POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2013/2014

PRACA DYPLOMOWA MAGISTERSKA

Michał Aniserowicz

[TYTUŁ] (Generator aplikacji opartych o architekturę CQRS?)

Praca wykonana pod kierunkiem
dra inż. Jakuba Koperwasa

Ocena: .....

.....

Podpis Przewodniczącego Komisji
Egzaminu Dyplomowego

Kierunek: Informatyka
Specjalność: Inżynieria Systemów Informatycznych
Data urodzenia: 1990.02.14
Data rozpoczęcia studiów: 2009.10.01

Życiorys

Urodziłem się 14.02.1990 w Białymstoku. Wykształcenie podstawowe odebrałem w latach 1997-2006 w Publicznej Szkole Podstawowej nr 9 w Białymstoku i Publicznym Gimnazjum nr 2 im. 42 Pułku Piechoty w Białymstoku. W latach 2006-2009 uczęszczałem do III Liceum Ogólnokształcącego im. K. K. Baczyńskiego w Białymstoku. Od roku 2009 jestem studentem studiów dziennych pierwszego stopnia na kierunku Informatyka na wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W marcu 2012 roku podjąłem pracę jako programista w firmie Fun and Mobile, gdzie po kilku miesiącach zostałem zastępcą przywódcy zespołu, do którego należę również obecnie. Moją pasją jest programowanie aplikacji w technologii .NET Framework.

.....
Podpis studenta

Egzamin dyplomowy:
Złożył egzamin dyplomowy w dniu: .....
z wynikiem: .....
Ogólny wynik studiów: .....
Dodatkowe uwagi i wnioski Komisji: .....
.....

Streszczenie

TODO

TODO (eng. title)

Summary

TODO

Spis treści

Rozdział 1

Wstęp

Rozdział 2

Duplikacja

Zasada "DRY" (ang. *"Don't Repeat Yourself"*), sformułowana przez Andrew Hunt'a i David'a Thomas'a, mówi: "Każda porcja wiedzy powinna mieć pojedynczą, jednoznaczną i autorytatywną reprezentację w systemie" [1]. Poprawne jej stosowanie skutkuje osiągnięciem stanu, w którym pojedyncza zmiana zachowania systemu wymaga modyfikacji tylko jednego fragmentu reprezentacji wiedzy. Należy podkreślić, że autorzy tej zasady przez "reprezentację wiedzy" rozumieją nie tylko kod źródłowy tworzony przez programistów systemu. Zaliczają do niej również dokumentację systemu, schemat bazy danych przez niego używanej oraz inne artefakty powstające w procesie wytwarzania oprogramowania (np. scenariusze testów akceptacyjnych), a nawet czynności wykonywane rutynowo przez programistów.

Celem stosowania zasady "DRY" jest unikanie duplikacji wiedzy zawartej w systemie. Duplikacją określa się fakt występowania w systemie dwóch reprezentacji tej samej porcji wiedzy. Należy wyjaśnić, że wielokrotne występowanie nazw klas lub zmiennych (bądź innych identyfikatorów) w kodzie programu nie jest duplikacją - identyfikatory nie są reprezentacją więdzy, a jedynie odnośnikami do wiedzy reprezentowanej przez artefakty, na które wskazują.

2.1 Rodzaje duplikacji

Wyróżnia się cztery rodzaje duplikacji ze względu na przyczynę jej powstania [1]:

- duplikacja wymuszona (ang. *imposed duplication*) - pojawia się, gdy programista świadomie duplikuje kod aplikacji (lub inną reprezentację wiedzy) uznając, że w danej sytuacji nie jest możliwe uniknięcie duplikacji;
- duplikacja niezamierzona (ang. *inadvertent duplication*) - występuje wtedy, gdy programista nie jest świadomy, że jego działania prowadzą do powstania duplikacji;
- duplikacja niecierpliwa (ang. *impatient duplication*) - jest wynikiem niedbalstwa programisty; ma miejsce w sytuacji, gdy programista, napotkawszy problem, świadomie wybierze rozwiązanie najprostsze, ale prowadzące do powstania duplikacji;

- duplikacja pomiędzy programistami (ang. *interdeveloper duplication*) - pojawia się, gdy kilku programistów tworzących tę samą aplikację wzajemnie duplikuje tworzony kod lub tworzy niezależne reprezentacje tej samej wiedzy.

Duplikacja może występować w różnych postaciach:

2.1.1 Duplikacja czynności wykonywanych przez programistów

Do rutynowych czynności wykonywanych podczas pracy programisty należą wszelkiego rodzaju aktualizacje: schematu bazy danych rezydującej na maszynie programisty czy wersji tworzonego oprogramowania zainstalowanej w środowisku testowym lub produkcyjnym.

Czynności te zwykle składają się z kilku kroków. Przykładowo, na aktualizację bazy danych może składać się:

1. Usunięcie istniejącej bazy.
2. Stworzenie nowej bazy.
3. Zainicjalizowanie schematu nowej bazy przy pomocy zaktualizowanego skryptu DLL (*Data Definition Language*).
4. Wypełnienie nowej bazy danymi przy pomocy zaktualizowanego skryptu DML (*Data Manipulation Language*).

Jeśli wszystkie z tych kroków wykonywane są ręcznie, to mamy do czynienia z duplikacją - każdy programista w zespole, zamiast tylko odwoływać się do wiedzy o tym, jak zaktualizować bazę danych, musi posiadać tę więdzę i wielokrotnie wcielać ją w życie krok po kroku.

2.1.2 Duplikacja kodu źródłowego aplikacji

Programistom niejednokrotnie zdarza się spowodować duplikację w kodzie źródłowym tworzonych programów. Powielone fragmenty kodu są zazwyczaj niewielkie, a podzielić je można na następujące kategorie [3]:

- prosta duplikacja wyrażeń (ang. *basic literal duplication*) - najprostszy rodzaj duplikacji; obejmuje fragmenty kodu, których treść jest identyczna (przykład: dwie identyczne metody umiejscowione w różnych modułach systemu);
- parametryczna duplikacja wyrażeń (ang. *parametric literal duplication*) - obejmuje fragmenty kodu, których treść różni się jedynie typami danych, na których te fragmenty operują;
- duplikacja strukturalna (ang. *structural duplication*) - obejmuje fragmenty kodu, które mają ten sam schemat działania, ale różnią się pojedynczymi instrukcjami (przykład: dwie pętle iterurające po tej samej kolekcji z tym samym warunkiem stopu, ale wykonujące różne operacje na elementach kolekcji);

- duplikacja czasowa (ang. *temporal duplication*) - określa fragment kodu, który jest niepotrzebnie wykonynway wiele razy (przykład: zliczenie elementów kolekcji podczas każdego sprawdzenia warunku stopu pętli iterurającej po tej kolekcji);
- duplikacja intencji (ang. *duplication of intent*) - występuje, gdy dwóch programistów umieści w różnych modułach aplikacji dwa fragmenty, których wynik działania (ale niekoniecznie treść) jest identyczny (jest to jeden z przypadków duplikacji pomiędzy programistami).

