narzędzie do profilowania i optymalizacji kodu
- Szereg gotowych komponentów (dostęp do bazy danych, komponenty GUI, WebServices, wiele innych)

RAD Studio XE3 wspiera wytwarzanie oprogramowania na platformy 32 i 64-bitowe systemów Windows (również w najnowszej wersji 8) oraz dla systemu Apple Mac OS X z wykorzystaniem zestawu komponentów GUI FireMonkey. Planowane jest wprowadzenie wsparcia dla systemów Linux oraz Android.

Definicja 6

Szybkie tworzenie aplikacji (ang. *Rapid Application Development*, RAD) jest to ideologia polegająca na udostępnieniu programiście dużych możliwości prototypowania oraz dużego zestawu gotowych komponentów (np. zapewniających dostęp do bazy danych). □

Rysunek 1 przedstawia przykładowy widok środowiska.



Rysunek 1- RAD Studio XE3

3.4. Microsoft SQL Server

Microsoft SQL Server jest systemem zarządzania bazą danych stworzonym i rozwijanym przez firmę Microsoft. Realizuje założenia relacyjnych baz danych, takie jak:

- Transakcje
- Więzy integralnościowe – klucze główne i obce, wartości domyślne
- Kontrola typów
- Poziomy bezpieczeństwa – np. prawa dostępu użytkowników do baz, tabel, krotek czy kolumn
- Backup danych
- Wyzwalacze
- Procedury składowane
- Widoki

Jako język zapytań w systemie wykorzystywany jest przede wszystkim Transact-SQL. Wspiera wiele typów danych – numeryczne, daty i czasu, łańcuchy znaków (w tym UNICODE), dane binarne (np. obrazy) i inne.

SQL Server jest jednym z najpopularniejszych systemów bazodanowych nie tylko ze względu na dobre wsparcie i ciągły rozwój, ale również dzięki dużej efektywności i stabilności. Tabela 2 przedstawia historię wydań systemu.

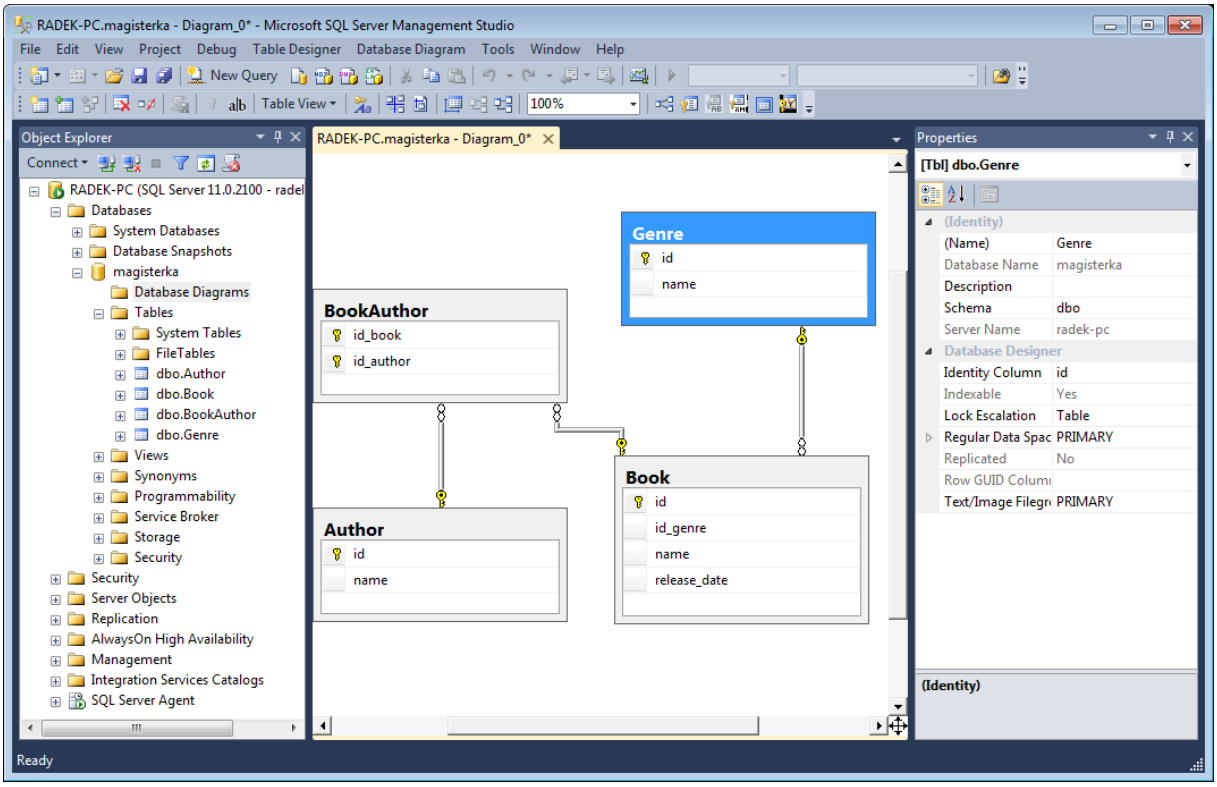
Wersja	Rok	Nazwa wydania	Nazwa kodowa
1.0 (OS/2)	1989	SQL Server 1.0	-
4.21 (WinNT)	1993	SQL Server 4.21	-
6.0	1995	SQL Server 6.0	SQL95
6.5	1996	SQL Server 6.5	Hydra
7.0	1998	SQL Server 7.0	Sphinx
-	1999	SQL Server 7.0 OLAP Tools	Plato
8.0	2000	SQL Server 2000	Shiloh
8.0	2003	SQL Server 2000 64-bit Edition	Liberty
9.0	2005	SQL Server 2005	Yukon
10.0	2008	SQL Server 2008	Katmai

10.50	2010	SQL Server 2008 R2	Kilimanjaro
11.00	2012	SQL Server 2012	Denali

Tabela 2 - Historia wydań Microsoft SQL Server

3.5. Microsoft SQL Server Management Studio

Microsoft SQL Server Management Studio jest potężnym narzędziem pozwalającym konfigurować, zarządzać i administrować komponentami Microsoft SQL Server (rysunek 2). Udostępnia edytor skryptów oraz graficzne narzędzia np. do budowy diagramów. Jego głównym elementem jest eksplorator obiektów, w którym można swobodnie przeglądać, pobierać i działać na każdym obiekcie serwera bazy danych.



Rysunek 2 - Microsoft SQL Server Management Studio 2012

4. Propozycja mapera dla systemów spadkowych

Mapery obiektowo-relacyjne pełnią wdzięczną rolę w nowoczesnych aplikacjach bazodanowych. Można znaleźć wiele narzędzi wykorzystujących najnowsze cechy obiektowych języków programowania, jednak maper dla systemów spadkowych musi uwzględniać ograniczenia nakładane przez stare technologie w nich wykorzystane.

4.1. Maper jako oddzielna aplikacja

Podczas tworzenia nowoczesnych aplikacji korzysta się ze zintegrowanych środowisk programistycznych (IDE) udostępniających szereg narzędzi wspomagających programistów w efektywnym tworzeniu oprogramowania. Większość dostępnych na rynku IDE umożliwia tworzenie rozszerzeń – komponentów integrowanych ze środowiskiem. Po zainstalowaniu rozszerzenia programista ma szybki dostęp do jego usług w obrębie środowiska.

Jednym z wielu czynników wpływających na utrzymywanie systemów spadkowych w starych technologiach jest cena za ich aktualizację. W przypadku języka Delphi jest to bezpośrednio powiązane z aktualizacją IDE. Ze względu na różnice między wydaniami w systemie rozszerzeń wtyczka napisana dla starszej wersji może nie współpracować z nowszym wydaniem środowiska. Wykonanie mapera jako oddzielnej aplikacji pozwala w szybki i łatwy sposób wprowadzić narzędzie na rynek bez problemów związanych z różnymi wersjami środowisk wykorzystywanych przez programistów.

Mimo wymienionych zalet narzędzia w postaci oddzielnej aplikacji, wtyczki IDE są pożądanym elementem dla każdego programisty. Są to graficzne komponenty zintegrowane ze środowiskiem programowania. Wtyczki dla środowiska RAD Studio implementowane są w języku Delphi jako interfejsy graficzne oparte na interfejsach COM. Tworząc wtyczki programista musi posiadać wiedzę na temat modułu *ToolsAPI*. Delphi nie udostępnia żadnej dokumentacji dla tego modułu, a w Internecie trudno znaleźć rzetelne i aktualne informacje na temat tworzenia wtyczek (tutaj [14] opisano system wtyczek dla Delphi 3). Na oficjalnej stronie producenta [15] można znaleźć szczątkowe informacje na temat wtyczek dla środowiska RAD Studio wersji 8.

4.2. Mapowanie poprzez plik konfiguracyjny

Większość nowoczesnych narzędzi mapujących opiera się na mechanizmie refleksji. Wymaga to odpowiedniego sposobu nazewnictwa i przestrzegania wielu zasad określonych

przez wykorzystywane narzędzie. Zaletą takiego podejścia, nazywanego „konwencja ponad konfigurację”, jest wyeliminowanie plików konfiguracyjnych, które często stają się dodatkowym źródłem błędów.

Technologie nieposiadające mechanizmu refleksji nie pozwalają na tak wygodne mapowanie obiektów. Mapowanie poprzez plik konfiguracyjny jest jedynym rozwiązaniem w takim przypadku. Sposób ten posiada drobną przewagę nad podejściem „konwencja ponad konfigurację” – programista nie musi tworzyć klasy z konstruktorami, określać pól wymagających utrwalenia ani tworzyć do nich metod dostępowych get set, ponieważ wygenerowany kod zawiera wszystko powyższe.

4.3. Generowanie i rozszerzanie kodu

Narzędzie mapujące powinno dostarczać gotowych rozwiązań dostępu do bazy danych – wygenerowany kod powinien implementować metody tworzenia, pobierania, zapisu i usuwania danych z bazy. Aplikacje bazodanowe nie ograniczają się jednak jedynie do realizacji operacji CRUD (ang. *Create Read Update Delete*). W większości przypadków należy rozszerzyć klasy o dodatkowe operacje realizujące funkcjonalności systemu. W przypadku zmian w modelu danych i ponownym wygenerowaniu kodu wcześniejsze zmiany nie mogą być utracone. Cel ten można osiągnąć stosując dziedziczenie, co zostało przedstawione na diagramie 3.

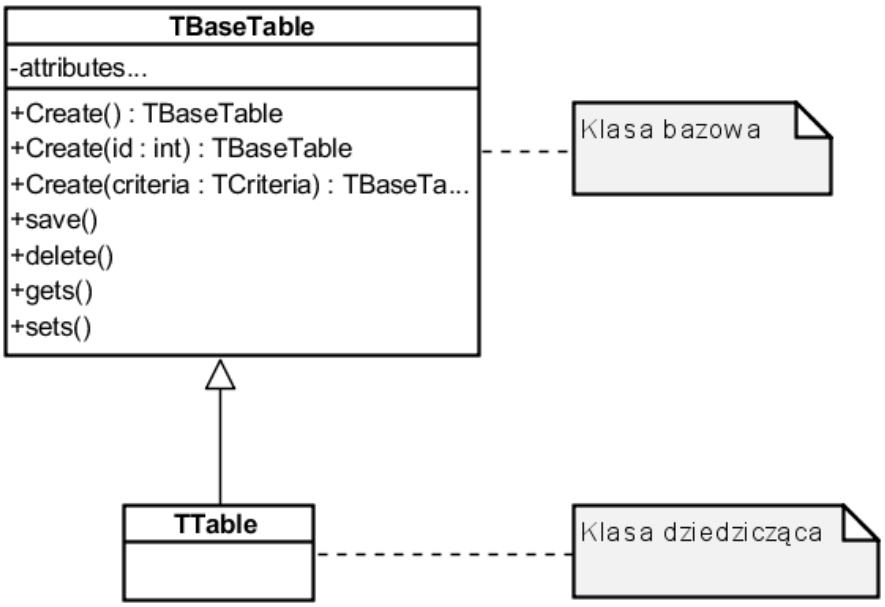


Diagram 3 – przykładowa struktura wygenerowanych klas

Wygenerowany kod zawiera zestaw klas bazowych oraz klas dziedziczących:

- **Klasa bazowa**: jest to klasa generowana każdorazowo podczas mapowania. Dokładny opis klasy bazowej zawarty jest w podrozdziale *Mapowanie tabel*
- **Klasa dziedzicząca**: jest to klasa generowana jednorazowo podczas pierwszego mapowania tabeli

Dzięki temu podejściu programista może rozszerzać klasę dziedziczącą o dodatkowe funkcjonalności, a ewentualne zmiany w modelu i ponowne wygenerowanie klas bazowych nie spowoduje ich utraty.

