POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Rafał Monkiewicz
Nr albumu 4001

Framework aplikacji bazodanowych

Praca magisterska
napisana pod kierunkiem

Dr inż. Mariusza Trzaski

Warszawa, luty 2011

Streszczenie

Niniejsza praca koncentruje się na temacie narzędzi dla programistów, które wspierają i przyśpieszają proces implementacji bazodanowych aplikacji biznesowych. Podczas implementacji programiści muszą wykonywać powtarzalne zadania, które z całą pewnością mogą zostać zautomatyzowane i zastąpione przez dedykowane oprogramowanie. Jednak nie powinno to wpłynąć na jakość kodu, łatwość jego utrzymywania oraz testowania. Większość tego typu rozwiązań dostępnych na rynku często zamiast ułatwiać wykonywane prace, dodatkowo je utrudnia wprowadzając do projektów złożoność oraz ukrywając zachowanie kodu. Programiści często są zmuszani do rezygnacji z własnych podejść na rzecz tych narzuconych przez framework. Dodatkowym problemem jest też fakt, że możliwości modyfikacji zachowania takich narzędzi w większości przypadków są bardzo ograniczone. Są one przeznaczone do bardzo konkretnych zadań, ich funkcjonalność jest mało uniwersalna, co w świecie ciągle rozwijających się technologii oraz ciągle zmieniających się wymagań funkcjonalnych jest nie do przyjęcia.

W pracy zostały przedstawione rozwiązania dostępne na rynku, opisano ich wady oraz zalety. Następnie na podstawie doświadczeń autora oraz przeprowadzonych badań stanu sztuki zaproponowany został zbiór wymagań, które powinny zostać spełnione przez odpowiednie narzędzie dla programistów. W dalszej części pracy scharakteryzowano podejście, którego celem było spełnienie tych wymagań. Dodatkowo została zwrócona uwaga na inne możliwe podejścia oraz na uzasadnienie ich odrzucenia. Skoncentrowano się na przedstawieniu podjętych decyzji architektonicznych, zastosowanych technologii, wzorców projektowych, bibliotek oraz na opisie samej implementacji prototypu rozwiązania – platformy o nazwie kodowej Timegen, która może integrować się ze środowiskiem programistycznym Microsoft Visual Studio. Zaprezentowano też sposób użycia uzyskanego rozwiązania oraz możliwości wprowadzania nowych funkcjonalności.

Pod koniec pracy opisano na ile postawione wymagania zostały spełnione. Zestawiono wady i zalety otrzymanego rozwiązania oraz przedstawione plany rozwoju projektu.

Spis treści

1. Wstęp

Każdy projekt informatyczny dotyczący oprogramowania jest inny. Oznacza to, że w procesie wytwarzania aplikacji biznesowych zawsze w mniejszym lub większym stopniu istnieje potrzeba pisania kodu źródłowego. W większości aplikacji biznesowych wymagane jest trwałe zachowywanie danych znajdujących się w systemie. W związku z tym potrzebny jest kod odpowiedzialny za komunikację z bazą danych. W obiektowo zorientowanych językach programowania wymagany jest również kod pozwalający na utrwalanie, modyfikację oraz usuwanie obiektów z bazy danych.

Tego typu potrzeby implikują wiele problemów. O ile przy implementacji komunikacji z bazą danych najczęściej wykorzystuje się gotowe biblioteki, to w przypadku pracy z obiektami lub chociażby definiowaniu ich, za każdym razem wymagany jest pewien nakład czasowy. Nawet pomimo tego, że zadania mogą być koncepcyjnie takie same, to ich implementacja będzie inna z powodu różnic dziedzin problemowych. Przykładowo obiekt klienta w systemie CRM firmy A będzie miał inna strukturę niż obiekt klienta w systemie CRM firmy B. W większości przypadków oznacza to, że za każdym razem będzie do napisania dodatkowy, a może nawet uznany już za podstawowy kod źródłowy. Kod, który nie jest zbytnio złożony i jest mało ambitny. Dla firm zajmujących się wytwarzaniem oprogramowania jest to równoznaczne z potrzebą alokowania dodatkowych zasobów czasowych, finansowych oraz ludzkich. Oznacza to również fakt, że programiści będą musieli wykonywać powtarzalne prace, które praktycznie nie wymagają od nich żadnego myślenia, ale są bardzo narażone na błędy wynikające chociażby z kopiowania i wklejania. Programiści powinni koncentrować się na rozwiązywaniu ważniejszych problemów, które wymagają ich pełnego zaangażowania i zarazem utrzymują ich morale na wysokim poziomie.

Przedstawione problemy pozostawiają szerokie pole do działania dla narzędzi przeznaczonych dla programistów. Duży nacisk kładziony jest na automatyzację powtarzalnych zadań programistycznych oraz przyśpieszenie procesu implementacji oprogramowania. Na rynku istnieją pewne rozwiązania, które starają się rozwiązywać tego typu problemy, jednak nie są one doskonałe.

1.1. Cel pracy

Głównym celem pracy jest zaproponowanie rozwiązania, które automatyzuje i przyśpiesza implementację warstwy dostępu do danych w aplikacjach biznesowych i nie robi tego kosztem jakości kodu. Aby osiągnąć główny cel zostały wyznaczone następujące cele pośrednie:

- zbadanie możliwych podejść, które pozwalają na przyśpieszenie implementacji,
- zbadanie stanu sztuki, czyli identyfikacja narzędzi nastawionych na rozwiązywanie problemów będących tematem pracy,
- identyfikacja zadań programistycznych, które mogą zostać zautomatyzowane i wykonywane przez oprogramowanie,

- zestawienie najbardziej pożądanych wymagań funkcjonalnych,
- implementacja prototypu narzędzia zgodnie z przyjętymi założeniami.

1.2. Rozwiązania przyjęte w pracy

Narzędzie wytworzone w ramach pracy zostało oparte o platformę Microsoft .NET Framework 4.0. Cała implementacja wykorzystuje język programowania C#. Prototyp został zintegrowany ze środowiskiem programistycznym Microsoft Visual Studio 2010 za pomocą rozwiązań udostępnianych przez Visualization and Modeling SDK. Ekrany służące do interakcji z użytkownikiem zostały wytworzone za pomocą Windows Presentation Foundation (WPF) wchodzącego w skład .NET Framework. Rozszerzalność rozwiązania jest wspierana za pomocą Managed Extensibility Framework (MEF). Do generacji kodu źródłowego został wykorzystany Text Template Tranformation Toolkit (T4). Założenia architektoniczne oraz wysoki poziom testowalności kodu zostały osiągnięte za pomocą wzorca projektowego Dependency Injection [3] oraz wspierającego go kontenera Inversion Of Control – Ninject.

Warstwy dostępu do danych będące produktami pracy z prototypem opierają się na relacyjnych bazach danych zarządzanych przez Microsoft SQL Server oraz obiektowych bazach danych udostępnianych przez środowisko ODRA. Kod generowany przez prototyp wykorzystuje frameworki NHibernate (mapowanie obiektowo-relacyjne w celu rozwiązania problemu niezgodności impedancji) oraz NOBC (umożliwia komunikację z bazami danych środowiska ODRA z poziomu platformy .NET Framework).

1.3. Rezultat pracy

Rezultatem pracy jest opracowanie zestawu wymagań i założeń, które spełniałoby odpowiednie narzędzie dla programistów. W ramach pracy został zaimplementowany prototyp takiego narzędzia, rozszerzający środowisko programistyczne Microsoft Visual Studio 2010 – platforma o nazwie kodowej Timegen.

Timegen spełnia najważniejsza wymagania funkcjonalne i udowadnia osiągalność przyjętych koncepcji. Pozwala na graficzne modelowanie struktur danych oraz generacji kodu źródłowego na ich podstawie. Rozszerzalność rozwiązania została udowodniona poprzez implementację dwóch dostawców o różnym zachowaniu, umożliwiających pracę z dwoma zupełnie innymi środowiskami bazodanowymi za pomocą dwóch różnych warstw dostępu do danych. Są to następująco:

- Timegen NHibernate Provider,
- Timegen ODRA Provider.

Timegen zbiera wszystkie informacje wymagane do przeprowadzanych przez niego operacji za pomocą przyjaznych dla użytkownika wizardów, które wtapiają się w środowisko programistyczne i

mogą być modyfikowane przez implementację dostawców. Duży nacisk kładziony jest na wygodę użytkowania wymaganą przez programistów podczas ich codziennej pracy z narzędziem.

1.4. Organizacja pracy

Rozdział drugi wprowadza w tematykę frameworków bazodanowych. Zawiera ich kategoryzację na podstawie wsparcia jakie oferują. Oprócz tego opisuje rozwiązania dostępne na rynku, koncentrując się na odnalezieniu ich najbardziej istotnych mocnych i słabych stron. Rozdział kończy się podsumowaniem stanu sztuki oraz zdefiniowaniem wymagań dla prototypu.

Rozdział trzeci koncentruje się na narzędziach oraz technologiach, które zostały wykorzystane przy tworzeniu tej pracy.

Rozdział czwarty zawiera ogólny opis proponowanego rozwiązania w odniesieniu do postawionych przed nim wymagań. Zawiera w sobie opis koncepcji całego rozwiązania, podjętych decyzji architektonicznych oraz uzyskanych funkcjonalności.

Rozdział piąty został poświęcony uzyskanemu prototypowi. Znajduje się w nim szczegółowy opis rozwiązania, decyzje podjęte na poziomie implementacji. Dodatkowo zawiera w sobie opis instalacji rozwiązania oraz przykłady użycia.

Rozdział szósty to zestawienie zalet i wad wytworzonego rozwiązania oraz przedstawienie planu jego dalszego rozwoju.

2. Frameworki aplikacji bazodanowych

W inżynierii oprogramowania używane są różne frameworki. Mogą dotyczyć one zarówno samego procesu wytwarzania oprogramowania jak i koncentrować się na bardziej konkretnych zadaniach. Ten rozdział dotyczy frameworków, które wspierają aplikacje bazodanowe.

Jednym z podstawowych założeń każdego frameworka jest zapewnienie środków umożliwiających rozwiązanie jakiegoś problemu. W przypadku frameworków aplikacji bazodanowych można wskazać zbiór cech charakteryzujący większość z nich:

- **Mapowanie obiektowo-relacyjne**

  Większość aplikacji biznesowych opiera się na relacyjnych bazach danych i jest zaimplementowana za pomocą obiektowych języków programowania. Powstaje problem znany jako niezgodność impedancji. Wynika on z tego, że wymagane jest przejście z modelu relacyjnego na model obiektowy. Dopóki relacyjne bazy danych nie zostaną wyparte z pierwszego planu przez obiektowe bazy danych, wymagane jest mapowanie obiektowo-relacyjne. Jest to jeden z problemów, z którym radzą sobie frameworki bazodanowe. Rozwiązania ORM (Object-Relational Mapping) są podstawową funkcjonalnością oferowaną przez frameworki. Najczęściej dostarczane jest mocno typowane API pozwalające na wykonywanie operacji na obiektach oraz ich wyszukiwania. Implementacja frameworka przetwarza wywołania metod na instrukcje zrozumiałe dla systemu zarządzania relacyjną bazą danych. Najczęściej jest to jeden z dialektów SQL. Przy pobieraniu danych z bazy, otrzymane informacje są zamieniane na obiekty.

- **Dostarczenie poziomu abstrakcji**

  Programista korzystający z frameworka zostaje wyposażony w biblioteki, które wznoszą go na wyższy poziom abstrakcji. Ten poziom zwiększa jego efektywność, bo część wymaganego kodu jest już dostarczona, a przy implementacji pozostałego kodu może wykorzystać generyczne mechanizmy zawarte w bibliotekach. Dodatkowo podstawowa implementacja, którą otrzymuje jest już przetestowana i najczęściej reprezentuje wysoki poziom jakości.

- **Konfiguracja**

  Każdy z frameworków wymaga pewnej konfiguracji. Musi zostać doprecyzowane zachowanie wewnętrznych mechanizmów poszczególnego rozwiązania. W przypadku frameworku ORM muszą zostać też określone reguły dotyczące przeprowadzania mapowania.

2.1. Poziomy wsparcia wytwarzania aplikacji bazodanowych i ich charakterystyka

Frameworki aplikacji bazodanowych mogą wspierać ich wytwarzanie na różnych poziomach. Dostępne jest wsparcie, które jest wykorzystywane tylko podczas wytwarzania aplikacji, jak również to które stanowi nierozłączną część aplikacji i towarzyszy jej również po wdrożeniu.

2.1.1. Biblioteki wspierające

Podstawowe wsparcie jest dostarczane do aplikacji za pomocą bibliotek. Są one wykorzystywane podczas implementacji i stanowią nierozłączną część kodu źródłowego aplikacji. Udostępniają generyczną implementację podstawowych mechanizmów warstwy dostępu do danych. Dostępne są również frameworki, które mogą realizować bardziej zaawansowane zadania takie jak zarządzanie transakcjami, śledzenie stanu obiektów czy cachowanie wyników zapytań.

Stosowanie takich frameworków wymaga pokonania pewnej krzywej uczenia się. Niektóre rozwiązania są nastawione na szybki start nawet dla osób mało doświadczonych, inne wymagają zrozumienia określonych koncepcji i nabycia wprawy w posługiwaniu się nimi. Każdy framework ma ściśle zaprojektowane zasady podstawowego funkcjonowania. Wiąże się to z tym, że programista musi przestrzegać pewnych konwencji podczas pisania kodu i używania bibliotek. Przykładem są chociażby wymagania stawiane przed klasami definiującymi obiekty, które będą zapisywane. Czasami klasy te muszą dziedziczyć po określonej klasie bazowej, w innych wypadkach klasy te mogą być POCO, ale z zastrzeżeniem, że ich atrybuty mają być wirtualne. Z każdym z rozwiązań są powiązane konkretne wzorce projektowe i dobre praktyki, których stosowanie pozwala na optymalne użycie oferowanych mechanizmów.

Biblioteki wspierające implementację warstwy dostępu do danych zawsze udostępniają API, które pozwala na łatwe użycie frameworka. API najpopularniejszych frameworków jest ciągle doskonalone i wykorzystuje najnowsze możliwości języków programowania.

Frameworki różnią się pod kątem dostępności ich kodu źródłowego. Niektóre są udostępniane na zasadzie Open Source. Pozwala to na poznawanie ich konstrukcji, sposobu działania oraz daje możliwości wprowadzania własnych modyfikacji. Dodatkowo programiści mają dostęp do nowych funkcjonalności, które jeszcze nie zostały do końca przetestowane, ale ich kod źródłowy został udostępniony. Dzięki temu mogą na bieżąco zapoznawać się z modyfikacjami oraz nowymi możliwościami danego frameworka. Jeżeli wokół danego rozwiązania jest skupiona duża społeczność specjalistów, to dystrybucja go na zasadzie Open Source niesie za sobą bardzo istotny plus – większość wykrytych błędow jest usuwana na bieżąco i naprawiony kod bardzo szybko trafia do dystrybucji frameworka. W przypadku rozwiązań, których kod źródłowy jest zamknięty dla społeczności, na naprawę błędów trzeba czekać aż do wydania kolejnej dużej wersji, czyli kilka miesięcy albo nawet lat.

2.1.2. Generatory kodu

Generatory kodu pozwalają na uzyskanie potrzebnego kodu źródłowego w szybkim czasie. Są to oddzielne aplikacje albo aplikacje zintegrowane ze środowiskami programistycznymi. Podstawą każdego generatora kodu jest zestaw danych wejściowych, które posłużą do wygenerowania kodu źródłowego na ich podstawie. Dobry generator kodu charakteryzują następujące cechy:

- możliwość wpłynięcia na postać kodu wynikowego,
- możliwość modyfikacji wygenerowanego kodu – zakładając, że każde uruchomienie generatora nadpisze kod uzyskany podczas poprzedniego przebiegu generacji (w tym również kod, który został dopisany przez programistę), wygenerowany kod musi być przystosowany do rozszerzania go. Najczęściej do tego celu wykorzystywane są możliwości obiektowych języków programowania i kompilatorów. Dobrym podejściem jest zastosowanie wzorca Double Derived, który w połączeniu z możliwością tworzenia klas częściowych (definicja jednej klasy może znajdować się w kilku plikach kodu źródłowego) pozwala na nadpisywanie implementacji wygenerowanej przez narzędzie. Generator używający Double Derived zamiast jednej klasy wygeneruje dwie. Pierwsza klasa będzie klasą bazową zawierająca całą implementację i umożliwiającą jej nadpisanie (metody klasy mają odpowiednie sygnatury – są wirtualne). Druga klasa to klasa dziedzicząca z klasy bazowej. Dzięki temu, że jest ona oznaczona jako częściowa, programista może sam utworzyć inne części tej klasy, w których zaprogramuje niestandardowe zachowanie klasy. Do tego celu programista użyje przesłaniania metod. Jego kod w klasach częściowych nie zostanie nadpisany podczas następnej generacji kodu źródłowego.
- wysoka jakość generowanego kodu, na którą składają się różne czynniki takie jak:
    - niepowtarzanie kodu – narzędzie generuje klasy, których implementacja jest później używana w wielu innych wygenerowanych klasach albo wygenerowany kod korzysta z zewnętrznych bibliotek. W przypadku aplikacji bazodanowych najczęściej będą to biblioteki zawierające implementację frameworków bazodanowych,
    - wzorce projektowe – wynikowy kod źródłowy implementuje pewne wzorce projektowe, które ułatwiają pracę z warstwą dostępu do danych. Mogą to być też wzorce, które są specyficzne dla użycia danego frameworka, w optymalny sposób wykorzystujące oferowane przez niego możliwości,
    - luźne powiązania i wysoki poziom testowalności – wygenerowany kod źródłowy jest podzielony na luźno powiązane ze sobą komponenty o określonych odpowiedzialnościach. Dodatkowo wygenerowany kod może

umożliwiać Dependency Injection [3], co w rezultacie przekłada się na udostępnienie możliwości testowania tego kodu.

Generacja kodu źródłowego może być również używana na potrzeby wytwarzania aplikacji bazodanowych. Najczęściej kod źródłowy będący wynikiem pracy generatora wykorzystuje biblioteki frameworków bazodanowych. Wygenerowany kod zależy od użytego narzędzia. Również stopień możliwości wpłynięcia na otrzymany kod źródłowy jest zależny od możliwości konfiguracyjnych danego rozwiązania.

Do wygenerowania kodu źródłowego warstwy dostępu do danych używane są następujące elementy:

- Model danych – model, na podstawie którego zostanie wygenerowany kod źródłowy. Zawiera definicje obiektów, które będą przechowywane w bazie danych oraz relacji pomiędzy nimi. Najczęściej odzwierciedla on strukturę bazy danych, ale może być też wzbogacony o dodatkowe informacje. Nie zawsze model musi w całości przekładać się na strukturę bazy danych. Przykładowo dwa obiekty z modelu mogą być odwzorowane jako jedna tabela w relacyjnej bazie danych. Z model danych wiążą się różne podejścia:
  - Model-First – model danych tworzony jest na początku i na jego podstawie powstaje odpowiadająca mu baza danych oraz kod źródłowy do komunikacji z nią,
  - Database-First – model tworzony jest na podstawie istniejącej bazy danych i na jego podstawie generowany jest kod źródłowy,
  - Code-First – baza danych tworzona jest bezpośrednio albo pośrednio (po drodze tworzony jest jeszcze model danych) na podstawie istniejącego kodu źródłowego.

  Istniejące rozwiązania udostępniają pewne możliwości synchronizacji trzech zidentyfikowanych powyżej elementów: baza danych, model danych, kod źródłowy. Model danych może być również ukryty dla użytkownika narzędzia, możliwa jest generacja kodu źródłowego bezpośrednio na podstawie metadanych opisujących bazę danych.
- Konfiguracja – poza samym modelem danych wymagane mogą okazać się dodatkowe dane, które dodatkowo określają jak będzie wyglądał wygenerowany kod źródłowy.
- Szablony – niektóre frameworki dają możliwość tworzenia nowych albo modyfikowania istniejących szablonów, które zostaną użyte do generowania kodu źródłowego. Szablony korzystają z dostarczonego modelu danych i konfiguracji. Definiują jak będzie ustrukturyzowany kod źródłowy oraz jak będzie wyglądała

wynikowa implementacja. Są to fragmenty kodu źródłowego, które zostaną wypełnione przekazanymi danymi.