2.1.3 Duplikacja opisu dziedziny aplikacji

W każdej aplikacji obiektowej korzystającej z bazy danych opis dziedziny aplikacji występuje w co najmniej dwóch miejscach. Te miejsca to:

- schemat bazy danych (DDL);
- definicje klas w kodzie źródłowym aplikacji.

Kolejnym typowym miejscem, w którym umieszcza się informacje o dziedzinie aplikacji jest jej dokumentacja.

W miarę rozrastania się systemu i powstawania kolejnych funkcjonalności i komponentów, pojawia się tendencja do powielania całości bądź części definicji dziedziny w różnych modułach. Dotyczy to głównie tych modułów, których zadaniem jest obróbka tych samych danych, ale w różny sposób. Przykładowo, aplikacja może pozwalać na dostęp do przechowywanych danych na następujące sposoby:

- wyświetlać je na stronach WWW,
- udostępniać je poprzez usługi sieciowe jako API (*Application Programming Interface*),
- umożliwiać ich eksport do arkusza kalkulacyjnego.

Komponenty realizujące te funkcjonalności mogą korzystać z pojedynczej implementacji dziedziny. Może to jednak nie być pożądane, szczególnie w przypadku, gdy każdy ze sposobów udostępnia inny zestaw danych. Przykładowo, strony WWW mogą wyświetlać podstawowe dane każdej encji będącej częścią dziedziny, podczas gdy API udostępnia pełne dane wszystkich encji, a eksport do arkusza kalkulacyjnego udostępnia pełne dane tylko niektórych encji.

W takim przypadku komponent odpowiedzialny za API operuje na pełnej dziedzinie, a komponenty stron WWW i eksportu do arkusza kalkulacyjnego - jedynie na jej fragmentach. Jeśli każdy z tych komponentów posiada niezależną implementację dziedziny aplikacji, to mamy do czynienia z duplikacją.

2.2 Skutki występowania duplikacji

Pojawienie się duplikacji w systemie ma zazwyczaj szkodliwe skutki. Rozprzestrzenia się ona tym szybciej, a wyrządzane szkody są tym dotkliwsze, im więszy jest rozmiar systemu.

Najbardziej oczywistym skutkiem występowania duplikacji jest wydłużenie się czasu poświęcanego przez programistów na nieskomplikowane lub powtarzalne zadania. Przykładowo, nieznajomość kodu współdzielonego przez wszystkie komponenty systemu (tzw. rdzenia, ang. *core*) powoduje, że programiści niepotrzebnie spędzają czas na implementowaniu podstawowych funcji, które są już dostępne w rdzeniu systemu. Innym przykładem może być praktyka ręcznego wykonywania czynności, które mogą być wykonywane automatycznie, w tle.

Po drugie, pojedyncza porcja wiedzy jest reprezentowana w wielu miejsach, a więc zmiana pojedynczego zachowania systemu wymaga modyfikacji wielu jego modułów. Wprowadzając dowolną modyfikację, programista musi ręcznie zlokalizować wszystkie miejsca wymagające zaktualizowania. Problem ten jest najbardziej odczuwalny podczas naprawy zgłoszonych błędów: będąc świadomym występowania duplikacji kodu, programista naprawiający błąd musi pamiętać, że poprawiany przez niego fragment może być powielony w wielu miejscach w systemie. Wszystkie te miejsca musi zlokalizować i poprawić [4]. Jeśli tego nie zrobi, w systemie pozostaną sprzeczne ze sobą reprezentacje danej wiedzy, a kolejni programiści nie będą świadomi, która reprezentacja jest poprawna. W dłuższej perspektywie taka sytuacja prowadzi do pojawienia się kolejnych błędów.

Po trzecie, duplikacja powoduje zbędne powiększenie rozmiarów systemu. Kod, w którym występują zduplikowane fragmenty jest dłuższy, a przez to mniej czytelny. Co więcej, zwiększona ilość kodu powoduje niepożądane zwiększenie rozmiarów plików wchodzących w skład systemu.

Należy również zwrócić uwagę na zagrożenie, jakie niesie ze sobą wystąpienie pojedynczej duplikacji, której istnienie zostanie zaaprobowane przez programistów wchodzących w skład zespołu tworzącego projekt. Może to spowodować osłabienie dyscypliny w zespole, a w efekcie doprowadzić do pojawienia się kolejnych duplikacji, a także innych złych praktyk programistycznych. Pojedyncze wystąpienie tego zjawiska może więc doprowadzić do rozprzestrzenia się innych niepożądanych zjawisk i pogorszenia jakości całego systemu [2].

2.3 Rozwiązanie: utrzymywanie listy zduplikowanych fragmentów

Najprostszym, a zarazem najbardziej naiwnym sposobem na zniwelowanie skutków duplikacji może być utrzymywanie listy fragmentów systemu, w których duplikacja występuje. Przykładowo, zespół może przygotować listę miejsc, które należy zmodyfikować, jeśli pojedyncza encja zostanie wzbogacona o nowe pole. Na takiej liście znalazłby się schemat bazy danych, implementacja modelu danych w kodzie źródłowym programu, dokumetacja systemu, itd.

Takie podejście ma jednak istotne wady:

- rozwiązanie to nie powstrzymuje programistów przed wprowadzaniem kolejnych duplikacji, a wręcz na to przyzwala;

- pojawienie się nowego miejsca występowania pól encji (np. alternatywnej implementacji modelu danych) wymaga zaktualizowania listy; niezaktualizowana lista może skutkować pominięciem nowych miejsc w szacunkach dotyczących czasu potrzebnego na implementację nowych funkcjonalności i w samej implementacji tych funkcjonalności;
- rozwiązanie to niweluje tylko drugi z wymienionych skutków występowania duplikacji - programista nie musi samodzielnie szukać miejsc, w których powinien wprowadzić modyfikacje.

2.4 Rozwiązanie: skrypty automatyzujące rutynowe czynności

Wiele czynności wykonywanych przez programistów da się zautomatyzować, tworząc skrypty je wykonujące. Przykłady:

- wszystkie kroki składające się na czynność aktualizacji shcematu bazy przedstawone w sekcji 2.1.1 mogą być wykonywane przez pojedynczy skrypt powłoki korzystający z kilku skryptów DDL i DML;
- operacja instalacji nowej wersji systemu na środowisku testowym lub produkcyjnym nie musi wiązać się z ręcznym przesyłaniem plików systemu na serwer FPT zdalnej maszyny - może odbywać się automatycznie, po uruchomieniu odpowiedniego skryptu lub zarejestrowaniu zmiany kodu źródłowego w systemie kontroli wersji [5].

