POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2013/2014

PRACA DYPLOMOWA MAGISTERSKA

Michał Aniserowicz

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

Praca wykonana pod kierunkiem
dra inż. Jakuba Koperwasa

Ocena: .....

.....

Podpis Przewodniczącego Komisji
Egzaminu Dyplomowego

Spis treści

Rozdział 1

Wstęp

- Wstęp
- Duplikacja
  - Rodzaje duplikacji
  - Skutki wystąpienia duplikacji
  - Możliwe rozwiązania
- Sprecyzowanie problemu
  - że rozwiązanie będzie oparte o generację
  - że zajmę się głównie dziedziną aplikacji
- Generacja
  - kiedy to się opłaca, a kiedy może powodować problemy
  - przykłady konkretnych dużych generatorów
  - wniosek: żaden generator nie będzie jednocześnie bogaty w funkcjonalności i dobry dla każdego typu aplikacji
  - zatem: rozwięzanie podzielę na dwie części
    - trzon narzędzia do generacji
    - generator aplikacji jednego typu
- Założenia co do generatora aplikacji pojedynczego typu
  - wybrać typ aplikacji
  - dlaczego CQRS
  - bo zdenormalizowana dziedzina
  - gdzie w CQRS zdefiniowana jest dziedzina (encje) aplikacji? Model do odczytywania nie zawiera przecież encji, a tylko widoki.
  - model "read" i model "write" częściowo na siebie zachodzą (lub nawet "read" zawiera "write"). Jak uniknąć duplikacji metadanych?

  - opisać CQRS
  - że często idzie w parze z Event Sourcing
  - opisać Event Sourcing
  - że całość dobrze idzie w parze z NoSQL
  - opisać NoSQL
    * opisać rodzaje baz NoSQL
    * wybrać bazę NoSQL i dlaczego Cassandra
    * opisać Cassandrę

- Implementacja generatora aplikacji pojedynczego typu
  - Ogólnie nie chodzi o to, żeby w ogóle nie pisać nazw klas / właściwości - tylko o to, żeby nie trzeba było pamiętać o wszystkich miejscach, gdzie dana encja jest używana.
  - czy w ogóle generować schemat bazy danych (no schema) (?)
  - wybrać sposób definicji dedziny
  - że będzie schema
  - ale że schema powinna być w jednym miejsu - będzie jako klasa w kodzie
  - wybór formatu
  - uml, emf się nie nadają - operują na encjach, a nie na polach
  - xml albo json, bez różnicy - czytelniejszy i bardziej intuicyjny wydaje się JSON (xml ma atrybuty i węzły zagnieżdzone - nie wiadomo czego użyć)
  - zdefiniowanie schematu opisu dziedziny na potrzeby CQRS
    * PresentIn
  - Wybrór klienta Cassandry (?)
  - Sformułowanie przykładu aplikacji
  - Zapisanie schemy w JSONie
  - Implementacja szablonów
  - Implementacja kolejnych modułów aplikacji
    * DAL (repozytorium bazowe zahardkodowane, konkretnych nie da się wygenerować)
    * BLL (obsługa zdarzeń zahardkodowana, da się wygenerować EventHandlery)
    * WEB (Nancy, da się wygenerować ViewModele, formy)

- Ocena rozwiązania
  - Co dało się wygenerować, a co nie
  - Jakiej duplikacji udało się uniknąć

  - jak łatwo wprowadza się zmiany w aplikacji
  - ile pracy wymagałoby dodanie nowej funkcjonalności (API, eksport) - przykład?
  - jaką inną aplikację można wygenerować (przykład)

- Wnioski
- Koniec

Rozdział 2

Duplikacja

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

Celem stosowania tej 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 są jedynie odnośnikami do wiedzy reprezentowanej przez artefakty, które identyfikują.

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 uznając, że w danej sytuacji jest to nie do uniknięcia;
- duplikacja niezamierzona (ang. *inadvertent duplication*) - występuje, gry programista nie jest świadomy, że doprowadza do duplikacji;
- duplikacja niecierpliwa (ang. *impatient duplication*) - jest wynikiem niedbalstwa programisty; ma miejsce w sytuacji, gdy programista świadomie wybierze rozwiązanie najprostsze w danej sytuacji;
- duplikacja pomiędzy programistami (ang. *interdeveloper duplication*) - pojawia się, gdy kilku programistów tworzących tę samą aplikację wzajemnie duplikuje tworzony kod.