4.4. Mapowanie tabel

Narzędzia mapujące model obiektowy na model relacyjny pozwalają na dużo swobody przy projektowaniu architektury systemu i hierarchii klas. W przypadku mapowania z modelu relacyjnego na obiektowy rozwiązaniem jest mapowanie w relacji *jeden-do-jednego* – każda tabela posiada własną reprezentację w postaci klasy programistycznej.

Tabela w relacyjnych bazach danych składa się z pól, które powinny być zmapowane na atrybuty klasy o odpowiednim typie. Mapowanie typów danych opisane jest w podrozdziale *Mapowanie typów danych*. Dla każdego pola powinna istnieć reprezentacja w postaci prywatnego atrybutu klasy wraz z metodami dostępowymi (get, set) do niego.

Istotną kwestią podczas mapowania tabel jest informacja o kluczach głównych i obcych. Dzięki identyfikacji tych elementów możliwe jest wygenerowanie konstruktorów klas przyjmujących identyfikatory wierszy tabeli w celu utworzenia obiektu dla odpowiedniej encji (opis pobierania danych zawiera podrozdział 4.7) oraz wygenerowanie atrybutów o odpowiednim typie w przypadku relacji definiowanych przez klucze obce. Mapowanie relacji opisane jest w podrozdziale *Mapowanie relacji.*

4.5. Mapowanie relacji

Relacyjne bazy danych definiują relacje za pomocą kluczy obcych. W modelu obiektowym odpowiednikiem relacji jest referencja do obiektu. Mapowanie obiektowo-relacyjne wymaga połączenia obu rozwiązań. Dla każdego klucza obcego powinna być wygenerowana para atrybutów *identyfikator* (reprezentujący identyfikator encji w tabeli) oraz *obiekt* (odpowiadający typem zmapowanej klasie będącej w relacji na podstawie danego

klucza oraz będący obiektem reprezentującym encję tabeli o konkretnym identyfikatorze). Identyfikator tabeli w bazie danych powinien pełnić rolę identyfikatora obiektu w modelu obiektowym. Nie jest to jednak referencja obiektu, tylko jednoznaczny identyfikator, na podstawie którego można pobrać dane z bazy i nasycić nimi instancję klasy. Dla zilustrowania mapowania różnych typów relacji wykorzystany jest przykład modelu bazy danych przedstawiony na diagramie 4.

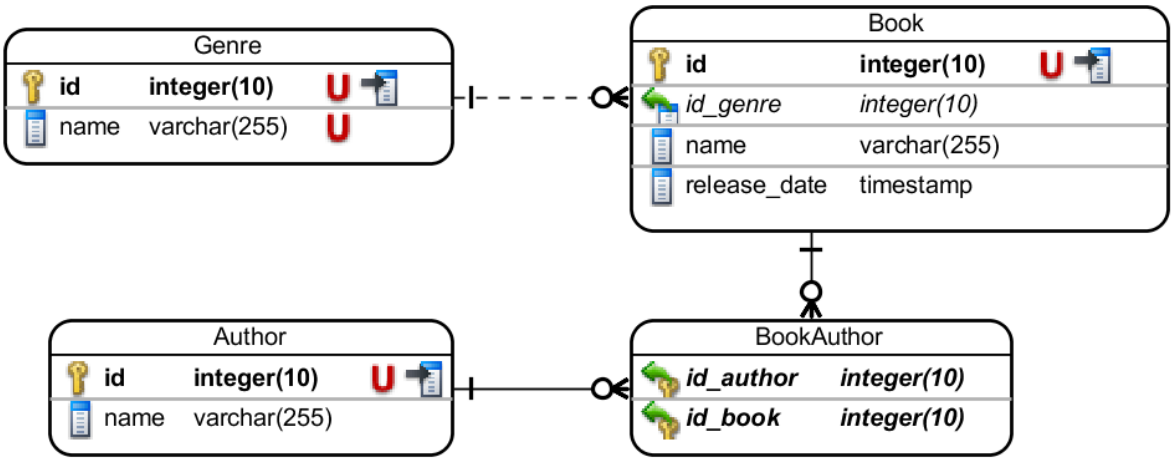


Diagram 4 - przykładowy model danych

4.5.1. Relacje *jeden-do-wielu*

Relacje *jeden-do-wielu* określają, że dana encja tabeli może posiadać wiele powiązanych encji innej tabeli. Tabela *TableW* będąca w relacji po stronie *wiele* zawiera klucz obcy wskazujący na klucz główny tabeli *TableJ* po stronie *jeden*. Mówi się wtedy, że jedna encja tabeli *TableJ* może być w relacji z wieloma encjami tabeli *TableW*. Na diagramie 2 relacja ta istnieje między tabelami *Genre* oraz *Book*, czyli jeden gatunek (ang. *genre*) książki może być w relacji z wieloma książkami (ang. *book*). Zmapowana klasa *TBook* dla tego przykładu powinna zawierać atrybut reprezentujący identyfikator (klucz) klasy *TGenre* w postaci liczby naturalnej oraz atrybut reprezentujący obiekt o typie *TGenre*.

4.5.2. Relacje *wiele-do-jednego*

Relacje *wiele-do-jednego* są odwrotnością relacji *jeden-do-wielu*. W bazie danych ich realizacja jest identyczna jak dla relacji *jeden-do-wielu*, jednak interpretacja następuje od tabeli zawierającej klucz obcy w kierunku tabeli zawierającej klucz główny. Dla omówionego przykładu z książkami i gatunkami książek brzmi ona następująco: wiele książek może być w

relacji z jednym gatunkiem. W tym przypadku zmapowana klasa *TGenre* powinna zawierać kolekcję obiektów klasy *TBook*.

4.5.3. Relacje *wiele-do-wielu*

Relacyjne bazy danych nie udostępniają generycznych metod definiowania relacji *wiele-do-wielu*. Aby osiągnąć ten cel stosowane są tabele pośredniczące. Tabele pośredniczące zawierają klucze obce tabel będących w relacji *wiele-do-wielu*. W obiektowych językach programowania relacja ta realizowana jest za pomocą kolekcji obiektów w każdej z klas będących w relacji. Tabele pośredniczące mogą jednak zawierać dodatkowe pola, przez co koniecznym jest utworzenie dodatkowej klasy ją mapującej. Na diagramie … tabele *Book* oraz *Author* są ze sobą w relacji *wiele-do-wielu* za pomocą tabeli pośredniczącej *BookAuthor*. Zmapowane klasy *TBook* oraz *TAuthor* powinny zawierać kolekcje obiektów klasy *TBookAuthor*, natomiast klasa *TBookAuthor* powinna zawierać pary atrybutów *identyfikator* oraz *obiekt* dla klas *TBook* oraz *TAuthor*. Dzięki takiemu podejściu chcąc np. pobrać wszystkie książki danego autora (ang. *author*) należałoby pobrać kolekcję obiektów *TBookAuthor* i dla każdej instancji pobrać obiekt *TBook*.

4.6. Mapowanie typów danych

Różnice w typach danych między modelem relacyjnym a obiektowym są jedną z cech niezgodności impedancji. Odpowiednia konwersja pozwala na prawidłowy zapis danych z obiektów do relacyjnej bazy danych i w drugą stronę. Tabela 3 przedstawia typy danych Microsoft SQL Server oraz ich odpowiedniki w języku Delphi.

MS SQL Server	Delphi
tinyint, smallint, int, bigint	byte, smallint, integer, int64
bit	boolean
numeric, decimal, smallmoney, money	currency
real, float	single, double
datetime	TDateTime
char, text, varchar, nchar, nvarchar, ntext	String

Tabela 3 – porównanie typów danych między MS SQL Server i Delphi

4.7. Logika operacji CRUD

Zmapowane klasy powinny dostarczać podstawowe metody do operacji CRUD, na które składają się tworzenie, pobieranie, zapis i usuwanie danych.

- Tworzenie – instancja obiektu reprezentującego nową encję tabeli powinna być tworzona przy użyciu konstruktora nieprzyjmującego żadnych parametrów (identyfikatora lub kryteriów)
- Pobieranie – pobieranie danych powinno być możliwe na dwa sposoby: poprzez podanie identyfikatora encji lub odpowiednich kryteriów wyszukania obiektu. Utworzenie obiektu reprezentującego istniejącą encję powinno odbywać się na zasadzie wywołania konstruktora obiektu przyjmującego identyfikator lub kryteria
- Zapis – operacja zapisu danych powinna tworzyć nowe encje w bazie danych oraz zapisywać zmiany w już istniejących krotkach tabel. Obiekt powinien wykonać odpowiedni proces zapisu na podstawie informacji o identyfikatorze
- Usuwanie – operacja usuwania powinna usuwać z bazy danych nie tylko encję reprezentowaną przez obiekt, na którym wywołana została metoda *delete*, ale również wszystkie encje będące w kompozycji z usuwaną krotką

5. Prototyp mapera

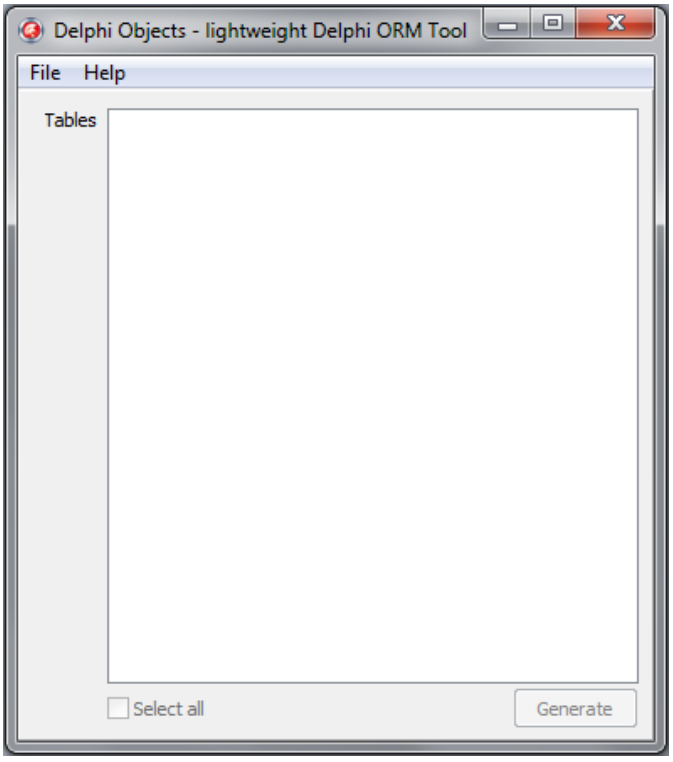
Prototyp mapera został wykonany zgodnie z założeniami opisanymi w rozdziale 4. W niniejszym rozdziale opisane zostało korzystanie z narzędzia i wygenerowanych klas oraz przedstawione zostały wybrane szczegóły implementacyjne prototypu.