Taka automatyzacja ma wiele zalet, w tym:

- w trakcie działania skryptu w tle, programista może skupić się na innych zadaniach,
- wiedza na temat kroków składających się na daną czynność jest reprezentowana w jednym miejscu (skrypcie),
- programiści lub inne skrypty odwołują się do tej wiedzy poprzez jej identyfikator (nazwę lub ścieżkę w systemie plików).

Należy jednak zauważyć, że podejście to rozwiązuje jedynie problem duplikacji czynności wykonywanych przez programistów. Pozostałe postacie duplikacji nie mogą być zniwelowanie w ten sposób.

2.5 Rozwiązanie: generyczna implementacja

Rozwiązaniem problemu duplikacji kodu źródłowego aplikacji może być generyczna implementacja aplikacji. W językach obiektowych jest ona możliwa do osiągnięcia na kilka sposobów:

2.5.1 Dziedziczenie

Podstawowym sposobem na uniknięcie duplikacji w kodzie aplikacji wykorzystujących języki obiektowe jest dziedziczene. Metody wspólne dla rodzin klas obiektów są umieszczane w ich klasach bazowych. Pozwala to uniknąć podstawowej duplikacji wyrażeń (patrz: sekcja 2.1.2).

2.5.2 Typy szablonowe i generyczne

Innych rodzajów duplikacji - parametrycznej duplikacji wyrażeń i duplikacji strukturalnej - można uniknąć poprzez zastosowanie typów generycznych i szablonów klas. Kod o tej samej strukturze i działaniu, ale operujący na danych innego typu, może być zamknięty w pojedynczej klasie generycznej.

2.5.3 Refleksja

W sytuacjach, w których występuje potrzeba zaimplementowania podobnej funkcjonalności dla kilku typów obiektów, które nie należą do jednej rodziny, można skorzystać z mechanizmu refleksji. Pozwala on osiągnąć rezultaty podobne do tych oferowanych przez klasy bazowe i typy generyczne.

Należy podkreślić, że przedstawione propozycje rozwiązują jedynie problem duplikacji kodu źródłowego aplikacji.

2.6 Rozwiązanie: użycie generatorów

Rozwiązaniem pozwalającym na uniknięcie duplikacji opisu dziedziny aplikacji jest zastosowanie różnego typu generatów.

Należy wyjaśnić, że kodu źródłowego (bądź innych artefaktów) wygenerowanego na podstawie innego kodu nie uznaje się za duplikację. Powodem jest to, że jeśli na podstawie danego fragmentu kodu generowanych jest wiele artefaktów, to aby wprowadzić zmiany we wszystkich tych artefaktach, należy jedynie zmodyfikować źródłowy fragment kodu i uruchomić proces generacji. Za "pojedynczą, jednoznaczną i autorytatywną" reprezentację danej porcji wiedzy (patrz: Zasada "DRY") uznaje się w tym przypadku źródłowy fragment kodu.

2.6.1 Generator modelu dziedziny aplikacji na podstawie schematu bazy danych

Najpowszechniejszym przykładem użycia generatorów kodu jest wykorzystanie generatora zestawu klas wchodzących w skład modelu dziedziny aplikacji na podstawie schematu bazy danych używanego przez tę aplikację. Takie podejście nosi nazwę "najpierw baza danych" (ang. *Database First*).

Przykładami narzędzi umożliwiających taką generację kodu są:

- EntityFramework [6] - rozbudowane narzędzie ORM (ang. *Object-Relational Mapping*) przeznaczone na platformę .Net;
- SQLMetal [7] - narzędzie dla platformy .Net, którego jedynym zadaniem jest generacja kodu źródłowego klas na podstawie schematu bazy danych;
- Django [8] - platforma aplikacji webowych dla języka Python.

2.6.2 Generator schematu bazy danych na podstawie modelu dziedziny aplikacji

Podejściem przeciwstawnym dla *"Database First"* jest podejście "najpierw kod" (ang. *Code First*). Jak sama nazwa wskazuje, generatory tego typu generują schemat bazy danych na podstawie klas należących do implementacji modelu w kodzie źródłowym aplikacji.

Przykłady generatorów *"Code First"* to:

- Hibernate [9] - narzędzie ORM dla platform Java i .Net;
- EntityFramework (patrz: sekcja 2.6.1);
- Django (patrz: sekcja 2.6.1).

2.6.3 Generator dokumentacji

Duplikacji opisu dziedziny aplikacji w jej dokumentacji można uniknąć poprzez zastosowanie narzędzi generujących tę dokumentację na podstawie kodu źródłowego aplikacji. Generatory tego rodzaju wymagają umieszczania w kodzie źródłowym specjalnie sformatowanych komentarzy, na podstawie których są w stanie wygenerować dokumentację w kilku formatach, takich jak PDF czy HTML. Przykłady takich narzędzi to:

- Doxygen [10] - popularne narzędzie obsługujące wiele języków programowania (w tym C++, Java C#), wspierające wiele formatów dokumentacji (w tym HTML, PDF, LaTeX, XML);
- JavaDoc [11] - generator dedykowany językowi Java, dymyślnie wspiera jedynie format HTML;
- C# XML Documentation [12] - format tworzenia dokumentacji wbudowany w język C#, na podstawie którego generowana jest dokumentacja w formacie XML;
- pydoc [13] - narzędzie będące częścią standardowego zestawu narzędzi deweloperskich języka Python; wspiera format tekstowy i HTML.

2.6.4 Inne generatory

Istnieje wiele innych generatorów, należących do typu programowania nazywanego programowaniem automatycznym (ang. *automatic programming*) [14]. Jednakże większość z nich, tak jak powyższe przykłady, eliminuje tylko pojedyncze rodzaje duplikacji.

Warto zwrócić uwagę na fakt, że różne typy generatorów za źródło danych obierają sobie różne definicje dziedziny: np. generator *Database First* bazuje na schemacie bazy danych, a generator dokumentacji - na kodzie źródłowym aplikacji. Używanie wielu różnych generatorów usuwających pojedyncze rozdzaje duplikacji w końcu doprowadziłoby do powstania "łańcucha" generatów, w którym wynik działania jedngo generatora byłby źródłem danych dla innego. Takie rozwiązanie mogłoby być trudne w utrzymaniu.

2.7 Rozwiązanie: pojedynczy generator wszystkich artefaktów systemu

Przedstawione rozwiązania mają pewne wspólne wady. Po pierwsze, wszystkie z nich usuwają tylko pojedyczne rodzaje duplikacji. Aby całkowicie wyeliminować duplikację z systemu, należałoby by użyć prawie wszystkich z nich.

