POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2013/2014

PRACA DYPLOMOWA MAGISTERSKA

Michał Aniserowicz

[TODO] (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, strukturę 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: struktury 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 struktury 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:

- struktura 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 znalazłaby się struktura 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 struktury bazy danych

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

Przykładami narzędzi umożliwiających taką generację kodu są:

- EntityFramework[1] - rozbudowane narzędzie ORM (ang. *Object-Relational Mapping*) przeznaczone na platformę .Net;
- SQLMetal[2] - narzędzie dla platformy .Net, którego jedynym zadaniem jest generacja kodu źródłowego klas na podstawie struktury bazy danych;
- Django[3] - platforma aplikacji webowych dla języka Python.

2.6.2 Generator struktury 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 strukturę bazy danych na podstawie klas należących do implementacji modelu w kodzie źródłowym aplikacji.

Przykłady generatorów *"Code First"* to:

- Hibernate[4] - 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[5] - popularne narzędzie obsługujące wiele języków programowania (w tym C++, Java C#), wspierające wiele formatów dokumentacji (w tym HTML, PDF, LaTeX, XML);
- JavaDoc[6] - generator dedykowany językowi Java, dymyślnie wspiera jedynie format HTML;
- C# XML Documentation[7] - format tworzenia dokumentacji wbudowany w język C#, na podstawie którego generowana jest dokumentacja w formacie XML;

[1] EntityFramework - strona projektu: https://entityframework.codeplex.com/
[2] SQLMetal - strona projektu: http://msdn.microsoft.com/pl-pl/library/bb386987(v=vs.110).aspx
[3] Django - strona projektu: https://www.djangoproject.com/
[4] Hibernate - strona projektu: http://hibernate.org/
[5] Doxygen - strona projektu: http://www.stack.nl/ dimitri/doxygen/
[6] JavaDoc - strona projektu: http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[7] C# XML Documentation - strona projektu: http://msdn.microsoft.com/en-us/library/b2s063f7(vs.71).aspx

- pydoc[8] - 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*) [6]. 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 strukturze 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.

[8] pydoc - strona projektu: https://docs.python.org/2/library/pydoc.html

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
  - rdzeń narzędzia do generacji
  - generator aplikacji jednego typu

Rozdział 5

Założenia dotyczące rdzenia narzędzia

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

5.1 Podstawowe założenia dotyczące rdzenia narzędzia

O ile elastyczność generatora konkretnego typu aplikacji może być ograniczona, o tyle rdzeń 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 (patrz: sekcja 5.6.1). Dzięki temu programista korzystający z komponentu nie będzie 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, rdzeń narzędzia nie pownien narzucać sposobu formatowania danych wejściowych (w tym przypadku - opisu dziedziny aplikacji).

Dla wygody autora narzędzia zakłada się, że zarówno jego rdzeń 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 rdzenia 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 rdzeń 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).

Rdzeń 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 rdzeń narzędzia, a także organizacja plików, które będzie w stanie wygenerować. Jak wspomniano wyżej, od rdzenia 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 strukturę 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_structure.sql
```

Wynik generacji:

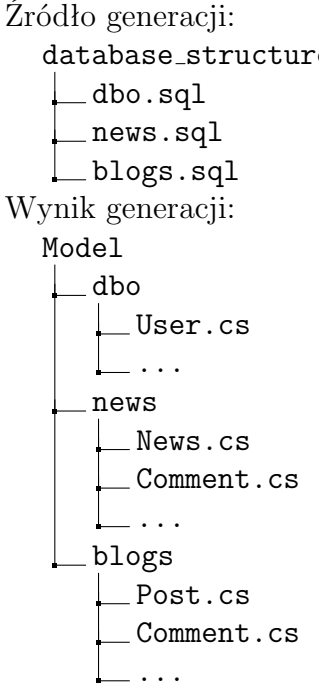
```
Model
├── User.cs
├── News.cs
├── Comment.cs
└── ...
```

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 struktury bazy danych na kilka plików - np. według nazw schematów (ang. *schema*), 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.



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. Rdzeń 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.

Źródło generacji:

```
database_structure
├── 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).

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.

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 przedstawiono kilka z możliwych wyborów:

Źródło generacji:

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

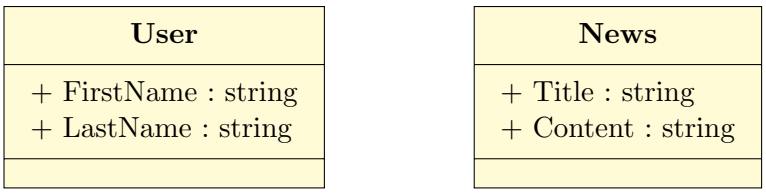
Wynik generacji:

```
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.

5.4.1 UML

Oczywistym wyborem sposobu opisu dziedziny aplikacji wydaje się być język UML[1], stworzony między innymi do tego właśnie celu. Do opisu przykładowej dziedziny posłużyć może diagram klas przedstawiony na rysunku 5.5.



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

Trzeba jednak zauważyć, ze taki opis nie jest wystarczająco elastyczny - posiada on zdefiniowany z góry zestaw atrybutów. Natomiast rzeczywiste scenariusze użycia generatora wymagać mogą atrybutów, których przewidzenie na etapie projektowania jego rdzenia 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;

[1] Unified Modeling Language: http://www.uml.org/

- 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 (patrz: rysunek 5.4). 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[2] jest powszechnie używany do opisu dziedziny. Jest o niego oparty na przykład język WSDL (język definicji usług sieciowych, ang. *Web Services Description Language*) [7] stosowany do opisu kontraktów (ang. *contract*) realizowaych przez usługi sieciowe (ang. *web service*).

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[3] posiada zestaw cech upodabniający go do języka XML. Jest on jednak prostszy i bardziej zwięzły - nie jest to język znaczników, więc nazwy atrybutów nie są duplikowane w znaczniku otwierającym i zamykającym. JSON również jest szeroko stosowany do opisu dziedziny, na przykład do opisu metadanych usług sieciowych opartych o protokół OData [8].

Rysunek 5.7 przedstawia przykładowy opis dziedziny zapisany w tym języku.

5.4.4 YAML

Język YAML[4] jest pod względem użyteczności podobny do języków XML i JSON. Jest to język najbardziej zwięzły z przedstawionych, jednak nie jest on używany w żadnym popularnym standardzie.

Przykładowy opis dziedziny zapisany języku YAML został przedstawiony na rysunku 5.7.

[2]Extensible Markup Language: http://www.w3.org/XML/
[3]JavaScript Object Notation: http://json.org/
[4]YAML Ain't Markup Language: http://www.yaml.org/

```
<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 zapisana w języku XML.

5.4.5 Domyślny język opisu dziedziny

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

- ma on 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.

```
[
 {
  "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 zapisana w języku JSON.

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:

```
---
- 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 zapisana w języku YAML.

1. Dane silnie typizowane, z czym wiąże się wymóg określenia schematu opisu dziedziny aplikacji. Konsekwencje tego wyboru:
   - wynikiem deserializacji opisu dziedziny aplikacji będzie obiekt konkretnego typu;
   - wystąpienie w opisie pola nieobecnego w schemacie dziedziny zostanie zignorowane lub spowoduje błąd deserializacji.
2. Dane dynamiczne; brak wymogu istnienia schematu opisu dziedziny. Konsekwencje wyboru:
   - wynikiem deserializacji opisu dziedziny będzie obiekt dynamiczny lub zbiór par klucz-wartość;
   - wystąpienie w opisie pola nieobecnego w schemacie będzie akceptowalne - pole takie znajdzie się w obiekcie powstałym w wyniku deserializacji.

Pierwsza możliwość wydaje się bardziej korzystna Przemawiają za nią następujące zalety:

- generator będzie sprawdzał spójność opisu dziedziny aplikacji;
- ewentualne błędy opisu (na przykład tzw. literówki) zostaną wykryte na etapie deserializacji;
- szablony generacji będą mogły 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 generacji kodu

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

5.6.1 XSLT

XSLT[5] jest językiem generacji dowolnych plików tekstowych na podstawie plików XML. Jest to standard zaproponowany przez organizację W3C [9]. 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[6] 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 ten silnik.

5.6.3 T4

T4[7] 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:

[5] Xml Stylesheets Transformations - strona języka: http://www.w3.org/TR/xslt20/
[6] Razor - strona projektu: https://github.com/Antaris/RazorEngine
[7] T4 - strona projektu: http://msdn.microsoft.com/pl-pl/library/bb126445.aspx

- 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 rdzeń generatora jak i generator konkretnego typu aplikacji stworzone zostaną w oparciu o platformę .NET, silnikiem generacji domyślnie wspieranym przez rdzeń generatora będzie T4. Powodem tego wyboru jest łatwość tworzenia jego szablonów w środowisku Visual Studio.

Przykładowy szablon został przedstawiony 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 rdzenia generatora zostały skompletowane. Następnym krokiem jest sformułowanie założeń dotyczących generatora aplikacji konkretnego typu.

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#.

Rozdział 6

Sprecyzowanie typu generowanych aplikacji

Po zebraniu założeń dotyczących rdzenia narzędzia generującego aplikacje, nadszedł czas na wybór typu aplikacji, które będą generowane przez narzędzie.

6.1 Wybór typu aplikacji

Narzędzie powinno generować taki typ aplikacji, aby można było zbadać jego użyteczność i ocenić, na ile eliminuje ono duplikację wiedzy o systemie. Aby było to możliwe, w generowanych aplikacjach powinno występować dużo miejsc, gdzie potencjalnie może pojawić się duplikacja.

Wymaganie to spełniają aplikacje o architekturze wielowarstwowej (ang. *multi-tier architecture, n-tier architecture* [10]).

6.1.1 Architektura wielowarstwowa

Architektura wielowarstwowa to taka, w której ogólne obszary przetwarzania danych w aplikacji są fizycznie rozdzielone pomiędzy osobne komponenty. Współpraca pomiędzy tymi komponentami jest zorganizowana w taki sposób, że komponent A może korzystać z funkcjonalności komponentu B tylko wtedy, gdy komponent B należy do warstwy logicznie umiejscowionej nie wyżej niż warstwa, do której należy komponent A.

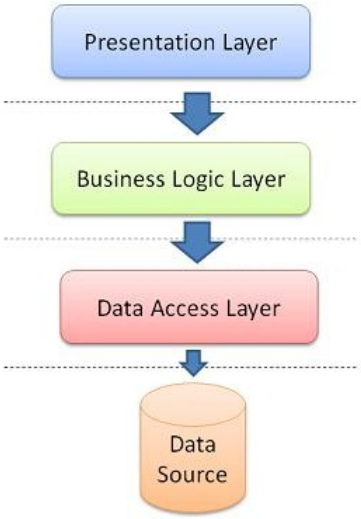
Przykładem, a jednocześnie najpopularniejszą realizacją tej architektury jest architektura trójwarstwowa (ang. *three-tier architecture*), która wprowadza podział aplikcji na trzy warstwy:

1. Warstwa prezentacji (ang. *Presentation Layer*) - odpowiada za komunikację z użytkownikiem aplikacji (np. poprzez interfejs graficzny) lub innymi systemami (np. poprzez usługi sieciowe); jest to warstwa logicznie najwyższa;

2. Warstwa logiki biznesowej (ang. *Business Logic Layer, BLL*) - odpowiada za przetwarzanie danych zgodnie z wymaganiami funkcjonalnymi aplikacji;

3. Warstwa dostępu do danych (ang. *Data Access Layer, DAL*) - udostępnia mechanizmy odczytu i zapisu danych składowanych przez aplikację (np. w pamięci lub w bazie danych); jest to warstwa logicznie najniższa.