Generatory kodu oferują wsparcie, które jest wykorzystywane tylko na etapie wytwarzania aplikacji. Natomiast uzyskany za ich pomocą kod źródłowy staje się integralną częścią aplikacji.

2.1.3. DSL

Doman-Specific Language czyli język dedykowany do rozwiązywania określonej dziedziny problemów. DSL może również znaleźć zastosowanie w narzędziach automatyzujących i przyśpieszających implementację aplikacji bazodanowych. W tym kontekście przydatne mogą okazać się języki opisujące struktury danych. Pod uwagę wzięte zostały języki dziedzinowe, które posiadają swoją graficzną reprezentację. Dla frameworków bazodanowych zadaniem graficznego języka dziedzinowego jest definiowanie modelu danych, który będzie pomostem pomiędzy bazą danych a kodem źródłowym. Podstawowymi elementami takiego języka dziedzinowego powinny być obiekty o zdefiniowanej strukturze oraz relacje jakie pomiędzy nimi zachodzą. Mając to na uwadze, taki język mógłby opierać się diagramach klas UML, diagramie związków encji albo innych o podobnych założeniach. Mógłby to być też język, który uwzględnia pojęcia definiowane przez Domain Driven Design, które jest podejściem szeroko wykorzystywanym do modelowania dziedziny problemowej samej aplikacji biznesowej.

Graficzny DSL powinien być łatwy do zrozumienia bez wcześniejszego dużego wkładu na naukę go. Programiści nie potrzebują kolejnego języka do nauki, zamiast tego spodziewają się czegoś co wykorzystuje dobrze im znane koncepcje np. model obiektowy. Jednocześnie powinien on pozwalać na zamodelowanie wszystkich potrzebnych konstrukcji oraz opisanie ich atrybutami. Jest to bardzo istotne, ponieważ generacja kodu źródłowego będzie odbywała się bezpośrednio na podstawie modelu utworzonego zgodnie z założeniami języka dziedzinowego. Dodatkowe informacje niezwiązane bezpośrednio z dziedziną problemową będą mogły być przekazana do generatora kodu źródłowego za pomocą jego konfiguracji.

Warto zaznaczyć, że graficzny DSL to najwyższy poziom wsparcia oferowanego przez frameworki bazodanowe. Opiera się on na dwóch położonych jeden pod drugim poziomach, odpowiednio generatorze kodu oraz bibliotekach wspierających. Generator kodu źródłowego wykorzystuje model danych wytworzony za pomocą graficznego DSL i na jego postawie generuje kod źródłowy wykorzystujący biblioteki wspierające.

2.2. Istniejące rozwiązania

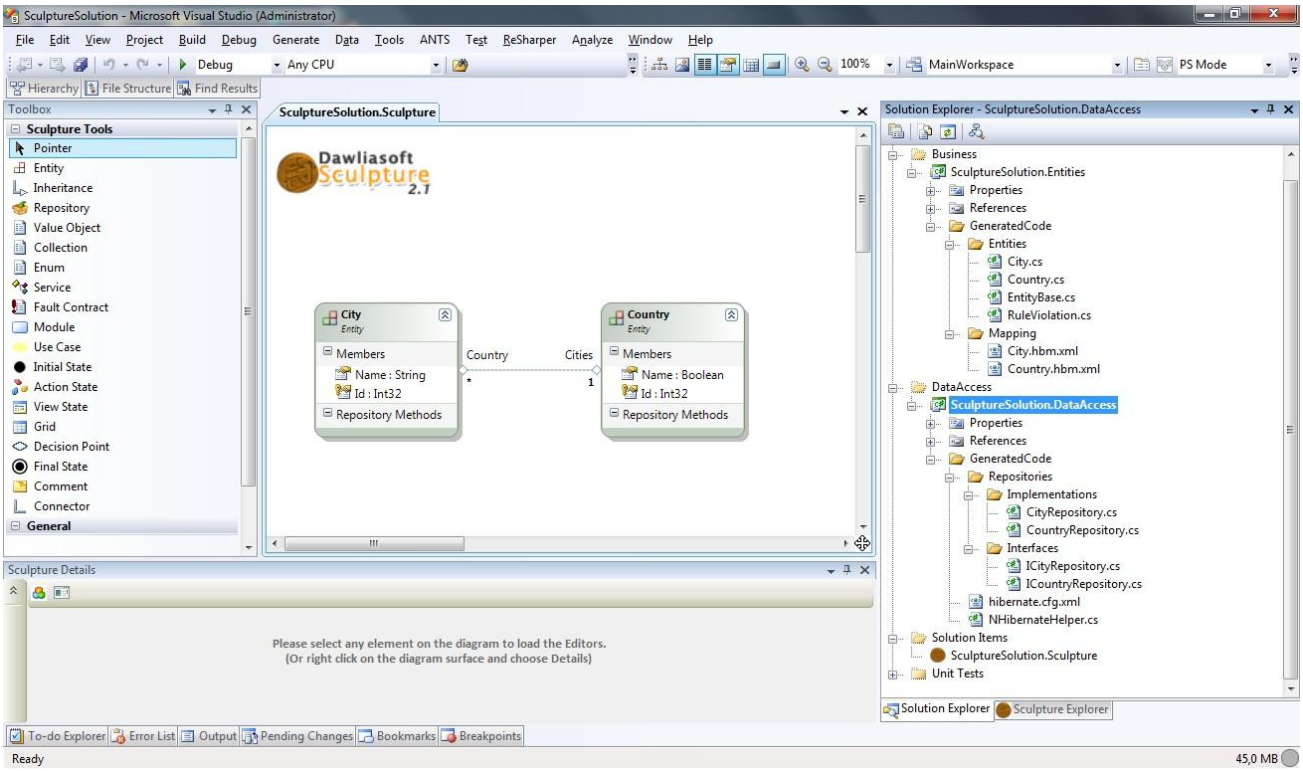
Na rynku istnieją różne rozwiązania wspierające implementację bazodanowych aplikacji biznesowych. W zestawieniu pod uwagę brane były rozwiązania, które oferują najwyższy poziom wsparcia, czyli dają możliwość użycia graficznego języka dziedzinowego w celu utworzenia modelu

danych, na podstawie którego generowany będzie kod źródłowy. Autor nie koncentrował się na możliwościach oferowanych przez same biblioteki wspierające.

2.2.1. Dawliasoft Sculpture 2.1

Sculpture to generator kodu źródłowego opierający się na podejściu Model-driven development. Wspiera wytwarzanie aplikacji biznesowych funkcjonujących na platformie Microsoft .NET. Pozwala na modelowanie wielu komponentów aplikacji, nie tylko warstwy dostępu do danych. Integruje się ze środowiskiem programistycznym Microsoft Visual Studio 2008. Podstawowe założenia Sculpture to:

- pozwala na modelowanie encji, usług, procesów biznesowych i widoków warstwy prezentacji,
- model jest niezależny od technologii, stanowi pewien poziom abstrakcji –oznacza to, że można w trakcie pracy z modelem zmieniać technologię, która zostanie użyta w aplikacji,
- wykorzystuje komponenty nazywane Molds (matryce), które definiują dodatkowe elementy modelu oraz dostarczają szablony do generacji kodu źródłowego. Wybranie matrycy jest jednoznaczne z wybraniem technologii, która zostanie użyta w aplikacji wynikowej. Dystrybucja zawiera dwie matryce dla warstwy dostępu do danych wykorzystujące Entity Framework oraz NHibernate. Programista może stworzyć własne matryce.



Rysunek 1 Praca z wykorzystaniem Sculpture w Visual Studio 2008

Rysunek 1 przedstawia okno Visual Studio z otwartą solucją wykorzystującą model Sculpture. Na podstawie widocznego modelu zostały wygenerowane dwa projekty Entities i DataAccess zawierające kod źródłowy dla encji oraz repozytoriów. Dodatkowo wygenerowany został projekt zawierający testy jednostkowe do zaimplementowania. Wykorzystana została matryca NHibernate.

Zalety rozwiązania:

- operowanie na poziomie abstrakcji, niezależność modelu od technologii,
- możliwość tworzenia dodatkowych matryc, które pozwalają na definiowanie nowych atrybutów modelu oraz szablonów wykorzystywanych podczas generacji kodu,
- plikami wynikowymi generacji nie muszą być tylko klasy ale też pliki XML,
- produkty generacji kodu są uporządkowane w projektach i katalogach,
- obsługa SQL Server, MySql, Oracle,
- wygenerowany kod źródłowy jest testowalny i został wygenerowany z użyciem wzorca Double Derived,
- do wygenerowanych projektów dodawane są referencje do wymaganych bibliotek
- generowane są testy jednostkowe

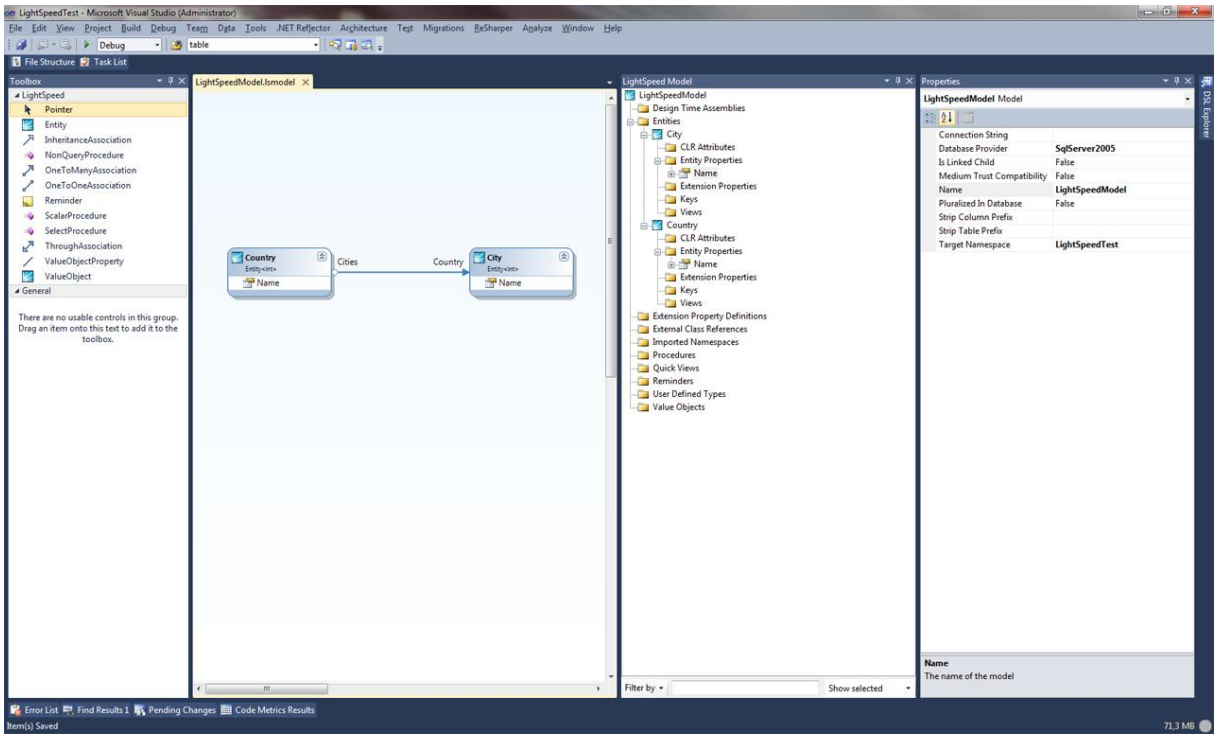
Wady rozwiązania:

- przyjęty poziom abstrakcji ogranicza model tylko i wyłącznie do relacyjnych baz danych,
- DSL zawiera w sobie elementy związane z różnymi dziedzinami problemowymi (encje mieszają się z usługami sieciowymi i widokami warstwy prezentacji),
- brak synchronizacji modelu z bazą danych,
- wygenerowany kod źródłowy nie jest generyczny,
- przyjęta abstrakcja narzuca wzorzec projektowy Repository,
- dla każdej encji wymagane jest definiowanie atrybutu, który posłuży jako klucz główny,
- nowoutworzony plik modelu nie przechodzi walidacji dopóki nie wprowadzi się szczegółów połączenia z bazą danych – trzeba odszukać ten atrybut i go skonfigurować,
- tworzenie nowych matryc posiada pewne ograniczenia i jest mało wygodne, dodatkowo nie jest odporne na błędy - produkty generacji definiowane są w pliku XML, gdzie można użyć odpowiednich fraz, które później zostaną zastąpione np. nazwą konkretnej encji.

2.2.2. Mindscape LightSpeed 3.0

LightSpeed to framework aplikacji bazodanowych na platformę Microsoft .NET oferujący mapowanie obiektowo-relacyjne oraz tworzenie modelu danych za pomocą zintegrowanego z Visual Studio (2008 lub 2010) graficznego języka dziedzinowego. Kluczowe cechy LightSpeed to:

- LightSpeed Designer, czyli graficzny kreator o dużych możliwościach, który pozwala na zaoszczędzenie dużej ilości czasu. Na podstawie modelu wytworzonego przy jego pomocy generowany jest kod źródłowy korzystający z frameworka LightSpeed,
- użycie dobrych praktyk poprzez stosowanie wzorców projektowych związanych z Domain Driven Design: Entity, Value Object, Repository, Unit of Work, Specification,
- testowalność wygenerowanego kodu źródłowego – możliwe jest wpinanie mocków kluczowych interfejsów oraz testowanie transakcji.



Rysunek 2 Praca z wykorzystaniem LightSpeed w Visual Studio 2010

Zalety rozwiązania:

- model zawiera dużo atrybutów służących do konfiguracji zachowania frameworka,
- wsparcie dla wielu silników bazodanowych,
- możliwość modelowania procedur składowanych,
- możliwość synchronizacji bazy danych z modelem – aktualizacja schematu bazy danych na podstawie modelu lub aktualizacja modelu na podstawie schematu bazy danych,
- przy synchronizacji można wybrać elementy, które zostaną uwzględnione,

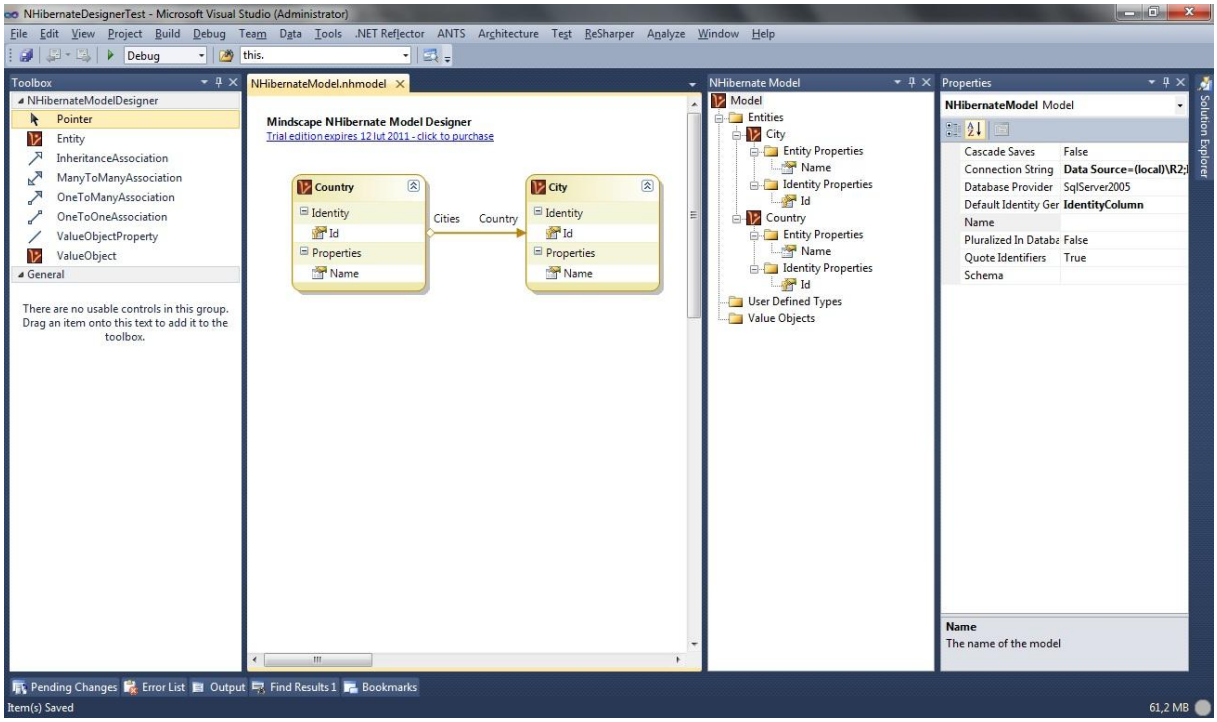
- wygenerowane klasy encji są oznaczone jako częściowe, co umożliwia ich rozszerzanie bez obawy, że niestandardowy kod zostanie nadpisany przez generator,

Wady rozwiązania:

- brak możliwości łatwego wpłynięcia na postać wygenerowanego kodu źródłowego,
- w modelu można użyć typów danych, które nie są obsługiwane przez bazę danych – synchronizacja atrybutu o takim typie danych nie będzie możliwa,
- klasy generowane są do jednego pliku docelowego,
- model nastawiony jest na pracę tylko z relacyjnymi bazami danych,
- wygenerowany kod przeznaczony jest tylko dla warstw dostępu do danych wykorzystujących framework ORM LightSpeed,
- relacja wiele do wiele w kodzie przekłada się na dodatkową encję pośrednią.

2.2.3. Mindscape NHibernate Designer

Mindscape NHibernate Designer oferuje podobne możliwości co Mindscape LightSpeed. Największą różnicą jest to, że wygenerowana warstwa dostępu do danych wykorzystuje NHibernate.



Rysunek 3 Praca z wykorzystaniem NHibernate Designer w Visual Studio 2010

Zalety rozwiązania:

- możliwość wyboru domyślnego sposobu generacji wartości identyfikatorów dla encji – po dodaniu pliku modelu do projektu wyświetlany jest wizard, który proponuje dostępne opcje,

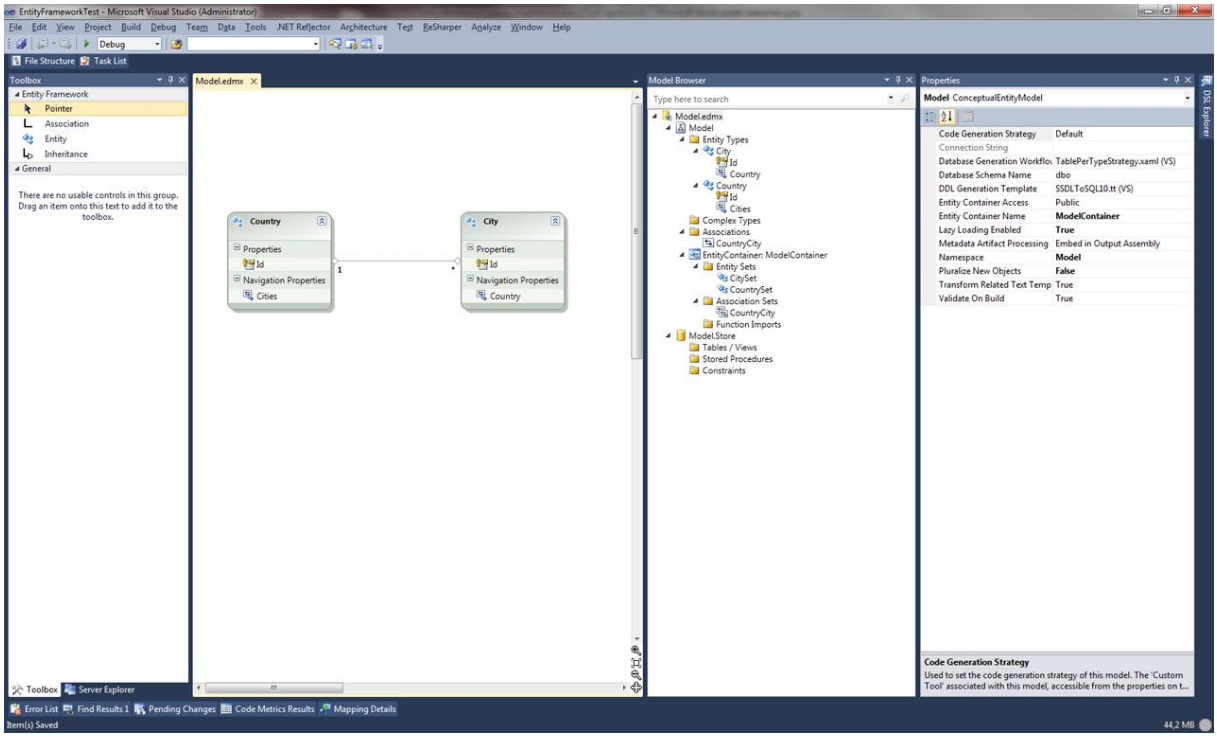
- wsparcie dla wielu silników bazodanowych (to bezpośrednio wynika z możliwości NHibernate),
- możliwość synchronizacji struktury bazy danych i modelu.