Duplikacja może występować w różnych postaciach:

2.1.1 Duplikacja czynności wykonywanych przez programistów

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

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

1. Usunięcie istniejącej bazy.
2. Stworzenie nowej bazy.
3. Zainicjalizowanie schematu nowej bazy przy pomocy skryptu DLL.
4. Wypełnienie nowej bazy danymi przy pomocy skryptu DML.

Jeśli wszystkie z tych kroków wykonywane są ręcznie, 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 wcielać ją w życie krok po kroku.

2.1.2 Duplikacja kodu źródłowego aplikacji

Programistom wielokrotnie 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 iterurajce 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 iterurajcej 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.

2.1.3 Duplikacja opisu dziedziny aplikacji

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

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

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

W miarę rozrastania się systemu i pojawiania się kolejnych funkcjonalności i modułów, występuje tendencja do powielania całości bądź części definicji dziedziny w różnych modułach. Dotyczy to modułów, których zadaniem jest obróbka tych samych danych, ale w różny sposób. Przykładowo, aplikacja może udostępniać przechowywane dane na następujące sposoby:

- wyświetlać je na stronie WWW,
- wystawiać jako API,
- eksportować do arkusza kalkulacyjnego.

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

W takim przypadku moduł odpowiedzialny za API operuje na pełnej dziedzinie, a moduły strony WWW i eksportu do arkusza kalkulacyjnego - na jej fragmentach. Jeśli każdy z tych modułów 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 moduły 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 wykonywanie 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 zaakceptowane 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 niepożadanych zjawisk i pogorszenia jakości całego systemu [2].

2.3 Rozwiązanie: zapisywanie wszystkich powiązanych i zduplikowanych miejsc

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

Takie podejście ma jednak istotne wady:

- rozwiązanie to nie powstrzymuje programistów przed wprowadzaniem kolejnych duplikacji, a wręcz na to przyzwala;
- istnienie takiej listy samo w sobie jest duplikacją - pojawienie się nowego miejsca występowania pól encji (np. alternatywnej implementacji modelu danych) wymaga zaktualizowania listy;
- 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 na wszystkie 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 [5].

Zalety:

- 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,
- 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 rodzaje 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.

2.5.3 Refleksja

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

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

2.6 Rozwiązanie: użycie generatorów

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

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

2.6.1 Generator modelu dziedziny aplikacji na podstawie schematu bazy danych

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

Przykładami narzędzi umożliwiających taką generację kodu są:

- NHibernate [6] - narzędzie ORM (ang. *Object-Relational Mapping*) przeznaczone na platformę .Net;
- EntityFramework [7] - narzędzie ORM dla platformy .Net;
- SQLMetal [8] - narzędzie dla platformy .Net, którego jedynym zadaniem jest generacja kodu źródłowego klas na podstawie schematu bazy danych;
- Django [9] - platforma aplikacji webowych dla języka Python.

2.6.2 Generator schematu bazy danych na podstawie modelu dziedziny aplikacji

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

Przykłady generatorów *"Code First"* to:

- Hibernate [6] - pierwowzór narzęcia NHibernate (patrz: sekcja 2.6.1), przeznaczony na platformę Java;
- EntityFramework (patrz: sekcja 2.6.1);
- Django (patrz: sekcja 2.6.1).

2.6.3 Generator dokumentacji

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

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

2.6.4 Inne generatory

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

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

2.7 Rozwiązanie: jeden generator wszystkiego

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

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

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

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

2.8 Podsumowanie

Jak widać, łatwo dostępne są jedynie rozwiązania pozwalające wyeliminować pojedyncze rodzaje duplikacji. Nie jest jednak znane pojedyncze, spójne rozwiązanie eliminujące wszystkie rodzaje duplikacji. Odpowiednim rozwiązaniem wydaje się być rozwiązanie łączące kilka z przedstawionych wyżej rozwiązań. W dalszej części pracy podjęta zostanie próba stworzenia takiego rozwiązania. Rozwiązanie to zostanie oparte o generację kodu, a celem jest skonstruowanie go w taki sposób, aby pozwolić na usunięcie jak największej liczby rodzajów duplikacji patrząc zarówno z perspektywy ich rodzajów, jak i przyczyn powstawania (patrz: sekcja 2.1).