Po drugie, rozwiązania zwalczające duplikację czynności wykonywanych przez programistów i duplikacje kodu źródłowgo aplikacji nie są rozwiązaniami prewencyjnymi - nie zapobiegają pojawianiu się nowych duplikacji. To, czy nowa rutynowa czynność zostanie zautomatyzowana przy pomocy skryptu powłoki, i czy nowy moduł aplikacji zostanie zaimplementowany w sposób generyczny, zależy jedynie od dyscypliny i dbałości zespołu tworzącego system. Aby uniknąć takiej sytuacji, wydaje się, że mechanizmy niwelujące duplikację powinny leżeć u podstawy systemu - tak, aby programiści intuicyjnie korzystali z nich, wprowadzając nowe czynności lub aktualizując dziedzinę aplikacji.

Odpowiednim rozwiązaniem wydaje się być zastosowanie pojedynczego generatora potrafiącego wygenerować wszystkie potrzebne w systemie artefakty. Powstanie takiego generatora dającego się zastosować w każdym projekcie jest jednak bardzo mało prawdopodobne, ponieważ:

- każdy projekt ma inne wymagania odnośnie wygenerowanych artefaktów;
- prawdopodobne nie istnieje format, który pozwalałby zdefiniować każdą dziedzinę w najlepszy dla niej - tj. najbardziej naturalny, a przy tym pozbawiony duplikacji - sposób.

2.8 Podsumowanie

Jak widać, powszechnie dostępne są jedynie rozwiązania pozwalające wyeliminować pojedyncze rodzaje duplikacji. Nie jest jednak dostępny pojedynczy, spójny sposób na wyeliminowanie duplikacji w obrębie całego systemu. Odpowiednim rozwiązaniem wydaje się być połączenie kilku z wyżej przedstawionych sposobów. W dalszej części pracy podjęta zostanie próba stworzenia takiego rozwiązania. Zostanie ono oparte o generację kodu,

a głównym założeniem jest skonstruowanie go w taki sposób, aby pozwolić na usunięcie jak największej liczby rodzajów duplikacji patrząc zarówno z perspektywy ich postaci, jak i przyczyn powstawania (patrz: sekcja 2.1).

Rozdział 3

Sprecyzowanie problemu

Celem praktycznej części niniejszej pracy jest opracowanie narzędzia pozwalającego w jak największym stopniu unikać duplikacji podrzas tworzenia aplikacji. Narzędzie to będzie oparte o mechanizmy generacji kodu źródłowego oraz innych artefaktów systemu.

Generacja została wybrała jako rozwiązanie problemu duplikacji dlatego, że pozwala zredukować duplikację nie tylko w kodzie źródłowym aplikacji, a w obrębie całego systemu. Co więcej, zastosowanie generacji u podstawy systemu może zapobiec pojawianiu się duplikacji w przyszłości - gdy pojawi się potrzeba stworzenia nowej funkcjonalności, nowego modułu aplikacji bądź nowego artefaktu systemu, programiści prawdopodobnie najpierw spróbują zaimplementować tę nową część systemu tak, aby była generowana na podstawie już istniejącej bazy.

Należy zaznaczyć, że od narzędzia będącego celem pracy nie jest wymagana całkowita eliminacja duplikacji w systemie. Główną postacią duplikacji, która będzie przedmiotem działania narzędzia jest duplikacja dziedziny aplikacji. Wybór padł na tę właśnie postać dlatego, że przejawia się ona w największym zakresie systemu. Co więcej, użycie mechanizmów generacji nie przekreśla eliminacji innych postaci duplikacji. Przykładowo, skrypty powłoki automatyzujące czynności wykonywane przez programistów również mogą być generowane lub mogą działać na wygenerowanych plikach - wtedy tym łatwiej będą się dostosowywać do zmian w systemie. Niektóre fragmenty kodu źródłowego logiki biznesowej lub testów jednostkowych aplikacji również mogą być generowane. Jednak uwaga poświęcona zostanie głównie definicji dziedziny aplikacji.

Rozdział 4

Generacja

- kiedy to się opłaca, a kiedy może powodować problemy
- generacja aktywna vs pasywna
- przykłady konkretnych dużych generatorów
- albo typów
  - na podstawie szablonu
  - z CodeDom
  - inne
- wniosek: żaden generator nie będzie jednocześnie bogaty w funkcjonalności i dobry dla każdego typu aplikacji
- zatem: rozwiązanie podzielę na dwie części
  - trzon narzędzia do generacji
  - generator aplikacji jednego typu

Rozdział 5

Założenia dotyczące trzonu narzędzia

Przed przystąpieniem do prac nad generatorem aplikacji konkretnego typu, należy sformułować założenia dotyczące komponentu realizującego sam proces generacji artefaktów systemu, nazwanego trzonem narzędzia.

5.1 Podstawowe założenia dotyczące trzonu narzędzia

O ile elastyczność generatora konkretnego typu aplikacji może być ograniczona, o tyle trzon powinien być na tyle uniwersalny, by mógł być użyty w celu wygenerowania wielu rodzajów plików tekstowych, w tym kodu źródłowego w dowolnym języku, skryptów DLL, skryptów powłoki, dokumentancji w formacie HTML lub XML itd. Wszystkie rodzaje plików powinny być generowane w ten sam sposób (np. na podstawie szablonów napisanych w jednym języku, takim jak xslt [15]), tak aby programista korzystający z komponentu nie musiał poznawać całej gamy języków lub narzędzi używanych do tworzenia szablonów generacji. Pożądaną funkcjonalnością jest możliwość łatwiej wymiany domyślnie używanego rodzaju szablonów na inny, tak aby użytkownik mógł w łatwy sposób użyć w nim szablonów stworzonych w języku, ktory zna najlepiej. Co więcej, trzon narzędzia nie pownien narzucać sposobu formatowania danych wejściowych (w tym przypadku - opisu dziedziny aplikacji).

Dla wygody autora narzędzia zakłada się, że zarówno jego trzon jak i generator aplikacji konkretnego typu zostanie stworzony w technolgii .NET Framework

5.2 Kroki generacji

Generacja odbywać się będzie w natępujących krokach:

5.2.1 Wczytanie definicji dziedziny aplikacji

Komponent na wejściu przyjmował będzie ścieżkę do pliku lub katalogu źródłowego (patrz: sekcja 5.3). Każdy plik katalogu źródłowego zostanie odwiedzony przez generator, a jego treść zostanie zdeserializowana do obiektu odpowiadającej mu klasy.

5.2.2 Zdeserializowanie definicji dziedziny aplkacji

Jak wymieniono w założeniach dotyczących trzonu generatora (patrz: sekcja 5.1), format opisu dziedziny aplikacji nie będzie narzucony z góry. Każdy plik źródłowy może być zapisany w innym formacie (np. XML lub JSON). Domyślnie wspierany będzie jeden format (patrz: sekcja 5.4), a użytkownik generatora będzie mógł dla każdego pliku źródłowego określić wybrany i dostarczony przez siebie sposób deserializacji.