Współpracę pomiędzy warstwami architektury trójwarstwowej przedstawia diagram zamieszczony na rysunku 6.1.



Rysunek 6.1: Współpraca pomiędzy warstwami architektury trójwarstwowej [11].

W takiej architekturze elementy dziedziny aplikacji często mają swoje odwzorowanie w każdej z warstw, na przykład:

- jako obiekty modelu w warstwie dostępu do danych;
- jako obiekty biznesowe (ang. *business object*) [12] w warstwie logiki biznesowej;
- jako modele widoków (ang. *view model*) [13] interfejsu użytkownika lub obiekty transportu danych (ang. *Data Transfer Object, DTO*) [14] usług sieciowych w warstwie prezentacji.

To sprawia, że aplikacja o architekturze wielowarstwowej jest narażona na powszechne występowanie duplikacji wiedzy na temat dziedziny aplikacji, a tym samym dobrze nadaje się jako typ aplikacji generowanych przez narzędzie.

6.2 CQRS

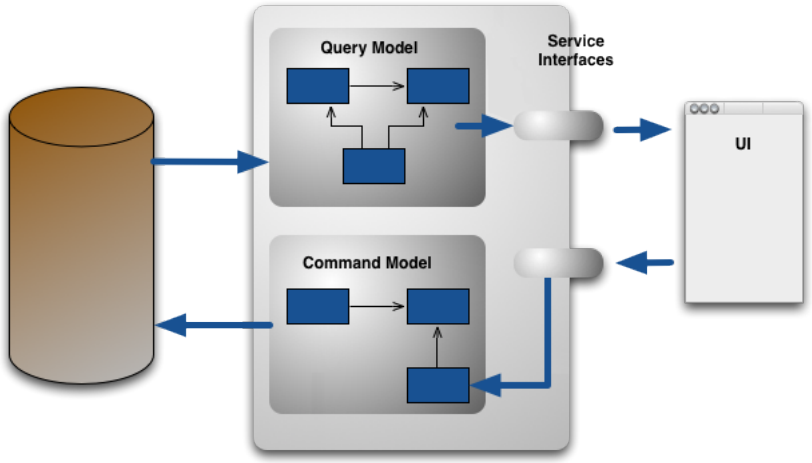
Przypadkiem szczególnym architektury wielowarstwowej jest architektura CQRS (*Command Query Responsibility Segregation*) [15]. Zakłada ona podział wszystkich działań w aplikacji na dwa rodzaje:

- zapytanie (ang. *query*) - działanie wiążące się z odczytaniem danych z bazy danych (lub innego źródła danych);
- komenda (ang. *command*) - działanie wiążące się z modyfikacją danych.

Działania te w architekturze CQRS są rozłączne. Ich wykonywaniem zajmują się dwa osobne modele danych aplikacji:

- model zapytań (ang. *Query Model*) - model przeznaczony do odczytu danych;
- model komend (ang. *Command Model*) - model przeznaczony do modyfikacji danych.

Modele te mogą być całkowicie rozłączne lub częściowo na siebie zachodzić. Koncepcyjny schemat tej architektury przedstawia rysunek 6.2.



Rysunek 6.2: Schemat architektury CQRS [16].

Podział odpowiedzialności pomiędzy komponenty przedstawia się następująco:

- model zapytań zajmuje się odczytywaniem danych z bazy danych;
- odpowiedzialnością modelu komend jest realizacja logiki biznesowej aplikacji, w tym weryfikacja poprawności danych, aktualizacja danych w bazie danych itd.;
- warstwa prezentacji (*UI*):
  - wyświetla dane pobrane z modelu zapytań za pośrednictwem interfejsów (*Service Interfaces*),
  - przekazuje - w postaci komend - akcje wykonywane przez użytkownika do modelu komend.

Wprowadzenie podziału pomiędzy zapytanie i komendę niesie ze sobą dwie ważne zalety:

- skomplikowana dziedzina aplikacji może być podzielona na dwie prostsze dziedziny, co ułatwia jej zrozumiene i operowanie na niej;
- zapytania i komendy mogą być wykonywane równolegle, co poprawia wydajność aplikacji;
- zapytania są wykonywane na specjalnie przygotowanych dla nich danych (np. zmaterializowanych widokach bazy danych), co ma bardzo pozytywny wpływ na ich wydajność.

W parze z zaletami idą jednak wady:

- synchronizacja obu modeli w przypadku, gdy korzystają one z osobnych źródeł danych może być kłopotliwa; problem ten nie występuje na przykład wtedy, gdy model komend operuje na tabelach bazy danych, a model zapytań - na zmaterializowanych widokach, których źródłem danych są te tabele (synchronizacja modeli odbywa się wtedy automatycznie po stronie bazy danych);
- aby każde zapytanie mogło być obsłużone jak najszybciej, model zapytań musi być w dużym stopniu zdenormalizowany.

Konsekwencją drugiej wady jest to, że w modelu zapytań masowo występuje duplikacja elementów dziedziny aplikacji. Jest to dobry powód do tego, aby generator generował aplikacje oparte właśnie o architekturę CQRS. Dodatkowo, wybór tej architektury stworzy okazję do przyjrzenia się innym problemom związanym z jej zastosowaniem. Te problemy to:

1. Model komend i model zapytań częściowo na siebie zachodzą lub nawet model zapytań w całości zawiera model komend - rodzi to dwa pytania:
   - jak w tej sytuacji uniknąć duplikacji wiedzy na temat dziedziny aplikacji?
   - który model wybrać na “pojedynczą, jednoznaczną i autorytatywną” (patrz: Zasada “DRY” w rozdziale 2) reprezentację wiedzy o dziedzinie aplikacji?
2. Model komend może nie być nigdzie fizycznie przechowywany - komendy mogą bezpośrednio aktualizować zdenormalizowaną strukturę tabel bazy danych. Gdzie w takim przypadku należy umieścić wiedzę na temat encji należących do dziedziny apliacji? (Model zapytań nie zawiera encji dziedziny, a tylko widoki na te encje.)

Architektura CQRS często idzie w parze z wykorzystaniem wzorca Event Sourcing i baz danych typu NoSQL. Zagadnienia te zostaną opisane w kolejnych sekcjach.

6.3 Event Sourcing

Event Sourcing [17] jest wzorcem architektonicznym, który realizuje przechowywanie stanu aplikacji poprzez przechowywanie wszystkich zdarzeń (ang. *event*), które zaszły w systemie od początku jego działania aż do stanu obecnego. Terminem "zdarzenie" określa się tutaj dowolną akcję, która ma wpływ na stan systemu.

6.3.1 Cechy zdarzenia

Zdarzenie posiada następujące cechy [18]:

- ma ono znaczenie dopiero wtedy, kiedy rzeczywiście wystąpi (np. "Wpis został skomentowany") - zdarzeń przyszłych, potencjalnych nie bierze się pod uwagę;
- jest ono niezmienne - jako że zdarzenie zaszło w przeszłości, nie może ono zostać zmienione ani cofnięte; jego skutki mogą jednak zostać zniwelowane przez inne zdarzenie (np. "Komentarz został usunięty");
- powinno ono mieć znaczenie biznesowe, a nie implementacyjne - opisanie zajścia zdarzenia słowami: "Do tabeli *Comment* dodano nowy rekord" niesie ze sobą mniejszą wartość biznesową niż: "Wpis został skomentowany".
- zazwyczaj zawiera ono informacje na temat kontekstu, w którym zaszło (np. "Użytkownik X dodał komentarz Y pod wpisem Z", "Moderator W usunął komentarz Y");
- informacja o zajściu zdarzenia jest informacją jednokierunkową od nadawacy (ang. *publisher*) do odbiorcy (ang. *subscriber*) lub wielu odbiorców; reakcja odbiorcy na zdarzenie nie jest bezpośrednio znana nadawcy.

Rejestrowanie zdarzeń odbywa się poprzez stworzenie obiektu opisującego daną akcję i zapisanie go w systemie. Zdarzenia rejestrowane są w dzienniku zdarzeń (ang. *event log* lub *event storage*). Za reagowanie na zdarzenia, a w efekcie faktyczną zmianę stanu systemu odpowiadają wyznaczone do tego obiekty implementujące odpowiednie procedury obsługi (ang. *event handlers*). Obiekty te pobierają nieprzetworzone zdarzenia z dziennika zdarzeń i wykonują odpowiadające im procedury. Pojedyczne zdarzenie może zostać przetworzone przez wiele takich obiektów, wykonujących wiele osobnych modyfikacji stanu systemu.

6.3.2 Zalety i wady wzorca

Wzorzec Event Sourcing wprowadza nastepujące zalety:

- dziennik zdarzeń dostarcza informacji o historii działań użytkowników systemu - informacje te mogą być przydatne podczas szukania przyczyn wystąpienia błędów;

- aplikację można łatwo usunąć i przywrócić do poprzedniego stanu poprzez ponowne zarejestrowanie wszystkich zaszłych w niej zdarzeń (np. podczas przenoszenia jej na inną maszynę);
- stan aplikacji można łatwo cofnąć do dowolnego punktu w czasie - wystarczy wyczyścić stan aplikacji i ponownie zarejestrować wszystkie zdarzenia, ktore zaszły przed wybranym punktem w czasie (może to ułatwić szukanie przyczyn wystąpienia błędów);
- w razie wykrycia zdarzenia będącego przyczyną wystąpienia błędu, można usunąć to zdarzenie z dziennika i ponownie zarejestrować wszystkie zdarzenia, które nastąpiły po nim; na podobnej zasadzie można skorygować kolejność zdarzeń zarejestowanych w złej kolejności (z tej zalety powinno się korzystać jedynie w sytuacjach awaryjnych, ponieważ narusza ona cechę zdarzeń mówiącą o ich niezmienności).

Wady:

- implementacja dziennika zdarzeń nie jest łatwa - powinna ona gwarantować, że zdarzenia zostaną zarejestrowane w kolejności ich zgłaszania (co nie jest oczywiste w aplikacjach wielowątkowych);
- z reguły reakcja na zdarzenie odbywa się asynchronicznie względem procesu zgłaszającego jego zajście - zmiany stanu systemu nie są więc widoczne natychmiast po zajściu zdarzenia
  - zjawisko to można zaobserwować na przykład na portalu YouTube[1] - zdarzenie "Obejrzano film" jest przetwarzane znacznie wolniej niż zdarzenie "Dodano komentarz" (wyświetlenia filmu sprawdzane są pod kątem fałszywych wyświetleń mających na celu sztuczne zwiększenie popularności filmu [19]), co skutkuje wyświetleniem pod danym filmem znacznie większej liczby komentarzy niż liczby wyświetleń (jeśli komentarzy i wyświetleń jest odpowiednio dużo);
- odtwarzanie stanu systemu, w którym zarejestrowano tysiące zdarzeń może trwać bardzo długo
  - rozwiązaniem tego problemu jest tworzenie co jakiś czas migawek (ang. *snapshot*) przechowujących stan systemu - aby odtworzyć stan systemu, wystarczy wtedy załadować najnowszą migawkę i ponownie zarejestrować tylko te zdarzenia, które zaszły po jej stworzeniu;
- zmiany w kodzie źródłowym systemu mogą spowodować sytuację, w której dawno zarejestrowane zdarzenia przestaną być kompatybilne z nową wersją systemu.