5.1. Aplikacja

Prototyp aplikacji jest lekkim narzędziem realizującym następujące funkcjonalności:

- Ładowanie pliku XML ze schematem bazy danych
- Sprawdzanie poprawności schematu bazy danych
- Generowanie klas dla wybranych tabel ze schematu bazy danych do wskazanego katalogu

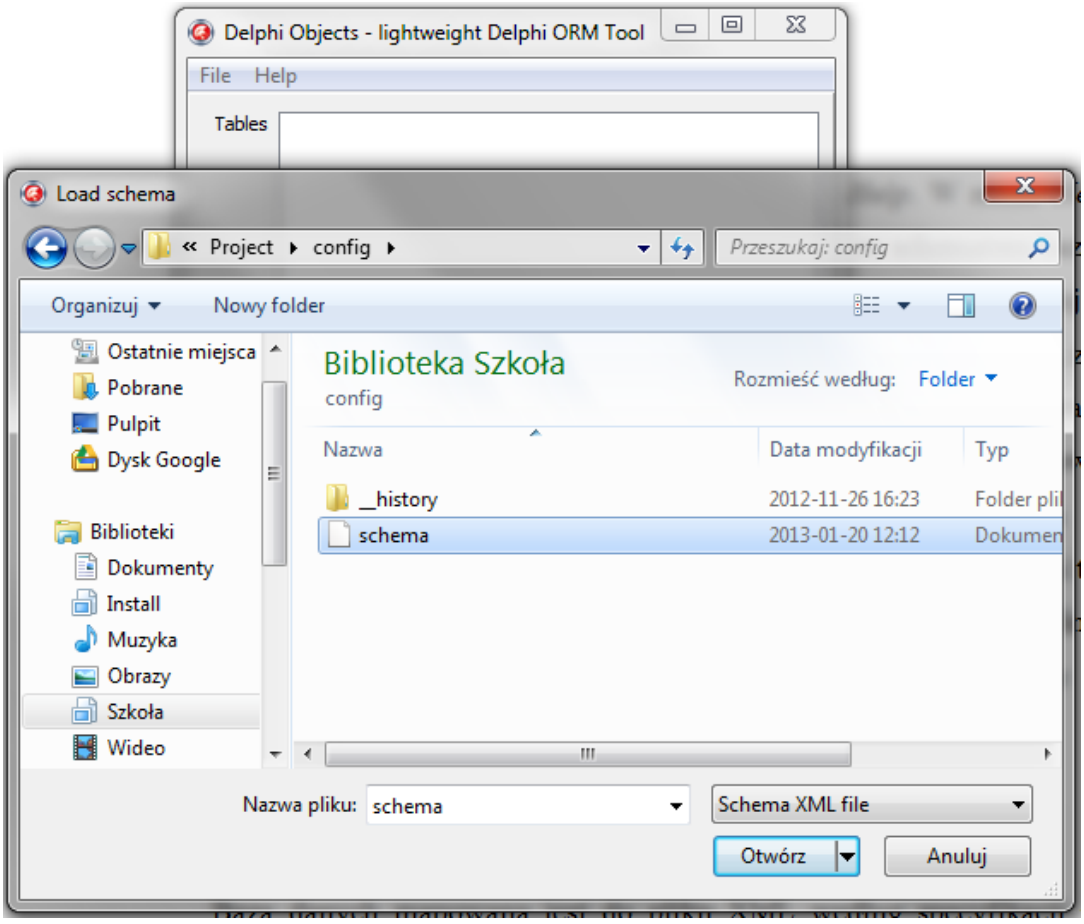
Na rysunku 3 przedstawiono główne okno aplikacji. Menu aplikacji zawiera dwie pozycje: *File* oraz *Help*.



Rysunek 3 - Prototyp mapera Delphi Objects

W menu *Help* można znaleźć informacje o aplikacji (*About*), specyfikację pliku XML ze schematem bazy danych (*Schema info*) oraz instrukcję korzystania z aplikacji (*How-to*). W

menu *File* znajduje się pozycja *Load, po której* wybraniu pojawia się okno dialogowe do wskazania pliku XML ze schematem bazy danych (rysunek 4). Po wskazaniu pliku aplikacja automatycznie sprawdza poprawność schematu zgodnie ze specyfikacją. W razie błędów użytkownik informowany jest o nich w wyskakującym oknie. Jeżeli schemat jest poprawny pojawia się komunikat o prawidłowym wczytaniu pliku, pole *Tabele* zostaje wypełnione nazwami tabel wykrytych w schemacie, a przyciski *Generate* oraz *Select all* stają się aktywne. Definiowanie schematu bazy danych oraz generowanie kodu zostało opisane w podrozdziałach *Mapowanie bazy danych* oraz *Generowanie kodu*.



Rysunek 4 - ładowanie pliku XML

5.2. Mapowanie bazy danych

Baza danych mapowana jest do pliku XML według specyfikacji przedstawionej na listingu 32.

Listing 32 - struktura i dozwolone elementy pliku XML ze schematem bazy danych

```
<database>
    <table>
        <name></name>
        <fields>
            <field>
                <name></name>
                <type></type>
                <foreignTable></foreignTable>
                <foreignKey></foreignKey>
                <primaryKey></primaryKey>
            </field>
        </fields>
    </table>
</database>
```

Poniżej opisane są poszczególne elementy schematu:

- <*database*> – jest to korzeń schematu, reprezentujący bazę danych. Może zawierać jeden, lub więcej elementów <*table*>
- <*table*> – reprezentuje tabelę w bazie danych i może definiować jej nazwę <*name*> (jako nierozdzielny ciąg znaków) oraz pola <*fields*>. W schemacie może pojawić się tylko jedno wystąpienie elementu <*table*> o tej samej nazwie. Każda tabela musi posiadać co najmniej jedno pole oznaczone jako klucz główny (<*primaryKey*>)
- <*fields*> może zawierać jeden, lub więcej elementów <*field*>, reprezentujących pojedyncze pola tabeli, o tej samej nazwie. Każdy element <*field*> składa się z następujących elementów:
    - <*name*> – nazwa pola – nierozdzielny ciąg znaków
    - <*type*> – typ danych pola: dozwolone typy danych:
        - Integer
        - Real
        - String
        - Datetime
        - Boolean

- <*foreignTable*> – opcjonalnie dla klucza obcego: nazwa tabeli, do której odnosi się klucz obcy
- <*foreignKey*> – opcjonalnie dla klucza obcego: nazwa pola w tabeli, do której odnosi się klucz obcy
- <*primaryKey*> – opcjonalnie dla klucza głównego: wartość *true*;

5.2.1. Przykładowy plik konfiguracyjny

W celu utworzenia przykładowego pliku schematu bazy danych widocznego na listingu 33 posłużył model bazy danych przedstawiony na diagramie 4.

W pliku zdefiniowano cztery tabele

- Genre – zawiera dwa pola:
  - Id: Integer { klucz główny }
  - Name: String
- Author – zawiera dwa pola:
  - Id: Integer { klucz główny }
  - Name: String
- Book – zawiera cztery pola:
  - Id: Integer { klucz główny }
  - Id_genre: Integer { klucz obcy: Genre.id }
  - Name: String
  - Release_date: Datetime
- BookAuthor – zawiera dwa pola:
  - Id_book: Integer { klucz główny; klucz obcy: Book.id }
  - Id_author: Integer { klucz główny; klucz obcy: Author.id }

Listing 33 - przykładowy plik konfiguracyjny

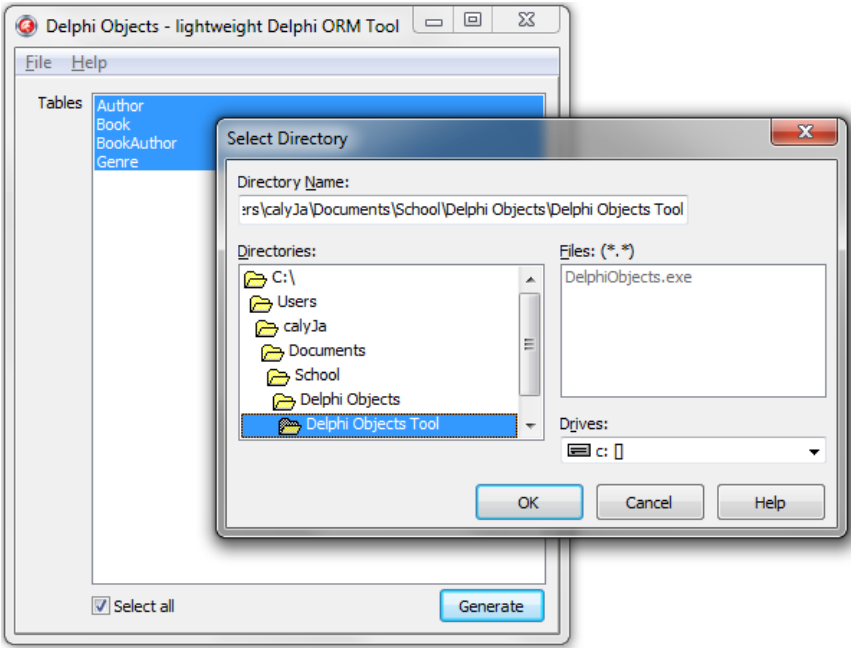
```
<database>
    <table>
        <name> Genre </name>
        <fields>
            <field>
                <name> id </name>
                <type> integer </type>
                <primaryKey> true </primaryKey>
            </field>
            <field>
                <name> name </name>
                <type> string </type>
            </field>
        </fields>
    </table>
    <table>
        <name> Book </name>
        <fields>
            <field>
                <name> id </name>
                <type> integer </type>
                <primaryKey> true </primaryKey>
            </field>
            <field>
                <name> id_genre </name>
                <type> integer </type>
                <foreignTable> Genre </foreignTable>
                <foreignKey> id </foreignKey>
            </field>
            <field>
                <name> name </name>
                <type> string </type>
            </field>
            <field>
                <name> release_date </name>
                <type> datetime </type>
            </field>
        </fields>
    </table>
    <table>
        <name> Author </name>
```

```
        <fields>
            <field>
                <name> id </name>
                <type> integer </type>
                <primaryKey> true </primaryKey>
            </field>
            <field>
                <name> name </name>
                <type> string </type>
            </field>
            <field>
                <name> height </name>
                <type> real </type>
            </field>
        </fields>
    </table>
    <table>
        <name> BookAuthor </name>
        <fields>
            <field>
                <name> id_book </name>
                <type> integer </type>
                <foreignTable> Book </foreignTable>
                <foreignKey> id </foreignKey>
                <primaryKey> true </primaryKey>
            </field>
            <field>
                <name> id_author </name>
                <type> integer </type>
                <foreignTable> Author </foreignTable>
                <foreignKey> id </foreignKey>
                <primaryKey> true </primaryKey>
            </field>
        </fields>
    </table>
</database>
```

Na podstawie przykładowego pliku konfiguracyjnego wygenerowane zostaną cztery klasy mapujące encje tabel oraz cztery klasy kontenerowe reprezentujące zbiory konkretnych encji. Dokładny opis generowania kodu przedstawiony jest w podrozdziale *Generowanie kodu*.

5.3. Generowanie kodu

Po wczytaniu pliku XML ze schematem bazy danych na liście *Tables* w oknie aplikacji pojawią się wykryte tabele. Aby wykonać mapowanie należy zaznaczyć tabele, dla których narzędzie ma wygenerować kod klas. Można to zrobić przytrzymując klawisz *Ctrl* i naciskając lewy klawisz myszy na odpowiednich elementach listy *Tables*, lub zaznaczając pole wyboru *Select all* w celu zaznaczenia wszystkich elementów listy. Po zaznaczeniu należy nacisnąć przycisk *Generate*. Aplikacja wyświetli okno dialogowe do wyboru katalogu, w którym umieszczone zostaną wygenerowane klasy (rysunek 5).