Zdeserializowany obiekt dziedziny może posłużyć jako wskazanie na kolejne pliki reprezentujące elementy dziedziny. Przykładem obrazującym potrzebę takiego działania może być sytuacja, w której dany plik zawiera ogólne informacje na temat encji, a pliki w podkatalogu sąsiadującym z tym plikiem zawierają definicje poszczególnych pól danej encji (patrz: sekcja 5.3.3). Wtedy wczytanie i deserializacja opisu pól encji odbędzie się dopiero po zdeserializowaniu opisu samej encji i na tej podstawie określeniu, który katalog zawiera kolejne pliki opisujące jej pola.

5.2.3 Wyodrębnienie jednostek generacji

Po uzyskaniu kompletnego obiektu - lub kolekcji obiektów - opisującego dziedzinę aplikacji, tzn. po zdeserializowaniu wszystkich plików zawierających dziedzinę aplikacji, generator będzie musiał przygotować kolejne obiekty, które będą podstawą do wygenerowania plików wynikowych. Obiekty te nazwano jednostkami generacji. Jednostką generacji może być na przykład opis pojedynczej encji dziedziny aplikacji lub, bardziej szczegółowo, pojedynczej tabeli bazy danych.

Aby zachować elastyczność, krok ten będzie musiał być zaimplementowany po stronie użytkownika komponentu, tzn. generatora aplikacji konkretnego typu. Umożliwi to obsłużenie scenariusza, w którym pojedynczy element opisu dziedziny aplikacji posłuży za źródło generacji wielu plików wynikowych (np. definicja tabeli bazy danych, definicja klasy w kodzie źródłowym aplikacji i kod HTML będący częścią dokumentancji systemu mogą być wygenerowane na podstawie pojedynczej jednostki generacji będącej opisem encji). Odpowiedzialność wyodrębnienia jednostek generacji z pełnego opisu dziedziny zostanie zrzucona na użytkownika komponentu dlatego, że trzon generatora nie jest w stanie go zautomatyzować bez utraty elastyczności. Aby jednak obsłużyć najprostsze scenariusze, pominięcie implementacji tego kroku w aplikacji będącej użytkownikiem komponentu zaskutkuje potraktowaniem głównego obieku (lub kolekcji obiektów) dziedziny jako jednostki (lub osobnych jednostek) generacji.

Uzyskane jednostki generacji bezpośrednio posłużą do wygenerowania plików wynikowych. Pojedyncza jednostka będzie mogła odpowiadać jednemu lub wielu plikom wynikowym.

5.2.4 Użycie szablonów do wygenerowania plików

Ostatnim krokiem będzie użycie jednostek generacji w celu wygenerowania plików wynikowych. Domyślny mechanizm generacji pliku wynikowego oparty będzie o wykorzystanie silnika generacji plików na postawie szablonu generacji (patrz: sekcja 5.6).

Trzon generatora przekaże daną jednostkę generacji odpowiedniemu szablonowi, a wygenerowana treść zostanie umieszczona w odpowiednim pliku. Zadaniem użytkownika

komponentu będzie dostarczenie zarówno szablonu generacji, jak i ściezki, pod którą ma się znaleźć wygenerowany plik.

Aby zachować elastyczność, wykorzystywany silnik generacji będzie mógł zostać wymieniony na inny przez użytkonika generatora.

5.3 Organizacja plików źródłowych i wynikowych

Podstawową decyzją, którą należy podjąć w trakcie precyzowania założeń dotyczących pierwszego kroku generacji jest organizacja i format plików źródłowych, na których pracował będzie trzon narzędzia, a także organizacja plików, które będzie w stanie wygenerować. Jak wspomniano wyżej, od trzonu oczekuje się jak największej elastyczności - dlatego powinien on wspierać kilka scenariuszy. Za przykład niech posłuży internetowy portal informacyjny.

5.3.1 Pojedynczy plik źródłowy

W tym scenariuszu całość dziedziny aplikacji (lub innych informacji o systmie) zawarta jest pojedynczym pliku. Przykładem zastosowania może być pojedynczy skrypt SQL zawierający schemat bazy danych używanej przez aplkację. Taki plik, oprócz swojego standardowego przeznaczenia, tj. konfigurowania bazy danych, pełniłby rolę źródła generatora *Code First*. Na jego podstawie generowane byłyby pliki zawierające defnijcje klas będących częścią implementacji modelu dziedziny w aplikacji.

Rysunek 5.1 obrazuje przykładową organizację plików portalu.

Źródło generacji:
`database_schema.sql`
Wynik generacji:

```
Model
├── User.cs
├── News.cs
├── Comment.cs
└── ...
```

Rysunek 5.1: Przykład organizacji plików przy generacji kilku plików wynikowych na podstawie pojedynczego pliku źródłowego.

5.3.2 Pojedynczy katalog z wieloma plikami źródłowymi

Kontynuując przykład, w miarę upływu czasu portal może rozrosnąć się na tyle, że wprowadzi możliwość prowadzenia blogów przez jego użytkowników. Wtedy może wystąpić potrzeba podzielenia schematu bazy danych na kilka plików - np. według nazw schematów (ang. *scheme*), w których znajdują się poszczególne tabele. Wszystkie te pliki w dalszym ciągu byłyby źródłem dla generatora, a wynikowe klasy mogłyby być umieszczone w osobnych katalogach (podzielonych według nazw schematow, w których znajdują się odpowiadające im tabele).

Rysunek 5.2 obrazuje przykład.

Źródło generacji:

```
database_schema
├── dbo.sql
├── news.sql
└── blogs.sql
```

Wynik generacji:

```
Model
├── dbo
│   ├── User.cs
│   └── ...
├── news
│   ├── News.cs
│   ├── Comment.cs
│   └── ...
└── blogs
    ├── Post.cs
    ├── Comment.cs
    └── ...
```

Rysunek 5.2: Przykład organizacji plików przy generacji kilku katalogów wynikowych na podstawie wielu plików źródłowych (dbo - schemat wspólny, pozostałe - schematy właściwe obszarom, którymi zajmuje się portal).

5.3.3 Drzewo katalogów z wielona plikami źródłowymi

W dłuższej perspektywie, w opisywanym przykładzie może pojawić się potrzeba wprowadzenia podkatalogów dla poszczególnych schematów. Pojedynczy podkatalog zawierałby osobne pliki zawierające definicje tabel, widoków i procedur składowanych obecnych w bazie danych. Trzon narzędzia generującego powinien być w stanie dotrzeć do wszystkich tych plików. Przykład takiej organizacji został przedstawiony na rysunku 5.3.