[1] YouTube: https://www.youtube.com/

6.3.3 Przykłady aplikacji wykorzystujących Event Sourcing

Aby przybliżyć działanie wzorca, poniżej przedstawiono kilka przykładów aplikacji używanych na co dzień, które są oparte o Event Sourcing lub częściowo go wykorzystują:

- systemy kontroli wersji - dziennik zdarzeń przechowuje zmiany dokonywane na plikach znajdujących się w wersjonowanym katalogu;
- edytory tekstu i edytory graficzne - dziennik zdarzeń przechowuje zmiany dokonywane na edytowanym pliku.

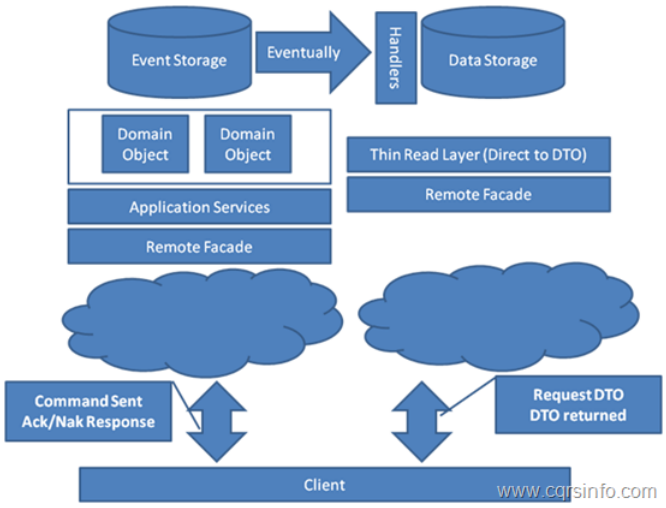
Wzorzec ten ma jednak zastosowanie nie tylko w aplikacjach użytkowych. W połączeniu z wzorcem CQRS, sprawdza się także w aplikacjach klasy *enterprise*.

6.3.4 Współpraca z CQRS

Zasady działania wzorca Event Sourcing w systemie opartym o architekturę CQRS są następujące [20]:

- rolę modelu komend pełni dziennik zdarzeń - jedynym zadaniem komend jest zarejestrowanie zdarzenia w dzienniku;
- stan systemu jest przechowywany w modelu zapytań;
- za synchronizację modelu zapytań z modelem komend odpowiadają procedury obsługi zdarzeń (*event handlers*) - synchronizacja ta odbywa się asynchronicznie;
- wzorzec Event Sourcing nie ma wpływu na warstwę prezentacji systemu.

Współpraca ta została schematycznie przedstawiona na rysunku 6.3.



Rysunek 6.3: Współpraca wzorca Event Sourcing z architekturą CQRS [21].

6.4 NoSQL

Bazy danych typu NoSQL (ang. *Not Only SQL*) odrzucają praktykę składowania danych w tabelarycznej, opartej na relacjach, ustrukturyzowanej formie będącej podstawą relacyjnych baz danych. Zamiast tego składują dane w innych, bardziej elastycznych postaciach, na przykład grafu lub zbioru par klucz-wartość. Pozwala to na wykonywanie niektórych operacji szybciej, niż są one wykonywane przez relacyjne bazy danych. Nie jest to jednak rozwiązanie bezwzględnie lepsze od relacyjnych baz danych - niektóre operacje są wykonywane wolniej w bazach typu NoSQL, niż w bazach relacyjnych.

Bazy danych typu NoSQL są nastawione na przechowywanie bardzo dużych ilości danych. Tym, co odróżnia je od baz relacyjnych jest fakt, że zazwyczaj domyślnie wspierają one rozproszenie danych pomiędzy wiele maszyn. Umożliwia to łatwą poziomą skalowalność systemów z nich korzystających.

6.4.1 Dostępne bazy NoSQL

Bazy danych typu NoSQL są tworzone pod kątem konkretnych problemów, jakie mają rozwiązywać. Dlatego istnieje wiele takich baz, które różnią się wydajnością bądź oferowanymi funkcjonalnościami. Można je podzielić ze względu na model przechowywania danych. Najczęściej stosowane modele to:

Zbiór par klucz-wartość

Para klucz-wartość (ang. *key-value pair*) jest zdecydowanie najprostszą strukturą danych używaną w bazach danych typu NoSQL. Przechowywane w postaci zbiorów takich par dane nie są w żaden sposób ustrukturyzowane. Pary klucz-wartość są podstawą dla innych modeli danych, na przykład rodzin kolumn, gdzie wiersze składają się z par kolumna-wartość. Są więc najbardziej elastycznym modelem, ale zazwyczaj oferującym ubogi zestaw funkcjonalności.

Przykłady baz przechowujących pary klucz-wartość to:

- DynamoDB[2] - baza danych będąca w stanie wydajnie działać pod bardzo dużym obciążeniem, jej kluczowymi celami są wydajność i skalowalność; stworzona w technologii Java;
- Azure Table Storage[3] - baza danych wspierająca przechowywanie danych w tabelarycznej, ustrukturyzowanej formie; stworzona na platformie .Net;
- Redis[4] - napisana w języku C wydajna baza danych wspierająca przechowywanie złożonych struktur takich jak listy, kolejki czy posortowane zbiory.

[2]DynamoDB - strona projektu: http://aws.amazon.com/dynamodb/
[3]Azure Table Storage - strona projektu: http://msdn.microsoft.com/en-us/library/dd179423.aspx
[4]Redis - strona projektu: http://redis.io/

Rodzina kolumn

Rodzina kolumn (ang. *column family*, *wide column*) jest modelem najbardziej przypominającym tabele w relacyjnej bazie danych. Dane przechowywane są w wierszach, które składają się z kolumn wraz z przypisanymi im wartościami. Pojedyczny wiersz może składać się z bardzo dużej liczby kolumn. W odróżnieniu od relacyjnych baz danych, wiersz nie musi posiadać wszystkich kolumn wchodzących w skład rodziny i może posiadać kolumny nienależące do rodziny. Każda wartość przypisana do danej kolumny wchodzącej w skład rodziny musi jednak być określonego dla tej kolumny typu.

W odróżnieniu od baz relacyjnych, bazy oparte o rodziny kolumn nie wspierają relacji (kluczy obcych) pomiędzy rodzinami. Aby osiągnąć satyfakcjonującą wydajność przeszukiwania, przechowuje się w nich dane o wysokim stopniu denormalizacji. Przykłady baz danych przechowujących rodziny kolumn:

- Cassandra[5] - stworzona w technologii Java baza danych, której główne cele to skalowalność, wysoka wydajność i brak pojedynczego punktu awarii (ang. *Single Point of Failure*, *SPOF*);
- Amazon SimpleDB[6] - komercyjna baza danych nastawiona na prostotę użytkowania i zarządzania.
- Hadoop[7] - platforma stworzona w technologii Java, wykonująca obliczenia na dużych ilościach danych realizowane przez wiele rozproszonych aplikacji, służy przede wszstkim do przetwarzania danych, a nie do ich składowania;

Graf

Niektóre bazy danych typu NoSQL za model danych obierają graf. Takie bazy nazywamy grafowymi bazami danych (ang. *graph databases*).

Węzłem grafu może być dowolny obiekt (np. zbiór par klucz-wartość), a krawędź może łączyć dwa dowolne węzły. Pojedyncza krawędź reprezentuje dowolną relację pomiędzy węzłami. Model ten jest więc bardzo elastyczny.

Zaletą grafu jest to, że podobnie jak relacja (w relacyjnych bazach danych), jest on strukturą danych o silnym podłożu matematycznym. Istnieje więc wiele wydajnych algorytmów operujących na tej strukturze (np. algorytmy wyszukiwania ścieżek). Dzięki temu struktura ta bardzo dobrze nadaje się do modelowania wszelkiego rodzaju danych przestrzennych (np. map) lub innych sieci (np. zbiorów użytkowników portali społecznościowych, gdzie wierzchołek reprezentuje osobę, a krawędź - relację "zna").

Do grafowych baz danych należą:

- Neo4J[8] - najpopularniejsza grafowa baza danych, cechuje się wysoką wydajnością i skalowalnością; wspiera mechanizm transakcji; stworzona w technologii Java;

[5]Cassandra - strona projektu: https://cassandra.apache.org/
[6]Amazon SimpleDB - strona projektu: http://aws.amazon.com/simpledb/
[7]Hadoop - strona projektu: http://hadoop.apache.org/
[8]Neo4J - strona projektu: http://www.neo4j.org/

- TITAN[9] - baza danych wykorzystująca inne rozwiązania (np. bazę Cassandra) korzytając z wymiennych mechanizmów; stworzona w technologii Java;
- Trinity[10] - stworzona na platformie .Net baza danych wspierająca wygodne, deklaratywne modelowanie i przeszukiwanie danych.

Dokument

Dokument (ang. *document*) jest typem danych, który z założenia reprezentuje duże porcje danych, z których tylko fragmenty są udostępniane na potrzeby pojedynczych zapytań. Dokumenty mogą posiadać dowolną strukturę, na co pozwala format, w jakim są przechowywane (np. XML lub JSON).

Bazy danych przechowujące dokumenty (ang. *document-oriented databases*) są więc przystosowane do przechowywania i przeszukiwania dużych jednostek danych. Do baz tego rodzaju należą:

- MongoDB[11] - umożliwia wygodne przeszukiwanie danych i tworzenie indeksów na dowolnych ich atrybutach, wspiera automatyczne dzielenie danych pomiędzy wiele maszyn (ang. *sharding*); stworzona w języku C++;
- CouchDB[12] - stworzona w języku Erlang baza danych udostępniająca przechowywane dane poprzez protokół HTTP; nastawiona na wykorzystanie w aplikacjach webowych i mobilnych;
- RavenDB[13] - baza danych umożliwiająca wykonywanie dowolnych operacji (np. przebudowy struktury) bez potrzeby zatrzymywania korzystających z niej aplikacji i udostępniająca wygodny sposób przeszukiwania danych; oparta na platformie .Net.

6.4.2 Wybrana baza typu NoSQL

Spośród dostępnych baz danych typu NoSQL należy wybrać jedną, która będzie używana przez aplikacje generowane przez narzędzie. Wydaje się, że z mechanizmem generacji dziedziny aplikacji najlepiej komponują się bazy za model danych obierające rodziny kolumn. Model ten jest najbardziej zblizony do klasycznego modelu relacyjnego - dobrze nadaje się więc do przechowywania ustrukturyzowanej dziedziny aplikacji, a sama struktura bazy danych może być jednym z generowanych artefaktów.