Rysunek 5 - okno wyboru katalogu do umieszczenia wygenerowanych klas

5.3.1. Umiejscowienie wygenerowanego kodu

Podczas pierwszego generowania kodu w miejscu wskazanym przez użytkownika zostaną utworzone dwa katalogi: *base* oraz *general*. W katalogu *base* umieszczone zostaną bazowe klasy mapujące, natomiast w katalogu *general* dwie klasy: *TMyEntity* oraz *TMyEntityContainer*, po których dziedziczą klasy bazowe. Klasy te służą do globalnego rozszerzania funkcjonalności klas reprezentujących encje tabel oraz klas kontenerowych. W głównym katalogu zapisane będą klasy rozszerzające klasy bazowe – to w nich programiści rozszerzać będą funkcjonalności pojedynczych klas. Dla przykładowego pliku konfiguracyjnego wygenerowane zostaną następujące klasy:

- *TMyEntity* – klasa, w której należy rozszerzać metody dla wszystkich obiektów reprezentujących encję tabeli
- *TMyEntityContainer* – klasa, w której należy rozszerzać metody dla wszystkich obiektów reprezentujących zbiór encji tabeli
- *TBaseAuthorEntity* – klasa bazowa dla encji tabeli *Author*
- *TBaseAuthorContainer* – klasa bazowa dla zbioru encji tabeli *Author*
- *TBaseBookEntity* – klasa bazowa dla encji tabeli *Book*
- *TBaseBookContainer* – klasa bazowa dla zbioru encji tabeli *Book*
- *TBaseGenreEntity* – klasa bazowa dla encji tabeli *Genre*
- *TBaseGenreContainer* – klasa bazowa dla zbioru encji tabeli *Genre*
- *TBaseBookAuthorEntity* – klasa bazowa dla encji tabeli *BookAuthor*
- *TBaseBookAuthorContainer* – klasa bazowa dla zbioru encji tabeli *BookAuthor*
- *TAuthor* – klasa dziedzicząca po *TBaseAuthorEntity*
- *TAuthorContainer* – klasa dziedzicząca po *TBaseAuthorContainer*
- *TBook* – klasa dziedzicząca po *TBaseBookEntity*
- *TBookContainer* – klasa dziedzicząca po *TBaseBookContainer*
- *TGenre* – klasa dziedzicząca po *TBaseGenreEntity*
- *TGenreContainer* – klasa dziedzicząca po *TBaseGenreContainer*
- *TBookAuthor* – klasa dziedzicząca po *TBaseBookAuthorEntity*
- *TBookAuthorContainer* – klasa dziedzicząca po *TBaseBookAuthorContainer*

Wygenerowane klasy wraz z hierarchią dziedziczenia przedstawione zostały na diagramie 8. W przypadku zmiany modelu należy ponownie wygenerować kod – zaleca się generowanie klas dla wszystkich tabel, można jednak zaznaczyć tylko te tabele, w których doszło do zmian. W przypadku tabel, dla których mapowanie wykonane było już wcześniej, wygenerowane zostaną jedynie klasy bazowe w celu aktualizacji zmian – klasy dziedziczące nie zostaną nadpisane. Dla nowych tabel wygenerowane zostaną zarówno klasy bazowe, jak i klasy dziedziczące. Klasy *TMyEntity* oraz *TMyEntityContainer* generowane są jednorazowo, podczas pierwszego generowania.

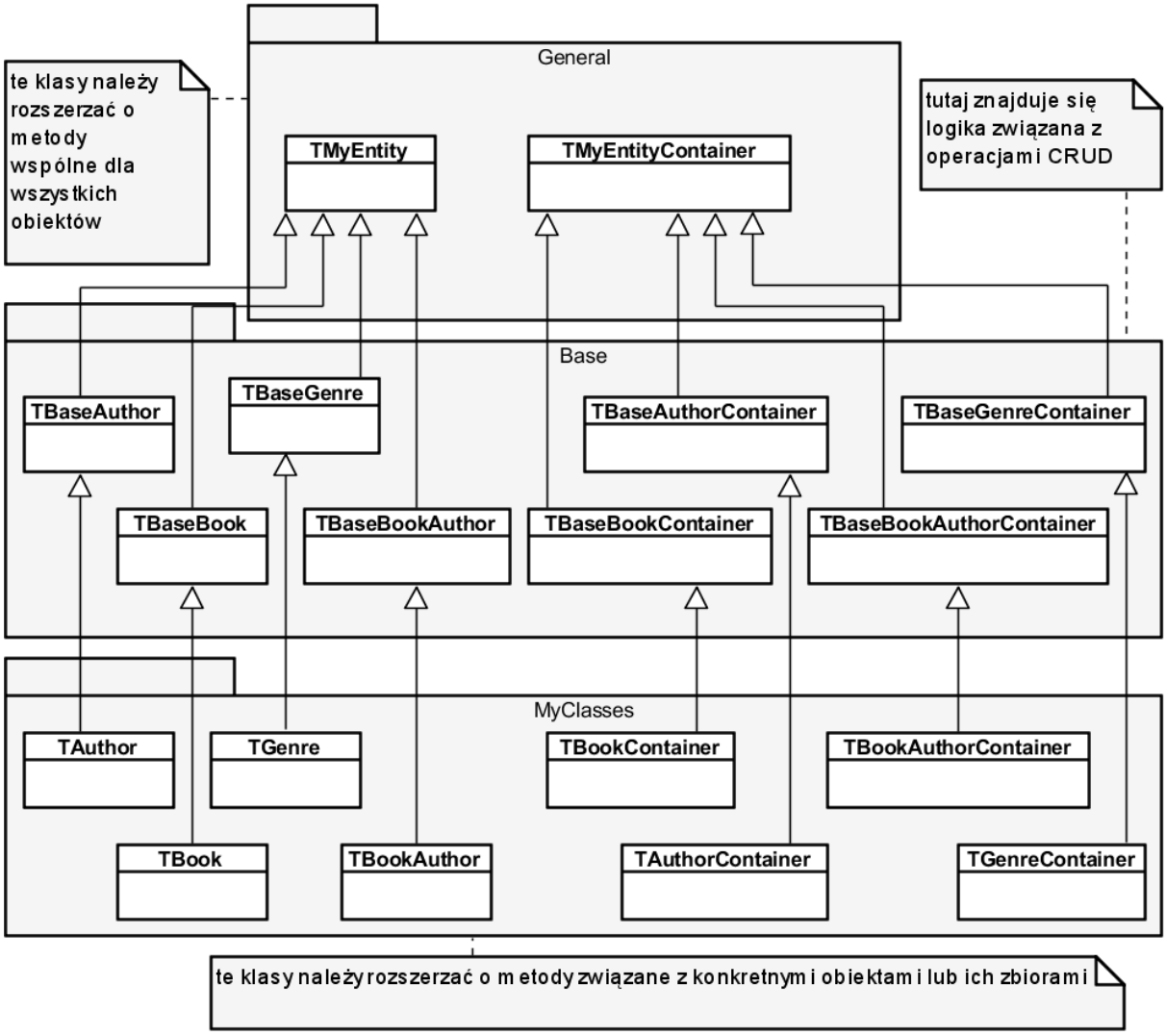


Diagram 5 – wygenerowane klasy dla przykładowego mapowania bazy danych

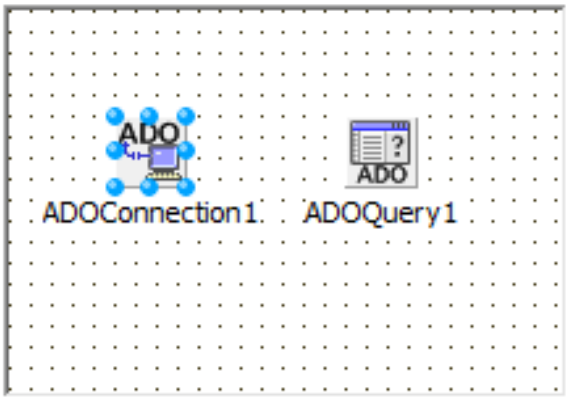
5.3.2. Dołączenie klas do projektu

Po wygenerowaniu klas należy dołączyć je do projektu Delphi w IDE Delphi Studio. Można to zrobić za pomocą menu *Project - Add to project...* lub skrótu klawiszowego *Shift + F11*. Poza klasami wygenerowanymi należy dodać klasy pomocnicze z katalogu *lib* mapera.

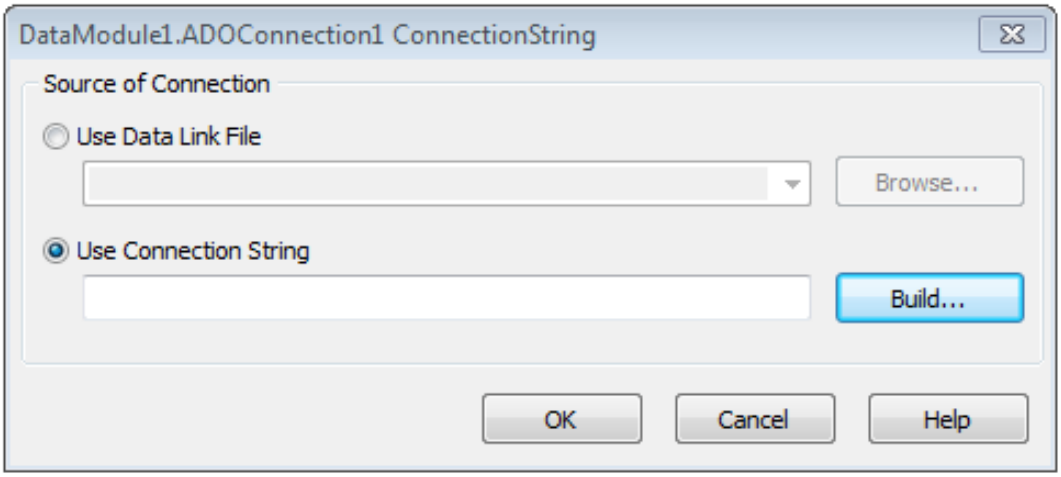
Po dodaniu klas należy skonfigurować połączenie bazy danych (podrozdział 5.3.3).

5.3.3. Konfiguracja połączenia

Połączenie definiowane jest w komponencie *DatabaseConfiguration*. Aby zdefiniować połączenie należy otworzyć komponent w IDE oraz kliknąć dwukrotnie w ikonę *ADOConnection1* (rysunek 6). Pojawi się okno konfiguracji połączenia (rysunek 7).



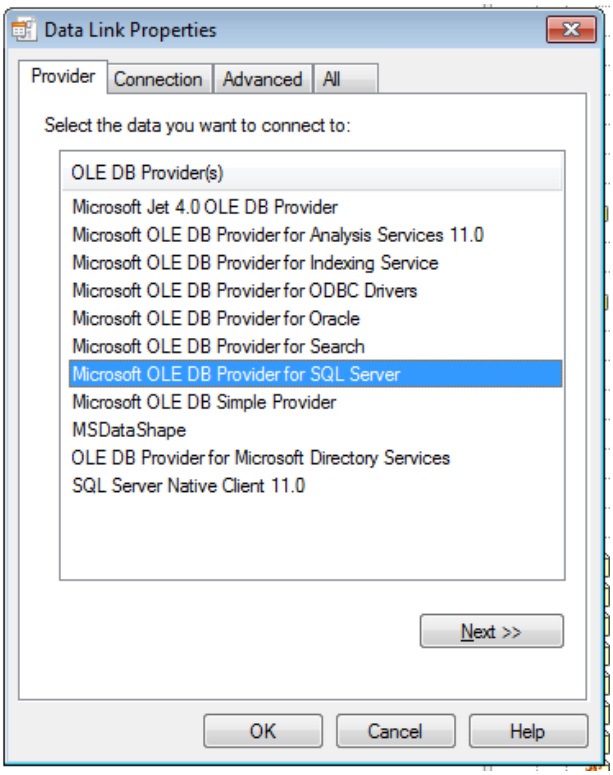
Rysunek 6 - komponent *DatabaseConfiguration*



Rysunek 7 – okno do wprowadzenia łańcucha połączenia z bazą danych

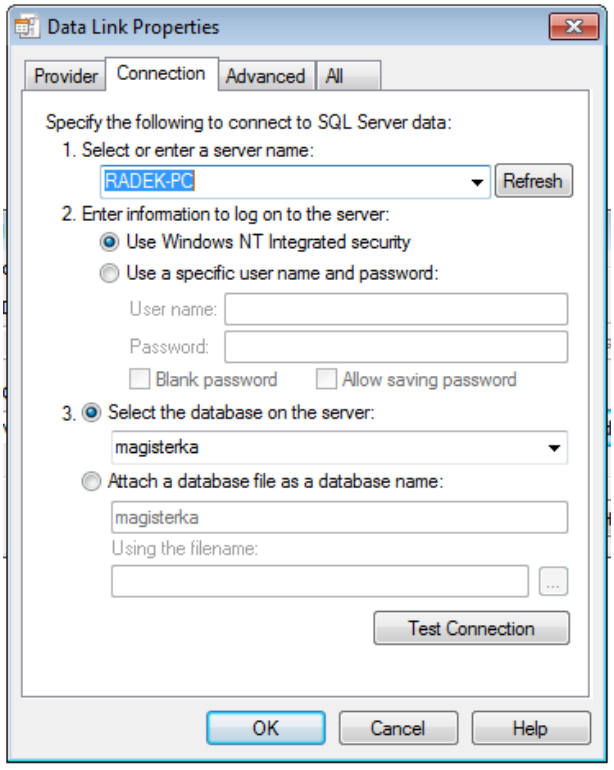
Połączenie można zdefiniować wpisując odpowiedni ciąg znaków, lub wykorzystując narzędzie do automatycznego generowania połączenia (przycisk *Build*). Automatyczne generowanie połączenia składa się z dwóch etapów:

- Wybranie dostawcy bazy danych – w przykładzie wybrany został *Microsoft OLE DB Provider for SQL Server* (rysunek 8)



Rysunek 8 – wybór dostawcy bazy danych

- Wybranie serwera bazy danych, zdefiniowanie danych do autoryzacji oraz wybranie bazy danych (rysunek 9)



Rysunek 9 – wprowadzanie danych logowania i wybór bazy danych

Po zdefiniowaniu połączenia można zacząć korzystać z klas w projekcie. Przykłady wykorzystania wygenerowanego kodu przedstawione zostały w podrozdziale *Przykłady zastosowania wygenerowanego kodu.*

5.4. Przykłady zastosowania wygenerowanego kodu

W celu zaprezentowania pracy z wygenerowanym kodem stworzony został projekt aplikacji okienkowej w IDE Delphi Studio, do którego dołączony został kod wygenerowanych klas i zdefiniowane zostało połączenie z bazą danych. W kolejnych podrozdziałach opisane zostały najistotniejsze metody obiektów mapujących oraz sposoby wykonania operacji CRUD.