Wady rozwiązania:

- cały wygenerowany kod źródłowy jest umieszczany w jednym pliku,
- brak możliwości wpłynięcia na postać otrzymanego kodu,
- narzędzie jest nastawione tylko na jedną technologię – NHibernate,
- model posiada zbyt mało opcji konfiguracyjnych w stosunku do tego co można skonfigurować w mapowaniach NHibernate,
- XML dla mapowań NHibernate jest zaszyty w postaci stringów w wygenerowanym kodzie źródłowym, co oznacza że programista w żaden sposób nie może ich zmodyfikować,
- narzędzie nie dodaje odpowiednich referencji do projektu, w efekcie projekt nie kompiluje się, a programista musi wiedzieć które wersje bibliotek NHibernate są kompatybilne z wygenerowanym kodem i dołączyć je samemu.

2.2.4. Microsoft Entity Framework 4.0

Rozwiązanie dostępne w Visual Studio 2010 bez instalacji żadnych dodatkowych rozszerzeń. Tak jak wyżej opisane narzędzia pozwala na utworzenie graficznego modelu danych i na jego podstawie wygenerowanie kodu źródłowego.



Rysunek 4 Praca z Entity Framework w Visual Studio 2010

Zalety rozwiązania:

- obsługa procedur składowanych,
- możliwość zastąpienia operacji CRUD realizowanych przez ORM procedurami składowanymi,
- możliwość utworzenia modelu na podstawie struktury istniejącej bazy danych,
- możliwość aktualizacji modelu po zmianach w strukturze bazy danych,
- możliwość generacji skryptu DDL na podstawie modelu i uruchomienie go na docelowej bazie danych,
- wizardy ułatwiające pracę z narzędziem,
- możliwość generacji schematu bazy danych na podstawie istniejącego kodu źródłowego,
- możliwość dostarczenia własnej strategii generowania skryptu DDL na podstawie modelu,
- możliwość dostarczenia własnego szablonu, który zostanie użyty do generacji kodu źródłowego.

Wady rozwiązania:

- model nastawiony jest tylko i wyłącznie na relacyjne bazy danych,
- kod źródłowy generowany jest do jednego pliku docelowego – przy większych modelach odnalezienie czegoś w pliku składającym się z kilku tysięcy linii jest utrudnione. Dodatkowe utrudnienie pojawia się, kiedy nad modelem pracuje kilka osób i w projekcie używany jest system kontroli wersji. Podczas wgrywania zmian na serwer programiści muszą rozwiązywać konflikty pojawiające się w pliku z wygenerowanym kodem nawet jeżeli zmiany, których dokonali dotyczyły innych klas,
- brak możliwości rozszerzania modelu – nie można definiować dodatkowych atrybutów, które wzbogacały by model.

2.3. Podsumowanie istniejących rozwiązań

Wszystkie wyżej przedstawione rozwiązania oferują możliwość graficznego modelowania struktury danych za pomocą określonego języka dziedzinowego. We wszystkich przypadkach jest to język bardzo zbliżony do modelu zgodnego z kanonami obiektowych języków programowania. Pozwala na definiowanie encji oraz asocjacji pomiędzy nimi. Istnieje również możliwość określania hierarchii dziedziczenia oraz dodawanie złożonych atrybutów encji. W części rozwiązań język dziedzinowy został rozszerzony o dodatkowe elementy. LightSpeed pozwala na dodawanie do modelu definicji procedur składowanych, natomiast Sculpture daje możliwość definiowania znacznie większej liczby bytów, ale niekoniecznie związanych z warstwą dostępu do danych.

Każdy z opisanych frameworków oferuje pewien poziom abstrakcji dostarczany za pomocą języka dziedzinowego. O ile z punktu widzenia rozszerzalności i uniwersalności rozwiązania, w większości frameworków zbyt usztywniona specjalizacja dziedziny domenowej jest minusem, to nastawienie na konkretną technologię można tłumaczyć tym, że stworzenie generycznego rozwiązania jest znacznie bardziej skomplikowane i kosztowniejsze. Dodatkowo nie zawsze jest taka potrzeba. Wszystko zależy od celu jaki został obrany przez firmę dostarczającą narzędzia dla programistów. Przykładowo firma Mindscape wypuściła dwa bardzo podobne do siebie rozwiązania, jedno jest dedykowane programistom opierającym swoje aplikacje o warstwę dostępu realizowaną za pomocą NHibernate, a drugie zwolennikom frameworka LightSpeed. Podstawne wydaje się pytanie czemu nie można było zintegrować obsługi NHibernate i LightSpeed w jednym produkcie. Zapewne na tą decyzję wpłynęły nie tylko kwestie techniczne, ale również marketingowe. Programiści najczęściej mają pełną świadomość jaki framework warstwy dostępu do danych będzie najodpowiedniejszy dla wytwarzanej przez nich aplikacji. W związku z tym wybierają odpowiednie narzędzie do pracy i nie muszą przy tym płacić za funkcjonalność, z której nie skorzystają. Sytuacja wygląda inaczej w przypadku, kiedy programista chce mieć jedno kompleksowe rozwiązanie, które pozwala mu na użycie różnych technologii. Przykładowo jeden projekt wymaga od niego użycia NHibernate, ale następny powinien być już oparty o mechanizmy Entity Framework. Wygodne może okazać się rozwiązanie, które oferuje obie opcje. Programista jest dobrze zaznajomiony z używanym przez siebie narzędziem i nie musi zmieniać swoich przyzwyczajeń, co miałoby miejsce gdyby musiał zacząć używać nowego oprogramowania. Takim generycznym rozwiązaniem jest Sculpture, które pozwala utworzyć model niezależny od technologii i wybrać ją dopiero później, przed generacją kodu. Wybór technologii może dodać nowe elementy do języka dziedzinowego, co odsłania nowe możliwości konfiguracyjne aspektów ściśle związanych z konkretną technologią. Tego typu rozszerzalność jest znaczącym plusem tego rozwiązania. Jego wartość dodatkowo podnosi możliwość tworzenia własnych rozszerzeń, co pozwala na użycie narzędzia do tworzenia warstw dostępu do danych, które nie są wspierane out of the box.

Wszystkie przedstawione rozwiązania nastawione są na pracę z relacyjnymi bazami danych. Ich języki dziedzinowe zawierają w sobie pojęcia takie jak tabele czy klucze główne, czyli ściśle związane z relacyjnymi źródłami danych,. Jest to dobre podejście w przypadku, kiedy rzeczywiście istnieje potrzeba użycia relacyjnej bazy danych i rozwiązania problemu niezgodności impedancji. Jednak co w przypadkach, kiedy aplikacja została zaprojektowana tak żeby korzystać z innego źródła danych np. z obiektowej albo dokumentowej bazy danych.

Znaczącą zaletą wymienionych rozwiązań jest ich integracja ze środowiskiem programistycznym. Brak tej integracji od razu dyskwalifikuje konkurencję. Narzędzia dla programistów powinny być w pełni zintegrowane ze środowiskiem używanym na co dzień przez

programistów. Zwłaszcza, że Visual Studio oferuje duże możliwości rozszerzalności. Pliki z wygenerowany kodem od razu są załączane do projektu, co pozwala na natychmiastowe rozpoczęcie pracy z uzyskanym kodem i kompilację go.

Najistotniejsza funkcjonalność, czyli generacja kodu źródłowego na postawie modelu jest wspierana przez wszystkie zaprezentowane rozwiązania. Najlepsze podejście do tej funkcjonalności posiada Sculpture, ponieważ wygenerowany kod nie trafia do tylko jednego pliku docelowego, tak jak dzieje się to w przypadku pozostałych rozwiązań. Umieszczenie klas w oddzielnych plikach jest dobrą praktyką i jest to naturalne podejście każdego programisty. Dodatkową zaletą narzędzia jest możliwość generowania nie tylko klas ale również innych potrzebnych plików np. w formacie XML. Pogrupowanie ich w folderach i projektach też pozytywnie wpływa na czytelność kodu i łatwość jego utrzymywania. W dużych aplikacjach logiczny podział kodu na projekty jest rzeczą wręcz obowiązkową. Odpowiedni podział kodu oraz zachowanie tylko narzuconych przez architekturę zależności pomiędzy projektami jest bardzo znaczącym dla aplikacji i niełatwym zadaniem. Pewnym rozczarowaniem jest fakt, że Sculpture nie daje możliwości wyboru projektów docelowych spośród projektów już istniejących w solucji. Z generowaniem kodu do wielu plików docelowych mogą jednak wiązać się dodatkowe utrudnienia dla narzędzia, które musi wiedzieć, które pliki wygenerowało i przy kolejnej generacji kodu nadpisać, dodać lub usunąć odpowiednie pliki.

Znaczącym aspektem w kontekście programów generujących kod źródłowy warstwy dostępu do danych bardzo istotna jest możliwość wpłynięcia na jego postać. Z jednej strony narzędzie powinno być tak zaprojektowane żeby generowało kod wysokiej jakości, który nie wymaga dodatkowej uwagi programisty. Jest to istotne dla mnie doświadczonych programistów, którzy niekoniecznie mają profesjonalną wiedzę na temat tworzenia kodu odpowiedzialnego za dostęp do danych. Darzą oni framework dużym zaufaniem, tworzą model danych, generują kod i później tylko z niego korzystają bez zagłębiania się w otrzymaną implementację. Jest to również przydatne, kiedy istnieje potrzeba szybkiego wytworzenia mało złożonej aplikacji albo prototypu. Z drugiej strony narzędzie powinno być na tyle elastyczne żeby umożliwić doświadczonym programistom dostosować zachowanie generatora do własnych potrzeb. Programiści bazując na swoim doświadczeniu często stosują odpowiednie praktyki, wzorce projektowe i posiadają własne implementacje rozwiązujące określone problemy. Pożądaną funkcjonalnością byłoby dostosowanie generatora kodu żeby uwzględniał te wszystkie rzeczy. Dodatkowo w bardziej złożonych aplikacjach istnieje potrzeba wzbogacenia warstwy dostępu do danych o dodatkowy kod odpowiedzialny np. za data caching, obsługę błędów albo logowanie. Sculpture w swoim mechanizmie rozszerzeń daje możliwość użycia własnych szablonów kodu oraz określenia plików, które powstaną podczas generacji. Entity Framework też daje możliwość dostarczenia własnego szablonu kodu, ale nadal kod będzie wygenerowany do jednego pliku docelowego. Mechanizm rozszerzeń Sculpture, czyli matryce, pozostawia jednak wiele do

życzenia. Określenie produktów generacji kodu i powiązanie ich z szablonami jest realizowane za pomocą plików XML. Jest to mało wygodny i nieodporny na błędy sposób. Natomiast zdefiniowanie warunków określających, czy dany element modelu ma zostać użyty do generacji kodu, odbywa się w kodzie, który później musi być załączony do matrycy jako dodatkowa biblioteka.

Wygenerowany kod powinien charakteryzować się wysoką jakością, powinien być wydajny, czytelny i łatwo testowalny. Sculpture i LightSpeed generują interfejsy dla podstawowych klas, dzięki czemu da się je łatwo przetestować w testach jednostkowych.

Istotnym aspektem jest również sposób radzenia sobie z naturą generacji kodu. Kod wygenerowany przez narzędzie jest nadpisywany przy kolejnej generacji. Oznacza to że istnieje potrzeba udostępnienia mechanizmów, które umożliwią programiście rozszerzanie i modyfikowanie tego kodu już po generacji. Najprostszą praktyką stosowaną przez wszystkie opisane rozwiązania jest oznaczenie wygenerowanych klas jako częściowe. Jeżeli wygenerowany kod zawiera metody, to warto również dać możliwość nadpisania ich implementacji. Sculpture daje taką możliwość poprzez zastosowanie wzorca Double Derived.

Na wygodę użycia danego narzędzia wpływają też takie rzeczy jak dodawanie odpowiednich referencji do projektu, w którym umieszczany jest wygenerowany kod. Przykładowo dla warstwy dostępu do danych opartej o NHibernate wymagane są jego biblioteki. NHibernate Designer nie dodaje wymaganych referencji do projektu, przez co nie można ukończyć jego kompilacji.

Przydatną funkcjonalnością, która znacznie ułatwia pracę nad warstwą dostępu do danych jest możliwość synchronizacji modelu danych ze strukturą bazy danych. Narzędzia oferowane przez MindScape oraz Microsoft pozwalają na utworzenie modelu danych na podstawie struktury istniejącej bazy danych. Istnieje również możliwość przeprowadzenia aktualizacji schematu bazy danych na podstawie modelu danych. MindScape pozwala na wybór elementów modelu, które mają zostać uwzględnione podczas tej operacji. Natomiast Entity Framework domyślnie generuje skrypt DDL dla całego modelu. Oznacza to, że wymagane jest całkowite nadpisanie schematu bazy danych, co wiąże się z utratą danych. Możliwe jest jednak dostarczenie innej strategii generacji skrypty DDL i zmiana tego zachowania.

2.4. Charakterystyka proponowanego rozwiązania

Odpowiednie narzędzie wspierające implementację warstwy dostępu do danych powinno posiadać najlepsze cechy wyżej opisanych rozwiązań oraz oferować nowe możliwości.

2.4.1. Założenia

Praca nie koncentruje się na implementacji frameworka ułatwiającego dostęp do danych, ani frameworka ORM. Jej celem nie jest wytworzenie bibliotek wspomagających. Zamiast tego główny nacisk został położony na uzyskanie platformy zintegrowanej ze środowiskiem programistycznym

Visual Studio 2010, która pozwala na graficzne modelowanie za pomocą przyjętego języka dziedzinowego i generację kodu źródłowego na jego podstawie. Bardzo istotnym założeniem było osiągnięcie wysokiego poziomu rozszerzalności. Tworzenie rozwiązań generujących kod w jeden określony sposób jest bardzo ryzykowną praktyką. W świecie ciągle zmieniających się wymagań oraz bezustannego rozwoju technologii takie rozwiązania z góry skazane są na bardzo krótki okres życia i nieopłacalność całej inwestycji. Właśnie dlatego tak duży nacisk został położony na elastyczność rozwiązania i przekazanie władzy programistom. To właśnie oni będą korzystali z narzędzia. Rozwiązanie nie powinno utrudniać ani ograniczać ich pracy. Powinno natomiast ją przyśpieszać i ułatwiać. Dodatkowo powinno zapewniać im odpowiedni poziom kontroli. Rozwiązanie jest kierowane do doświadczonych programistów, którzy mają pełną świadomość kodu, którego używają i wiedzą co chcą osiągnąć, ale potrzebują narzędzia, które pozwoli im zautomatyzować zadania takie jak implementacja warstwy dostępu do danych. Nawet jeżeli korzystają z autorskich implementacji to będą w stanie zintegrować je z oferowaną platformą i zaoszczędzić dużo czasu. Uzyskany czas będą mogli poświęcić na rozwiązywanie bardziej złożonych problemów takich jak implementacja logiki biznesowej aplikacji

Istotą projektów informatycznych dotyczących wytwarzania aplikacji biznesowych jest fakt, że każdy z nich jest inny. Jednak dla większości z nich da się wydzielić pewne zadania, które zawsze będą takie same. Są to fragmenty oprogramowania, które koncepcyjnie zawsze będą rozwiązywały ten sam problem, ale ze względu na różnice dziedzin problemowych poszczególnych projektów nie da się napisać całości kodu raz i później zdać się tylko na ponowne użycie. Jednym z takich fragmentów jest warstwa dostępu do danych. Tylko tego typu fragmenty mogą być celem generatora kodu. Dla pozostałych części aplikacji, których nawet nieznaczne różnice pomiędzy projektami mogą stanowić dużą przeszkodę dla narzędzi, bardziej opłacalna jest bezpośrednia implementacja i wsparcie się odpowiednimi bibliotekami. Dopóki nic się nie zmieni w temacie wysokopoziomowego ponownego użycia, narzędzie takie jak platforma będąca przedmiotem pracy mają duże pole do działania.

2.4.2. Wymagania

Podstawowe wymagania stawiane przed proponowanym rozwiązaniem zostały wymienione poniżej:

- utworzenie podstawowego języka dziedzinowego
    - język jest prosty i zawiera podstawowe koncepcje obiektowego języka programowania takie jak:
        - obiekt,
        - atrybut obiektu zdefiniowany w postaci nazwy i typu danych – dla każdego atrybutu można określić czy jest on opcjonalny,

        - asocjacja binarna – dla każdej strony asocjacji można zdefiniować rolę oraz liczność,
        - dziedziczenie
    - DSL nie zawiera elementów specyficznych dla konkretnych typów źródeł danych, przykładowo nie ma w nim pojęć ściśle związanych z relacyjnymi bazami danych,
    - DSL nie zawiera elementów specyficznych dla konkretnej technologii implementacji warstwy dostępu do danych,
    - DSL posiada swoją graficzną reprezentację,
    - za jego pomocą można tworzyć model danych,
    - poprawność modelu jest walidowana
- generacja kodu źródłowego
    - produktu generacji od razu dołączane są do aktywnej solucji,
    - możliwość generowania kodu do wielu plików docelowych, które mogą być umieszczane w różnych projektach i folderach,
    - poza kodem generowane mogą być też inne, dowolne zasoby np. skrypty SQL albo pliki konfiguracyjne XML,
    - możliwość dodawanie dowolnych, już istniejących plików do solucji,
    - możliwość dodawania referencji do projektów
- integracja ze środowiskiem programistycznym Visual Studio 2010
    - łatwa i bezproblemowa instalacja,
    - nowy typ plików pozwalających na pracę z oferowanym narzędziem – pliki tego typu można dodawać tak samo łatwo jak standardowe,
    - mechanizm wizardów umożliwiający interakcję użytkownika platformy z narzędziem,
    - architektura pozwalająca na zintegrowanie rozwiązania z przyszłymi wersjami środowiska programistycznymi
- mechanizm rozszerzalności
    - rozszerzenia języka domenowego,
    - możliwość zbierania dodatkowych danych konfiguracyjnych od użytkownika,
    - możliwość dostarczenia własnej implementacji całego procesu generacji kodu i umieszczania go w solucji,
    - rozszerzenia tworzone w kodzie, z wykorzystaniem wygodnego API,
    - rozszerzenia są łatwe do zainstalowania

2.4.3. Funkcjonalności

Podstawowe funkcjonalności oferowane przez rozwiązanie to:

- tworzenie modelu danych za pomocą graficznego kreatora,
- generacja kodu źródłowego na podstawie modelu,
- synchronizacja modelu danych ze strukturą źródła danych.