Rozdział 3

Sprecyzowanie problemu

Celem praktycznej części niniejszej pracy jest stworzenie narzędzia pozwalającego stworzyć aplikację, w jak największym stopniu unikając duplikacji. 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 systemy tak, aby była generowana na podstawie już istniejącej bazy.

Należy zaznaczyć, że od narzędzia będącego celem pracy nie jest wymagana możliwość całkowitej eliminacji duplikacji w systemie. Głownym rodzajem duplikacji, który będzie przedmiotem działania narzędzia jest duplikacja dziedziny aplikacji. Wybór padł na ten właśnie rodzaj dlatego, że przejawia się on w największym zakresie systemu. Co więcej, użycie mechanizmów generacji nie przekreśla eliminacji innych rodzajów 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, aczkolwiek uwaga poświęcona zostanie głównie definicji dziedziny aplikacji.

Rozdział 4

Generacja

Rozdział 5

Implementacja trzonu narzędzia

Zanim przystąpimy do prac nad generatorem aplikacji konkretnego typu, zajmijmy się samym procesem generacji artefaktów systemu, nazwanym tutaj "trzonem" narzędzia.

5.1 Podstawowe założenia dotyczące trzonu narzędzia

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

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

5.2 Kroki generacji

Generacja odbbywać się będzie w natępujących krokach:

5.2.1 Wczytanie definicji dziedziny aplikacji

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

5.2.2 Zdeserializowanie definicji dziedziny aplkacji

Jak wymieniono w założeniach dotyczących trzonu generatora (patrz: 5.1, format opisu dziedziny aplikacji nie będzie narzucony z góry. Każdy plik źródłowy może być zapisany w innym formacie (np. JSON lub XML). Domyślnie wspierany będzie jeden format 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 plik zawiera ogólne informacje na temat encji, pliku w podkatalogu sąsiadującym z tym plikiem zawierają definicje poszczególnych pól danej encji (patrz: sekcja 5.3.3). Wtedy wczytanie i deserializacja pól encji odbędzie się dopiero po zdeserializowaniu opisu samej encji i na tej podstawie określeniu, który katalog zawiera kolejne pliki opisujące elementy do niej należące.

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 (tutaj: jednostki generacji), które będą podstawą do wygenerowania plików wynikowych. Jednostką generacji może być na przykład pojedyncza encja dziedziny aplikacji.

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

Uzyskane jednostki generacji bezpośrednio posłużą do wygenerowania plików wynikowych - pojedyncza jednostka odpowiadać będzie pojedynczemu plikowi wynikowemu.

5.2.4 Użycie szablonów do wygenerowania plików

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

Trzon generatora przekaże daną jednostkę generacji odpowiedniemu szablonowi, a wygenerowana treśc zostanie umieszczona w odpowiednim pliku. Zadaniem użytkownika generatora będzie dostarczenie zarówno szablonu generacji, jak i ściezki, pod którą ma się znaleźć wygenerowany plik.

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

5.3 Organizacja plików wejściowych i wyjściowych

Pierszą decyzją, którą należy podjąć w trakcie implementacji, jest organizacja i format plików źródłowych, na których pracował będzie trzon narzędzia, także organizacja plików, które będzie w stanie wygenerować. Jak wspomniano wyżej, od trzonu oczekuje się jak największej elastyczności - dlatego trzon powinien 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 pojedynczy skrypt SQL zawierający schemat bazy danych używanej przez aplkację. Taki plik, oprócz swojego standardowego przeznaczenia, tj. konfigurowania bazy danych, pełniłby rolę źródła generatora *Code First*. Na jego podstawie generowane byłyby pliki zawierające defnijcje klas będących częścią implementacji modelu dziedziny w aplikacji.