5.4.1. Opis klas bazowych

Każda klasa bazowa reprezentujące encję tabeli (*Entity*) posiada prywatne atrybuty reprezentujące pola tabeli. Co najmniej jeden z atrybutów każdej z klas jest reprezentacją klucza głównego – zgodnie ze specyfikacją schematu bazy danych *każda tabela musi posiadać co najmniej jeden klucz główny*. Dla każdego atrybutu zaimplementowane są publiczne metody dostępowe *get, set*.

Każda klasa posiada trzy konstruktory *Create* – pierwszy służy do tworzenia nowych obiektów, pozostałe do tworzenia istniejących obiektów za pomocą klucza głównego lub obiektu kryteriów (klasa *TCriteria*). W celu zapisania obiektu w bazie danych wykorzystywana jest metoda *save*, w celu usunięcia metoda *delete*.

W przypadku relacji tworzone są dodatkowe atrybuty wraz z metodami dostępowymi do obiektów o odpowiednim typie. Dla relacji *jeden-do-wielu* implementowany jest kontener obiektów reprezentujących tabelę po stronie *wiele*. Dla relacji *wiele-do-jednego* implementowany jest obiekt reprezentujący tabelę po stronie *jeden*.

Klasy kontenerowe zawierają dwa konstruktory *Create* – pierwszy z nich służy do utworzenia nowego kontenera obiektów, drugi zaś do pobrania zbioru istniejących obiektów na podstawie kryteriów. Dodatkowo każdy kontener posiada metody zapisu i usuwania z bazy danych (*save, delete*) oraz podstawowe metody obiektów kontenerowych jak dodanie i usunięcie obiektu z kolekcji (*addItem, removeItem*). Ilość obiektów kolekcji można uzyskać za pomocą metody *getCount*.

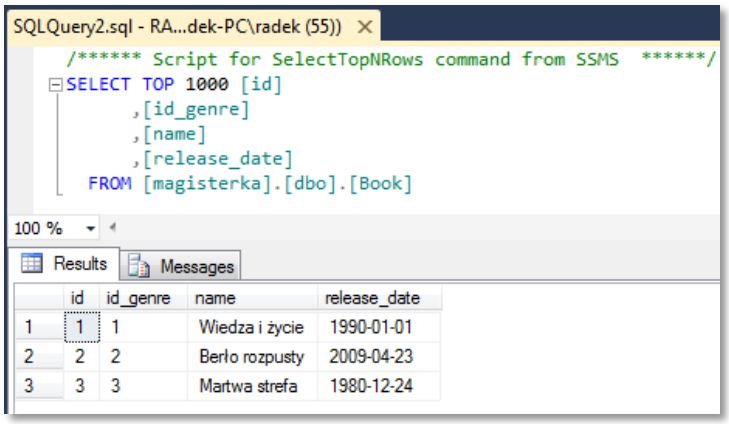
5.4.2. Opis klasy kryteriów

Klasa *TCriteria* jest jednym z najważniejszych elementów mapera. Za jej pomocą można pobrać pojedyncze obiekty, lub ich kolekcje na podstawie złożonych kryteriów wyszukiwania bez konieczności pisania kodu SQL. Konstruktor klasy kryteriów za parametr przyjmuje nazwę tabeli (można ją pobrać ze statycznej metody *getTableName* klasy reprezentującej daną tabelę). Klasa konstruktora realizuje założenia wzorca *płynny interfejs*, tzn. każda metoda klasy zwraca jako wynik instancję klasy, na której została wywołana. Pozwala to na elegancki i intuicyjny sposób tworzenia kryteriów. Przykładowe metody klasy, to np.:

- *addInnerJoin* – pozwala dodać złączenie typu *INNER*
- *addIntCondition* – pozwala dodać warunek dla liczby naturalnej
- *addGroupBy* – pozwala dodać grupowanie
- *addAscOrderBy* – pozwala dodać sortowanie rosnące
- *addHaving* – pozwala dodać warunki dla grupowania
- *setDistinct* – pozwala wyodrębnić wynik zapytania

5.4.3. Tworzenie obiektów na podstawie klucza głównego

Aby utworzyć obiekt na podstawie klucza głównego należy użyć konstruktora *Create* przyjmującego za parametr identyfikator obiektu. Rysunek 10 przedstawia wpisy w tabeli *Book* w bazie danych. Na listingu 34 pokazano jak utworzyć obiekt klasy *Book* na podstawie identyfikatora o wartości 1.

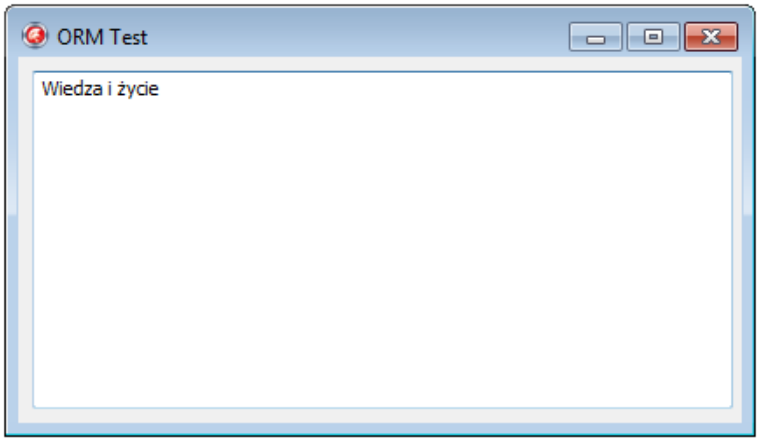


Rysunek 10 – wpisy w tabeli *Book*

Listing 34 – tworzenie obiektu klasy *TBook* na podstawie identyfikatora

```
var
  book: TBook;
begin
  book := TBook.Create(1);
  Memo1.Text := book.getName();
end;
```

Wynik wykonania powyższego kodu pokazuje rysunek 11.



Rysunek 11 – wynik wykonania kodu z listing 34

Dla klas mapujących tabele z kluczem obcym składającym się z wielu pól należy wywołać konstruktor przekazując jako osobne parametry wszystkie identyfikatory. Dla przykładu tabela pośrednicząca *BookAuthor* posiada dwa klucze główne: *id_book* oraz *id_author*. Aby utworzyć obiekt

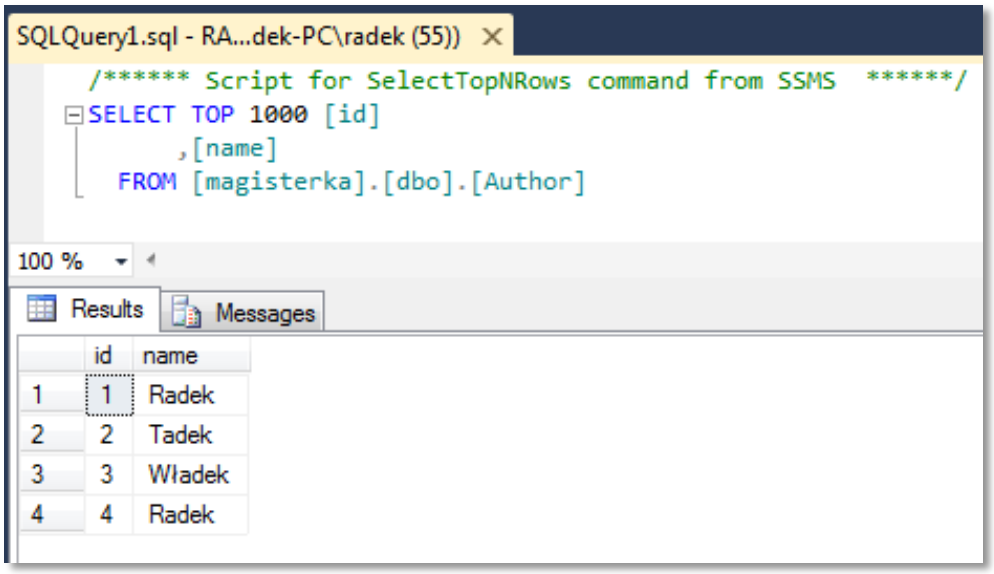
5.4.4. Tworzenie obiektów na podstawie kryteriów

Aby utworzyć obiekt na podstawie kryteriów należy wywołać konstruktor klasy przekazując jako parametr obiekt klasy *TCriteria*. Obiekt kryteriów powinien być odpowiednio skonfigurowany przed przekazaniem do konstruktora. Dla przykładu na listingu … obiekt klasy *Author* tworzony jest na podstawie kryteriów z następującymi warunkami:

- Autor powinien nazywać się *Radek*
- Autor powinien być autorem książki o tytule *Wiedza i życie*

Aby warunek ograniczający tytuł książki mógł zadziałać, wymagane było określenie warunków złączenia tabel (*addInnerJoin*). Rysunek 12 przedstawia zawartość tabeli *Author*.

Autor o identyfikatorze *1* jest autorem książki o tytule *Wiedza i życie* natomiast autor o identyfikatorze *4* jest autorem książki *Martwa strefa*.



Rysunek 12 – wpisy w tabeli *Author*

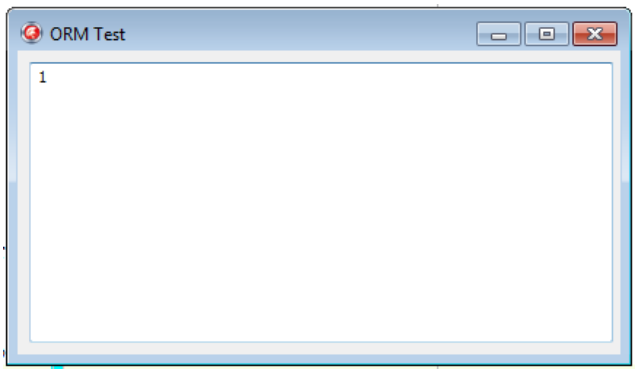
Metody klasy *TCriteria* za parametry przyjmują głównie nazwy pól tabel oraz wartości je ograniczające. Aby zmniejszyć ryzyko związane z literówkami, zalecane jest stosowanie stałych atrybutów klas reprezentujących nazwy pól tabel (np. *TBook.name*).