Innym przykładem wykorzystującym ten scenariusz jest sytuacja, w której pojedynczy plik wynikowy generowany jest na podstawie wielu plików źródłowych. Przykładem może być zorganizowanie opisu dziedziny aplikacji w taki sposób, aby informacje na temat każdego pola każdej encji umieszczone były w osobnym pliku. Rysunek 5.4 obrazuje przykład.

Taka organizacja może mieć zastosowanie w przypadku, gdy w systemie w wielu miejscach występują jedynie fragmenty encji (np. w licznych widokach bazy danych). Wtedy każde pole może wymagać skonfigurowania dla niego miejsc, w których występuje, czego skutkiem mogą być rozległe opisy każdego z pól. Opisy te najwygodniej byłoby przechowywać w osobnych plikach.

Źródło generacji:

```
database_schema
├── dbo
│   ├── tables.sql
│   ├── views.sql
│   └── procedures.sql
├── news
│   ├── tables.sql
│   ├── views.sql
│   └── procedures.sql
└── blogs
    ├── tables.sql
    ├── views.sql
    └── procedures.sql
```

Wynik generacji:

```
Model
├── dbo
│   ├── User.cs
│   └── ...
├── news
│   ├── News.cs
│   ├── Comment.cs
│   └── ...
└── blogs
    ├── Post.cs
    ├── Comment.cs
    └── ...
```

Rysunek 5.3: Przykład organizacji plików przy generacji wielu katalogów wynikowych na podstawie wielu katalogów źródłowych (procedury składowane nie są podmiotem generacji).

5.4 Sposób zdefiniowania dziedziny aplikacji

O ile założenie o elastyczności generatora powinno dopuszczać zdefiniowanie dziedziny aplikacji w dowolny sposób, używając dowolnego formatu plików zawierających opis poszczególnych elementów tej dziedziny, o tyle generator powinien obsługiwać pewien domyślny sposób formatowania. Poniżej przestawiono kilka z możliwych wyborów:

5.4.1 UML

Oczywistym wyborem sposobu opisu dziedziny aplikacji wydaje się być język UML, stworzony między innymi do tego właśnie celu. Do opisu przykładowej dziedziny posłużyć może diagram klas przestawiony na rysuku 5.5.

Trzeba jednak zauważyć, ze taki opis nie jest wystarczająco elastyczny - posiada on zdefiniowany z góry zestaw atrybutów, . Natomiast rzeczywiste scenariusze użycia genera-

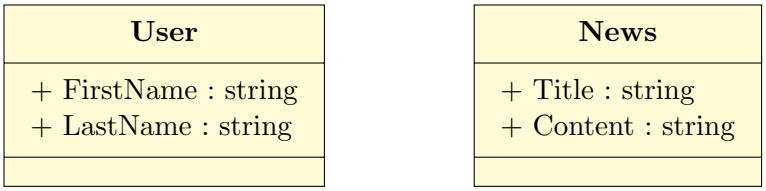
Źródło generacji:

```
domain
├── user.json
├── user_fields
│   ├── FirstName.json
│   ├── LastName.json
│   └── ...
├── news.json
├── news_fields
│   ├── Title.json
│   ├── Content.json
└── ...
```

Wynik generacji:

```
Model
├── User.cs
├── News.cs
└── ...
```

Rysunek 5.4: Przykład organizacji plików przy generacji wielu katalogów wynikowych na podstawie wielu katalogów źródłowych. Pojedynczy plik wynikowy jest generowany na podstawie wielu plików źródłowych.



Rysunek 5.5: Przykładowa dziedzina aplikacji przestawiona na diagramie klas będącym częścią języka UML.

tora wymagać mogą atrybutów, których przewidzenie na etapie projektowania generatora jest niemożliwe. Przykładowo, do opisu encji mogą należeć takie atrybuty, jak:

- opis encji, który powinien znaleźć się w dokumentancji systemu;
- wersja systemu, w której encja została wprowadzona;
- widoki bazy danych, kórych źródłem jest encja.

Diagram klas nie przewiduje przechowywania żadej z tych informacji.

Co więcej, podstawową jednostką diagramu klas UML jest encja, co samo w sobie jest ograniczeniem elastyczności. Opis dziedziny aplikacji tworzony przez użytkownika generatora może natomiast za postawową jednostkę obierać na przykład pojedyncze pole encji. Do opisu pojedynczego pola encji, oprócz jego nazy i typu, mogą należeć takie atrybuty, jak:

- opis pola, który powinien znaleźć się w dokumentancji systemu;
- wersja systemu, w której pole zostało wprowadzone;
- widoki bazy danych, w których występuje pole.

Diagram klas nie przewiduje przechowywania żadej z tych informacji.

5.4.2 XML

Język XML jest powszechnie używany do opisu dziedziny, Jest o niego oparty na przykład język WSDL (język definicji usług sieciowych, ang. *Web Services Description Language*) [16] stosowany do opisu kontraktów (ang. *contract*) realizowaych przez usługi sieciowe (ang. *web service*).

Język XML jest pozbawiony wad języka UML - można w nim zamodelować dowolne atrybuty. Tę samą dziedzinę, która została przedstawiona na diagramie klas języka UML (rysunek 5.5), ale wzbogaconą o niedostępne na tym diagramie atrybuty, przedstawia rysunek 5.6.

5.4.3 JSON

Język JSON posiada te same cechy, co język XML. Jest on jednak prostszy i bardziej zwięzły - nie jest to język znaczników, także nazwy atrybutów nie są duplikowane w znaczniku otwierającym i zamykającym. On również jest używany do opisu dziedziny, na przykład do opisu metadanych usług sieciowych opartych o protokół OData [17].

Rysunek 5.7 przedstawia przykładowy opis dziedziny zapisany w tym języku.

5.4.4 YAML

Język YAML jest pod względem użyteczności podobny do języków XML i JSON.

Przykładowy opis dziedziny zapisany w tym języku został przedstawiony na rysunku 5.7.

5.4.5 Domyślny język opisu dziedziny

Jako że język UML nie spełnia wymagań trzonu generatora, domyślnie wspieranym językiem opisu dziedziny aplikacji będzie jeden z pozostałych trzech opisanych wyżej. Będzie to język JSON, za kórym przemawiają nastepujące zalety:

- ma najprostszą składnię za wszystkich trzech kandydatów;
- jest bardziej czytelny dla człowieka niż XML;
- jest szerzej znany i stosowany niż YAML.