Współpraca takiej bazy danych z architekturą CQRS i wzorcem Event Sourcing będzie wyglądać następująco:

- baza typu NoSQL będzie pełnić rolę modelu zapytań (komponent *Data Storage* na rysunku 6.3);

[9]TITAN - strona projektu: https://github.com/thinkaurelius/titan/wiki
[10]Trinity - strona projektu: http://research.microsoft.com/en-us/projects/trinity/
[11]MongoDB - strona projektu: http://www.mongodb.org/
[12]CouchDB - strona projektu: http://couchdb.apache.org/
[13]RavenDB - strona projektu: https://github.com/ravendb/ravendb

- model zapytań będzie w dużym stopniu zdenormalizowany, a więc przystosowany do przechowywania go w postaci rodzin kolumn;
- procedury obsługi zdarzeń będą modyfikować dane zawarte w tej bazie;
- wykorzystanie tej bazy nie będzie miało wpływu na model komend i dziennik zdarzeń - zdarzenia mogą, ale nie muszą być przechowywane bazie NoSQL.

Spośród baz danych opartych o model rodziny kolumn najbardziej popularna jest Cassandra. Przemawia za tym jej wysoka wydajność i skalowalność potwierdzona w badaniach [22]. Dlatego to właśnie z tej bazy korzystać będą generowane aplikacje.

6.5 Cassandra

Cassandra to baza danych o hierarchicznej architekturze i skomplikowanym modelu danych, a jednocześnie udostępniająca prosty i intuicyjny język przeszukiwania danych. Jej główne zalety to wysoka wydajność i odporność na awarie.

6.5.1 Architektura

Hierchia elementów architektury bazy Cassandra jest następująca:

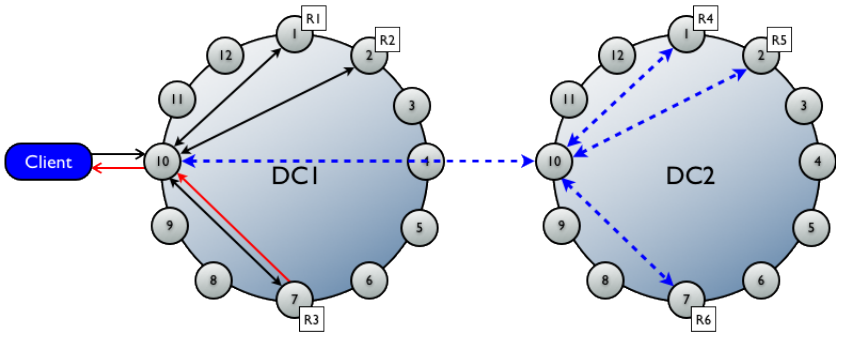
1. Podstawową jednostką architektury jest węzeł (ang. *node*) - to właśnie on przechowuje dane składowane w bazie. Zazwyczaj jednemu węzłowi odpowiada jedna maszyna (choć możliwe jest stworzenie wielu węzłów na pojedynczej maszynie).
2. Grupa węzłów tworzy centrum danych (ang. *data center*). Działa ono zazwyczaj jako grupa maszyn wirtualnych na pojedynczej maszynie fizycznej lub grupa powiązanych maszyn fizycznych działających w tej samej sieci.
3. Grupa centrów danych tworzy klaster (ang. *cluster*). Centra danych należące do klastra mogą być geograficznie odległe od siebie.

Rozproszenie danych pomiędzy węzłami ma na celu zminimalizowanie ryzyka utraty dużej ilości danych w razie awarii pojedynczej maszyny. Aby zminimalizować ilość danych traconych w razie awarii, dane mogą być replikowane pomiędzy węzłami. Jeśli jeden z węzłów utraci dane, zostaną one uzupełnione ich kopiami z innych węzłów. Co więcej, żaden z węzłów nie jest głównym właścicielem danej porcji danych. Sprawia to, że architektura jest wolna od pojedynczych punktów awarii.

Przykładowy klaster wykorzystujący replikację danych został przedstawiony na rysunku 6.4.

6.5.2 Model danych

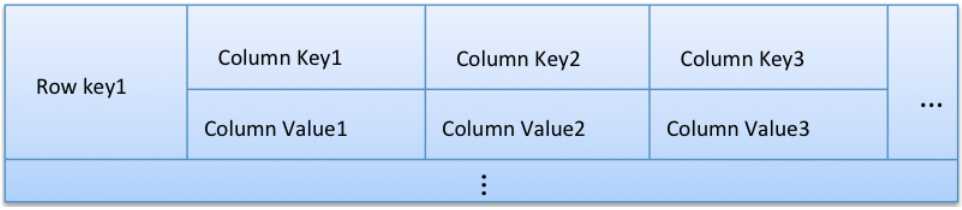
Cassandra przechowuje dane w postaci rodziny kolumn (patrz: sekcja 6.4.1) [24]. Każdy wiersz rodziny jest identyfikowany poprzez klucz główny (ang. *primary key*). W odróżnieniu od tabel w bazach relacyjnych, każda rodzina kolumn musi posiadać klucz główny.



Rysunek 6.4: Operacja zapisu pojedycznego wiersza danych do klastra składającego się dwóch centrów danych ($DC1$, $DC2$), z których każde składa się z dwunastu węzłow (1 - 12). Dane są replikowane w sześciu kopiach ($R1$ - $R6$) [23].

Należy podkreślić, że rodziny kolumn nie mogą posiadać znanych w bazach relacyjnych kluczy obcych (ang. *foreign key*) - kolumn wskazujących na inne rodziny.

Strukturę pojedynczego wiersza rodziny przedstawia rysunek 6.5.



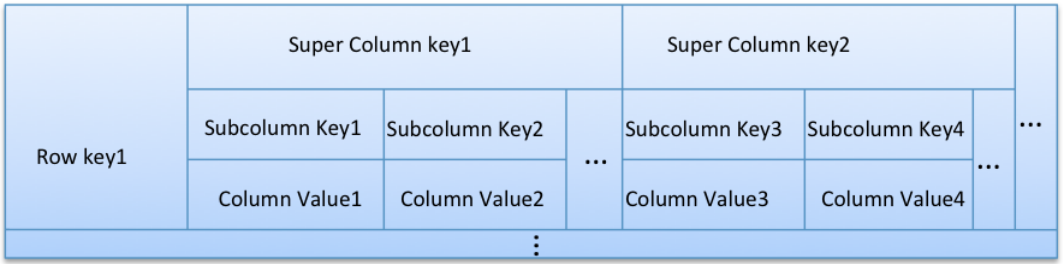
Rysunek 6.5: Schemat modelu danych przechowywanych przez bazę Cassandra (*Row key* - wartość klucza głównego) [24].

Wiersze należące do tej samej rodziny przechowywane są w kolejności posortowanej według wartości jej klucza głównego. Kolumny należące do wiersza sortowane są według ich nazw.

Superkolumny

Klucz główny rodziny może składać się z wielu kolumn. W takim przypadku pierwszą z kolumn wchodzących w skład klucza głównego nazywa się kluczem partycjonującym (ang. *partition key*). Pozostałe kolumny klucza głównego nazywa się superkolumnami (ang. *Super Column*). Strukturę wiersza należacego do rodziny kolumn posiadającej superkolumnę przedstawia rysunek 6.6.

Superkolumny, podobnie jak klucz partycjonujący sortowane są według ich wartości - różnica polega na tym, że klucze partycjonujące sortowane są w obrębie całej rodziny kolumn, a superkolumny - w obrębie pojedynczego wiersza.



Rysunek 6.6: Schemat rodziny kolumn posiadającej superkolumnę (*Super Column key* - wartość superkolumny) [24].

Wartości kolumn

Wartościami kolumn mogą być różnego rodzaju wartości skalarne znane z baz relacyjnych, takie jak napisy (*text*, *ascii*), liczby całkowite (*int*, *bigint*) lub zmiennoprzecinkowe (*float*, *double*), czy wartości logiczne (*boolean*).

Jednakże w odróżnieniu od baz relacyjnych, wartościami kolumn mogą być również kolekcje: lista (*list*), zbiór unikalnych elementów (*set*) i mapa par klucz-wartość (*map*). Kolekcje mają następujące ograniczenia [25]:

- maksymalny rozmiar pojedynczego elementu kolekcji wynosi 64 KB;
- kolekcje są odczytywane i zwracane w całości - dlatego nie powinny zawierać dużej liczby elementów.

Indeksy

Domyślnie rodzina kolumn może być przeszukiwana jedynie po wartościach jej klucza partycjonującego. Aby móc przeszukiwać wiersze rodziny po wartościach innych kolumn (w tym superkolumn), należy stworzyć na nich indeksy.

Indeks (ang. *index*) to, podobnie jak w relacyjych bazach danych, osobna (nie będąca częścią rodziny kolumn) struktura danych przechowująca posortowane wartości indeksowanych kolumn wraz ze wskazaniami na wiersze, do których te wartości należą.

Pojedynczy indeks przechowuje tylko wartości kolumn należących do wierszy, które znajdują się w jego węźle.

6.5.3 CQL

Do przeszukiwania i modyfikowania danych przechowywanych w bazie danych Cassandra służy język CQL (Cassandra Query Language). Przypomina on język SQL, jednak jest znacznie prostszy. Ograniczenia języka CQL względem SQL to na przykład [26]:

- brak podzapytań (ang. *subquery*);
- brak złączeń (ang. *join*);

- brak sortowania według wartości dowolnych kolumn - sortowanie jest możliwe tylko według wartości klucza partycjonującego (domyślnie) lub pierwszej superkolumny.

Niemniej jednak, język CQL umożliwia efektywne zarządzanie strukturą danych przechowywanych w bazie i przeszukiwanie tych danych. Przykłady komend i zapytań tego języka zostały przedstawione na rysunkach: 6.7, 6.8 i 6.9.

```
CREATE TABLE "User" (
 "UserName" text,
 "FirstName" text,
 "LastName" text,
 PRIMARY KEY ("UserName")
);

CREATE INDEX ON "User" ("LastName");
```

Rysunek 6.7: Komendy języka CQL tworzące rodzinę kolumn o nazwie *User* i kluczu głównym *UserName*, a także indeks na kolumnie *LastName* tej rodziny.

```
INSERT INTO "User" ("UserName", "FirstName", "LastName")
VALUES ('JanKowalski', 'Jan', 'Kowalski');
```

Rysunek 6.8: Komenda języka CQL rejestrująca użytkownika "JanKowalski".

```
SELECT * FROM "User"
WHERE "LastName" = 'Kowalski';
```

Rysunek 6.9: Komenda języka CQL wyszukująca użytkowników o nazwisku "Kowalski".

Rozdział 7

Implementacja generatora

Generator aplikacji konkretnego typu będzie korzystał z trzonu narzędzia w celu wygenerowania fragmentu aplikacji opartej o architekturę CQRS, korzystającej z wzorca Event Souring i z bazy danych Cassandra. Głównym celem generatora jest wyeliminowanie z systemu duplikacji opisu dziedziny aplikacji. Zostanie to osiągnięte przez umieszczenie pełnego opisu dziedziny w jednym miejscu i na jego podstawie generowanie innych artefaktów systemu, które duplikowałyby tę wiedzę.

W celu przedstawienia kolejnych kroków implementacji narzędzia, sformułowany zostanie przykład generowanej aplikacji.