3. Narzędzia i technologie zastosowane przy realizacji pracy

Podczas projektowania i implementacji prototypu rozwiązania zostały użyte poniżej opisane narzędzia oraz technologie.

3.1. Microsoft Visual Studio 2010

Środowisko programistyczne dedykowane wytwarzaniu oprogramowania na platformę Microsoft .NET. Wersja 2010 pozwala na tworzenie rozwiązań opartych o najnowszą wersję .NET Framework 4.0. Visual Studio 2010 jest złożonym środowiskiem oferującym bardzo dużo funkcjonalności wspierających projektowanie, implementację oraz testowanie oprogramowania. Jest dostępne w kilku edycjach zaczynając od darmowej wersji Express i kończąc na najbardziej rozbudowanej wersji Ultimate.

Visual Studio 2010 oferuje zupełnie nowe możliwości jego rozszerzania. Wprowadzono narzędzie Extension Manager, które pozwala w łatwy sposób zarządzać rozszerzeniami. Ich instalacja i deinstalacja stała się bardzo łatwa. Jest to związane z tym, że powstał nowy format przeznaczony dla paczek instalacyjnych zawierających rozszerzenia (VSIX). Teraz całe rozszerzenie zawiera się w jednym pliku/paczce, która zawiera w sobie manifest oraz ładunek. Jest to duże ułatwienie dla dystrybucji rozszerzeń.

W pracy Visual Studio 2010 posłużyło przede wszystkim do wytworzenia prototypu rozwiązania, ale również jako środowisko, w którym podczas badania stanu sztuki testowane były różne rozszerzenia. Visual Studio 2010 zostało też wybrane, jako środowisko, z którym będzie integrował się prototyp.

3.2. Microsoft SQL Server 2008 R2

Jest to platforma bazodanowa typu klient-serwer. Do zapytań jest używany głównie język Transact-SQL. Microsoft SQL Server 2008 R2 jest oznaczony wersją 10.5 i został wydany przez Microsoft w roku 2010.

W pracy posłużył on jako docelowy silnik bazodanowy jednej z implementacji warstw dostępu do danych oferowanych przez prototyp.

3.3. Microsoft SQL Server Management Studio

Narzędzie pozwalające na połączenie się z SQL Server i zarządzanie znajdującymi się tam bazami danych. Narzędzie zawiera zarówno edytor skryptów, jak i graficzne narzędzia, które pozwalają na wykorzystywanie funkcjonalności serwera.

3.4. UML

Język formalny służący do modelowania systemów informatycznych posiadający swoją reprezentację graficzną. Najnowsza wersja języka (2.2) wyróżnia 17 różnych typów diagramów.

Przy projektowaniu prototypu zostały użyte notacje służące do tworzenia diagramów struktur, a konkretnie diagramów klas.

3.5. Microsoft Visio 2010

Narzędzie służące do tworzenia diagramów i schematów wykorzystujące grafikę wektorową. Visio umożliwia również tworzenie diagramów związanych z wytwarzaniem oprogramowania. Ta funkcjonalność została wykorzystana przy pracy do wykonania diagramów klas UML.

3.6. WPF

Windows Presentation Foundation (WPF) jest to silnik graficzny oraz API dostępne w .NET Framework, które bazuje na języku XAML.

Technologia WPF została wykorzystana do implementacji warstwy prezentacji prototypu – ekranów wyświetlanych przez kreatorów konfiguracji.

3.7. Microsoft Expression Blend 4

Narzędzie do tworzenia graficznych interfejsów użytkownika. Jest edytor WYSIWYG przeznaczony do projektowania interfejsów bazujących na języka XAML, czyli aplikacji WPF albo Silverlight. Narzędzie wchodzi w skład zestawu oprogramowania Expression Studio.

W pracy zostało wykorzystane jako pomoc przy utworzeniu widoków formularzy wizardów zbierających dane od użytkownika.

3.8. JetBrains ReSharper 5.0

Rozszerzenie do Visual Studio zwiększające produktywność oraz dodające nowe możliwości refaktoryzacji kodu. Oprócz tego oferuje ono statyczną analizę kodu w całej solucji. Pozwala to na wykrywanie błędów w kodzie bez jego kompilacji. Rozszerzenie jest używane przez autora pracy podczas realizacji wszystkich zadań programistycznych.

3.9. Microsoft .NET Framework 4.0

Platforma programistyczna składająca się ze środowiska uruchomieniowego oraz zestawu bibliotek, które służą jako podstawa do budowania aplikacji.

3.10. Microsoft Visual C# 4.0

Obiektowy język programowania. Skompilowany kod napisany w języku C# jest wykonywany na platformie uruchomieniowej .NET Framework. Cały kod źródłowy prototypu został napisany z użyciem tego języka.

3.11. Code Contracts

Mechanizm pozwalający na stosowanie programowania kontraktowego (Design by contract). Dedykowane API pozwala na pisanie kodu służącego do weryfikacji działania konkretnych elementów programu i sprawdzania czy wszystkie warunki wymagane do prawidłowego działania kodu zostały spełnione.

W prototypie głownie używane były sprawdzenia warunków początkowych poszczególnych metod. Bardzo przydatna okazała się również możliwość definiowania warunków dotyczących interfejsów. Tego typu warunki są egzekwowane od każdej klasy implementującej dany interfejs.

3.12. T4

Text-Template Transformation Toolkit (T4) jest frameworkiem zawartym w Visual Studio, generującym tekst na podstawie szablonów. T4 zostało użyte w implementacji prototypu do generacji kodu źródłowego oraz innych artefaktów.

3.13. Visual Studio Visualization and Modeling SDK (VMSDK)

VMSDK pozwala na tworzenie narzędzi programistycznych bazujących na modelach [7]. Narzędzia te można później zintegrować z Visual Studio. Podstawą użycie VMSDK jest utworzenie modelu definiującego język dziedzinowy, który będzie obowiązywał w wytwarzanym narzędziu. Można powiedzieć, że VMSDK jest to DSL, który służy do budowania innych DSL.

W prototypie VMSDK zostało użyte w celu zintegrowania rozwiązania z Visual Studio oraz umożliwienia graficznego tworzenia modelu danych za pomocą zaprojektowanego języka dziedzinowego.

3.14. ESENT

Środowisko uruchomieniowe ESE zawarte w każdym systemie operacyjnym Windows. ESE (Extensible Storage Engine) jest to technologia składowania danych opracowana przez Microsoft. Jest to rdzeń takich mechanizmów jak Microsoft Exchange Server albo Active Directory. Zadaniem ESE jest umożliwienie aplikacjom składowanie oraz odczytywanie danych przy użyciu metody ISAM (Indexed Sequential Access Method). Istotą ISAM jest indeksowanie danych w celu ich szybkiego odczytu.

W prototypie technologia ESENT została użyta w celu przechowywania dodatkowych danych, które nie są bezpośrednio związane z językiem dziedzinowym i co za tym idzie, nie są serializowane do pliku modelu danych.

3.15. ESENT Managed Interop

Biblioteka pozwalająca na dostęp do ESENT z poziomu kodu zarządzalnego, czyli w tym przypadku z poziomu C#. Biblioteka oferuje bardzo przydatny mechanizm – PersistentDictionary. Jak

sama nazwa wskazuje jest to kolekcja składająca się z par klucz-wartość. Kolekcja jest przechowywana w bazie danych ESENT. Taka konstrukcja pozwoliła na szybką implementację mechanizmu key-value store, który jest używany przez prototyp do przechowywania dodatkowych danych w uniwersalny sposób.

3.16. Ninject

Biblioteka zawierająca implementację kontenera Inversion of Control. Została wybrana spośród wielu innych rozwiązań tego typu dostępnych na platformę .NET z powodu jej dużych możliwości przy zachowaniu prostoty użycia.

Ninject pozwolił na realizację wielu założeń architektonicznych prototypu m.in. zastosowania wzorca projektowego Dependency Injection oraz odpowiedniego pozyskiwania obiektów oraz ich zależności.

3.17. MEF

Managed Extensibility Framework (MEF) to część .NET Framework pozwalająca na tworzenie rozszerzalnych aplikacji, które nie muszą mieć świadomości, jakie rozszerzenia zostaną użyte.

MEF został użyty w prototypie do realizacji mechanizmu rozszerzalności.

3.18. MVVM Light Toolkit

Biblioteka wspierająca zastosowanie wzorca projektowego MVVM w aplikacjach używających technologii WPF. Właśnie do tego celu biblioteka została użyta w implementacji prototypu.

3.19. ODRA

ODRA (Object Database for Rapid Application development) jest to obiektowo zorientowane środowisko programistyczne i bazodanowe. Oparte jest na obiektowym języku programowania SBQL (Stack-Based Query Language) ściśle powiązanym z możliwością tworzenia zapytań.

ODRA została użyta jako technologia, na której opiera się jedna z implementacji warstwy dostępu do danych prototypu.

3.20. NHibernate

Framework ORM dla aplikacji opartych o platformę .NET Framework. Jest to rozwiązanie udostępniane na licencji Open Source , wokół którego skupiona jest duża społeczność specjalistów IT. Żeby skorzystać z tego frameworka trzeba utworzyć pliki XML zawierające mapowania pomiędzy modelem relacyjnym a obiektowym.

NHibernate został użyty jak technologia, na której opiera się jedna z implementacji warstwy dostępu do danych prototypu.

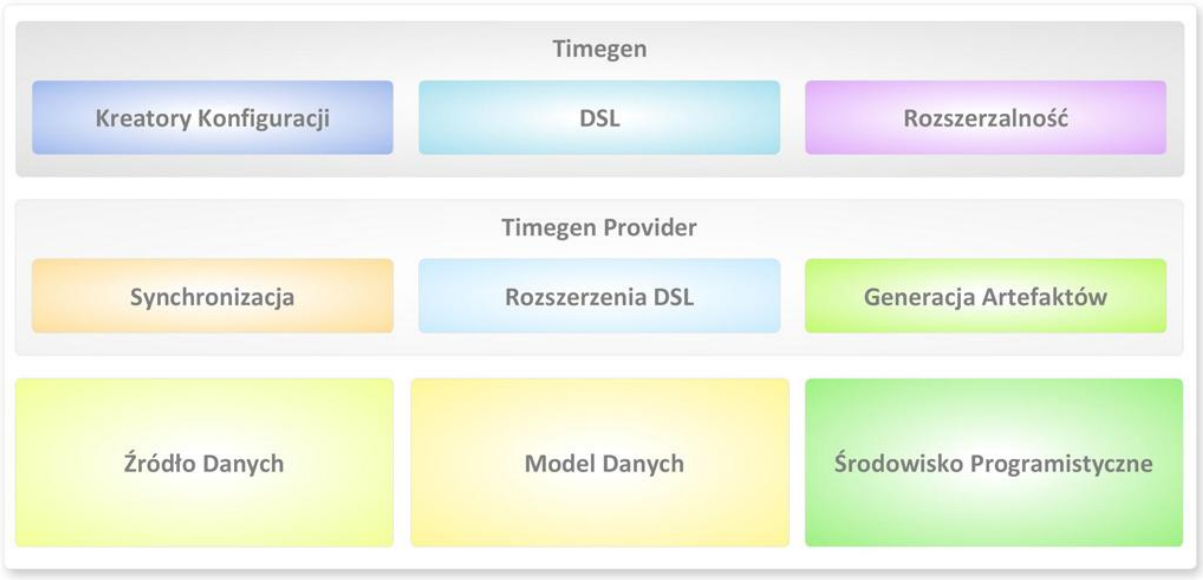
3.21. Fluent NHibernate

Biblioteka ułatwiająca pracę z NHibernate. Pozwala na definiowanie mapowań bezpośrednio w kodzie z użyciem API zaprojektowanego w stylu fluent.

4. Propozycja nowego rozwiązania

W celu zrealizowania założeń i postawionych wymagań powstała koncepcja oraz prototyp platformy o nazwie kodowej Timegen.

4.1. Architektura



Rysunek 5 Komponenty platformy Timegen

Rysunek 5 przedstawia ogólną architekturę platformy Timegen. Przy projektowaniu rozwiązania zwrócono dużą uwagę na elastyczność rozwiązania. W związku z tym powstała koncepcja mechanizmu rozszerzeń. Istotny jest fakt, że sama platforma Timegen nie oferuje żadnych kompletnych funkcjonalności. Wymagany jest Timegen Provider, który dostarczy implementację umożliwiającą pracę z konkretną technologią dostępu do danych.

Wszystkie operacje dostarczane przez platformę bazują na trzech komponentach:

- źródło danych,
- model danych,
- środowisko programistyczne.

4.1.1. Źródło danych

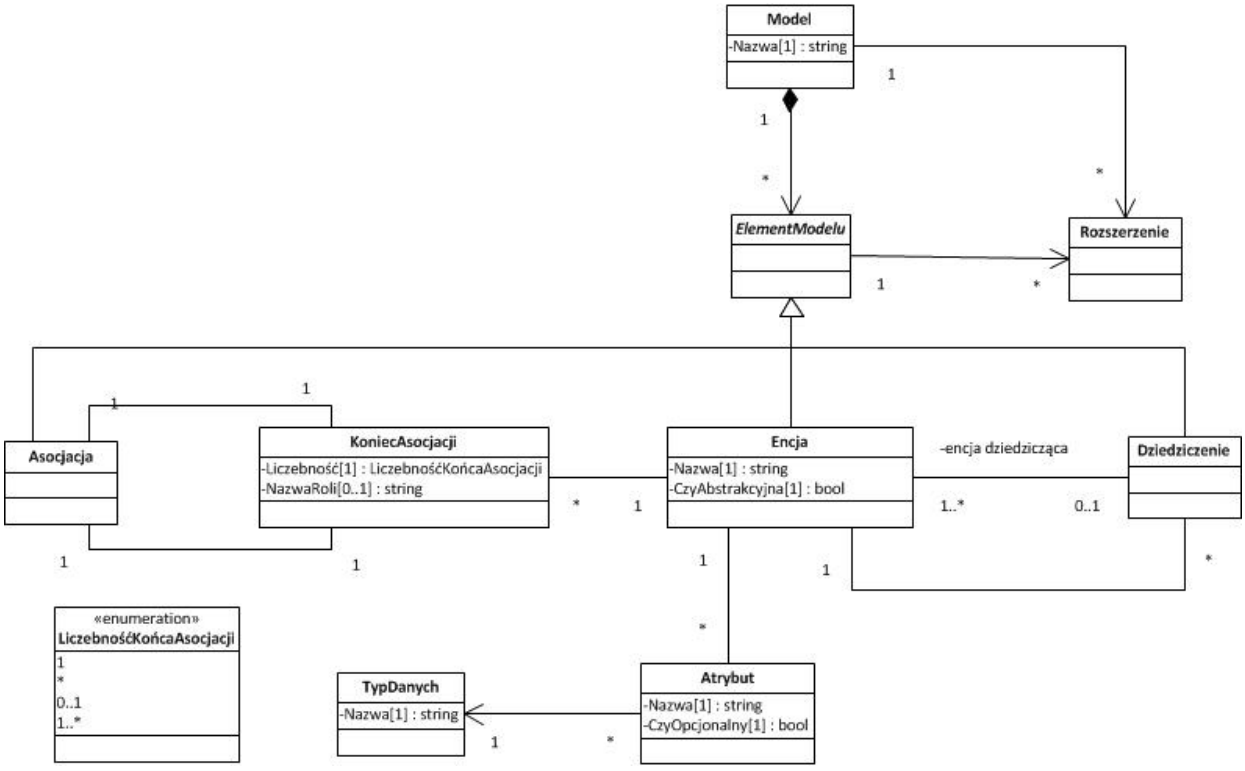
Źródło danych, czyli bardzo istotny element w aplikacjach biznesowych. Odpowiedzialny za trwałe przechowywanie danych wykorzystywanych przez aplikację. Platforma zakłada możliwość ingerencji w strukturę konkretnego źródła danych. W celu synchronizacji modelu danych ze źródłem danych, źródło danych powinno udostępniać metadane opisujące jego strukturę oraz możliwość modyfikacji tej struktury.

Timegen nie narzuca potrzeby wykorzystania określonego rodzaju źródła danych ani związanej z nim technologii. Jednak uzasadnione jest użycie tylko takich źródeł danych, których struktura może

w jakiś sposób przełożyć się na model danych. Oznacza to, że źródło danych powinno wymuszać podział na poszczególne byty i w jakiś sposób definiować powiązania pomiędzy nimi. W związku z tym platforma skierowana jest głównie do aplikacji opierających się na relacyjnych i obiektowych bazach danych.

4.1.2. Model danych

Model definiujący w jakiś sposób podzielone i powiązane ze sobą będą dane używane przez aplikację. Timegen dostarcza podstawowy model danych, który nie zawiera w sobie pojęć konkretnych dla danego typu źródła danych czy technologii. Stworzenie modelu danych będzie zadaniem użytkownika platformy. Elementy modelu mogą zostać również stworzone podczas jego synchronizacji ze źródłem danych.



Rysunek 6 Diagram klas UML modelu danych

Rysunek 6 przedstawia strukturę modelu danych oferowanego przez platformę Timegen. Podstawowym założeniem jest to, że model składa się z różnych elementów. Dostępne są elementy następujących typów:

- **Encja**

  Posiada swoją nazwę oraz zestaw atrybutów. Każdy z atrybutów musi zostać nazwany i musi mieć określony typ danych.

- **Asocjacja**

  Asocjacja binarna, której każdy z końców musi być powiązany z jedną encją i mieć określoną liczność. Dodatkowo istnieje możliwość określenia nazwy roli jaką encja pełni w asocjacji.

- **Dziedziczenie**

  Pozwala na zdefiniowanie drzewa dziedziczenia. Dziedziczenie wielokrotne nie jest wspierane.

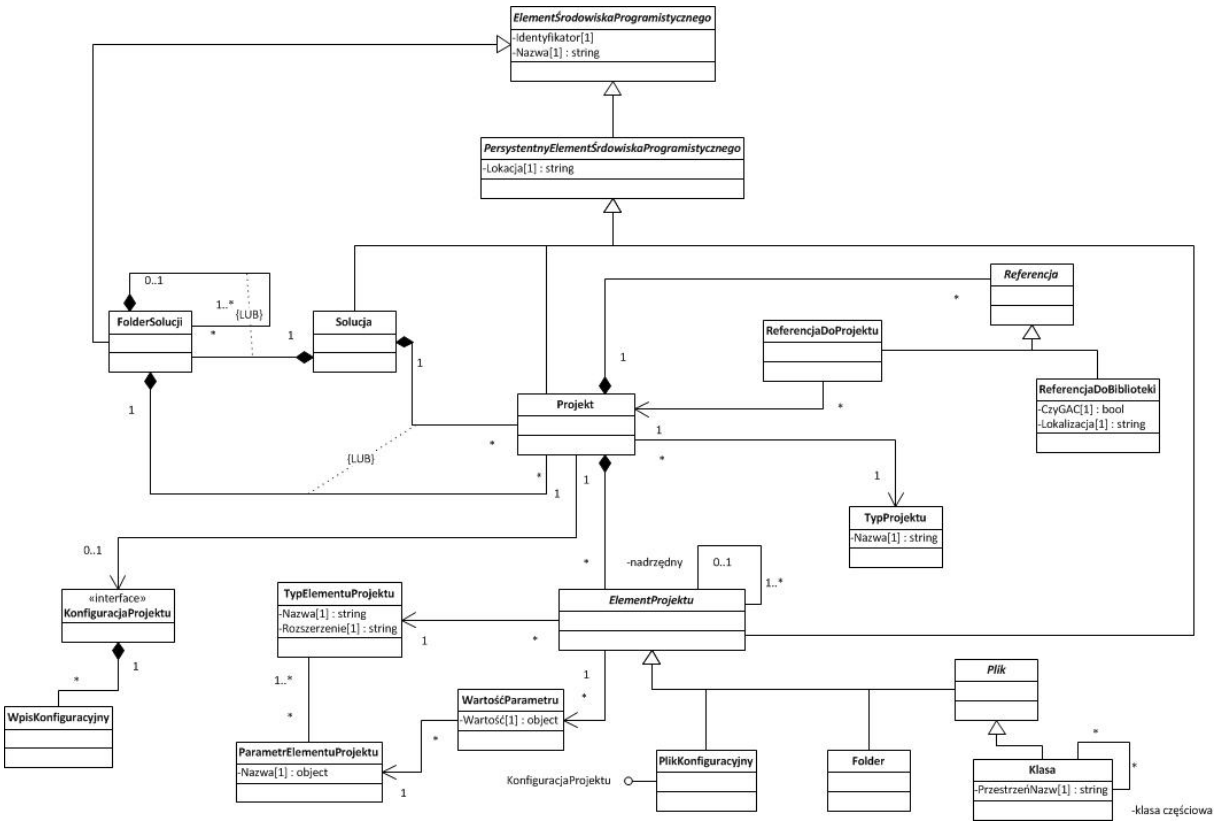
Każdy element modelu jak również sam model posiada rozszerzenia zawierające dodatkowe dane. Rozszerzenia mogą być definiowane przez providerów.