Listing 35 – pobieranie obiektu klasy *TAuthor* na podstawie kryteriów wyszukiwania

```
var
  author: TAuthor;
  criteria: TCriteria;
begin
  criteria := TCriteria.Create(TAuthor.getTableName());
  criteria.
    addStrCondition(TAuthor.NAME, 'Radek').
    addInnerJoin(TBookAuthor.getTableName(), TAuthor.ID, TBookAuthor.ID_AUTHOR).
    addInnerJoin(TBook.getTableName(), TBookAuthor.ID_BOOK, TBook.ID).
    addStrCondition(TBook.NAME, 'Martwa strefa');

  author := TAuthor.Create(criteria);
  Memo1.Text := IntToStr(author.getid());
end;
```

Wynik wykonania kodu z listingu 35 przedstawiony jest na rysunku 13



Rysunek 13 – wynik wykonania kodu z listingu 35

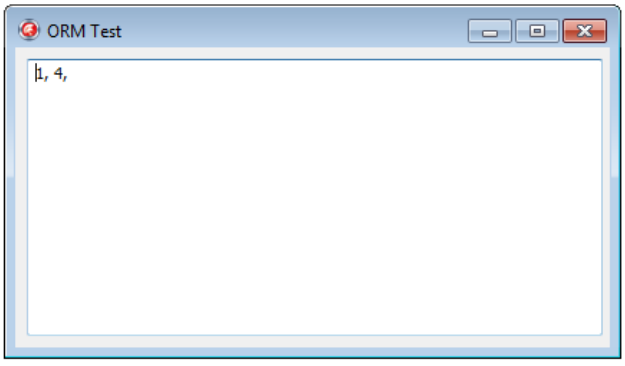
Powyższy przykład tworzenia pojedynczego obiektu na podstawie kryteriów, chociaż możliwy do wykonania, w praktyce nie będzie często stosowany. Kryteria przydają się w momencie tworzenia zbioru danych. Listing 36 przedstawia tworzenie zbioru autorów, którzy nazywają się *Radek*.

Listing 36 – pobierania kolekcji obiektów na podstawie kryteriów wyszukiwania

```
var
  authors: TAuthorContainer;
  criteria: TCriteria;
  i: Integer;
begin
  criteria := TCriteria.Create(TAuthor.getTableName());
  criteria.addStrCondition(TAuthor.NAME, 'Radek');
  authors := TAuthorContainer.Create(criteria);

  for i := 0 to authors.getLength() - 1 do
  begin
    Memo1.Text := Memo1.Text + IntToStr(authors.getItem(i).getId());
  end;
end;
```

Wynik wykonania powyższego kodu widoczny jest na rysunku 14.



Rysunek 14 – wynik wykonania kodu z listingu 36

5.4.5. Zapisywanie i usuwanie obiektów

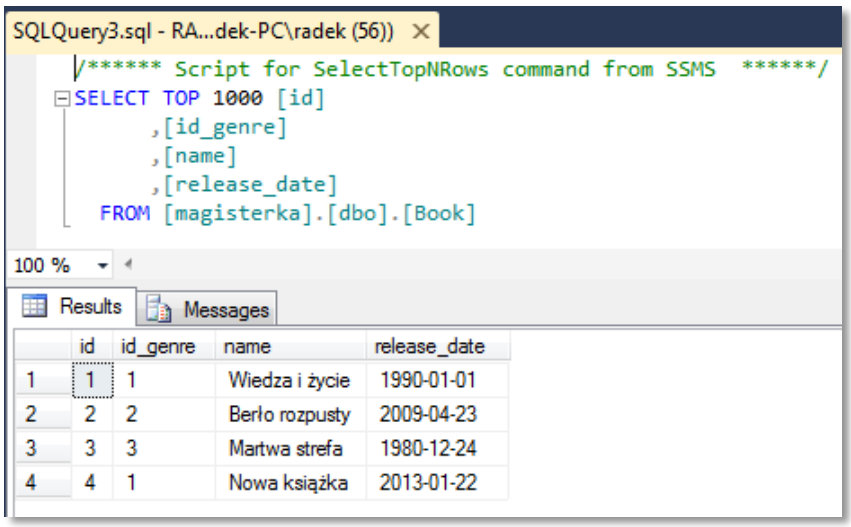
Operacje zapisu i usuwania można wykonywać zarówno na pojedynczych obiektach jak i kontenerach obiektów. Wywołanie operacji na kontenerze obiektów wywołuje tą samą operację na każdym elemencie kontenera. Zarówno metody zapisu i usuwania objęte są transakcją.

Aby wykonać zapis na obiekcie należy wywołać metodę *save*. W przypadku nowego obiektu dane zostaną zapisane w tabeli jako nowa krotka, w przeciwnym razie zmodyfikowane dane obiektu zostaną utrwalone w istniejącym rekordzie. Listing 37 przedstawia utrwalenie nowej książki w bazie danych.

Listing 37 – zapisywanie obiektu klasy *TBook*

```
var
  criteria: TCriteria;
  book: TBook;
  genre: TGenre;
begin
  criteria := TCriteria.Create(TGenre.getTableName());
  criteria.addStrCondition(TGenre.NAME, 'Nauka');
  genre := TGenre.Create(criteria);

  book := TBook.Create();
  book.setName('Nowa książka');
  book.setGenre(genre);
  book.setRelease_date(Now());
  book.save();
end;
```

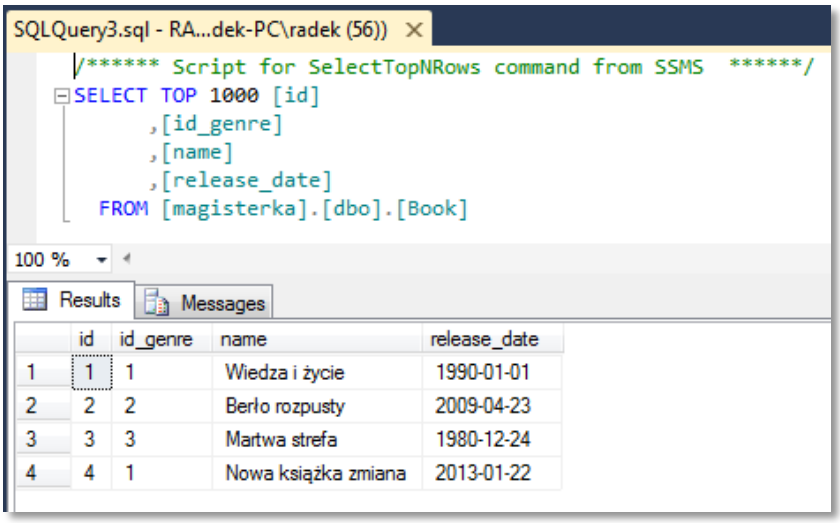


Rysunek 15 – stan tabeli po zapisaniu nowej książki

Na listingu 38 zmieniona zostaje jej nazwa i ponownie zostaje zapisana do bazy. Rysunki 15 i 16 przedstawiają stan tabeli *Book* po zapisaniu nowej książki oraz po zmianie jej nazwy i utrwaleniu zmian.

Listing 38 – zapisywanie zmian w obiekcie

```
var
  book: TBook;
begin
  book := TBook.Create(4);
  book.setName('Nowa książka zmiana');
  book.save();
end;
```



Rysunek 16 – stan tabeli po zapisaniu zmian

Usuwając wpis z bazy danych należy zwrócić uwagę na tabele będące z nim w relacji. Dla przykładu usuwając książkę naturalnym wydaje się usunięcie wpisów z tabeli pośredniczącej *BookAuthor*. Inaczej wygląda sytuacja w przypadku tabeli *Genre* – usuwając gatunek książki nie powinny być usuwane wszystkie książki o tym gatunku. W wygenerowanych metodach *delete* wykonywane jest usuwanie jedynie wpisu, który dana instancja obiektu reprezentuje. Jeżeli programista chce, by metoda *delete* usuwała również inne wpisy, powinien zaimplementować tą funkcjonalność. Na listingu 39 przedstawiona została metoda *deleteAll* zaimplementowana w klasie *TBook*. Jej zadaniem jest usunięcie wszystkich wpisów z tabeli *BookAuthor*, następnie usunięcie konkretnego wpisu z tabeli *Book*. Obydwie operacje zawarte są jednej w transakcji.

Listing 39 – implementacja metody *deleteAll* w klasie *TBook*

```
procedure TBook.deleteAll;
var
  fTransactionFlag: boolean;
  i: Integer;
begin
  fTransactionFlag := false;
  try
    begin
      if (dbConf.getConnection().InTransaction) then
      begin
        fTransactionFlag := true;
      end
      else
      begin
        dbConf.getConnection().BeginTrans();
      end;

      // usuwanie wpisów z tabeli BookAuthors
      for i := 0 to getBookAuthors().getLength() - 1 do
      begin
        getBookAuthors.getItem(i).delete();
      end;
      // ------------------------------------

      // wywołanie metody delete dla instancji klasy Book
      delete();
      // ------------------------------------

      if NOT(fTransactionFlag) then
      begin
        dbConf.getConnection().CommitTrans();
      end;
    end;
  except
    if NOT(fTransactionFlag) then
    begin
      dbConf.getConnection().RollbackTrans();
    end;
    Raise Exception.Create(className + ': error on delete!');
  end;
end;
```

5.5. Opis wybranych szczegółów implementacyjnych

Najistotniejszymi elementami prototypu mapera jest ładowanie pliku XML ze schematem bazy danych i jego konwersja na model obiektowy oraz implementacja generatora klas mapujących. Niemniej ważnym są szczegóły implementacyjne wygenerowanych klas. Poniżej przedstawione zostały wybrane szczegóły modelu obiektowego schematu bazy danych, generatora oraz klas mapujących.

- **Model obiektowy schematu bazy danych**

Na podstawie wskazanego pliku XML ze schematem bazy danych tworzona jest obiektowa reprezentacja bazy danych, której model prezentuje diagram 5.

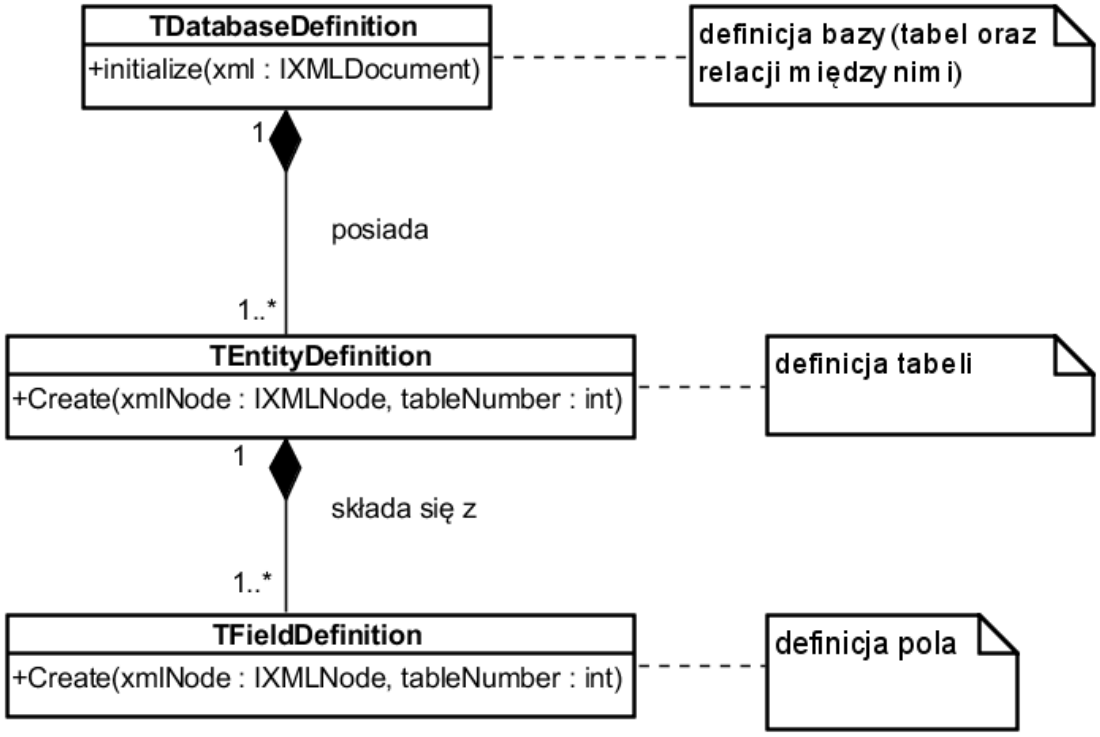


Diagram 6 - model obiektowy schematu bazy danych

Plik XML parsowany jest za pomocą biblioteki omniXML – darmowego parsera plików XML dla Delphi [16]. Po utworzeniu obiektu definicji bazy danych uruchamiana jest operacja *initialize*, która parsuje przekazany obiekt klasy *IXMLDocument* i tworzy obiekty klasy *TEntityDefinition* (diagram 6). Konstruktor klasy *TEntityDefinition* przyjmuje jako parametr obiekt *IXMLNode* reprezentujący element *table* schematu bazy danych, na którego podstawie tworzone są obiekty klasy *TFieldDefinition* (diagram 7). Obiekty reprezentujące tabele bazy danych, poza wiedzą o własnych polach, posiadają informacje o swojej nazwie, złączeniach wewnętrznych (operacja *addInner* na diagramie 7) oraz zewnętrznych (operacja *prepareOuters* na diagramie 6). Na podstawie złączeń generowane są odpowiednie referencje w klasach mapujących – dla złączeń wewnętrznych są to obiekty klas mapujących, dla złączeń zewnętrznych kontenery obiektów klas mapujących.