Aby zachować elastyczność, język opisu dziedziny będzie jednak można łatwo wymienić.

```
<Entities>
 <Entity Name="User">
  <Description>The user of the system.</Description>
  <IntroducedIn>1.1</IntroducedIn>
  <Fields>
   <Field Name="FirstName">
    <Type>string</Type>
    <Description>The first name of the user.</Description>
   </Field>
   <Field Name="LastName">
    <Type>string</Type>
    <Description>The last name of the user.</Description>
   </Field>
  </Fields>
 </Entity>
 <Entity Name="News">
  <Description>The piece of news.</Description>
  <IntroducedIn>1.0</IntroducedIn>
  <Fields>
   <Field Name="Title">
    <Type>string</Type>
    <Description>The title of the piece of news.</Description>
   </Field>
   <Field Name="Content">
    <Type>string</Type>
    <Description>The content of the piece of news.</Description>
   </Field>
  </Fields>
 </Entity>
</Entities>
```

Rysunek 5.6: Przykładowa dziedzina aplikacji opisana językiem XML.

5.5 Czy wymagać stworzenia schematu definicji dziedziny aplikacji?

Kolejną decyzją jest wybór typu danych, do którego deserializoway będzie opis dziedziny aplikacji. Wyboru należy dokonać pomiędzy dwoma przeciwstawnymi podejściami:

1. Wymóg określenia schematu dziedziny aplikacji:
   - deserializacja definicji dziedziny aplikacji do obiektu konkretnego typu;
   - wystąpienie w opisie pola nieobecnego w schemacie dziedziny jest ignorowane lub powoduje błąd deserializacji.
2. Brak takiego wymogu

```
[
 {
  "Name": "User",
  "Description": "The user of the system.",
  "IntroducedIn": "1.1"
  "Fields": [
   {
    "Name": "FirstName",
    "Type": "string",
    "Description": "The first name of the user.",
   },
   {
    "Name": "LastName",
    "Type": "string",
    "Description": "The last name of the user.",
   },
  ]
 },
 {
  "Name": "News",
  "Description": "The piece of news.",
  "IntroducedIn": "1.0",
  "Fields": [
   {
    "Name": "Title",
    "Type": "string",
    "Description": "The title of the piece of news.",
   },
   {
    "Name": "Content",
    "Type": "string",
    "Description": "The content of the piece of news.",
   },
  ]
 }
]
```

Rysunek 5.7: Przykładowa dziedzina aplikacji opisana językiem JSON.

- deserializacja definicji dziedziny aplikacji do obiektu dynamicznego lub słownika;
- wystąpienie w opisie pola nieobecnego w schemacie jest akceptowalne - pole takie znajdzie się w obiekcie powstałym w wyniku deserializacji.

Pierwsza możliwość wydaje się lepsza, a to za sprawą następujących zalet:

```
---
- Name:         User
  Description:  The user of the system.
  IntroducedIn: 1.1
  Fields:
   - Name:        FirstName
     Type:        string
     Description: The first name of the user.

   - Name:        LastName
     Type:        string
     Description: The last name of the user.

- Name:         News
  Description:  The piece of news.
  IntroducedIn: 1.0,
  Fields:
   - Name:        Title
     Type:        string
     Description: The title of the piece of news.

   - Name:        Content
     Type:        string
     Description: The content of the piece of news.
...
```

Rysunek 5.8: Przykładowa dziedzina aplikacji opisana językiem YAML.

- generator sprawdza spójność opisu dziedziny aplikacji;
- ewentualne błędy (literówki) opisu zostaną wykryte na etapie deserializacji;
- szablony generacji mogą pracować na danych silnie typizowanych.

Domyślnie wspieranym podejściem będzie więc deserializacja opisu dziedziny do obiektu silnego typu. Aby jednak zachować elastyczność, możliwa będzie deserializacja do typu dynamicznego.

5.6 Wybór silnika do generacji kodu

Ostatnią decyzją dotyczącą założeń dotyczących trzonu generatora jest wybór silnika generacji kodu (ang. *templating engine*) przez niego używanego. Domyślnie używany silnik wybrano spośród następujących możliwości:

5.6.1 XSLT

XSLT (Xml Stylesheets Transformations) jest językiem generacji dowolnych plików tekstowych na podstawie plików XML. Jest to standard zaproponowany przez organizację W3C. Jego zaletą jest to, że szablony generacji tworzone są w języku XML, co zwalnia programistę z potrzeby poznawania kolejnego języka. Jest on jednak mało czytelny, a wprowadzanie zmian w szablonie jest niewygodne.

5.6.2 Razor

Razor jest silnikiem generacji tekstu stworzonym na potrzeby platformy ASP.NET MVC. Służy głównie do generacji kodu HTML, jednak może być używany w celu tworzenia dowolnych plików tekstowych.

Wadą tego wyboru - niezależną od samego silnika - jest to, że szablony Razor są powszechnie używane w typowych aplikacjach opartych o platformę ASP.NET MVC, a więc same mogą stanowić pliki wynikowe dla generatora. Stworzenie szablonu Razor generującego inny szablon Razor jest możliwe, ale taki szablon byłby bardzo nieczytelny - co dyskwalifikuje tę opcję.

5.6.3 T4

T4 to silnik generacji tekstu wbudowany w środowisko programistyczne Microsoft Visual Studio. Jest on przeznaczony do generowania plików tekstowych dowolnego typu na podstawie danych przekazanych szablonowi.

Szablon T4 nie jest interpretowany, a kompilowany do kodu języka C#, co daje następujące korzyści:

- szybkość generacji tekstu jest większa niż w przypadku pozostałych silników;
- szablon może odwoływać się do bibliotek zewnętrznych i wywoływać ich metody (np. w celu pobrania potrzebnych danych z bazy danych);
- szablon może wywoływać inne szablony lub dziedziczyć po innym szablonie.

Taka integracja ze środowiskiem programistycznym i platformą .NET niesie za sobą dalsze konsekwencje:

- tworzenie szablonów jest ułatwione ze względu na podświetalnie i podpowiadanie składni (zarówno języka szablonu i jak korzystającego z niego kodu C#),
- silnik ten jest niedostępny dla programistów innych platform.

5.6.4 Wybrany silnik

Ze względu na fakt, że zarówno trzon generatora jak i generator konkretnego typu aplikacji stworzone zostaną w oparciu o platformę .NET, silnikiem generacji domyślnie wspieranym przez trzon generatora będzie T4. Powodem tego wyboru jest łatwość tworzenia jego szablonów w środowisku Visual Studio.

Szablon:

```
<#@ template language="C#" #>
<#@ parameter type="Sample.Schema.Entity" name="Entity" #>
namespace Sample
{
 /// <summary> <#= Entity.Description #> </summary>
 public class <#= Entity.Name #>
 {
<# foreach (var field in Entity.Fields) { #>
  /// <summary> <#= field.Description #> </summary>
  public <#= field.Type #> <#= field.Name #> { get; set; }
<# } #>
 }
}
```

Wynik:

```
namespace Sample
{
 /// <summary> The user of the system. </summary>
 public class User
 {
  /// <summary> The first name of the User. </summary>
  public string FirstName { get; set; }

  /// <summary> The last name of the User. </summary>
  public string LastName { get; set; }
 }
}
```

Rysunek 5.9: Przykładowy szablon T4 i wynikowy kod C#.