7.1 Przykład generowanej aplikacji

Przykładowa aplikacja będzie serwisem internetowym, który umożliwia użytkownikom prowadzenie blogów. Użytkownik serwisu (encja *User*) będzie mógł:

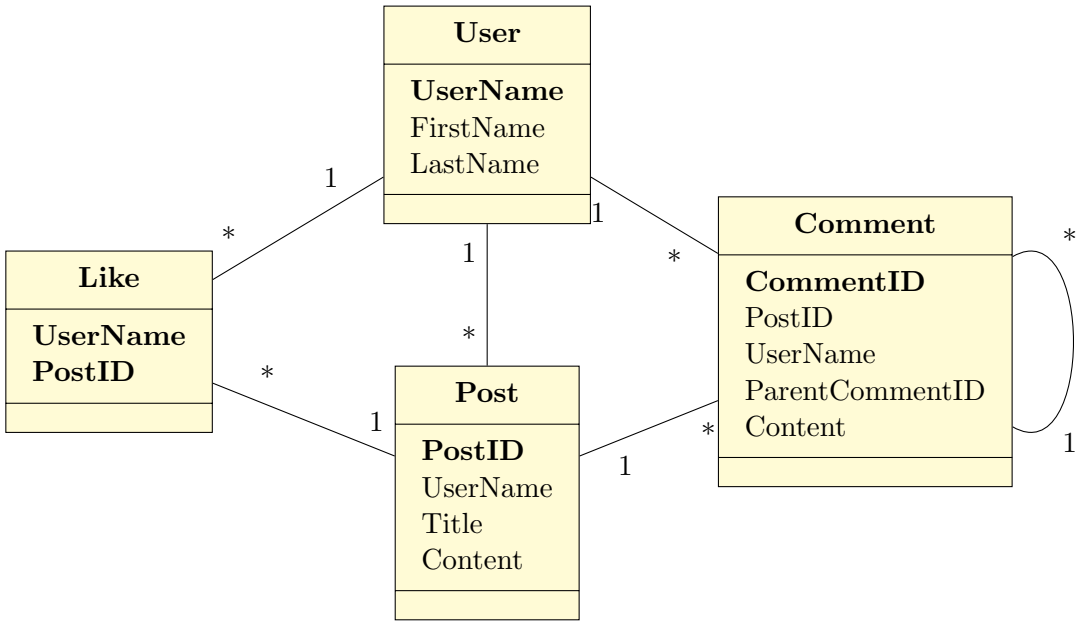
- opublikować wpis (encja *Post*) na swoim blogu (zdarzenie *PostPublishedEvent*);
- polubić (asocjacja *Like*) wpis innego użytkownika (zdarzenie *PostLikedEvent*);
- skomentować (encja *Comment*) wpis innego użytkownika (zdarzenie *PostCom* − *mentedEvent*);
- skomentować komentarz innego użytkownika (również zdarzenie *PostComment* − *edEvent*).

Encje serwisu zostały przedstawione na rysunku 7.1[1] - jest to model komend systemu. Natomiast zachodzące w systemie zdarzenia pokazano na rysunku 7.2.

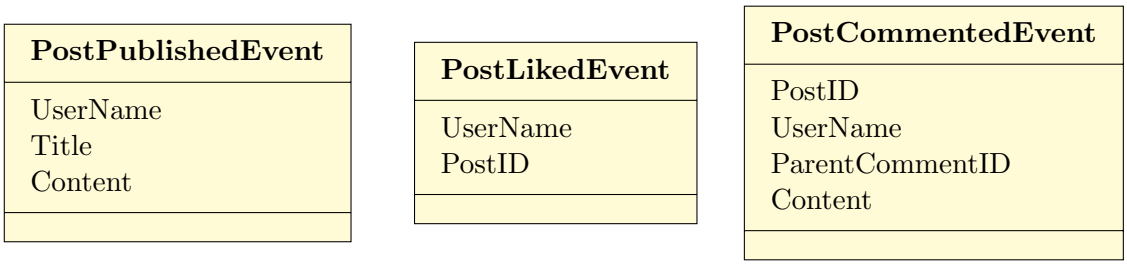
Na potrzeby architektury CQRS należy także zdefiniować model zapytań. Elementy wchodzące w skład tego modelu nazywane będą widokami. Widoki powinny udostępniać dane zdenormalizowane w takim stopniu, aby umożliwić jak najefektywniejsze wykonywane najczęstszych zapytań wykonywanych w systemie [24].

W omawianym przykładzie najczęściej wykonywane będą zapytania na potrzeby:

[1] Należy zaznaczyć, że na potrzeby przykładu dziedzina została w jak największym stopniu uproszczona (np. nie znalazła się w niej encja *Blog*, która mogłaby przechowywać nazwę, opis i logo bloga).



Rysunek 7.1: Encje należące do dziedziny przykładowej aplikacji. Pola wchodządze w skład kluczy głównych zostały pogrubione.



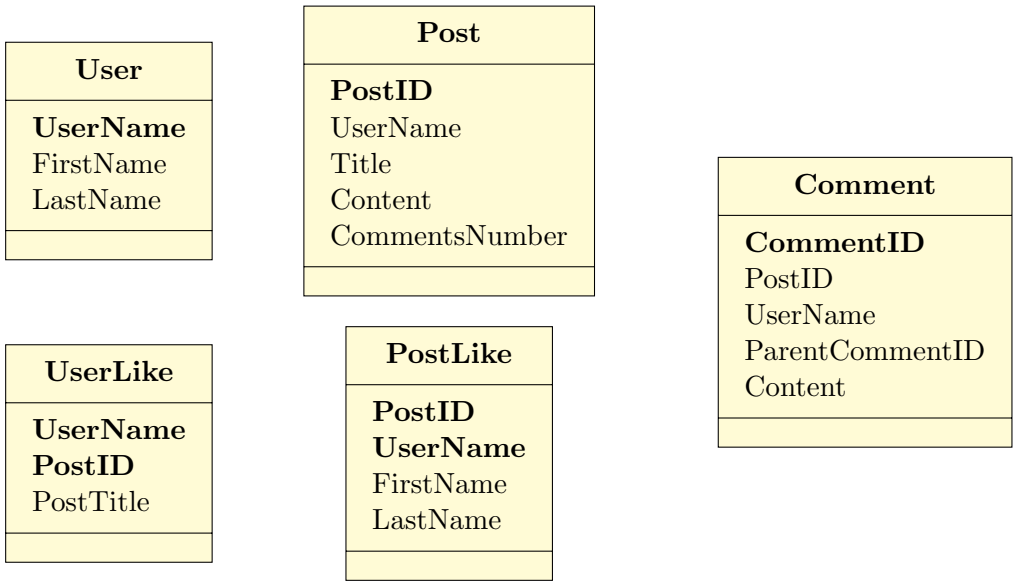
Rysunek 7.2: Zdarzenia występujące w przykładowej aplikacji.

- wyświetlenia danego bloga, tzn. wszystkich wpisów jego autora wraz z liczbą ich komentarzy - w tym celu wykonane zostanie pojedyncze odwołanie do widoku *Post*;
- wyświetlenia danego wpisu wraz z jego komentarzami i "polubieniami" - w tym celu wykonane zostaną odwołania do widoków: *Post*, *Comment* i *PostLike*;
- wyświetlenia profilu użytkownika wraz z lubianymi przez niego wpisami - w tym celu wykonane zostanie jedno odwołanie do widoku *User* i jedno do widoku *UserLike*.

Model zapytań został przedstawiony na rysunku 7.3.

Pomiędzy znormalizowanym modelem komend (rysunek 7.1), a zdenormalizowanym modelem zapytań występują dwie istotne różnice:

1. Encja *Like* została rozbita na dwia widoki: *UserLike* i *PostLike*. Przechowują one dane inne niż tylko identyfikatory powiązanych encji. Umożliwia to wykonywanie



Rysunek 7.3: Schemat modelu zapytań przykładowej aplikacji.

stałej (równej 2) liczby zapytań do bazy danych w celu wyświetlenia profilu użytkownika wraz z tytułami lubianych przez niego wpisów (lub wyświetlania wpisu wraz imionami i nazwiskami użytkowników go lubiących). Pozostawienie asocjacji *Like* w niezmienionym kształcie skutkowałoby potrzebą wykonania w tym celu $n+2$ zapytań, gdzie n to liczba wpisów lubianych przez danego użytkownika (lub liczba użytkowników lubiących dany wpis). Brak operacji złączenia w języku CQL uniemożliwia wydobycie wszystkich potrzebnych danych w jednym zapytaniu.

2. Encja *Post* otrzymała pole *CommentsNumber*. Umożliwia to wyświetlenie wpisów wraz z liczbą ich komentarzy bez potrzeby odwoływania się do widoku *Comment*. Używając języka SQL, aby uzyskać te dane przy użyciu pojedynczego zapytania, wystarczyłoby wykorzystać złączenie i funkcję agregującą *Count*. Jednakże język CQL nie umożliwia stworzenia takiego zapytania (a tym samym wymusza jak najmniejszy koszt zapytania).

7.2 Generowane artefakty

Artefaktami generowanymi na podstawie opisu dziedziny aplikacji będą:

- skrypty DDL definiujące strukturę bazy danych;
- klasy wchodzące w skład modelu zapytań (patrz: sekcja 6.2);
- klasy reprezentujące zdarzenia zachodzące w systemie (patrz: sekcja 6.3);
- klasy reprezentujące dane wyświetlane w interfejsie użytkownika (*view models*, patrz: sekcja 6.1.1);

- dokumentację systemu w formacie HTML zawierającą spis widoków należących do modelu zapytań;
- pomocnicze skrypty powłoki automatyzujące rutynowe czynności wykonywane przez programistów (patrz: sekcje 2.1.1, 2.4).

We wszystkich tych elementach zazwyczaj powielana jest wiedza na temat dziedziny aplikacji. Elementy mniej narażone na duplikację dziedziny, takie jak klasy realizujące logikę biznesową aplikacji, nie będą generowane przez narzędzie.

Należy zwrócić uwagę na fakt, że klasy reprezentujące poszczególne encje (patrz: rysunek 7.1) nie muszą być generowane. Wystarczy, że ich pola występują w modelu zapytań i w klasach zdarzeń.

7.3 Sposób definicji dziedziny

Zgodnie z założeniami dotyczącymi implementacji generatora, wszystkie artefakty wymienione w sekcji 7.2 powinny być generowane na podstawie pojedynczego opisu dziedziny aplikacji. Podczas implementacji tego aspektu pojawiają się następujące problemy:

7.3.1 Czy tworzyć schemat opisu dziedziny?

Problem ten poruszony został w sekcji 5.5. Stwierdzono, że słusznym wyborem jest stworzenie schematu opisu dziedziny aplikacji. Pozostaje jednak pytanie, gdzie należy ten schemat zdefiniować i przechowywać.

Nie ulega wątpliwości, że aby uniknąć duplikacji, schemat opisu dziedziny powinen być zdefiniowany tylko w jednym miejscu. Odpowiednim takim miejscem wyjdaje się być klasa należąca do kodu źródłowego generatora aplikacji CQRS. To właśnie tam schemat ten będzie używany. Gdyby był on przechowywany gdzie indziej, odpowiadająca mu klasa i tak musiałaby być stworzona lub wygenerowana - bez niej nie byłaby możliwa deserializacja opisu dziedziny do obiektu silnie typizowanego.