4.1.3. Środowisko programistyczne

Komponent platformy odpowiedzialny za integrację ze środowiskiem programistycznym. Jest z nim powiązane API, które pozwala na wykonywanie operacji na strukturze solucji z poziomu kodu platformy Timegen.

Timegen dostarcza własny model środowiska programistycznego oraz wygodne API do operowania na nim. Pozwala to na odseparowanie całej platformy od konkretnej wersji środowiska programistycznego Visual Studio. Teoretycznie daje to również możliwość zintegrowania platformy z innym środowiskiem programistycznym. Dodatkową zaletą takiego rozwiązania jest ułatwienie pracy z bardzo niejasnym API udostępnionym przez Microsoft – Visual Studio automation object model [4].

Rysunek 7 przedstawia model używany przez platformę Timegen do odwzorowania struktury solucji otwartej w środowisku programistycznym. Uwzględnia on projekty, referencje, foldery i pliki.



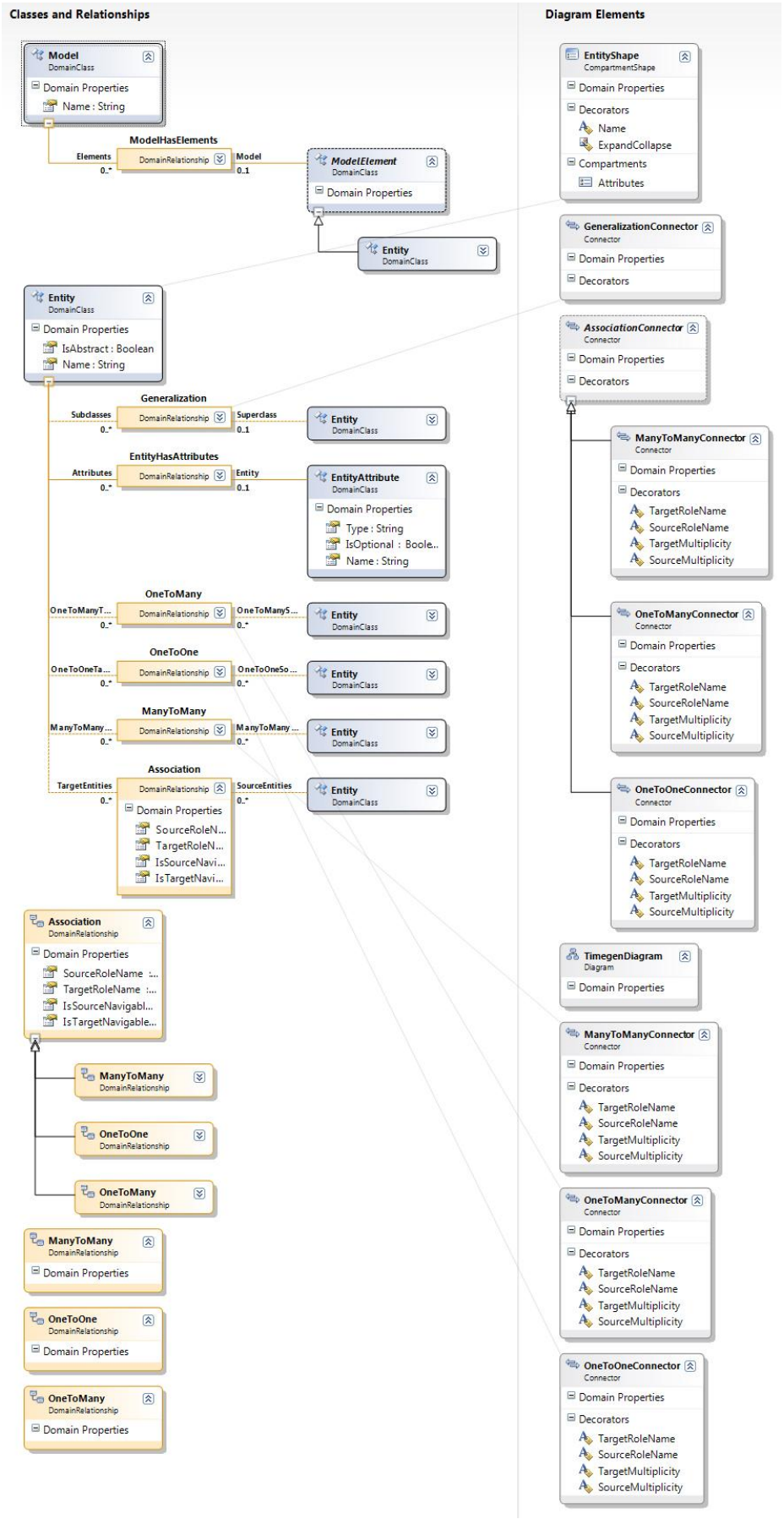
Rysunek 7 Diagram klas UML środowiska programistycznego

4.1.4. Timegen

Główna część platformy, której odpowiedzialność to udostępnienie graficznego języka dziedzinowego potrzebnego do tworzenia modelu danych. Graficzny DSL jest zintegrowany z Visual Studio 2010. Timegen udostępnia również mechanizm kreatorów konfiguracji, którego zadaniem jest możliwość uruchamiania wizardów przeprowadzających użytkownika przez formularze, w których to podejmuje on decyzje i wprowadza potrzebne dane. Oprócz tego Timegen implementuje mechanizm rozszerzalności pozwalający na obsługę providerów.

4.1.4.1. DSL

Ten komponent umożliwia graficzne modelowania z użyciem języka domenowego, który zawiera elementy określone przez model danych. Duża część tego wymagania została zrealizowana za pomocą VMSDK. Zastosowanie takiego rozwiązania przyniosło korzyści w postaci integracji z Visual Studio oraz graficznego DSL, który dla użytkowników końcowych jest sercem całej platformy Timegen. Graficzny DSL został zaprojektowany za pomocą narzędzi będących częścią VMSDK. Rysunek 8 przedstawia język dziedzinowy zaprojektowany na potrzeby platformy Timegen.



Rysunek 8 DSL zaprojektowany za pomocą VMSDK

Pierwsza część modelu DSL (Classes and Relationships) przedstawia elementy, które są częścią języka dziedzinowego. Druga część (Diagram Elements) zawiera elementy graficzne, które będą widoczne dla użytkownika. Elementy graficzne są powiązane z odpowiednimi elementami języka dziedzinowego, które będą reprezentowały. Wyszczególnione zostały następujące elementy graficzne (kształty):

- kształt kontenerowy dla encji, w którego wnętrzu mogą być umieszczane pozycje związane z atrybutami,
- kształt konektora, który łączy ze sobą dwa kształty encji i został przewidziany dla odpowiednich asocjacji i dziedziczenia.

Zaprojektowany DSL ze względu wygody jego użytkowania w pewnych miejscach nie jest zgodny z przyjętym modelem danych. DSL definiuje oddzielne byty dla każdej z typów asocjacji. Oznacza to, że na pewnym etapie obiekty języka dziedzinowego będą musiały zostać przekształcone tak, żeby były zgodne z modelem danych założonym przez platformę Timegen.

Po zainstalowaniu zaprojektowanego DSL w Visual Studio pojawia się możliwość dodawania plików nowego typu. Są to pliki pozwalające na tworzenie diagramów za pomocą tego języka dziedzinowego. W pliku diagramu przechowywane są zserializowane dane elementów dodanych do diagramu oraz informacje na temat rozmieszczenia tych elementów. Dzięki temu użytkownik może zapisywać stan diagramu i efekty swojej pracy.

4.1.4.2. Kreatory Konfiguracji

Mechanizm pozwalający na prezentowanie użytkownikowi sekwencji ekranów/kroków w celu zgromadzenia danych wymaganych do konfiguracji oraz innych zadań. Platforma Timegen wykorzystuje ten mechanizm w trzech miejscach:

- rozpoczęcie pracy z modelem (utworzenie pliku danego typu, który jest powiązany z Timegen – rozszerzenie .timegen),
- synchronizacja ze źródłem danych,
- generacja artefaktów.

Timegen narzuca tylko jeden krok – wybór providera w momencie rozpoczynania pracy z modelem. Pozostałe kroki kreatorów są dostarczane przez providerów.

4.1.4.3. Rozszerzalność

Rozszerzalność platformy Timegen opiera się na obsłudze providerów. Biblioteki dll providerów umieszczone w odpowiednim miejscu na dysku są rozpoznawane i wczytywane za pomocą MEF. Jeżeli biblioteka zawiera w sobie klasę implementująca konkretny interfejs, to zostanie wzięta pod uwagę.

Wykorzystywana jest również mechanizm rozszerzalności udostępniany przez VMSDK, który też wykorzystuje MEF [7]. Pozwala on na projektowanie nowych elementów, które wzbogacą język dziedzinowy.

4.1.5. Timegen Provider

Timegen Provider jest to biblioteka dll z implementacją spełniająca określone wymagania i dostarczająca funkcjonalność wspierająca tworzenie warstwy dostępu do danych dla konkretnego typu źródeł danych oraz technologii. Samej platformy Timegen nie można użyć bez providera.

Provider musi dostarczyć definicje oraz implementacje związane z:

- typami danych, które obsługuje – tylko tych typów będzie można użyć przy modelowaniu,
- generacją artefaktów. Timegen nie narzuca konkretnego sposobu generacji kodu, przez co implementacje dostarczane przez providerów nie są ograniczane. Provider musi zaimplementować metodę, z której ma dostęp do:
    - modelu danych,
    - obiektu przechowującego konfigurację/wynik kreatora konfiguracji uruchamianego przed generacją,
    - środowiska programistycznego – dzięki temu provider może bezpośrednio modyfikować aktywną solucję, czyli np. dodawać do projektów pliki z wygenerowanym kodem źródłowym.

  Przykładowe implementacje providerów wytworzone razem z prototypem wykorzystują szablonów T4 w celu generacji kodu źródłowego oraz innych artefaktów.
- uzyskiwaniem modelu danych – implementacja, która zwróci drzewo obiektów określające obecny stan modelu danych utworzonego za pomocą DSL. W tym miejscu dokonywane jest przekształcenie modelu użytego w graficznym DSL na model zdefiniowany przez platformę Timegen.
- definicją providera – provider musi udostępnić informacje, które go opisują takie jak nazwa czy opis,
- synchronizacją ze źródłem danych.

Provider może dostarczyć definicje oraz implementacje związane z:

- krokami kreatorów (dla każdego kroku zarówno jego prezentacja w postaci widoku w technologii WPF jak i implementacja logiki go obsługująca) – kroki zostaną wyświetlone przez mechanizm kreatorów konfiguracji,
- rozszerzeniami modelu danych – rozszerzenia, które aplikują się do elementów modelu lub do samego modelu,

- rozszerzeniami DSL zgodnymi ze specyfikacją VMSDK [2] – rozszerzenia modelu danych wymagają też wprowadzenia rozszerzeń do graficznego języka dziedzinowego.

Timegen udostępnia również mechanizm służący do utrwalania danych, które nie są związane z językiem dziedzinowym. Jest to key-value store oparty o wbudowaną w system operacyjny Windows bazę danych ESENT. Dla każdego diagramu jest zakładana nowa baza. Ten mechanizm jest dostępny dla providerów z poziomu API. Daje on możliwość zapamiętywana praktycznie dowolnych danych pomiędzy kolejnymi sesjami pracy użytkownika z diagramem.

4.2. Funkcjonalność

Funkcjonalność platformy Timegen wynika z jej podstawowych założeń. Bazowa funkcjonalność może być wzbogacona o dodatkowe, małe funkcjonalności dostarczane przez providerów.

4.2.1. Timegen

Bazowa funkcjonalność platformy Timegen to:

- tworzenie modelu danych za pomocą graficznego języka dziedzinowego zintegrowanego ze środowiskiem programistycznym:
  - diagram dodaje się do projektu poprzez wybranie pliku o rozszerzeniu .timegen,
  - do dyspozycji użytkownika jest przybornik z elementami języka (kształtami), które można dodać do przestrzeni roboczej diagramu metodą drag and drop,
  - elementy można dowolnie rozmieszczać na diagramie,
  - właściwości każdego z elementów można edytować za pomocą okna właściwości,
  - plik diagramu można zapisać i powrócić do niego następnym razem – cały stan diagramu zostanie zachowany
- przeprowadzenie wstępnej konfiguracji, która jest wykonywana tylko przy pierwszym otwarciu pliku diagramu (realizowane za pomocą kreatora konfiguracji):
  - wybór providera,
  - wstępna konfiguracja providera – zależne od wybranego providera
- synchronizacja modelu danych ze strukturą źródła danych (realizowane za pomocą kreatora konfiguracji) – funkcjonalność obsługiwana przez providera,
- generacja kodu źródłowego (opcja dostępna z menu kontekstowego diagramu):
  - wybór parametrów generacji - funkcjonalność realizowana za pomocą kreatora konfiguracji zależnego od wybranego providera.

4.2.2. Timegen NHibernate Provider

Pozwala na stworzenie warstwy dostępu do danych opierającej się na SQL Server i NHibernate. Provider definiuje następujące rozszerzenia:

- wstępna konfiguracja:
  - wybór serwera bazodanowego poprzez zdefiniowanie szczegółów połączenia,
  - wybór typu danych używanego dla identyfikatorów obiektów
- rozszerzenia języka dziedzinowego:
  - dla modelu dwa dodatkowe atrybuty związane ze wstępną konfiguracją:
    - connection string,
    - typ danych identyfikatorów
  - dla encji nazwa tabeli z bazy danych,
  - dla asocjacji wiele do wielu nazwa tabeli asocjacyjnej
- generacja kodu źródłowego:
  - rozszerzenia kreatora konfiguracji:
    - wybór projektu docelowego,
    - określenie, dla których encji mają zostać wygenerowane klasy częściowe
  - sposób generacji kodu:
    - dodanie do projektu bibliotek (plików dll) związanych z NHibernate,
    - dodanie do projektu referencji do tych bibliotek,
    - utworzenie folderu Entities,
    - dodanie do tego folderu klas odpowiadającym encjom z diagramu,
    - utworzenie folderu Mappings,
    - dodanie do tego folderu klas będącymi mapowaniami wymaganymi przez NHibernate – wygenerowany kod mapowań wykorzystuje Fluent NHibernate,
    - dodanie klasy SessionFactoryProvider, która jest odpowiedzialna za tworzenie sesji NHibernate – wygenerowany kod zawiera konfigurację, która wykorzystuje wytworzone mapowania oraz connection string do bazy danych.

4.2.3. Timegen ODRA Provider

Pozwala na stworzenie warstwy dostępu do danych opierającej się na obiektowej bazie danych ODRA i konektor do niej przeznaczony dla platformy .NET (NOBC). Wygenerowany kod

wykorzystuje generyczną implementację warstwy dostępu do danych – Devenue.Odra. Jest to biblioteka zaimplementowana przez autora pracy. Rozszerza ona NOBC i oferuje:

- obsługę reference result – dane powiązanych obiektów są automatycznie pobierane z bazy, programista pisząc kod może określić jak głęboki ma być poziom wypełniania referencji,
- wsparcie w postaci obsługi LINQ – programista może pisać mocno typowane zapytania używając LINQ. Zapytania są tłumaczone na SBQL i wysyłane do serwera.
- abstrakcyjną implementację podstawowych operacji takich jak:
    - GetById,
    - GetAll,
    - Save,
    - SaveOrUpdate,
    - Delete

  Programista dostaje obsługę tych operacji bez potrzeby pisania żadnego kodu.

Provider definiuje następujące rozszerzenia:

- wstępna konfiguracja:
    - wybór serwera ODRA poprzez podanie szczegółów połączenia,
- rozszerzenia języka dziedzinowego:
    - dla modelu atrybut związany ze wstępną konfiguracją – connection string,
- generacja kodu źródłowego:
    - rozszerzenia kreatora konfiguracji:
        - zdefiniowanie nazwy modułu ODRA,
        - wybór projektu docelowego
    - sposób generacji kodu:
        - dodanie do projektu bibliotek (plików dll) NOBC i Devenue.Odra,
        - dodanie do projektu referencji do tych bibliotek,
        - utworzenie folderu Entities,
        - dodanie do tego folderu klas odpowiadającym encjom z diagramu,
        - utworzenie folderu Dao,
        - dodanie do tego folderu interfejsów oraz ich implementacji zgodnych ze wzorcem projektowym Data Access Object (DAO) – dla każdej encji tworzony jest oddzielny interfejs oraz klasa DAO, która dziedziczy z klasy AbstractOdraDao będącej częścią biblioteki Devenue.Odra. Dzięki temu wszystkie klasy DAO bez żadnego dodatkowego wkładu

implementują podstawowe operacje związane z operowaniem na danych,

- dodanie pliku z definicją modułu ODRA – moduł zawiera definicje struktury bazy danych,
- dodanie klasy BootStrapper, która zawiera kod konfigurujący fabrykę tworząca połączenia z bazą danych. Konfiguracja zawiera przekazanie odpowiedniego connection stringa.

5. Prototyp

W celu udowodnienia osiągalności założeń został zaimplementowany prototyp oferujący większość przewidywanych funkcjonalności.

5.1. Wzorce projektowe

Najistotniejszym wzorcem projektowym zastosowanym w prototypie jest Dependency Injection (DI). Podstawowe zalety użycia tego wzorca to:

- luźno powiązane ze sobą komponenty o określonych odpowiedzialnościach,
- możliwość łatwego testowania kodu za pomocą testów jednostkowych,
- możliwość zastosowania kontenera Inversion of Control (IoC), który oferuje bardzo duże wsparcie dla DI. W prototypie wykorzystany został kontener IoC Ninject.

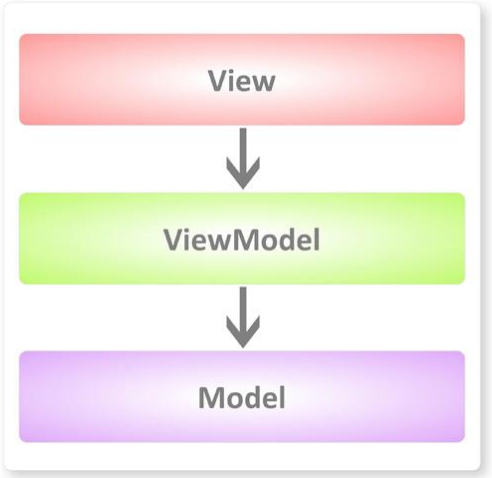
Ninject pozwala na definiowanie modułów [8], które rejestrują w kontenerze konkretne implementacje interfejsów. Jest to szczególnie przydatne w przypadku, kiedy aplikacja składa się z kilku części logicznych, z których każda powinna uczestniczyć w procesie rozruchu aplikacji i rejestracji typów w kontenerze. Użycie kontenera IoC ułatwia pozyskiwanie instancji obiektów. W przypadku, kiedy do utworzenia danego obiekty wymagane są inne obiekty będące jego zależnościami, to są one automatycznie pobierane z kontenera. Jest to mechanizm określany jako constructor injection [3] i jest on szeroko stosowany w implementacji prototypu.