Rysunek 5.1 obrazuje przykład.

Źródło generacji:
`database_schema.sql`
Wynik generacji:

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

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

5.3.2 Pojedynczy katalog z wieloma plikami źródłowymi

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

Rysunek 5.2 obrazuje przykład.

Źródło generacji:

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

Wynik generacji:

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

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

5.3.3 Drzewo katalogów z wielona plikami źródłowymi

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

Rysunek 5.3 obrazuje przykład.

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

Taka organizacja może mieć zastosowanie w przypadku, gdy w systemiie w wielu miejscach występują jedynie fragmenty encji (np. w widokach bazy danych). Wtedy każde pole może wymagać skonfigurowania dla niego miejsc, w których występuje.

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 do-

Źródło generacji:

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

Wynik generacji:

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

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

myślny sposób formatowania. Poniżej przestawiono kilka z możliwych wyborów:

5.4.1 UML

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

Trzeba jednak zauważyć, ze taki opis nie jest wystarczająco elastyczny - posiada on zdefiniowany z góry zestaw atrybutów, . Natomiast rzeczywiste scenariusze użycia generatora wymagać mogą atrybutów, których przewidzenie na etapie projektowania generatora jest niemożliwe. Przykładowo, do opisu encji mogą należeć takie atrybuty, jak:

- opis encji, który powinien znaleźć się w dokumentancji systemu;
- wersja systemu, w której encja została wprowadzona;

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.

TODO

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

- widoki bazy danych, kórych źródłem jest encja.

Diagram klas nie przewiduje przechowywania żadej z tych informacji.

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

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

Diagram klas nie przewiduje przechowywania żadej z tych informacji.

5.4.2 XML

Język XML jest powszechnie używany do opisu dziedziny, Jest o niego oparty na przykład język WSDL (język definicji usług sieciowych, ang. *Web Services Description Langu-*

age) [15] stosowany do opisu kontraktów (ang. *contract*) realizowaych przez usługi sieciowe (ang. *web service*).

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

5.4.3 JSON

Rysunek 5.7.

5.4.4 YAML

Rysunek 5.8.

- domyślnie będzie JSON, ale można to łatwo podmienić

5.5 Czy wymagać stworzenia schematu definicji dziedziny aplikacji?

- są dwie opcje:
  - (no schema) definicję dziedziny deserializować do dynamic, szablonom przekazywać dynamic (jest dowolność, ale nie wyłapie się błędów podczas deserializacji)
  - definicję dziedziny deserializować do konkretnego typu, szablonom przekazywać konkretne typy
- druga opcja wydaje się lepsza (opis dziedziny powinien być spójny)
- ale mechanizm będzie generyczny, decyzja będzie należała do konkretnego generatora

5.6 Wybór silnika do generacji kodu (templating engine)

- opisać dostępne (t4, razor, xslt)
- dlaczego t4? - np. razor nie, bo razora tez mozemy chciec generowac - generowanie razora razorem mogloby byc utrudnione

```
<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>
    <IntroducedIn>1.1</IntroducedIn>
    ...
   </Field>
   <Field Name="LastName">
    <Type>string</Type>
    <Description>The last name of the user</Description>
    <IntroducedIn>1.2</IntroducedIn>
    ...
   </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>
    <IntroducedIn>1.0</IntroducedIn>
    ...
   </Field>
   <Field Name="Content">
    <Type>string</Type>
    <Description>The content of the piece of news</Description>
    <IntroducedIn>1.0</IntroducedIn>
    ...
   </Field>
   ...
  </Fields>
 </Entity>
</Entities>
```

Rysunek 5.6: Przykładowa dziedzina aplikacji opisana językiem XML.

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

Rysunek 5.7: Przykładowa dziedzina aplikacji opisana językiem JSON.

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

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

   ...

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

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

   ...
```

Rysunek 5.8: Przykładowa dziedzina aplikacji opisana językiem YAML.

Rozdział 6

Implementacja generatora aplikacji pojedynczego typu

Założenia dotyczące całości:

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

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

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

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

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

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

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

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

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

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

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

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

Oświadczenie

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

.....

Michał Aniserowicz