Przykładowy szablon został przestawiony na rysunku 5.9.

Podobnie jak w przypadku pozostałych podjętych dezycji, używany przez generator silnik będzie mógł być zastąpiony przez inny.

5.7 Podsumowanie

Wymagania dotyczące trzonu generatora zostały skompletowane. Następnym krokiem jest sformułowanie założeń dotyczących generatora aplikacji konkretnego typu.

Rozdział 6

Implementacja generatora aplikacji pojedynczego typu

Założenia co do generatora aplikacji pojedynczego typu

- wybrać typ aplikacji
- dlaczego CQRS
- bo zdenormalizowana dziedzina
- gdzie w CQRS zdefiniowana jest dziedzina (encje) aplikacji? Model do odczytywania nie zawiera przecież encji, a tylko widoki.
- model "read" i model "write" częściowo na siebie zachodzą (lub nawet "read" zawiera "write"). Jak uniknąć duplikacji metadanych?
- opisać CQRS
- że często idzie w parze z Event Sourcing
- opisać Event Sourcing
- że całość dobrze idzie w parze z NoSQL
- opisać NoSQL
  - opisać rodzaje baz NoSQL
  - wybrać bazę NoSQL i dlaczego Cassandra
  - opisać Cassandrę

Implementacja generatora aplikacji pojedynczego typu

- Ogólnie nie chodzi o to, żeby w ogóle nie pisać nazw klas / właściwości - tylko o to, żeby nie trzeba było pamiętać o wszystkich miejscach, gdzie dana encja jest używana.
- czy w ogóle generować schemat bazy danych (no schema) (?)

- wybrać sposób definicji dedziny
- że będzie schema
- ale że schema powinna być w jednym miejsu - będzie jako klasa w kodzie
- wybór formatu
- uml, emf się nie nadają - operują na encjach, a nie na polach
- xml albo json, bez różnicy - czytelniejszy i bardziej intuicyjny wydaje się JSON (xml ma atrybuty i węzły zagnieżdzone - nie wiadomo czego użyć)
- zdefiniowanie schematu opisu dziedziny na potrzeby CQRS
  - PresentIn
- Wybrór klienta Cassandry (?)
- Sformułowanie przykładu aplikacji
- Zapisanie schemy w JSONie
- Implementacja szablonów
- Implementacja kolejnych modułów aplikacji
  - DAL (repozytorium bazowe zahardkodowane, konkretnych nie da się wygenerować)
  - BLL (obsługa zdarzeń zahardkodowana, da się wygenerować EventHandlery)
  - WEB (Nancy, da się wygenerować ViewModele, formy)

Założenia dotyczące całości:

- generator ma generować dziedzinę aplikacji CQRS wykorzystującej Cassandrę:
  - schemat DLL
  - klasy C#: model Read, model Write
  - dokumentancja HTML
- ...

Szczególna uwaga zostanie poświęcona aplikacjom opartym o architekturę CQRS i wykorzystującym bazy danych typu NoSQL. Specyficzną cechą takich aplikacji jest to, że operują one na modelach o wysokim stopniu denormalizacji, co wiąże się z masowo występującą duplikacją metadanych.

Rozdział 7

Ocena rozwiązania

- Co dało się wygenerować, a co nie
- Jakiej duplikacji udało się uniknąć
- jak łatwo wprowadza się zmiany w aplikacji
- ile pracy wymagałoby dodanie nowej funkcjonalności (API, eksport) - przykład?
- jaką inną aplikację można wygenerować (przykład)

Rozdział 8

Podsumowanie

- Wnioski
- Koniec

Bibliografia

[1] *The Pragmatic Programmer.* Hunt A., Thomas D. Addison-Wesley. Westford 2013. ISBN 0-201-61622-X. *The Evils of Duplication*, s. 26-33.

[2] *The Pragmatic Programmer.* Hunt A., Thomas D. Addison-Wesley. Westford 2013. ISBN 0-201-61622-X. *Software Entropy*, s. 4-6.

[3] *Duplication in Software* [online]. Just Software Solutions. http://www.justsoftwaresolutions.co.uk/design/duplication.html [dostęp: lipiec 2014].

[4] *Repetition* [online]. The Bad Code Spotter's Guide. Diomidis Spinellis. http://www.informit.com/articles/article.aspx?p=457502&seqNum=5 [dostęp: lipiec 2014].

[5] *Continous delivery* [online]. ThoughtWorks. http://www.thoughtworks.com/continuous-delivery [dostęp: lipiec 2014].

[6] *Enity Framework* [online]. CodePlex. https://entityframework.codeplex.com/ [dostęp: lipiec 2014].

[7] *SqlMetal.exe (Code Generation Tool)* [online]. Microsoft Developer Network. http://msdn.microsoft.com/pl-pl/library/bb386987(v=vs.110).aspx [dostęp: lipiec 2014].

[8] *The Web framework for perfectionists with deadlines* [online]. django. https://www.djangoproject.com/ [dostęp: lipiec 2014].

[9] *Hibernate. Everything data.* [online]. Hibernate. http://http://hibernate.org/ [dostęp: lipiec 2014].

[10] *Doxygen: Main Page* [online]. Doxygen. http://www.stack.nl/ dimitri/doxygen/ [dostęp: lipiec 2014].

[11] *Javadoc Tool Homepage* [online]. Oracle. http://www.oracle.com/technetwork/java/javase/docu
jsp-135444.html [dostęp: lipiec 2014].

[12] *XML Documentation* [online]. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/b2s063f7(vs.71).aspx [dostęp: lipiec 2014].

[13] *pydoc — Documentation generator and online help system* [online]. Python v2.7.8 documentation. https://docs.python.org/2/library/pydoc.html [dostęp: lipiec 2014].

[14] *Automatic programming* [online]. Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Automatic_programming [dostęp: lipiec 2014].

[15] *XSL Transformations (XSLT) Version 2.0* [online]. W3C. http://www.w3.org/TR/xslt20/ [dostęp: lipiec 2014].

[16] *Web Services Description Language (WSDL) 1.1* [online]. W3C. http://www.w3.org/TR/wsdl [dostęp: lipiec 2014].

[17] *Open Data Protocol* [online]. OData. http://www.odata.org/ [dostęp: lipiec 2014].

Oświadczenie

Oświadczam, że Pracę Dyplomową pod tytułem *"[TYTUŁ]"*, którą kierował dr inż. Jakub Koperwas, wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

.....

Michał Aniserowicz