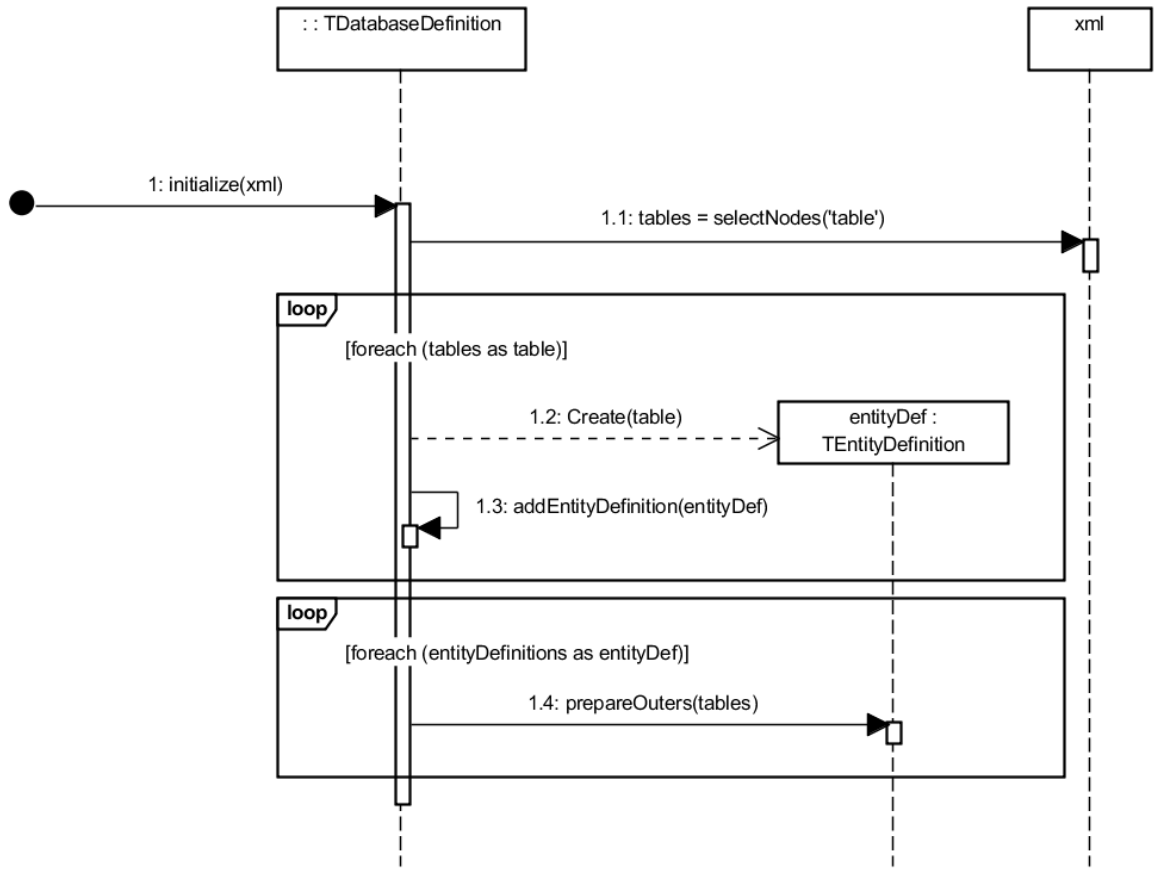


Diagram 7 - diagram sekwencji dla operacji *initialize*

Konstruktor klasy *TFieldDefinition*, reprezentującej pole w tabeli bazy danych, jako parametr przyjmuje obiekt *IXMLNode*, z którego pobierane są dane do nasycenia:

- Nazwa
- Typ danych
- Czy jest kluczem głównym
- Czy jest kluczem obcym
    - Nazwa tabeli powiązana z kluczem obcym
    - Nazwa pola powiązanego z kluczem obcym

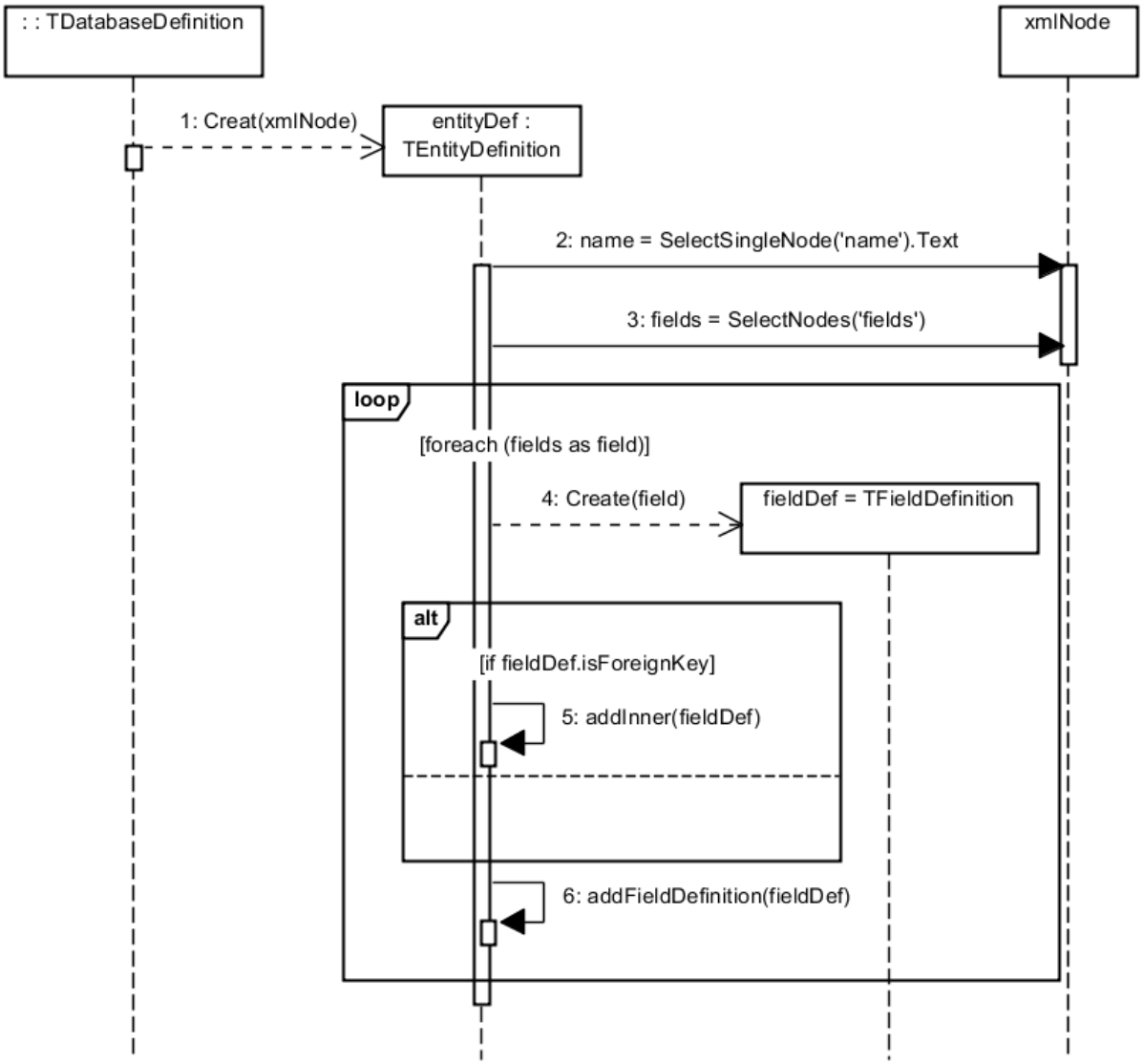


Diagram 8 - diagram sekwencji dla operacji tworzenia obiektu *TEntityDefinition*

- ## Generator i klasy mapujące

Pliki z klasami mapującymi generowane są za pomocą klasy *TORM*. Proces generowania plików jest trywialny – tworzona jest tablica ciągów znaków, której każdy element reprezentuje jedną linię kodu w zapisywanym pliku wynikowym. Dla każdego obiektu *TEntityDefinition*, pobranego z obiektowej reprezentacji schematu bazy danych *TDatabaseDefinition*, tworzone są cztery pliki:

- *BaseTableEntity* – plik zawierający kod klasy bazowej mapującej tabelę *Table*
- *BaseTableContainer* – plik zawierający kod klasy bazowej kontenera klas mapujących tabelę *Table*
- *Table* – plik zawierający klasę rozszerzającą klasę *TBaseTableEntity*

- *TableContainer* – plik zawierający klasę rozszerzającą klasę *TBaseTableContainer*

Wygenerowane pliki podzielone są na sekcje zgodnie ze specyfikacją języka Delphi. Pierwsza linia zawiera nazwę jednostki (ang. *unit*), po czym rozpoczyna się sekcja *interface*. W sekcji *interface* dołączane są wykorzystywane pliki (*uses*) oraz definiowana jest klasa (*type*). Po definicji klasy znajduje się ostatnia sekcja pliku: *implementation*. W sekcji *implementation* dołączane są dodatkowe, wykorzystywane pliki (*uses*) oraz implementowane są metody zadeklarowane w sekcji *interface*. Listing 40 zawiera przykładowy kod ze szkieletem (deklaracją) klasy bazowej, mapującej tabelę *Book*. Tabela *Book* posiada cztery pola:

- Id – klucz główny tabeli
- Id_genre – klucz obcy reprezentujący gatunek książki (tabela *Genre*)
- Name – nazwa książki
- Release_date – data wydania książki

Na podstawie danych z obiektowej reprezentacji tabeli utworzonych zostało sześć atrybutów klasy *TBook*:

- fId: integer
- fId_genre: integer
- fName: String
- fRelease_date: TDateTime
- fGenre: TObject
- fBookAuthors: TBookAuthorContainer

Poza atrybutami reprezentującymi pola tabeli, wygenerowane zostały dwa dodatkowe:

- fGenre – reprezentuje obiekt gatunku książki, tworzony na podstawie identyfikatora encji tabeli *id_genre*
- fBookAuthors – reprezentuje kontener obiektów dla tabeli pośredniczącej między *Book* oraz *Author*. Jest to zbiór autorów książki.