7.3.2 Schemat opisu dziedziny aplikacji

Najważniejszym aspektem określenia sposobu definicji dziedziny aplikacji jest zdefiniowanie schematu tej dziedziny. Schemat ten powinien pozwalać na zapisanie dowolnej dziedziny w wygody sposób, tak aby wprowadzanie modyfikacji dziedziny było jak najłatwiejsze.

Podejście naiwne

Najprostszym sposobem opisu dziedziny jest jawne wypisanie wszystkich elementów, które powinny być generowane przez narzędzie. Umożliwi to bardzo łatwą generację artefaktów systemu. Co więcej, sposób ten jest bardzo elastyczny - elementy dziedziny nie są ze sobą w żaden sposób powiązane.

Fragment opisu dziedziny przykładowej aplikacji wykorzystujący tę notację przedstawia rysunek 7.4.

```
{
 "Views": [
  {
   "Name": "User",
   "Fields": [
    { "Name": "UserName", "Type": "text", "IsKey": true },
    { "Name": "FirstName", "Type": "text" },
    { "Name": "LastName", "Type": "text" }
   ]
  },
  {
   "Name": "PostLike",
   "Fields": [
    ...
    { "Name": "UserName", "Type": "text", "IsKey": true },
    { "Name": "FirstName", "Type": "text" },
    { "Name": "LastName", "Type": "text" }
   ]
  },
  ...
 ],
 "Events": [
  {
   "Name": "PostPublishedEvent",
   "Fields": [
    { "Name": "UserName", "Type": "text" },
    ...
   ]
  },
  ...
 ]
}
```

Rysunek 7.4: Fragment opisu dziedziny aplikacji zawierający jawny opis wszystkich artefaktów systemu. *Views* - elementy modelu zapytań; *IsKey* - wartość oznaczająca, że dane pole należy do klucza głównego danego widoku; *Events* - klasy zdarzeń.

Takie podejście ma jednak istotną wadę - wprowadza szeroko występującą duplikację wiedzy. Przykładowo, wiedza na temat pól encji *User* jest zduplikowana w widokach *User* i *PostLike*. Oba widoki duplikują informacje o typach pól tej encji[2]. Wprowadza to kilka problemów:

- wzbogacenie pojedynczej encji o nowe pole (lub usunięcie istniejącego pola) wymaga aktualizacji opisu wszystkich widoków i zdarzeń, w których to pole będzie

[2]Powielenie nazw pól nie jest duplikacją (patrz: rozdział 2).

występować (lub występuje);

- zmiana typu pojedynczego pola wymaga aktualizacji opisu wszystkich widoków i zdarzeń, w których to pole występuje;

Jeśli opisy poszczególnych widoków i zdarzeń umieszczone będą w osobnych plikach, aktualizacje te będą tym bardziej uciążliwe.

Obranie pola jako podstawowego elementu opisu dziedziny

Aby rozwiązać problem duplikacji, można jako pojedynczy element dziedziny potraktować nie widok czy zdarzenie, a pojedyncze pole. Taki opis zawierałby informację o każdym z pól dziedziny systemu wraz z informacją o widokach i zdarzeniach, do których dane pole należy.

Fragment opisu dziedziny przykładowej aplikacji wykorzystujący tę notację przedstawia rysunek 7.5.

To rozwiązanie wyeliminowało duplikację z opisu dziedziny aplikacji, rozwiązując wszystkie problemy powodowane przez podejście opisane wyżej. Jednakże i ono posiada pewne wady:

- zarządzanie taką listą pól może być utrudnione:
  - pola nie są w żaden sposób pogrupowane,
  - na liście mogą występować pola o tej samej nazwie (należące do innych encji);
- algorytm wyodrębnienia jednostek generacji na podstawie opisu dziedziny może być skomplikowany;
- opis nie zawiera żadnej informacji na temat encji występujących w systemie (patrz: problemy postawione na końcu sekcji 6.2).

Obranie encji jako podstawowego elementu opisu dziedziny

Kolejnym podejściem może być potraktowanie encji jako pojedynczego elementu dziedziny - pomimo faktu, że encje nie wystąpią w żadnym artefakcie systemu. Taki opis byłby podobny do poprzedniego rozwiązania, jednak pola byłyby pogrupowane według encji, do których należą.

Pełny opis pojedynczej encji wykorzystujący tę notację przedstawia rysunek 7.6.

Rozwiązanie to jest najlepsze spośród omówionych:

- nie występuje w nim duplikacja;
- występuje w nim hierarchiczna organizacja:
  - pola są pogrupowane,
  - występowanie wielu pól o tej samej nazwie nie jest problemem;
- rozwiązuje ono oba problemy postawione na końcu sekcji 6.2:

```
[
 {
  "Name": "UserName",
  "Type": "text",
  "PresentInViews": [
   { "Name": "User", "IsKey": true },
   { "Name": "PostLike", "IsKey": true },
   { "Name": "Post" },
   ...
  ],
  "PresentInEvents": [
   "PostPublishedEvent",
   "PostLikedEvent",
   "PostCommentedEvent"
  ]
 },
 {
  "Name": "FirstName",
  "Type": "text",
  "PresentInViews": [
   { "Name": "User" },
   { "Name": "PostLike" }
  ]
 },
 ...
]
```

Rysunek 7.5: Fragment opisu, w którym za pojedyczy element dziedziny przyjęto pole. $PresentInViews$ - informacja o widokach, w których pole występuje; $PresentInEvents$ - informacja o zdarzeniach, w których pole występuje.

- wiedza na temat modelu komend (równoważnego modelowi encji) znalazła się w opisie dziedziny,
- na "pojedynczą, jednoznaczną i autorytatywną reprezentację wiedzy o dziedzinie aplikacji" wybrany został model komend.

Jedyną wadą tego podejścia w stosunku do rozwiązania opisanego jako pierwsze jest to, że algorytm wyodrębnienia jednostek generacji na podstawie takiego opisu dziedziny może być skomplikowany. Aby skonstruować pojedynczy widok lub zdarzenie, będzie on musiał odwiedzić opis każdej encji i przeanalizować zawartość jego listy $PresentInViews$ lub $PresentInEvents$ (patrz: sekcja 7.2). Wada ta jednak nie przesłania zalet tego rozwiązania.

```
{
 "Name": "User",
 "IsView": true
 "Fields": [
  {
   "Name": "UserName",
   "Type": "text",
   "IsKey": true,
   "PresentInViews": [
    { "Name": "PostLike", "IsKey": true },
    { "Name": "UserLike", "IsKey": true },
    { "Name": "Post", "IsSearchable": true },
    { "Name": "Comment" }
   ],
   "PresentInEvents": [
    "PostPublishedEvent",
    "PostLikedEvent",
    "PostCommentedEvent"
   ]
  },
  {
   "Name": "FirstName",
   "Type": "text",
   "PresentInViews": [ { "Name": "UserLike" } ]
  },
  {
   "Name": "LastName",
   "Type": "text",
   "PresentInViews": [ { "Name": "UserLike" } ]
  }
 ]
}
```

Rysunek 7.6: Pełny opis encji $User$ w notacji, w której za pojedyczny element dziedziny przyjęto encję. $IsView$ - wartość oznaczająca, że wszystkie pola encji obecne są w widoku $User$ (likwiduje to potrzebę wymieniania tego widoku w węźle $PresentInViews$ każdego pola); $IsSearchable$ - wartość oznaczająca, że dla danego pola powinien istnieć w danym widoku indeks.

7.3.3 Sposób definicji typów pól

Kolejnym problemem dotyczącym definicji dziedziny aplikacji jest sposób określania typów pól należących do encji. Bierze się on stąd, że w różnych językach występują różne typy danych, a te same typy mogą posiadać różne nazwy. Przykładowo, typ reprezentujący ciąg znaków w języku CQL to *text*, a w języku C# - *string*.

Możliwe rozwiązania

W przypadku omawianego generatora, dostępne są cztery możliwości rozwiązania tego problemu:

- używanie w opisie dziedziny jednie typów zaczerpniętych z języka CQL:
  - uprościłoby to generowanie definicji rodzin kolumn bazy danych - typy kolumn mogą zostać pobrane bezpośrednio z opisu pól encji,
  - skutkowałoby to wymogiem mapowania typów języka CQL na typy języka C# podczas generacji klas;
- używanie w opisie dziedziny jednie typów zaczerpniętych z języka C#:
  - uprościłoby to generowanie klas,
  - wymagałoby to mapowania typów języka C# na typy języka CQL podczas generacji definicji rodzin kolumn;
- używanie własnych nazw typów - byłoby to podejście uniwersalne, niezależne od języków wykorzystywanych w systemie, ale wymagające mapowania typów podczas generacji każdego z artefaktów;
- uwzględnianie obu typów w opisie każdego pola (np. jako osobne wartości: $CQLType$ i $CSharpType$):
  - generowanie zarówno klas, jak i definicji rodzin kolumn byłoby ułatwione,
  - konieczność definiowania dwóch typów dla każdego pola mogłaby być uciążliwa.

Wybór rozwiązania

Pomiędzy typami języka C# i typami języka CQL występuje korzystna zależność: każdemu typowi języka CQL odpowiada pojedynczy typ języka C#. Przykładowo, typom *ascii*, *text* i *varchar* języka CQL odpowiada wspomniany już typ *string* języka C#. Jak widać, zależność ta nie działa w obie strony.

Odpowiednim rozwiązaniem jest więc definiowanie typów pól encji przy użyciu typów języka CQL. Dlatego w dotychczas prezentowanych przykładach wszystkie pola encji, widoków i zdarzeń posiadały typy języka CQL.

Należy zaznaczyć, że gdyby generowane aplikacje korzystały z innej bazy danych czy innego języka programowania, lub gdyby korzystały z więcej niż dwóch języków (np. przechowywały model komend w bazie danych SQL), to wybrane rozwiązanie mogłoby być niewystarczające. Można by wtedy skorzystać z ostatniego z wymienionych rozwiązań, jednak w przypadku konieczności definiowania trzech typów dla każdego pola byłoby ono zbyt uciążliwe. Należałoby wtedy stosować własne nazwy typów (trzecie rozwiązanie) i zaimplementować mapowanie ich na typy zdefiniowane w każdym z używanych języków.

7.3.4 Organizacja plików

Ostatnim z problemów, jakie należy rozwiązać ustalając sposób definicji dziedziny aplikacji, jest organizacja plików składających się na jej opis (patrz: sekcja 5.3).

Jedną z wymienionych zalet wybranego schematu definicji dziedziny jest fakt, że jest ona hierarchiczna - w dziedzinie występują encje, a każda encja składa się z pól. Pozwala to na wybór spomiędzy trzech organizacji plików opisu dziedziny:

- umieszczenie pełnego opisu dziedziny w pojedynczym pliku;
- umieszczenie opisu każdej encji w osobnym pliku (rysunkek 7.7);
- umieszczenie opisu każdego pola w osobnym pliku (rysunkek 7.8).

```
Domain
├── Domain.json
└── Entities
    ├── Comment.json
    ├── Post.json
    └── User.json
```

Rysunek 7.7: Organizacja, w której opis każdej encji znajduje się w osobnym pliku.

```
Domain
├── Domain.json
└── Entities
    ├── Comment.json
    ├── Comment_Fields
    │   ├── CommentId.json
    │   ├── Content.json
    │   └── ...
    ├── Post.json
    ├── Post_Fields
    │   ├── Content.json
    │   ├── PostID.json
    │   └── ...
    └── ...
```

Rysunek 7.8: Organizacja, w której opis każdego pola znajduje się w osobnym pliku.