Stosowanie wzorca DI wymusza myślenie o zależnościach pomiędzy obiektami oraz odpowiedzialnościach danych klas. Sprowadza się to do przestrzegania podstawowej zasady programowania obiektowego – Single responsibility principle [5]. Pojęcie mówi o tym, że każdy obiekt ma jedną, określoną odpowiedzialność. Biorąc to pod uwagę, można wyznaczyć obiekty, które do realizacji swoich celów będą wykorzystywały inne obiekty, które będą dla nich spełniały określone odpowiedzialności. Jest to implementowane za pomocą Dependency Injection oraz kompozycji, czyli zawierania się poprzez referencję jednych obiektów w drugich. Takie podejście ułatwia testy jednostkowe, ponieważ do danego obiektu można wstrzyknąć inne implementacje przygotowane specjalnie pod kątem testowania – mock objects.

Wyżej opisane praktyki zakładają, że pomiędzy obiektami tworzone są kontrakty. Jeden obiekt wie, czego może oczekiwać od drugiego obiektu i oczekuje od niego określonej funkcjonalności. Wywołuje udostępnianą przez niego metodę i otrzymuje jej wynik, nie jest dla niego istotne w jaki sposób ten wynik jest pozyskiwany. To jest już odpowiedzialność innego obiektu. Takie podejście prowadzi do dobrego i przemyślanego designu aplikacji. Wyżej wspomnianymi kontaktami w obiektowych językach programowania są interfejsy programistyczne.

Kolejnym wzorcem projektowym zastosowanym w implementacji jest Model View ViewModel (MVVM). Wartość dodana przez zastosowanie tego wzorca z punktu widzenia platformy Timegen nie

jest aż tak dużą jak DI i IoC, ale na pewno wpływa pozytywnie na architekturę aplikacji i jakość kodu. MVVM jest to wzorzec projektowy warstwy prezentacji i składa się z elementów wyszczególnionych na rysunku 9.



Rysunek 9 Elementy wzorca projektowego MVVM

Model reprezentuje bezpośrednio obiekty będące danymi albo warstwę dostępu do danych. Widok jest to część graficznego interfejsu użytkownika. ViewModel (model widoku) jest to model specjalnie dostosowany pod kątem wykorzystania go w widoku. Udostępnia właściwości, które mogą być odczytywane i modyfikowane przez widok. Zawiera również obsługę akcji przychodzących z GUI. W celu powiązania widoku z modelem widoku stosowane są takie mechanizmy jak Data Binding. Polega to na tym, że odpowiednie elementy widoku są powiązane z konkretnymi właściwościami udostępnianymi przez model widoku. Dzięki temu stan modelu widoku oraz informacje prezentowane w GUI są ze sobą zsynchronizowane. Do odbierania akcji przychodzących z GUI wykorzystywany jest mechanizm commandingu. Oznacza to, że akcje użytkownika są przekazywane do modelu widoku w postaci różnych komend i tam są już odpowiednio obsługiwane.

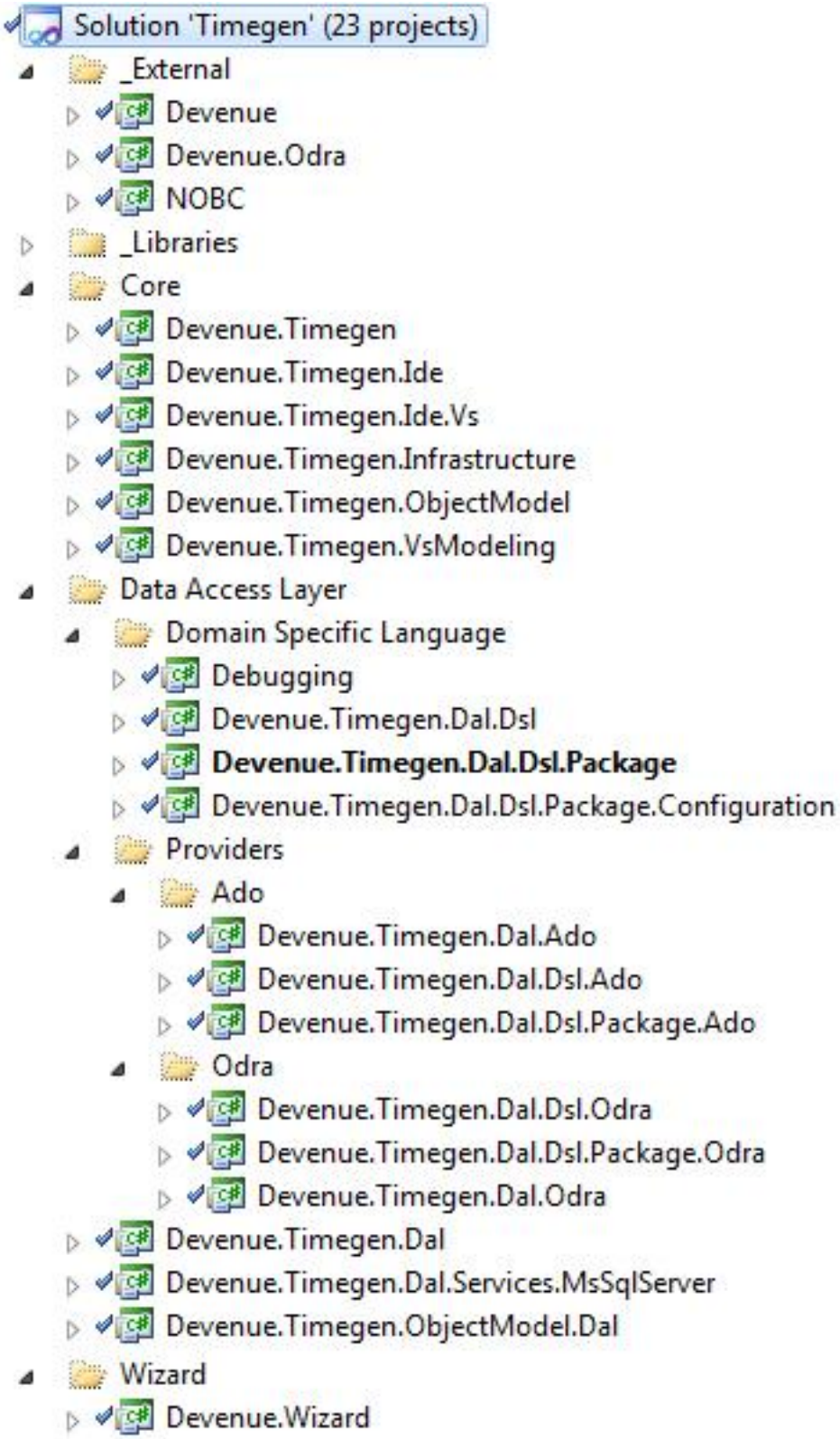
Wzorzec projektowy jest szeroko wykorzystywany w aplikacjach opartych o technologię WPF, która udostępnia potężne możliwości Data Bindingu. W prototypie został on wykorzystany przy implementacji mechanizmu kreatorów konfiguracji. Ułatwieniem było użycie frameworka wspierającego MVVM – MVVM Light Toolkit.

5.2. Implementacja

Podczas implementacji prototypu duża uwaga zwrócona została na wysoką jakość kodu oraz dobrą architekturę.

5.2.1. Podział logiczny kodu

Implementacja została podzielona na obszary logiczne i komponenty. Podział bezpośrednio odwzorowuje struktura solucji zaprezentowana na rysunku 10.



Rysunek 10 Podział solucji Timegen

- External – projekty zewnętrzne wykorzystywane przez implementację
    - Devenue – zestaw klas pomocniczych związanych np. z serializacją obiektów do formatu XML,
    - Devenue.Odra – rozszerzenie NOBC (omówione w poprzednim rozdziale),
    - NOBC – konektor do baz danych środowiska ODRA
- Core – podstawowa implementacja platformy. Implementacja tej części wcale nie przesądza o tym, że platforma ma zostać wykorzystana do pracy nad warstwą dostępu do danych. Możliwe będzie zastosowanie poniższych bibliotek przy tworzeniu podobnego rozwiązania dedykowanego dla innej dziedziny problemowej.
    - Devenue.Timegen – podstawowe interfejsy i klasy związane z modelem danych, providerami i rozszerzeniami,
    - Devenue.Timegen.Ide – abstrakcje związane ze środowiskiem programistycznym,
    - Devenue.Timegen.Ide.Vs – implementacja środowiska programistycznego opierająca się o Automation Object Model [4], pozwalająca na ingerowanie w aktywną solucję,
    - Devenue.Timegen.Infrastructure – implementacja mechanizmu key-value store wykorzystująca ESENT Managed Interop,
    - Devenue.Timegen.ObjectModel – podstawowe abstrakcje związane z modelem danych,
    - Devenue.Timegen.VsModeling – implementacja transakcyjności modelu danych realizowanego za pomocą DSL wytworzonego za pomocą VMSDK
- Data Access Layer – implementacja związania z zagadnieniem warstwy dostępu do danych wykorzystująca Core
    - Domain Specific Language – implementacja graficznego DSL
        - Debugging – projekt pomocniczy używany do debugowania rozwiązania,
        - Devenue.Timegen.Dal.Dsl – zawiera definicję bazowego graficznego języka dziedzinowego,
        - Devenue.Timegen.Dal.Dsl.Package – implementacja odpowiedzialna za integrację graficznego DSL z Visual Studio. Podczas kompilacji tego projektu powstaje plik VSIX [1], za pomocą którego można instalować platformę Timegen jako rozszerzenie Visual Studio.
        - Devenue.Timegen.Dal.Dsl.Package.Configuration – projekt konfiguracyjny zawierający bootstrappera, który w momencie rozruchu

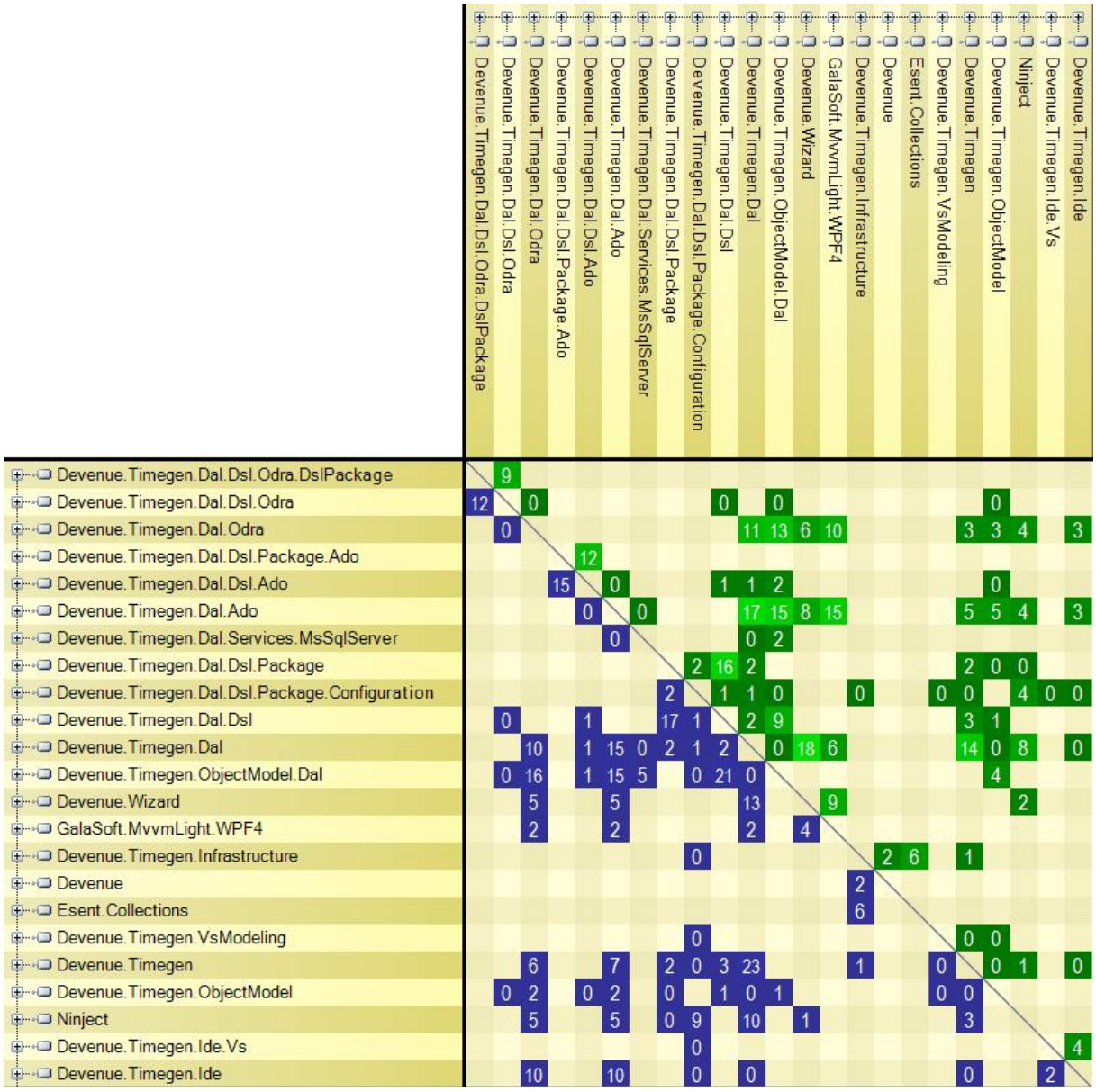
platformy rejestruje wszystkie potrzebne implementacje w kontenerze IoC

- Devenue.Timegen.Dal – abstrakcje rozszerzające Devenue.Timegen o pojęcia związane z warstwami dostępu do danych,
- Devenue.Timegen.ObjectModel.Dal – abstrakcje rozszerzające Devenue.Timegen.ObjectModel o pojęcia związanie z modelem danych dla warstwy dostępu do danych,
- Devenue.Timegen.Dal.Services.MsSqlServer – implementacja usługi dostarczającej informacje o typach danych obsługiwanych przez providera korzystającego z SQL Servera. Wydzielona do oddzielnego projektu pod kątem umożliwienia ponownego użycia.
- Providers – część implementacji odpowiedzialna za implementacje providerów
  - Ado – implementacja providera NHibernate
    - Devenue.Timegen.Dal.Ado – implementacja wszystkich usług wymaganych od providera, w tym definicje rozszerzeń, obsługa generacji kodu źródłowego oraz dodatkowe kroki dla kreatorów konfiguracji,
    - Devenue.Timegen.Dal.Dsl.Ado – definicje rozszerzeń dla graficznego języka dziedzinowego opartego o VMSDK,
    - Devenue.Timegen.Dal.Dsl.Package.Ado – projekt odpowiedzialny za integrację rozszerzenia DSL z istniejącym DSL. Podczas kompilacji tego projektu powstaje plik VSIX [1] pozwalający na zainstalowanie w Visual Studio rozszerzenia do DSL i co za tym idzie, całego providera.
  - Odra – implementacja providera ODRA (analogicznie jak w przypadku providera NHibernate)

- Wizard
  - Devenue.Wizard – uniwersalny mechanizm obsługi kreatorów konfiguracji

5.2.2. Zależności poszczególnych części implementacji

Ważne dla architektury całego rozwiązania jest zachowanie odpowiednich powiązań pomiędzy poszczególnymi częściami implementacji [6]. Rysunek 11 przedstawia macierz zależności pomiędzy assembly. Komórka jest zielona, kiedy biblioteka w kolumnie jest wykorzystywana przez bibliotekę w wierszu. Komórka jest niebieska, kiedy biblioteka w kolumnie wykorzystuje bibliotekę w wierszu. Liczba w danej komórce jest to liczba klas, metod, atrybutów biorących udział w powiązaniu.

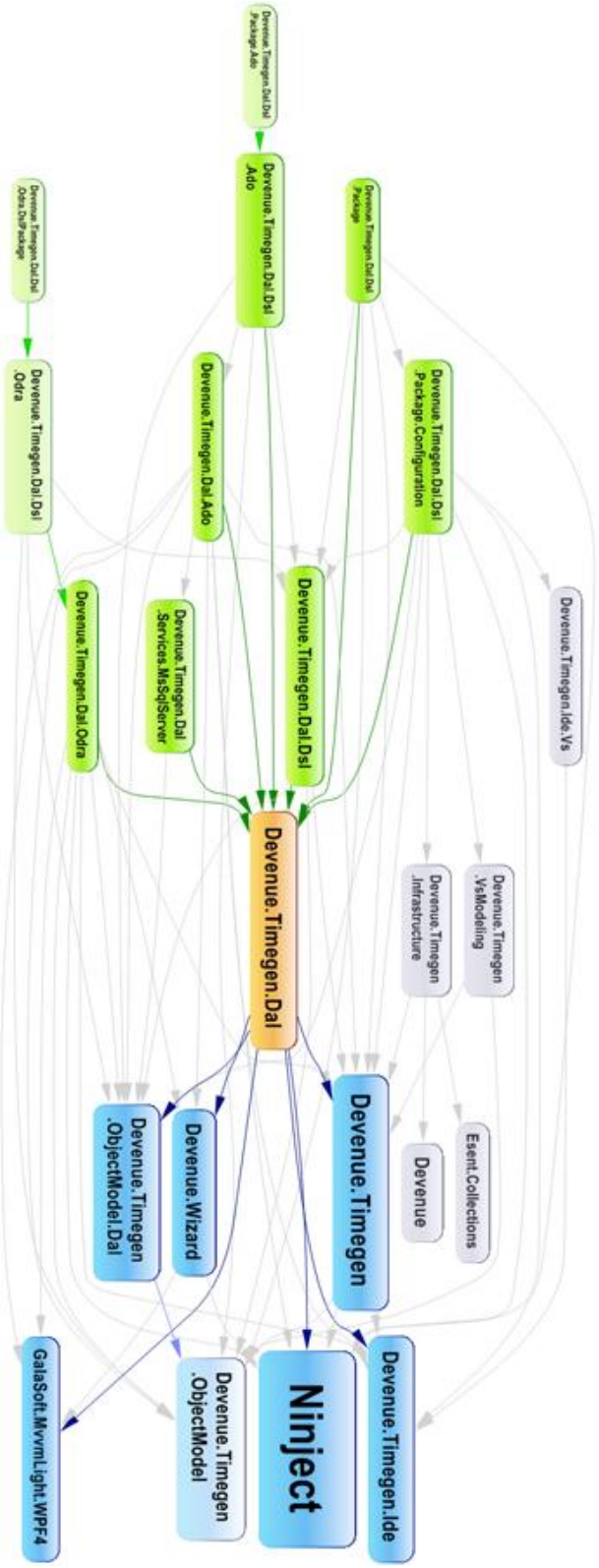


Rysunek 11 Macierz powiązań pomiędzy bibliotekami

Ważne dla architektury było odseparowanie mechanizmów platformy od implementacji związanych z konkretnymi technologiami. Logika biznesowa platformy operuje na dobrze znanych klasach i interfejsach. Nie jest ona skażona i uzależniona od danej technologii. Przede wszystkim środowisko programistyczne, na którym może operować kod providera jest dla niego widoczne jako zestaw interfejsów o wygodnym API. Implementacja oparta o Automation Object Model i komunikację z Visual Studio za pomocą COM jest ukryta dla providera i wstrzykiwana za pomocą Dependency Injection. Provider operuje na środowisku programistycznym na podstawie zawartego kontraktu, określonego przez interfejs programistyczny.