Listing 40 - klasa bazowa *TBaseBookEntity*

```
TBaseBookEntity = class (TMyEntity)
private
  fid: integer;
  fid_genre: integer;
  fname: string;
  frelease_date: TDateTime;
  fGenre: TObject;
  fBookAuthors: TBookAuthorContainer;
  fIsNew: Boolean;
protected
public
  const ID = 'Book.id';
  const ID_GENRE = 'Book.id_genre';
  const NAME = 'Book.name';
  const RELEASE_DATE = 'Book.release_date';
  class function getTableName(): String; static;
  class function getFieldNames(): TStringArray; static;
  class function getPrimaryKeyNames(): TStringArray; static;
  procedure setid(aVal: integer);
  function getid(): integer;
  procedure setid_genre(aVal: integer);
  function getid_genre(): integer;
  procedure setname(aVal: String);
  function getname(): String;
  procedure setrelease_date(aVal: TDateTime);
  function getrelease_date(): TDateTime;
  procedure setGenre(aVal: TObject);
  function getGenre(): TObject;
  procedure setBookAuthors(aVal: TBookAuthorContainer);
  function getBookAuthors(): TBookAuthorContainer;
  constructor Create(aCriteria: TCriteria); overload;
  constructor Create(); overload;
  constructor Create(aid: integer); overload;
  procedure serialize(); override;
  procedure save();
  procedure delete();
  destructor Destroy(); override;
  procedure prepareEntity(ADOQuery: TADOQuery);
  function isNew(): boolean;
end;
```

Dla każdego z atrybutów zdefiniowane zostały metody dostępowe *get* i *set*, wygenerowane zostały również stałe reprezentujące nazwy pól w bazie danych (pomocne podczas definiowania kryteriów wyszukiwania). Na samym dole deklaracji klasy widoczne są najważniejsze metody klasy mapującej – operacje związane z CRUD. Poniższe listingi przedstawiają implementację przykładowych metod *Create(aid), prepareEntity(ADOQuery) oraz save().*

Listing 41 - implementacja konstruktora tworzącego obiekt na podstawie identyfikatora tabeli

```
constructor TBaseBookEntity.Create(aid: integer);
var
  query: String;
begin
  inherited Create();
  query := 'SELECT * FROM ' + getTableName() + ' WHERE ' +
    'id = ' + IntToStr(aid);
  dbConf.executeQuery(query);
  prepareEntity(dbConf.getQuery());
end;
```

Listing 42 - metoda nasycająca obiekt danymi z zapytania SQL

```
procedure TBaseBookEntity.prepareEntity(ADOQuery: TADOQuery);
begin
  setid(ADOQuery.FieldByName('id').AsInteger);
  setid_genre(ADOQuery.FieldByName('id_genre').AsInteger);
  setname(ADOQuery.FieldByName('name').AsString);
  setrelease_date(ADOQuery.FieldByName('release_date').AsDateTime);
end;
```

Operacje na bazie danych wykonywane są z pomocą komponentów *TADOConnection* oraz *TADOQuery*. Tworzenie obiektu na podstawie identyfikatora encji tabeli polega na utworzeniu zapytania w języku SQL, wykonania go przez obiekt *ADOConnection* oraz przekazania wyników w postaci obiektu *ADOQuery* do metody *prepareEntity*. Metoda *prepareEntity* nasyca obiekt wywołując odpowiednie metody na obiekcie *ADOQuery*.

Operacja zapisu obiektu do bazy danych zawarta jest w transakcji wywołanej na obiekcie *ADOConnection*. W przypadku, gdy obiekt jest nowy, wykonywane jest zapytanie *INSERT* i obiekt zapisywany jest jako nowy wpis do tabeli. W przeciwnym razie obiekt zostaje zaktualizowany za pomocą zapytania *UPDATE*. Konwertowanie wartości atrybutów klasy do odpowiedniego formatu w języku SQL wykonywane jest z wykorzystaniem pomocniczej klasy *TSQLF* (SQL Format).

Listing 43 - operacja zapisu obiektu do bazy danych

```
procedure TBaseBookEntity.save();
var
  fTransactionFlag: boolean;
  query: String;
begin
  fTransactionFlag := false;
  try
    begin
      if (NOT fNeedSave) then
        exit;
      if (dbConf.getConnection().InTransaction) then
      begin
        fTransactionFlag := true;
      end
      else
      begin
        dbConf.getConnection().BeginTrans();
      end;
      if (isNew()) then
      begin
        query := 'INSERT INTO ' + getTableName() + ' (' +
             'id_genre, ' +  'name, ' + 'release_date' +
          ') VALUES (' +
            TSQLF.Int(getid_genre()) + ', ' +
            TSQLF.EscapeSpecialChars(getname()) + ', ' +
            TSQLF.Date(getrelease_date()) +
          ');';
        dbConf.executeInsertQuery(query);
        query := 'SELECT MAX(id) AS id FROM Book;';
        dbConf.executeQuery(query);
        setid(dbConf.getQuery().FieldByName('id').AsInteger);
        fIsNew := false;
      end
      else
      begin
        query := 'UPDATE ' + getTableName() + ' SET ' +
            'id_genre = ' + TSQLF.Int(getid_genre()) + ', ' +
            'name = ' + TSQLF.EscapeSpecialChars(getname()) + ', ' +
            'release_date = ' + TSQLF.Date(getrelease_date()) +
          ' WHERE ' +  'id = ' + TSQLF.Int(getid()) ;
        dbConf.executeInsertQuery(query);
      end;
      if NOT(fTransactionFlag) then
      begin
        dbConf.getConnection().CommitTrans();
      end;
    end;
  except
    if NOT(fTransactionFlag) then
    begin
      dbConf.getConnection().RollbackTrans();
    end;
    Raise Exception.Create(className + ': error on save!');
  end;
end;
```

6. Podsumowanie

Mapery obiektowo-relacyjne stały się bardzo istotnym elementem w tworzeniu nowoczesnych aplikacji bazodanowych. Rozwój tego typu narzędzi nabrał dodatkowo tempa po wprowadzeniu zaawansowanych mechanizmów refleksji w językach obiektowych. Wszystkie nowości kierowane są jednak w stronę najnowszych technologii pozostawiając systemy spadkowe na straconej pozycji. Dziedzina maperów obiektowo-relacyjnych dla systemów spadkowych powinna zostać zauważona i rozwijana.

W ramach pracy wykonano prototyp mapera dla języka obiektowego Delphi, realizujący podstawowe mapowanie encji zgodnie z założeniami postawionymi w rozdziale 4. Aplikacja korzysta z pliku schematu bazy danych XML, na podstawie którego generuje kod klas realizujących podstawowe operacje CRUD. Klasa kryteriów wyszukiwania *TCriteria* jest jedną z głównych zalet aplikacji – dzięki niej programiści mogą tworzyć zaawansowane kryteria wyszukiwania obiektów bez konieczności pisania kodu SQL.

6.1. Zalety i wady przyjętych rozwiązań

Najistotniejszą zaletą aplikacji jest generowanie kodu klas mapujących działających w starszych wersjach języka Delphi (2005 lub wyższa). Wykonany został prototyp jako oddzielną aplikację z myślą o deweloperach posiadających przestarzałe środowiska programowania, lub korzystających z alternatywnych narzędzi do edytowania kodu Delphi (np. *Emacs*). Dzięki temu maper nie uzależnia ich od konkretnego narzędzia zewnętrznego. Plik ze schematem bazy danych nie ma skomplikowanej struktury, po której poznaniu jego stworzenie staje się stosunkowo łatwym zadaniem. Operacje CRUD wykonywane są na obiektach w bardzo łatwy i przejrzysty sposób. Pobieranie zbiorów obiektów na podstawie kryteriów wyszukiwania wspomaga klasa *TCriteria*, która realizuje wzorzec *płynny interfejs* i posiada bogaty zestaw metod związanych z tworzeniem zapytań SQL. Programista może dzięki temu w elegancki sposób stworzyć skomplikowane kryteria wyszukiwania bez konieczności pisania kodu SQL. Pomagają w tym również stałe atrybuty w wygenerowanych klasach odpowiadające polom w tabeli. Kolejną zaletą narzędzia są wbudowane transakcje w operacje zapisu i usuwania zbiorów obiektów.

Plik schematu bazy danych XML, mimo swojej prostoty, wprowadza dodatkowy element podatny na błędy. Ze względu na brak mechanizmu refleksji w starszych technologiach Delphi jest to akceptowalne rozwiązanie, istnieje jednak możliwość

całkowitego wyeliminowania pliku konfiguracyjnego poprzez odczytywanie informacji o schemacie z metadanych serwera baz danych. Mimo wymienionych zalet związanych z realizacją prototypu jako oddzielnej aplikacji, brak narzędzia w postaci wtyczki do IDE jest dużym utrudnieniem dla osób korzystających ze zintegrowanych środowisk programistycznych.

6.2. Proponowane plany rozwoju

Prototyp mapera realizuje podstawowe zadania aplikacji mapującej obiektowo-relacyjnie. Możliwych ścieżek rozwoju jest wiele, a najważniejsze zostały opisane poniżej.

- Pobieranie schematu bazy danych z serwera baz danych:

  Serwery relacyjnych baz danych udostępniają metadane na temat baz danych, co pozwala wygenerować kod klas mapujących bez konieczności tworzenia pliku XML ze schematem. Jest to najważniejszy kierunek rozwoju prototypu mapera.

- Interfejs GUI dla operacji CRUD:

  Kod klas mapujących realizuje operacje CRUD, lecz aby wykorzystać go w aplikacji okienkowej należy zaimplementować interfejs graficzny do odczytu, wprowadzania, zapisu i usuwania danych. Proces ten jest powtarzalny, a szablony dla tych operacji mogą zostać wygenerowane na podstawie pliku schematu bazy danych. Jest to kolejny, istotny element warty rozwoju.

- Obsługa większej ilości typów:

  Prototyp aplikacji obsługuje ograniczoną ilość typów danych. Rozbudowanie narzędzia o mapowanie dodatkowych typów danych zwiększy możliwości jego wykorzystania.

- Możliwość wyboru dostawcy bazy danych:

  Definiowanie połączenia z bazą danych odbywa się poprzez generyczny dla języka Delphi komponent *TADOConnection*. Pozwala on na wybór różnych dostawców bazy danych, jednak prototyp aplikacji skierowany jest na obsługę Microsoft SQL Server, przez co wybór innego dostawcy może spowodować błędy w wykonaniu niektórych operacji SQL. Rozwinięcie prototypu dla obsługi większej ilość serwerów baz danych zwiększy jego atrakcyjność.

- Funkcje agregujące jako metody pomocnicze klas mapujących:

Dobrym pomysłem na rozwinięcie funkcjonalności wygenerowanych klas mapujących jest wprowadzenie metod pomocniczych reprezentujących funkcje agregujące języka SQL typu *SUM*, *MAX*, czy *AVG*. Metody mogą być zaimplementowane jako statyczne metody klas mapujących encje tabeli przyjmujące za parametr nazwę pola, np. *TBook.funcMAX(TBook.RELEASE_DATE);*

Prace cytowane

[1]. Trzaska M.: The Smart Persistence Layer. ICSEA 2011: The Sixth International Conference on Software Engineering Advances. October 23-29, 2011 - Barcelona, Spain. ISBN: 978-1-61208-165-6. pp. 206-212

[2]. Radosław Adamus, Piotr Habela, Krzysztof Kaczmarski, Michał Letner, Krzysztof Stencel, Kazimierz Subieta. Stack-Based Architecture and Stack-Based Query Language. http://www.odbms.org/download/030.02%20Subieta%20Stack-Based%20Architecture%20and%20Stack-Based%20Query%20Language%20March%202008.PDF, Warszawa 2008

[3]. Kazimierz Subieta. Niezgodność impedancji. Wykłady z przedmiotu *Języki i Środowiska Programowania Baz Danych*.

http://edu.pjwstk.edu.pl/wyklady/jps/scb/wyklad2/section1.html, Warszawa 2006

[4]. Martin Fowler. Data Mapper. *Catalog of Patterns and Enterprise Application Architecture*. http://www.martinfowler.com/eaaCatalog/dataMapper.html

[5]. Daniele Teti. *DORM – the Delphi ORM*. Verona 27-28.10.11

[6]. Delphi-ORM Wiki Pages. *Attributes Mapping Examples*.

http://code.google.com/p/delphi-orm/wiki/AttributeMappingRFC

[7]. TMS Aurelius PDF Developers Guide. Official Documentation. http://www.tmssoftware.com.br/aurelius/doc/aurelius_manual.pdf

[8]. Wikipedia. *Domain Driven Design*. http://pl.wikipedia.org/wiki/Domain-Driven_Design

[9]. Wikipedia. *ECO*. http://en.wikipedia.org/wiki/ECO_(Domain_Driven_Design)

[10]. Wikipedia. *Wzorzec Projektowy*. http://pl.wikipedia.org/wiki/Wzorzec_projektowy_(informatyka)

[11]. Craig Larman. UML i Wzorce Projektowe. Analiza i projektowanie obiektowe oraz iteracyjny model wytwarzania oprogramowania. Wydanie trzecie. Helion 2011

[12]. Wikipedia. *Delphi*. http://pl.wikipedia.org/wiki/Delphi

[13]. Delphi – podstawy programowania. http://chomikuj.pl/melasa4/Delphi/Delphi+-+podstawy+programowania+-+*c5*9brodowisko+Delphi,87147216.pdf, Olsztyn 2004

[14]. Ray Lishner. Hidden Paths of Delphi 3. http://www.tempest-sw.com/hiddenpaths/

[15]. Allen Bauer. IDE Integration pack for Delphi 8 & C++ Builder 1.0. http://edn.embarcadero.com/article/31918

[16]. omniXML – Simple way to use XML in Delphi. http://code.google.com/p/omnixml/

Spis rzeczy

Listingi

Diagramy

Rysunki