Ostatni ze sposobów pozwala na uporządkowanie elementów skomplikowanych dziedzin, w których występują encje posiadające wiele pól, a opis każdego pola jest rozbudowany (oprócz typu pola i list miejsc, w których występuje, zawiera wiele dodatkowych wartości).

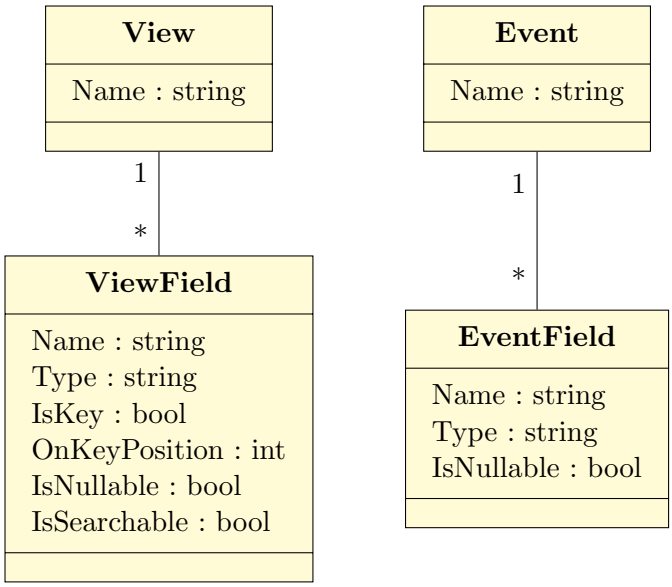
Dziedzina przykładowej aplikacji nie jest jednak skomplikowana. Mimo to, pojedynczy plik z jej opisem miałby ponad 100 linii tekstu. Dlatego najlepszy w tym przypadku jest wybór drugiej z wymienionych możliwości.

7.4 Wyodrębnienie jednostek generacji z opisu dziedziny aplikacji

Zdefiniowanie dziedziny aplikacji i zapisanie jej w odpowiedniej strukturze plików umożliwia wykonanie pierwszego i drugiego kroku działania generatora (patrz: sekcja 5.2). Kolejny krok wymaga implementacji algorytmu wyodrębnienia jednostek generacji z opisu dziedziny (patrz: sekcja 5.2.3).

Jednostki generacji

Wszystkie generowane artefakty systemu (patrz: skecja 7.2) generowane będą na podstawie obiektu (lub kolekcji obiektów) jednej z dwóch jednostek generacji: widoku lub zdarzenia. Klasy jednostek generacji zostały przedstawione na rysunku 7.9.



Rysunek 7.9: Klasy jednostek generacji. $OnKeyPosition$ - określa pozycję pola w kluczu głównym rodziny kolumn (dotyczy złożonych kluczy głównych); $IsNullable$ - określa, czy pole może przyjąć wartość *null* (dotyczy typów prostych, takich jak *int*).

Algorytm

Algorytm wyodrębnienia jednostek generacji z opisu dziedziny przebiega następująco:

1. Stwórz pustą listę widoków V i pustą listę zdarzeń E.
2. Dla każdego pola f każdej encji e należącej do dziedziny:
   (a) Jeśli $e.IsView$:
      i. Jeśli V nie zawiera widoku o nazwie $e.Name$, stwórz go i dodaj do V.

ii. Na podstawie f stwórz obiekt $ViewField$ i dodaj go do widoku o nazwie $e.Name$.

(b) Dla każdego widoku v z $f.PresentInViews$:

i. Jeśli V nie zawiera widoku o nazwie $v.Name$, stwórz go i dodaj do V.

ii. Na podstawie f stwórz obiekt $ViewField$ i dodaj go do widoku o nazwie $v.Name$.

(c) Dla każdej nazwy zdarzenia ev z $f.PresentInEvents$:

i. Jeśli E nie zawiera zdarzenia o nazwie ev, stwórz je i dodaj do E.

ii. Na podstawie f stwórz obiekt $EventField$ i dodaj go do zdarzenia o nazwie ev.

3. Zwróć V i E.

Algorytm zapewnia kilka ważnych funkcjonalności:

- w pojedynczym widoku lub zdarzeniu mogą znaleźć się pola należące do kilku różnych encji;
- dla danego widoku nie musi istnieć odpowiadająca mu encja; na przykład nie istnieją encje $PostLike$ i $UserLike$ - widoki te występują jedynie w listach $PresentInViews$ pól innych encji;
- dla danej encji nie musi istnieć odpowiadający jej widok; może nie istnieć widok, który zawiera wszystkie jej pola.

7.5 Szablony generacji

Aby umożliwić wykonanie ostatniego kroku generacji, należy dostarczyć generatorowi szablony generacji. Na podstawie jednostek generacji wyodrębnionych z opisu dziedziny aplikacji wygenerują one artefakty systemu.

Zgodnie z listą generowanych artefaktów podaną w sekcji 7.2, stworzono następujące szablony:

- szablon generujący definicję rodziny kolumn bazy danych Cassandra na podstawie widoku (rysunek 7.10);
- szablon generujący klasę widoku na podstawie widoku (rysunek 7.11);
- szablon generujący klasę zdarzenia na podstawie zdarzenia;
- szablon generujący model widoku (*view model*) na podstawie widoku;
- szablon generujący plik HTML na podstawie zbioru widoków (rysunek 7.12);
- szablony generujące skrypty automatyzujące czynności wykonywane przez programistów:

- szablon generujący skrypt tworzący rodziny kolumn w bazie danych na podstawie zbioru wszystkich widoków dziedziny (rysunek 7.13);
- szablon generujący skrypt pobierający wszystkie dane przechowywane w danej rodzinie kolumn na podstawie widoku odpowiadającego tej rodzinie.

Szablon:

```
<#@ template language="C#" #>
<#@ parameter type="Schema.Model.View" name="Metadata" #>
CREATE TABLE "<#= Metadata.Name #>" (
<# foreach (var field in Metadata.Fields) { #>
 "<#= field.Name #>" <#= field.Type #>,
<# } #>
 <#= CqlHelper.FormatPrimaryKey(Metadata) #>
);

<# foreach (var field in Metadata.Fields.Where(x => x.IsSearchable)) { #>
CREATE INDEX ON "<#= Metadata.Name #>" ("<#= field.Name #>");
<# } #>
```

Wynik dla widoku *Post*:

```
CREATE TABLE "Post" (
 "PostID" timeuuid,
 "Title" text,
 "Content" text,
 "CommentsNumber" int,
 "UserName" text,
 PRIMARY KEY ("PostID")
);

CREATE INDEX ON "Post" ("UserName");
```

Rysunek 7.10: Szablon generacji definicji rodzin kolumn i przykładowy wynik jego działania. *FormatPrimaryKey* - metoda formatująca kolumny klucza głównego, mająca zastosowanie głównie dla kluczy złożonych.

7.6 Podsumowanie

Wszystkie kroki generacji aplikacji zostały zaimplementowane. Dzięki temu generacji podlega znaczna część artefaktów systemu, które korzystają z wiedzy dotyczącej jego dziedziny.

Należy zaznaczyć, że osiągnięta implementacja może posłużyć do wygenerowania aplikacji CQRS o dziedzinach innych niż podana w przykładzie. Co więcej, schemat opisu

Szablon:

```
<#@ template language="C#" #>
<#@ parameter type="View" name="Metadata" #>
public class <#= Metadata.Name #> : IView
{
<# foreach (var field in Metadata.Fields) { #>
 public <#= TypesMap.GetDotNetType(field.Type, field.IsNullable) #>
  <#= field.Name #> { get; set; }
<# } #>
}
```

Wynik dla widoku *Post*:

```
public class Post : IView
{
 public Guid PostID { get; set; }
 public string Title { get; set; }
 public string Content { get; set; }
 public int CommentsNumber { get; set; }
 public string UserName { get; set; }
}
```

Rysunek 7.11: Szablon generacji klas i przykładowy wynik jego działania. $GetDotNetType$ - mapuje typ języka CQL na typ języka C# (patrz: sekcja 7.3.3).

dziedziny aplikacji opracowany w sekcji 7.3 może być wykorzystany do generowania aplikacji innego rodzaju, np. aplikacji o klasycznej architekturze trójwarstwowej, korzystającej z bazy danych SQL.

Ogólny schemat postępowania podczas konstruowania generatora dowolnej aplikacji jest następujący:

1. Zdefiniowanie encji wchodzących w skład dziedziny.
2. Zapisanie encji i ich pól w wybranym formacie (tu: JSON).
3. Określenie artefaktów systemu, w których powielana jest wiedza na temat encji (tu: struktura bazy danych, model zapytań, zdarzenia, modele widoków).
4. Zidentyfikowanie artefaktów równoważnych pod względem wykorzystywanej wiedzy i wyznaczenie jednostek generacji, na podstawie których generowane będą te artefakty (tu: widoki, zdarzenia).
5. Dla każdego pola każdej encji określenie, w których jednostkach generacji występować będzie to pole.
6. Uzupełnienie opisu pól każdej z encji o listy jednostek generacji, w których będzie występować dane pole (tu: $PresentInViews$, $PresentInEvents$).

Szablon:

```
<#@ template language="C#" #>
<#@ parameter type="View[]" name="Metadata" #>
<html>
<# foreach (var entity in Metadata) { #>
 <div>
  <#= entity.Name #>:
  <ul>
<# foreach (var field in entity.Fields) { #>
   <li><#= field.Name #> (<#= field.Type #>)</li>
<# } #>
  </ul>
 </div>
<# } #>
</html>
```

Wynik:

```
<html>
 <div>
  Post:
  <ul>
   <li>PostID (timeuuid)</li>
   <li>Title (text)</li>
   ...
  </ul>
 </div>
 ...
</html>
```

Rysunek 7.12: Szablon generacji dokumentacji i fragment wyniku jego działania.

7. Implementacja algorytmu wyodrębnienia jednostek generacji z opisu dziedziny.
8. Stworzenie szablonów generacji każdego z artefaktów systemu określonych w punkcie 3.
9. Określenie organizacji plików wyjściowych generatora (krok ten jest trywialny, nie został opisany w niniejszym rozdziale).

Szablon:

```
<#@ template language="C#" #>
<#@ parameter type="View[]" name="Metadata" #>
<# foreach (var view in Metadata) { #>
SOURCE 'column_families/<#= view.Name #>.cql';
<# } #>
```

Wynik:

```
SOURCE 'column_families/Comment.cql';
SOURCE 'column_families/Post.cql';
SOURCE 'column_families/User.cql';
```

Rysunek 7.13: Szablon generacji skryptu tworzącego rodziny kolumn w bazie danych i wynik jego działania. Szablon zakłada, że wyniki działania szablonu 7.10 znajdują się w katalogu *column_families* w plikach o nazwach odpowiadających nazwom widoków.