Kolejnym miejscem, w którym platforma uniezależnia się od technologii jest model danych. Provider przed rozpoczęciem generacji otrzymuje drzewo obiektów reprezentujące model danych. Są

to obiekty zdefiniowane przez platformę Timegen. Wymagane jest przekształcenie modelu, który użytkownik utworzy za pomocą graficznego DSL na model zrozumiały dla platformy Timegen. Przekształcenie jest to odpowiedzialność odpowiedniej usługi. Jej implementacja jest dostarczana przez Timegen, więc podczas pracy nad nowym providerem, jego twórca może z niej skorzystać ale może również dostarczyć własną. Odseparowanie konkretnych implementacji od mechanizmów platformy widać na rysunku 12.



Rysunek 12 Graf zależności bibliotek

5.2.3. Istotne rozwiązania i mechanizmy

Podczas implementacji zostało zastosowane wiele interesujących podejść i mechanizmów. Wybrane z nich zostały opisane poniżej.

5.2.3.1. Rozruch platformy Timegen

Aby rozpocząć pracę z platformą Timegen należy uzyskać referencję do obiektu korzenia – `TimegenDalRoot`. Wykorzystywana jest do tego statyczna klasa `Container` z projektu konfiguracyjnego oraz udostępniania przez nią metoda `Bootstrap`. Ta metoda jest odpowiedzialna za rozruch całej platformy, czyli musi w niej nastąpić rejestracja implementacji w kontenerze IoC. Ładowane są dwa moduły konfiguracyjne:

- `TimegenDalDslModule`,
- `TimegenDalModule`.

Początkowa konfiguracja i rozruch powinna zostać dokonana w punkcie wejściowym do aplikacji. W przypadku aplikacji zintegrowanej z Visual Studio za pomocą VMSDK takim punktem jest obsługa zdarzenia `DocumentLoaded`. Zdarzenie jest wywoływane w momencie otwarcia pliku z diagramem Timegen. Potrzebne jest sprawdzenie czy jest to nowoutworzony plik czy już wcześniej istniejący. Sprawdzenie sprowadza się do wykorzystania key-value strore i sprawdzenia czy istnieje w nim wartość o podanym kluczu. Jeżeli nie istnieje to znaczy, że trzeba uruchomić kreatora wstępnej konfiguracji w celu wyboru providera. Jest to wymagane przed rozpoczęciem pracy z diagramem. Zanim użytkownik będzie mógł wybrać providera, to ich definicje muszą zostać załadowane.

5.2.3.2. Ładowanie dostępnych providerów

Do tego celu wykorzystywany jest `ITimegenProvidersLoader` i udostępniana przez niego metoda `DiscoverProviders`. Ważne jest żeby obiekty wszystkich dostępnych providerów nie były tworzone jeżeli nie ma takiej potrzeby. Dlatego wprowadzono pojęcie `IDefinitionService`. Każdy provider musi zaimplementować taką usługę. Usługę można traktować jako deskryptor providera. Każdy deskryptor musi dostarczać informacje takie jak nazwa czy opis providera. Do wczytania deskryptorów używany jest MEF. Jeżeli provider został zainstalowany, to znaczy, że jego biblioteki znajdują się w tym samym drzewie katalogowym co obecnie wykonywany kod. Ta informacja jest wykorzystywana w celu utworzenia katalogu agregującego i dodanie do niego katalogów wskazujących na odpowiednie foldery. Katalog agregujący jest przekazywany do kontenera kompozycji, którego można użyć w celu wczytania wymaganych implementacji z zewnętrznych bibliotek dll znajdujących się w określonym miejscu na dysku. Do wczytania deskryptorów używana jest klasa pomocnicza `DescriptorsLoader` (przykład 1).

Przykład 1 – klasa `DescriptorsLoader`

```
private class DescriptorsLoader
{
    [ImportMany]
    private IEnumerable<IDefinitionService> ProviderDescriptors { get; set; }

    public IEnumerable<IDefinitionService> LoadDescriptors(CompositionContainer
container)
    {
        CompositionBatch batch = new CompositionBatch();
        batch.AddPart(this);

        container.Compose(batch);

        return ProviderDescriptors;
    }
}
```

Kiedy dostępni providerzy zostaną zaprezentowani użytkownikowi i dokona on wyboru jednego z nich, następuje wczytanie fabryk providerów i wybranie odpowiedniej z nich na podstawie typu wybranego providera. Pozwala to na uniknięcie tworzenia wszystkich obiektów providerów, skoro potrzebny jest tylko jeden. Kontrakt dla fabryki przedstawia przykład 2.

Przykład 2 – interfejs `ITimegenProviderFactory`

```
public interface ITimegenProviderFactory
{
    Type ProviderType { get; }
    ITimegenProvider InstantiateProvider(IKernel kernel);
}
```

Metoda `InstantiateProvider` jest odpowiedzialna za stworzenie obiektu providera. Dostawca providera dostaje do dyspozycji jądro kontenera IoC. Oznacza to, że może zarejestrować w nim swoje implementacje i użyć go do pobrania obiektu providera, który zwróci. Jest to o tyle dobre rozwiązanie, że pozwala ono na wykorzystanie przez providera z dowolnej implementacji zarejestrowanej w kontenerze. Wystarczy tylko, że provider zdefiniuje zależności w swoim konstruktorze, a zostaną one automatycznie wypełnione przez kontener IoC.

5.2.3.3. Wizard

Devenue.Wizard to uniwersalny mechanizm zaprojektowany w celu zbierania informacji od użytkownika w postaci kolejno następujących po sobie kroków. Definicję kontraktu samego obiektu wizarda przedstawia przykład 3.

Przykład 3 – interfejs `IWizard`

```
public interface IWizard<TResult, TParameters, TContext>
    where TResult : class
    where TParameters : class
    where TContext : class, IWizardContext
{
    ICommand MoveForwardCommand { get; }
    ICommand MoveBackwardCommand { get; }
    ICommand CancelCommand { get; }
    bool CanMoveForward { get; }
```

```
    bool CanMoveBackward { get; }
    IWizardStep<TContext> CurrentStep { get; }
    ICommand FinishCommand { get; }

    WizardResult<TResult> Run(WizardParameters<TParameters> parameters);
}
```

Nie wymusza on konkretnej technologii warstwy prezentacji, ale jest on najoptymalniejszy dla rozwiązań bazujących na Data Bindingu. Doskonale sprawdza się w połączeniu z WPF. Podstawowym założeniem jest generyczność. Wizard określa jakie przyjmuje parametry, jakie zwraca wyniki i na jakim kontekście operuje. Kontekst jest to obiekt danego typu, który pozwala przechowywać konkretne dane pomiędzy krokami. Przykładową definicję wizarda generacji kodu przedstawia przykład 4.

Przykład 4 – interfejs `IGenerationWizard`

```
public interface IGenerationWizard : IWizard<GenerationWizardResult,
GenerationWizardParameters, GenerationWizardContext> { }
```

Z kontekstem wiążą się dwie zasady. Musi on zostać utworzony na podstawie parametrów i po ostatnim kroku musi on zostać przetłumaczony na wyniki. Jest to jednoznaczne z kontraktem przedstawionym w przykładzie 5.

Przykład 5 – interfejs `IWizardContextProvider`

```
public interface IWizardContextProvider<TResult, TParameters, TContext>
    where TResult : class
    where TParameters : class
    where TContext : class, IWizardContext
{
    TContext InitializeContext(WizardParameters<TParameters> parameters);
    WizardResult<TResult> FinalizeContext(TContext context, bool cancelled);
}
```

Określona zostały też wymagania dotyczące poszczególnych kroków wizarda (przykład 6).

Przykład 6 – interfejs `IWizardStep`

```
public interface IWizardStep<TWizardContext>
    where TWizardContext : class, IWizardContext
{
    event Action<IList<IWizardStep<TWizardContext>>> StepsRegistrationRequested;

    WizardStepDescription Description { get; }

    void Load(TWizardContext context);
    WizardStepTransitionInfo Unload(TWizardContext context, WizardStepUnloadReason
unloadReason);
}
```

Każdy krok musi obsługiwać swój początek oraz koniec wyświetlania. Przy tej drugiej operacji może wstrzymać przejście do następnego kroku np. z powodu wprowadzenia nieprawidłowych danych przez użytkownika. Krok może też zgłosić do mechanizmu wizarda potrzebę dynamicznego dodania kroków. Jest to używane w momencie wstrzyknięcia kroków rozszerzających wizarda, zdefiniowanych przez providera.

Nierozłączną częścią każdego wizarda są widoki warstwy prezentacji, które będą wyświetlane użytkownikowi. Widoki muszą być zarejestrowane w kontenerze IoC, z którego są pobierane na podstawie konkretnego kroku.

Jeżeli nie wiadomo konkretnie jakiego typu będą wyniki zwracane przez wizarda, to wymagany jest też generyczny mechanizm ich obsługi (przykład 7).

Przykład 7 – interfejs `ICustomWizardResultHandler`

```
public interface ICustomWizardResultHandler<in TCustomResult>
    where TCustomResult : class
{
    void Handle(TCustomResult customResult);
}
```

Ten mechanizm obsługi jest używany np. w celu zapisania w modelu connection stringa skonfigurowanego podczas wstępnej konfiguracji. Wszystkie modyfikacje modelu danych powinny być dokonywane w sposób transakcyjny.

5.2.3.4. Transakcyjność modelu danych

Timegen dostarcza własną abstrakcję dotyczącą transakcyjności operacji na modelu danych. W prototypie istnieje jedna jej implementacja opierająca się o `Microsoft.VisualStudio.Modeling.Transaction`. Użycie z poziomu platformy na wyżej przedstawionym przykładzie zostało przedstawione w przykładzie 8.

Przykład 8 – użycie mechanizmu transakcyjności

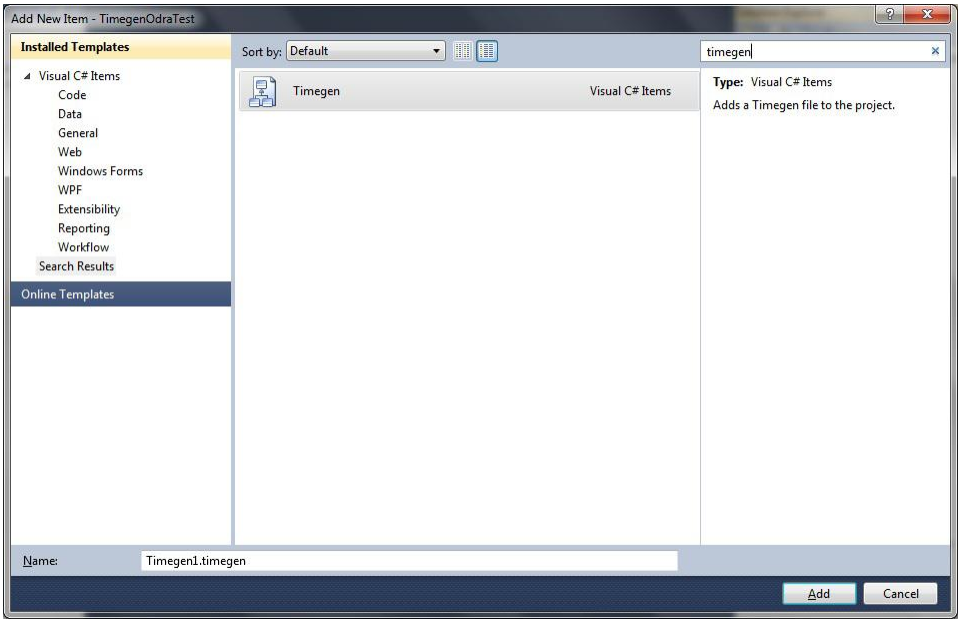
```
using (IModelTransaction<IDalModel> transaction =
DalProviderConfiguration.ModelService.BeginTransaction())
{
    IDalModelExtension extension = transaction.Model.Extensions.First(x => x is
IDalModelExtension) as IDalModelExtension;

    extension.ConnectionString = connectionStringBuilder.ConnectionString;

    transaction.Commit();
}
```

5.3. Instalacja

Instalacja platformy Timegen w Visual Studio 2010 jest bardzo prosta. Wystarczy, że zostanie uruchomiony plik VSIX [1] zawierający rozszerzenie. Każdy provider jest oddzielnym plikiem VSIX, który też należy zainstalować.

5.4. Przykłady użycia

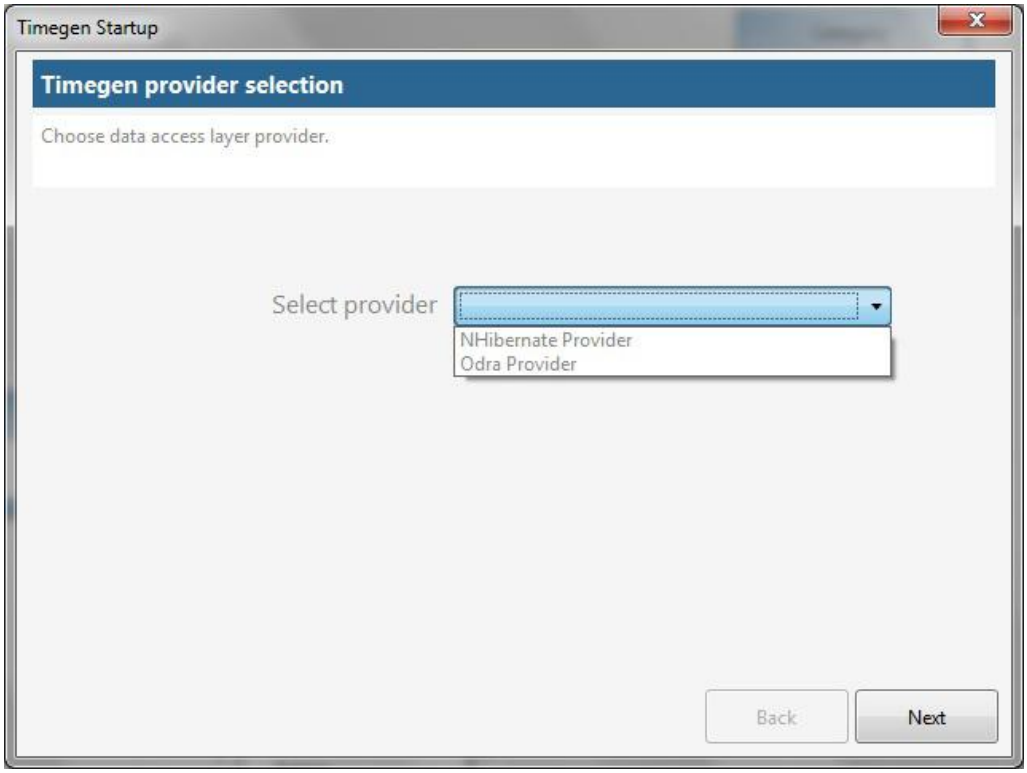
W celu rozpoczęcia pracy z platformą Timegen wymagane jest dodanie nowego pliku określonego rodzaju do projektu (rysunek 13).



Rysunek 13 Wybór pliku nowego rodzaju

Po dodaniu nowego pliku otworzy się okno edytora graficznego. Zostanie uruchomiony kreator wstępnej konfiguracji, który przeprowadzi użytkownika przez wymagane kroki. Pierwszym z nich jest wybór providera (rysunek 14). Wybór providera wpływa na następujące elementy:

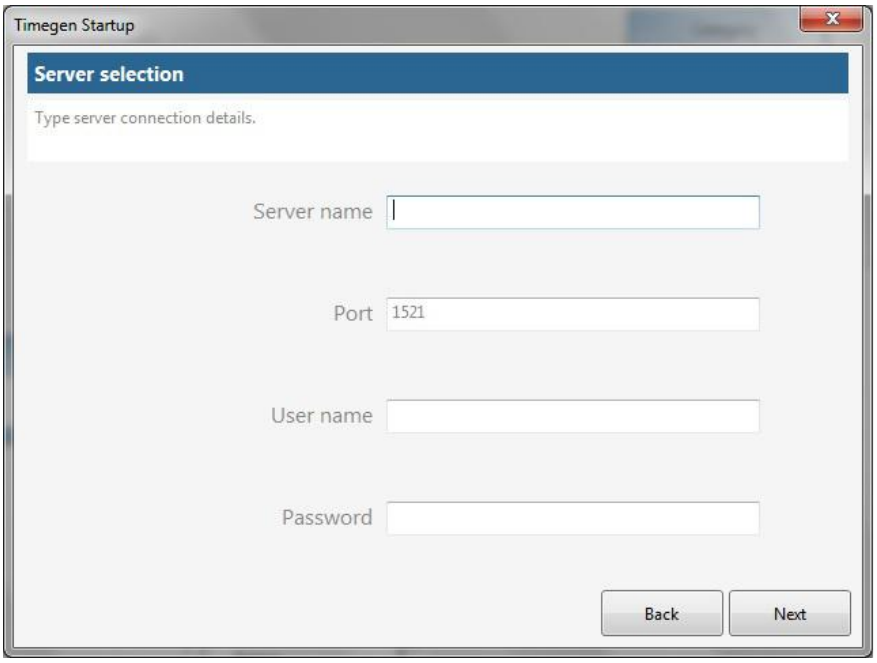
- kroki wyświetlane użytkownikowi przez wizardy,
- atrybuty elementów diagramu,
- pliki wytworzone podczas procesu generacji.



Rysunek 14 Wybór providera

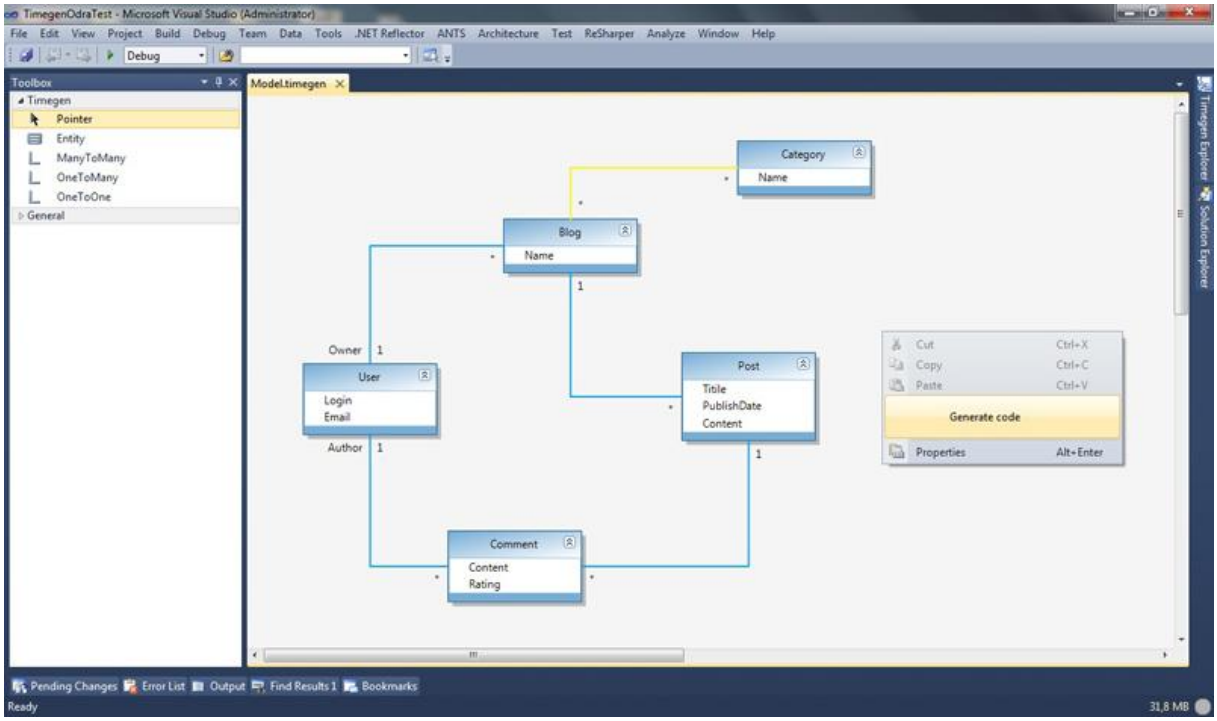
5.4.1. Odra Provider

W przypadku wyboru Odra Provider użytkownik będzie musiał wprowadzić dane konfiguracyjne takie jak szczegóły połączenia z bazą danych (rysunek 15) czy nazwę modułu.



