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

TODO
- problem - w projektach często występuje duplikacja w wielu miejscach - w CQRS to w ogóle

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, 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 występuje duplikacja opisu dziedziny aplikacji. Występuje ona w co najmniej dwóch miejsach:

1. schemacie bazy danych (DDL),
2. definicjach klas w kodzie źródłowym aplikacji.

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

W miarę jak rozrasta się projekt informatyczny, pojawia się tendencja do duplikowania fragmentów dziedziny poruszanego przez niego problemu. Duplikacja ta rozprzestrzenia się pomiędzy modułami aplikacji, których zadaniem 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.

Jeśli moduły te posiadają osobne implementacje dziedziny aplikacji, to każda zmiana dziedziny wymaga zmodyfikowania implementacji dziedziny w każdym z modułów - co wiąże się z dużymi kosztami.

Z drugiej strony, identyczność zestawu danych udostępnianego przez różne moduły może nie być pożądana. Przykładowo, na stronie WWW wyświetlane mogą być jedynie podstawowe dane danej encji, podczas gdy pełne dane dostępne są po wyeksportowaniu arkusza kalkulacyjnego. Takie wymaganie wymusza duplikację części dziedziny aplikacji pomiędzy różnymi implementacjami tej dziedziny.

TODO: Sformułować konkretny przykład aplikacji.

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

2.6 Rozwiązanie: użycie licznych generatorów

2.6.1 Generator code-first / database-first

Rozwiązaniem problemu może być osiągnięcie implementacji, w której pełna dziedzina aplikacji definiowana jest w jednym miejscu, a jej implementacje są generowane automatycznie w odpowiednich modułach. Taka sytuacja sprawia, że duplikacja pomiędzy modułami przestaje być problemem - aby wprowadzić zmiany we wszystkich implementacjach, wystarczy wprowadzic pojedynczą modyfikację w definicji dziedziny aplikacji.

Najprostszym przykładem realizującym to rozwiązanie jest użycie generatora definicji klas na podstawie schematu bazy danych (tzw. podejście database-first) lub generatora schematu bazy danych na podstawie definicji klas (tzw. podejście code-first). Przykładami takich generatorów są:

- Hibernate,
- EntityFramework,
- LinqToSQL.

TODO: Wymienić więcej, dla różnych technolgii, dać przypisy.

Jednakże takie generatory eliminują tylko jeden rodzaj duplikacji, wymieniony na początku rozdziału.

2.6.2 Generator dokumentacji

Duplikacji w dokumentacji aplikacji można uniknąć poprzez zastosowanie narzędzi generujących tę dokumentację na podstawie kodu źródłowego aplikacji. Narzędzia takie pozwalają na wygenerowanie dokumentacji w kilku formatach, takich jak PDF czy HTML. Ich przykłady to:

- JavaDoc,
- ...

TODO: Wymienić więcej, dla różnych technolgii, dać przypisy.

W przypadku tych narzędzi występuje ten sam problem, co w przypadku generatorów schematu bazy danych lub definicji klas - eliminują one tylko jeden rodzaj duplikacji.

2.6.3 Inne generatory

Aby uniknąć pozostałych rodzajów duplikacji:

- duplikowania nazw i typów kolumn tabel w definicjach widoków bazy danych,
- duplikowania nazw i typów pól klas pomiędzy implementacjami dziedziny aplikacji w różnych jej modułach,

należałoby skorzystać z kolejnych generatorów.

Warto zauważyć, że różne typy generatorów za źródło danych obierają sobie różne definicje dziedziny: np. generator database-first bazuje na języku DDL, 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 bardzo trudne w utrzymaniu.

2.7 Rozwiązanie: jeden generator wszystkiego

Należy zauważyć, że żadne z przedstawionych wyżej rozwiązań nie jest rozwiązaniem prewencyjnym - nie zapobiega pojawianiu się nowych duplikacji. To, czy nowa rutynowa czynność zostanie zamknięta w skrypcie 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:

- każdy projekt ma inne wymagania odnośnie wygenerowanych artefaktów,
- prawdopodobne nie istnieje format, który pozwalałby zdefiniować każdą dziedzinę w najlepszy (najbardziej naturalny) dla niej sposób.

Przedstawione rozwiązania eliminują tylko pojedyncze rodzaje duplikacji. Rozwiązanie potencjalnie optymalne to mix wszystkiego po trochu.

TODO: Czemu skupiam się na dziedzinie aplikacji - wydaje się bardziej kosztowna niż duplikacja czynności, a łatwiejsza do usunięcia niż duplikacja kodu.

W dalszej części pracy najwięcej zostanie poświęcone rodzajom duplikacji, których usunięcie nie wymaga ciągłej dyscypliny od programistów (bo na to każde rozwiązanie może być zawodne).

Rozdział 3

Implementacja

Celem niniejszej pracy jest próba stworzenia takiego generatora. Założenia funkcjonalne:

- generator ma być na tyle elastyczny, aby móc obsłużyć wiele sposobów zdefiniowania dziedziny aplikacji,
- generator powinien pozwalać na wygenerowanie dowolnych plików tekstowych (skryptów SQL, kodu źródłowego, dokumentancji HTML itd.),
- generator nie musi generować logiki biznesowej aplikacji - wystarczy dziedzina

Założenia niefunkcjonalne:

- generator zostanie stworzony w technolgii .NET Framework,
- ...

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 danych.

TODO: Opisać CQRS. TODO: Opisać Event Sourcing. TODO: Opisac NoSQL. TODO: Opisać rodzaje baz NoSQL. TODO: Wybrać bazę NoSQL i dlaczego Cassandra. TODO: Opisać Cassandrę.

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].

Oświadczenie

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

.....

Michał Aniserowicz