Rozdział 8

DSL

- że ma być czytelne dla człowieka - że jak najbardziej business-friendly - że wynik taki sam jak dla opisu w poprzednim rozdziale - różnica tylko w opisie dziedziny i algorytmie wyodrębniania jednostek generacji (szablony i wszystko inne takie samo)

8.1 Składnia

- że typy własne (id zmiast timeuuid), żeby było zrozumiałe

8.1.1 Encje i ich pola

- nazwy encji poprzedzone są znakiem małpy:
  - przykład: `@User`;
- nazwy pól wchodzących w skład klucza encji wymienione są w jej identyfikatorze, w nawiasach okrągłych bezpośrednio po nazwie encji:
  - przykład: `@User(UserName)`,
  - nazwy pól wchodzących w skład klucza złożonego oddzielone są przecinkami (bez spacji),
  - klucz encji wystarczy zdefiniować w jednym (dowolnym) wystąpieniu jej identyfikatora,
  - w opisie dziedziny nie mogą wystąpić dwie różne definicje klucza tej samej encji,
  - definicja klucza encji musi wystąpić przy co najmniej jednym wystąpieniu jej identyfikatora;
- pola należące do encji występują po jej identyfikatorze, oddzielone od niego znakiem dwukropka:
  - przykład: `@User:FirstName`;
- typ pola podany jest nawiasach kwadratowych bezpośrednio po nazwie pola

  - przykłady: `@Post(PostID[id])`, `@Post:CommentsNumber[int]`,
  - typ pola wystarczy zdefiniować przy jednym (dowolnym) wystąpieniu jego nazwy,
  - w opisie dziedziny nie mogą wystąpić dwie różne definicje typu tego samego pola,
  - polom, których definicja typu nie występuje ani razu, nadawany jest typ *text*;
- relację "wiele do jednego" pomiędzy encjami oznacza się znakiem dwukropka, oddzielającym identyfikatory tych encji:
  - przykład: `@Post:@User:FirstName`,
  - znaczenie: *z jedną instancją encji* `@User` *powiązanych jest wiele instancji encji* `@Post`;
- relację "jeden do wielu" pomiędzy encjami oznacza się dwoma znakami dwukropka, oddzielającymi identyfikatory tych encji:
  - przykład: `@Post::@Comment:Content`,
  - znaczenie: *z jedną instancją encji* `@Post` *powiązanych jest wiele instancji encji* `@Comment`.

8.1.2 Zdarzenia

- sekcja definicji zdarzeń następuje w opisie dziedziny po linii o treści "`Events:`";
- definicja pojedynczego zdarzenia składa się z dwóch części:
  - nazwy zdarzenia,
  - opisu zdarzenia, zawierającego identyfikatory i pola encji wchodzących w jego skład;
- przykład: `PostPublished: @Post with @Post:Content can be published`;
- definicje kolejnych zdarzeń są od siebie oddzielone pustymi liniami.

Pojedyncza definicja zdarzenia jednoznacznie określa pojedyncze zdarzenie wchodzące w skład modelu komend systemu.

8.1.3 Widoki

- sekcja definicji widoków następuje w opisie dziedziny po linii o treści "`Views:`";
- definicja widoku to pojedyncze zdanie zawierające identyfikatory i pola encji w nim występujących;
- przykład: `I can display @Post with @Post:Title and @Post:Content`;
- definicje kolejnych widoków są od siebie oddzielone pustymi liniami.

Pojedyncza definicja widoku nie określa jednoznacznie pojedynczego widoku wchodzącego w skład modelu zapytań systemu. Określa natomiast treść wyświetlaną na pojedynczym ekranie aplikajcji, co z biznesowego punktu widzenia jest bardziej intuicyjne.

Model zapytań będzie więc generowany stopniowo na podstawie analizy wszystkich definicji widoków zawartych w opisie dziedziny. Po analizie pojedynczej definicji widoku będzie następować aktualizacja całej definicji modelu zapytań.

8.2 Przykład

(patrz: 7.1)
przedstawia rysunek 8.1

```
Events:

PostPublished:
@User(UserName) of @User:UserName can publish a @Post(PostID[id]) with
@Post:Title and @Post:Content.

PostLiked:
@User of @User:UserName can like a @Post by its @Post:PostID.

PostCommented:
@User of @User:UserName can write a @Comment(CommentID[id]) of
@Comment:Content under a post of ID @Comment:PostID or another comment
of ID @Comment:ParentCommentID[id].

========

Views:

I can display @Post list with @Post:@User:UserName, @Post:Title,
@Post:Content and @Post:CommentsNumber[int].

I can display @Post details with @Post:@User:UserName, @Post:Title,
@Post:Content, @Post::@Comment:Content, @Post::@Comment:@User:UserName,
@Post::@User:FirstName and @Post::@User:LastName.

I can display user profile with @User:UserName, @User:FirstName,
@User:LastName and @User::@Post:Title.
```

Rysunek 8.1: Pełny opis dziedziny systemu w języku DSL.

Należy zwrócić uwagę na czytelność i zwięzłość tego opisu. Żaden z przedstawionych wcześniej sposobów opisu dziedziny nie mógł pokazany w całości na jednej stronie, podczas

gdy powyższy opis mieści się bez problemu.

8.3 Algorytm

1. Stwórz pustą listę widoków V i pustą listę zdarzeń E.
2. Dla każdej definicji zdarzenia zawartej w sekcji definicji zdarzeń:
   (a) todo
3. Dla każdej definicji widoku zawartej w sekcji definicji widoków:
   (a) todo

8.4 Kroki postępowania

8.5 Podsumowanie

- że wystarczający na potrzeby tego systemu - że gdyby pojedyncze pole miało więcej właściwości, to lipa - że bardzo business-friendly

Rozdział 9

Ocena rozwiązania

- co dało się wygenerować, a czego nie
- jakiej duplikacji udało się uniknąć
- ile pracy wymagałoby dodanie nowej funkcjonalności (API, eksport do arkusza) - przykład?
- jaką inną aplikację można wygenerować (przykład)
- jak łatwo wprowadza się zmiany w dziedzinie aplikacji
  - problem: jak obsłużyć zmiany struktury bazy danych? (migracje)

Generacji podlega znaczna część systemu, w tym wszystkie pliki dotyczące bazy danych i dokumentacji. Ręcznie należy zaimplementować:

- dostęp do danych przechowywanych w bazie danych;
- procedury obsługi zdarzeń;
- mechanizm zgłaszania zdarzeń, zapisywania ich w dzienniku zdarzeń i wywoływania procedur ich obsługi;
- kod aplikacji webowej;
- kod HTML stron serwisu;

Rozdział 10

Obsługa zmiany dziedziny aplikacji

- generator potrafi zrobić snapshot dziedziny / jednostki generacji
- snapshot będzie porównywany z obecnym stanem (lub innym snapshotem)
- co serializować w snapshocie (czyli na podstawie czego migrować):
  - dziedzinę:
    - zalety:
      - odporne na zmiany schematu jednostek generacji
    - wady:
      - i tak raczej trzeba będzie generować jednostki generacji
      - do schematu dziedziny mogą tylko dochodzić nowe rzeczy (nic nie moze się zmienić / wypaść)
      - trzeba i tak tworzyć jednostki generacji
      - nie odporne na zmiany mechanizmu wyznaczania jednostek generacji na podstawie dziedziny (ten sam będzie używany do starych i nowych snapshotów)
  - jednostki generacji
    - zalety:
      - efekty migracji i tak dotyczyć będą jednostek generacji, a nie dziedziny
      - nie trzeba czytać dziedziny ze snapshota, tylko jednostki generacji (ale aktualną dziedzinę i tak trzeba czytać)
      - można robić snapshot tylko niektórych jednostek generacji (np. tylko Views)
      - odporne na zmiany schematu dziedziny
      - zmiana jednostek generacji może wynikać z poprawki błędu działania mechanizmu generacji jednostek generacji - w tym przypadku taka poprawka też będzie potraktowana jako migracja
    - wady:
      - do schematu jednostek generacji mogą tylko dochodzić nowe rzeczy (nic nie moze się zmienić / wypaść)

        - na podstawie snapshota nie da się łatwo odtworzyć dziedziny (ale czy jest to potrzebne?)

- implementacja dla:
    - migracje bazy danych (cql)
    - changelog w dokumentacji?

Rozdział 11

Podsumowanie

- Wnioski

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

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

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

[9] *World Wide Web Consortium (W3C)* [online]. W3C. http://www.w3.org/ [dostęp: lipiec 2014].

[10] *N-Tier Data Applications Overview* [online]. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/bb384398.aspx [dostęp: lipiec 2014].

[11] *Three tier architecture in asp.net* [online]. Dot Net Peoples. http://dotnetpeoples.blogspot.com/2011/04/three-tier-architecture-in-aspnet.html [dostęp: lipiec 2014].

[12] *Business object* [online]. Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Business_object [dostęp: lipiec 2014].

[13] *View model* [online]. Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/View_model [dostęp: lipiec 2014].

[14] *Data Transfer Object* [online]. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/ms978717.aspx [dostęp: lipiec 2014].

[15] *Exploring CQRS and Event Sourcing* Betts D., Domínguez J., Melnik G., Simonazzi F., Subramanian M. Microsoft. 2012. ISBN 978-1-62114-016-0.

[16] *CQRS* [online]. Fowler M. http://martinfowler.com/bliki/CQRS.html [dostęp: lipiec 2014].

[17] *Event Sourcing Pattern* [online]. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/dn589792.aspx [dostęp: lipiec 2014].

[18] *Introducing Event Sourcing* [online]. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/jj591559.aspx [dostęp: lipiec 2014].

[19] *Why do YouTube views freeze at 301?* [online]. Numberphile. YouTube. https://www.youtube.com/watch?v=oIkhgagvrjI [dostęp: lipiec 2014].

[20] *CQRS and Event Sourcing* [online]. Wordpress. http://cqrs.wordpress.com/documents/cqrs-and-event-sourcing-synergy/ [dostęp: lipiec 2014].

[21] *CQRS Info* [online]. CQRS Info. http://www.cqrsinfo.com/ [dostęp: lipiec 2014].

[22] *Solving Big Data Challenges for Enterprise Application Performance Management* [online]. Rabl T., Sadoghi M., Jacobsen H. http://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf [dostęp: lipiec 2014].

[23] *Multiple data center write requests* [online]. DataStax Cassandra 1.2 Documentation. http://www.datastax.com/documentation/cassandra/1.2/cassandra/architecture/architectureClie [dostęp: sierpień 2014].

[24] *Cassandra Data Modeling Best Practices, Part 1* [online]. Patel J. eBay tech blog. http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/ [dostęp: lipiec 2014].

[25] *Using collections* [online]. DataStax CQL 3.0 Documentation. http://www.datastax.com/documentation/cql/3.0/cql/cql_using/use_collections_c.html [dostęp: lipiec 2014].

[26] *What's New in CQL 3.0* [online]. Cannon P. DataStax Developer Blog. http://www.datastax.com/dev/blog/whats-new-in-cql-3-0 [dostęp: sierpień 2014].

Oświadczenie

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

.....

Michał Aniserowicz