Rysunek 15 Ekran wprowadzania szczegółów połączenia z bazą danych

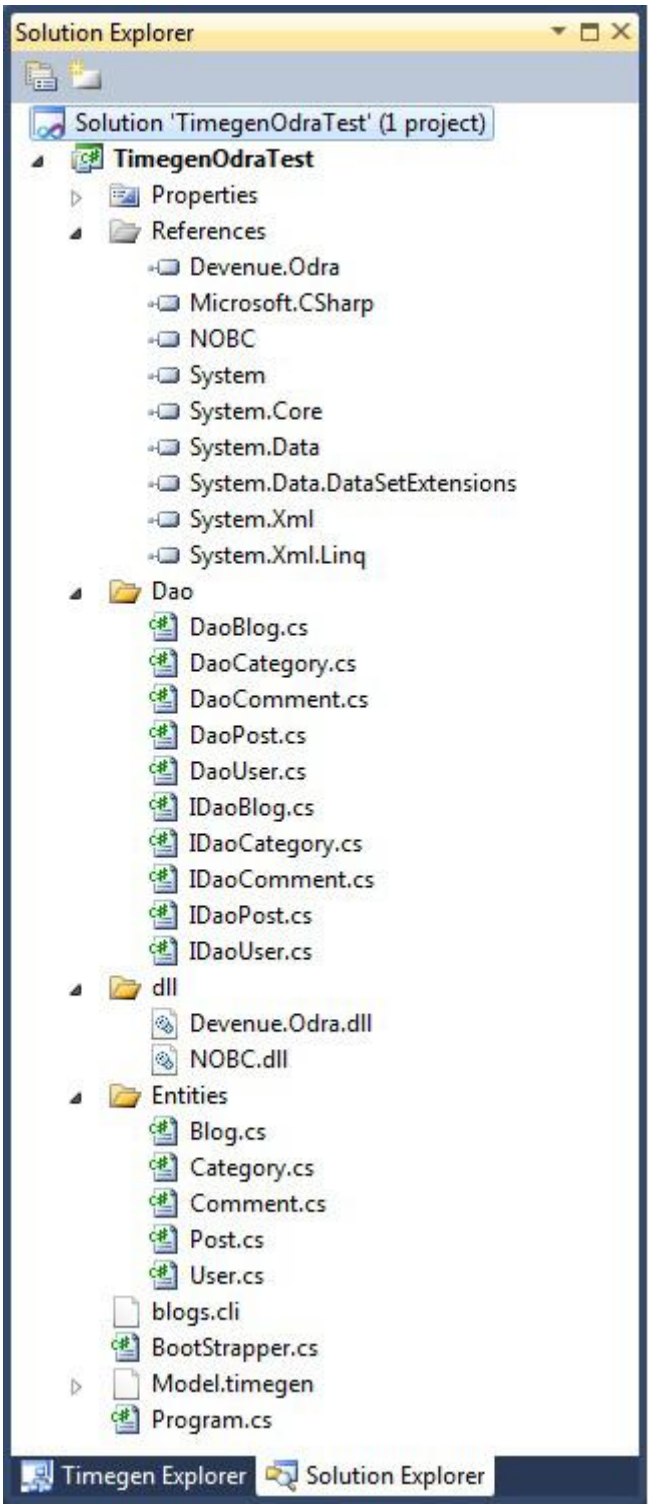
Po zakończeniu konfiguracji użytkownik zaczyna pracę z graficznym językiem dziedzinowym. Do jego dyspozycji są elementy dostępne w zasobniku. Elementy może przeciągać do edytora metodą drag and drop. Rysunek 16 przedstawia okno edytora graficznego platformy Timegen z załadowanym providerem ODRA oraz utworzonym modelem danych.



Rysunek 16 Okno edytora graficznego platformy Timegen

Po utworzeniu modelu użytkownik może rozpocząć proces generowania kodu źródłowego na jego podstawie. Używa do tego opcji „Generate code” dostępnej w menu kontekstowym.

Użytkownikowi prezentowane są kroki, których zadaniem jest zebranie informacji wymaganych do przeprowadzenia generacji. Użytkownik wybiera projekt docelowy i potwierdza operację. Efektem końcowym jest wygenerowanie i dodanie odpowiednich plików do projektu określonego przez użytkownika. Widok solucji po przeprowadzeniu generacji kodu przedstawia rysunek 17.

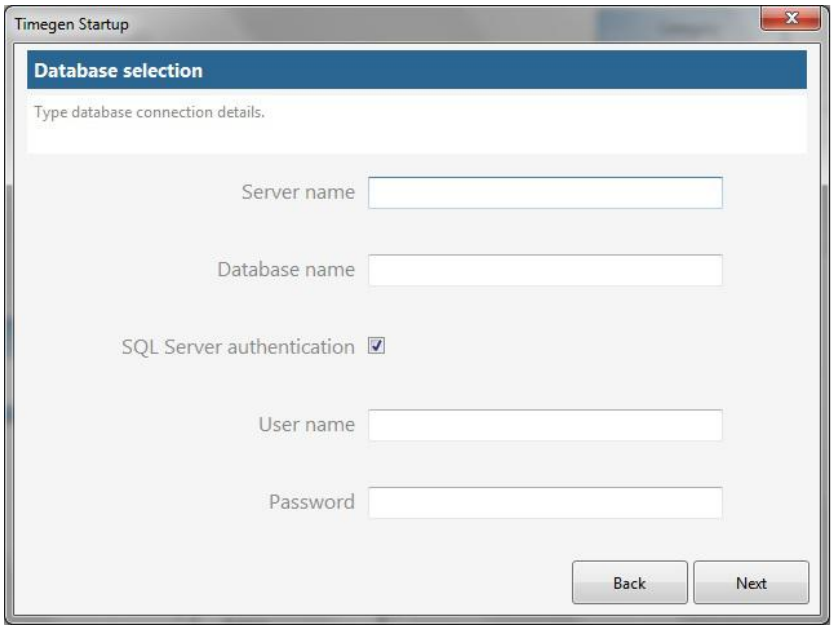


Rysunek 17 Widok solucji po generacji kodu przez providera ODRA

5.4.2. NHibernate Provider

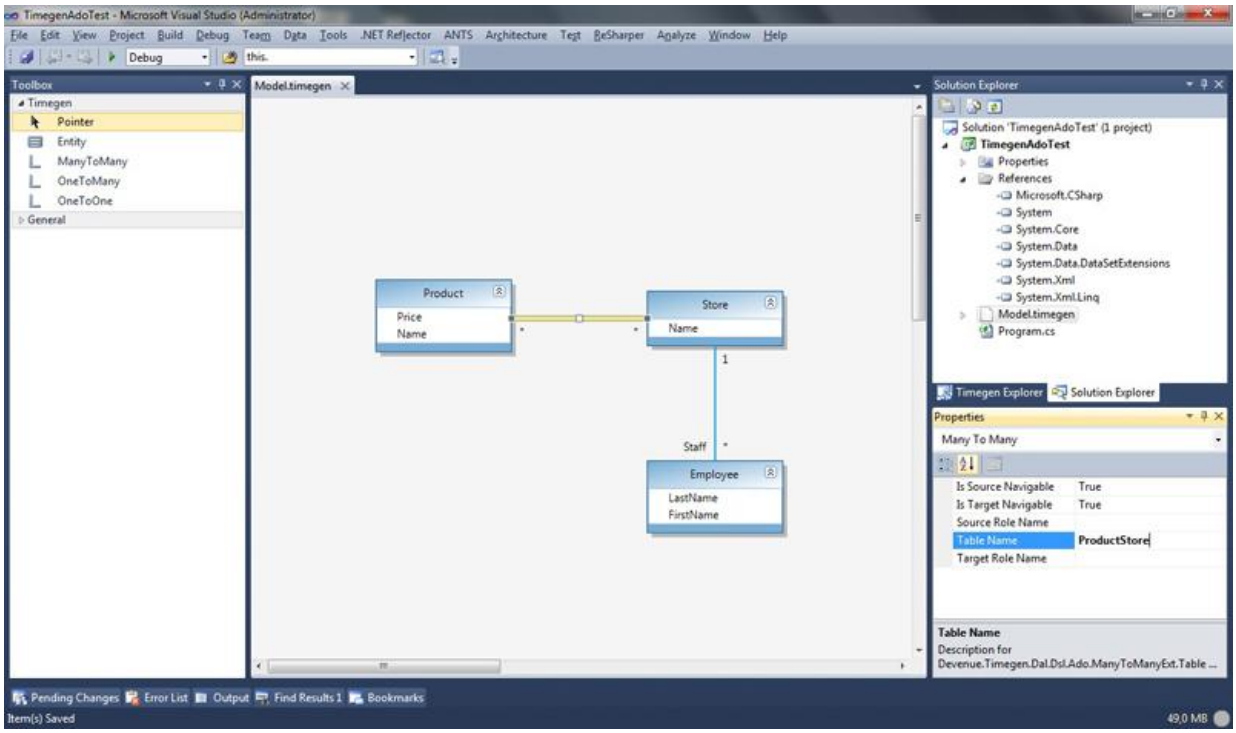
W przypadku wyboru NHibernate Provider użytkownik wyraża chęć wytworzenia warstwy dostępu do danych wykorzystującej SQL Server i opierającej się na NHibernate. Użytkownik zostanie

poproszony o wprowadzenie danych konfiguracyjnych m.in. szczegółów połączenia z bazą danych (rysunek 18). Formularz różni się od tego wyświetlanego w przypadku innego providera.



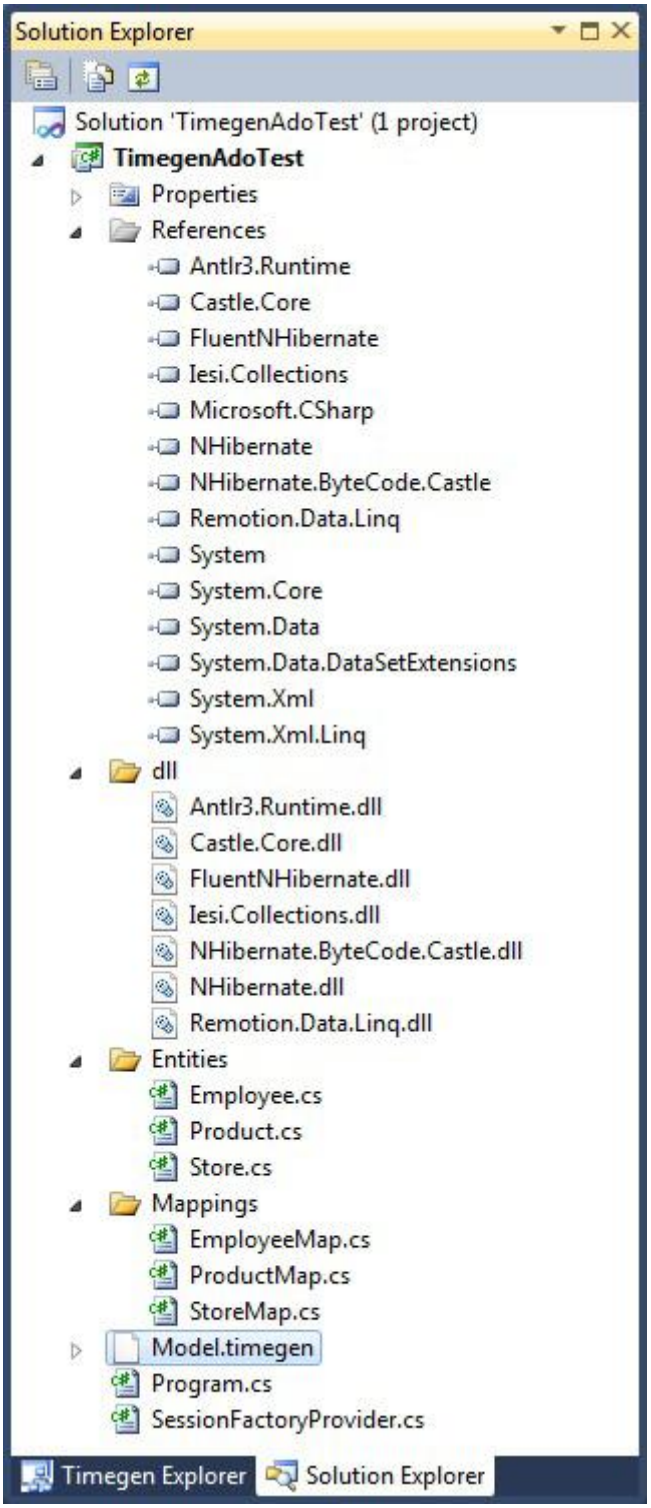
Rysunek 18 Ekran wprowadzania szczegółów połączenia z bazą SQL Server

Po zakończeniu konfiguracji użytkownik zaczyna pracę z graficznym językiem dziedzinowym wzbogaconym o atrybuty zdefiniowane przez providera. Rysunek 19 przedstawia okno edytora graficznego platformy Timegen z załadowanym providerem NHibernate. Widać, że został już utworzony prosty model danych. Dodatkowo na rysunku można zaobserwować obecny stan solucji oraz rozszerzenie relacji wiele do wiele (Table Name) specyficzne dla wybranej technologii.



Rysunek 19 Okno edytora graficznego Timegen

Uruchomienie generacji kodu i przejście przez wizarda, dla przedstawionego modelu spowoduje dodanie nowych plików do projektu (rysunek 20).



Rysunek 20 Wygląd solucji po generacji kodu przez providera NHibernate

6. Zalety, wady oraz plany rozwojowe

Zaimplementowany prototyp spełnia najważniejsze założenia postawione przed frameworkiem aplikacji bazodanowych. Nie zostały jednak zaimplementowane wszystkie funkcjonalności oraz pewne elementy na pewno można usprawnić. Bardzo przydatne okazałyby się komentarze testerów zebrane podczas beta testów albo informacje na temat dodatkowych funkcjonalności oczekiwanych od rozwiązania.

6.1. Zalety oraz wady przyjętych rozwiązań

Dobrą decyzją było zastosowanie technologii VMSDK, dzięki której w miarę bezproblemowy sposób udało się zintegrować całe rozwiązanie ze środowiskiem programistycznym Visual Studio. Dodatkowym atutem jest powiązana z tym możliwość łatwej instalacji platformy Timegen jako rozszerzenia do Visual Studio. Dzięki przemyślanej architekturze rozwiązania oraz wykorzystaniu MEF można z całą pewnością powiedzieć, że uzyskana platforma jest rozszerzalna.

Również pozytywnym aspektem rozwiązania jest wprowadzenie koncepcji providerów. W efekcie platformę można wykorzystać do pracy z wieloma technologiami. Jest to również znaczącym czynnikiem wpływającym na zasadność całego rozwiązania, ponieważ narzędzia generujące kod muszą być elastyczne żeby ich wytworzenie w ogóle było opłacalne. Rozwiązania generujące kod zawsze w jeden sposób po prostu nie są dobrymi rozwiązaniami ze względu na to, że technologie wytwarzania aplikacji biznesowych szybko się zmieniają i rozwijają.

Istotne jest żeby wytworzenie nowego providera w całości odpowiadającego oczekiwaniom programisty, nie było czymś skomplikowanym. Częściowo udało się to osiągnąć poprzez podjęcie decyzji, że całego providera można utworzyć za pomocą kodu C#, poprzez implementacje odpowiednich interfejsów i wykorzystanie udostępnionego API. Programista nie musi poznawać nieprzyjaznych formatów plików konfiguracyjnych ani martwić się o instalację providera. Udostępnione API z pewnością można jeszcze dostosować, tak żeby było bardziej intuicyjne.

Największą zaletą całego rozwiązania jest fakt, że spełnia ono podstawowe założenie, czyli ułatwienie i przyspieszenie pracy programisty nad implementacją warstwy dostępu do danych. Timegen realizuje to w sposób wygodny dla użytkownika – używa kreatorów konfiguracji składających się z prostych, przejrzystych i opisanych kroków.

6.2. Plany rozwoju

Podstawowym działaniem, które zostanie podjęte w celu rozwoju rozwiązanie będzie użycie go podczas wytwarzania komercyjnych aplikacji biznesowych. Pozwoli to na identyfikację istniejących błędów oraz na doskonalenie użyteczności platformy Timegen.

Na chwilę obecną można wskazać potencjalne punkty rozwoju:

- funkcjonalność synchronizacji modelu danych ze strukturą źródła danych, która nie została zaimplementowana w prototypie,
- wsparcie dla nieobsługiwanego w prototypie elementu języka dziedzinowego – dziedziczenia,
- możliwość implementacji przez providera dodatkowych reguł walidacji poprawności utworzonego przez użytkownika modelu,
- usprawnienie API środowiska programistycznego,
- ułatwienie tworzenia providerów poprzez dostosowanie udostępnianego API,
- udostępnienie większej ilości podstawowych implementacji usług, które muszą być dostarczane przez providerów – dzięki temu nakład pracy wymagany na zaimplementowanie nowego providera będzie mniejszy,
- utworzenie providera albo rozwinięcie samej platformy tak żeby pozwalała na tworzenie modeli zgodnych z Domain Driven Design – pozwoli to na modelowanie aplikacji biznesowej nie myśląc o problemie przechowywania danych, ale na poziomie wyżej, czyli koncentrując się na sercu aplikacji biznesowej – obiektowym modelu dziedziny problemowej.

Istotnym krokiem podjętym w celu rozwoju rozwiązania będzie przedstawienie go innym osobom zajmującym się wytwarzaniem aplikacji biznesowych na co dzień. Zebrane komentarze posłużą jako cenna pomoc w celu ukierunkowania dalszego rozwoju platformy.

7. Podsumowanie

Celem pracy było zdefiniowanie wymagań dla frameworka aplikacji bazodanowych przyśpieszającego pracę programistów oraz próba implementacji takiego rozwiązania. Przeprowadzone zostało badanie stanu sztuki w celu poznania rozwiązań dostępnych na rynku oraz określeniu ich mocnych i słabych stron. Na tej podstawie oraz również na podstawie doświadczenia autora pracy zostały zdefiniowane wymaganie stawiane przed oczekiwanym rozwiązaniem.

W celu udowodnienia osiągalności postawionych wymagań został zaprojektowany i zaimplementowany prototyp o nazwie kodowej Timegen. Prototyp realizuje wszystkie najważniejsze założenia oraz implementuje prawie wszystkie funkcjonalności. Pozwala na graficzne tworzenie modelu danych oraz na jego podstawie generację kodu źródłowego warstwy dostępu do danych. Prototyp został zintegrowany ze środowiskiem programistycznym Visual Studio 2010. Duża uwaga została poświęcona elastyczności i rozszerzalności całego rozwiązania.

Platforma Timegen jest dowodem na to, że cel pracy został spełniony.

8. Bibliografia

[1] Mike Snell, Lars Powers. Sams, 2010. Microsoft Visual Studio 2010 Unleashed. ISBN-13: 978-0-672-33081-0.

[2] Steve Cook, Gareth Jones, Stuart Kent, Alan Cameron Wills. Addison-Wesley Professional, 2007. Domain-Specific Development with Visual Studio DSL Tools. ISBN-13: 978-0321398208.

[3] Dhanji R. Prasanna. Manning Publications, 2010. Dependency Injection. ISBN-13: 978-1933988559.

[4] Keyvan Nayyeri. Wrox, 2008. Professional Visual Studio Extensibility. ISBN-13: 978-0470230848.

[5] Robert C. Martin. Prentice Hall, 2008. Clean Code: A Handbook of Agile Software Craftsmanship. ISBN-13: 978-0132350884.

[6] Krzysztof Cwalina, Brad Abrams. Addison-Wesley Professional, 2008. Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (2nd Edition). ISBN-13: 978-0321545619.

[7] Visual Studio Visualization and Modeling SDK Official Site.
http://code.msdn.microsoft.com/vsvmsdk

[8] Ninject Wiki.
https://github.com/ninject/ninject/wiki