Autor

Receptury projektowe – niezbędnik początkującego architekta

Część V: Mapowanie relacyjno-obiektowe prawdziwych obiektów – rzecz o DDD i JPA

Stosując mapery relacyjno-obiektowe, zwykle nie zastanawiamy się nad problemami związanymi z niespójnością danych wynikającą ze stosowania Lazy Loadingu, granicą spójności obiektów podczas zapisu kaskadowego oraz pułapkami naiwnego blokowania optymistycznego.
Niniejszy artykuł został osadzony w kontekście standardu Java Persistence API, jednak poruszone problemy i rozwiązania aplikują się dla każdego ORM. Artykuł nawiązuje również do Domain Driven Design, jednak przedstawione podejścia opierają się na ogólnym paradygmacie Object Oriented (i jego niezgodności z paradygmatem relacyjnym).

O serii „Receptury projektowe"

Intencją serii „Receptury projektowe" jest dostarczenie usystematyzowanej wiedzy bazowej początkującym projektantom i architektom. Przez projektanta lub architekta rozumiem tutaj rolę pełnioną w projekcie. Rolę taką może grać np. starszy programista lub lider techniczny, gdyż bycie architektem to raczej stan umysłu niż formalne stanowisko w organizacji.

Wychodzę z założenia, że jedną z najważniejszych cech projektanta/architekta jest umiejętność klasyfikowania problemów oraz dobieranie klasy rozwiązania do klasy problemu. Dlatego artykuły będą skupiały się na rzetelnej wiedzy bazowej i sprawdzonych recepturach pozwalających na radzenie sobie z wyzwaniami niezależnymi od konkretnej technologii, wraz z pełną świadomością konsekwencji płynących z podejmowanych decyzji projektowych.

AGREGAT DDD – PRAWDZIWY OBIEKT

W niniejszym artykule przyjmiemy założenie, że pracujemy z obiektami w rozumieniu Object Oriented, czyli bytami, które posiadają odpowiedzialność (metody) i hermetyzują swoje wewnętrzne struktury w celu przygotowania modelu na zmiany.

Część czytelników zapewne zetknęła się z projektami, w których stosowane są jedynie struktury danych (klasy z polami oraz metodami get/set, tudzież atrybutami). W takim podejściu zachowanie znajduje się w osobnych procedurach (zwanych Serwisami). Problemy przedstawione w tekście aplikują się również do takiego proceduralnego (znanego z Pascala i C) podejścia do modelowania.

EFEKTYWNE MAPOWANIE GRANICY AGREGATU

W podejściu Domain Driven Design Agregatem nazywamy graf obiektów, który stanowi spójną jednostkę pracy (zmiany biznesowej). Agregat posiada wyróżnioną encję zwaną korzeniem (Aggregate Root), która stanowi swego rodzaju API dla całego Agregatu. Wykonując operacje na Agregacie, wywołujemy metody biznesowe na korzeniu, korzeń natomiast dba o spójność reguł biznesowych nałożonych na składowe Agregatu (inne encje).

Na Listingu 1 widzimy przykładowy Agregat Order, który zawiera w sobie listę zamówionych pozycji (pozycja wskazuje na zamówiony produkt, oraz posiada ilość zamówionych sztuk). Natomiast z punktu widzenia Object Oriented najważniejsze jest zachowanie, jakie oferuje Agregat Order: dodanie/usunięcie produktu, zatwierdzenie zamówienia i zwrócenie całkowitego kosztu.

W kolejnych sekcjach artykułu zajmiemy się problemami związanymi z zapewnieniem spójności tych operacji i stanu w bazie danych poprzez maper relacyjno – obiektowy.

Listing 1. Przykładowy Agregat wraz z mapowaniem w standardzie JPA: pobieranie zawieranych obiektów w sposób EAGER, powiązanie one-to-many bez tabel linkujących oraz włączenie wszystkich operacji kaskadowych

```
@Entity
@Table (name = "Orders")
public  class Order {

   @Id @GeneratedValue
   private Long entityId;

   //technika A: blokowanie optymistyczne
   //(również całego Agregatu)
   @Version
   private Long version;

   //taktyka B: snapshot danych klienta
   //(odcięcie się od zmian w agregacie Client)
   @Embedded
   private ClientData client;

   @Embedded
   private Money totalCost;

   //technika C: operacje zarządzania persystencją
   //wykonywane kaskadowo na składowych Agregatu
   //(również kasowanie usuniętych elementów listy)
   @OneToMany(cascade = CascadeType.ALL, orphanRemoval=true,
       fetch = FetchType.EAGER)
   //technika D: wskazanie klucza obcego w tabeli
   //OrderLine - skutkuje uniknięciem tworzenia tabeli
   linkującej
   @JoinColumn("order_id")
```

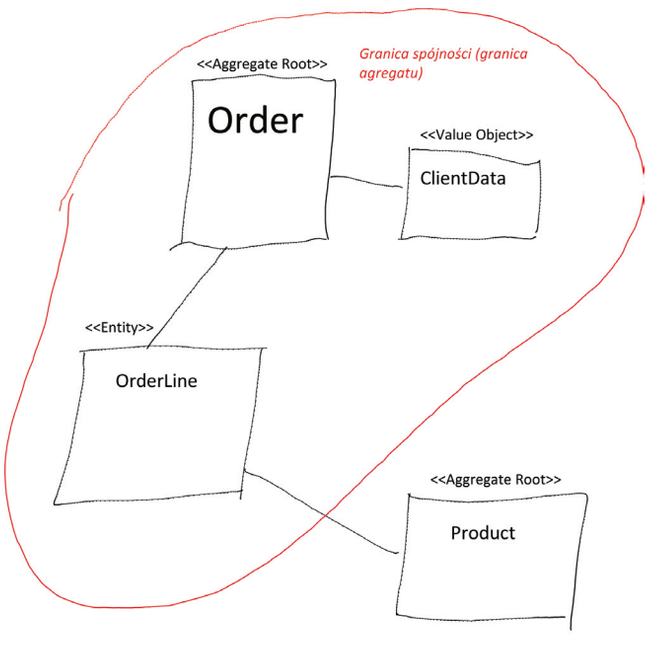
```
    private List<OrderLine> items;

    public void addProduct(Product product, int quantity) {
       //...
       OrderLine line = find(product);

          if (line == null) {
             items.add(new OrderLine(product, quantity));
          } else {
             line.increaseQuantity(quantity);
          }
    }

    public void confirm(Client client) {
       //..
       this.client = client.generateSnapshot()
    }

    public Money getTotalCost() {
       return totalCost;
    }

    //taktyka E: podejście funkcyjne (Agregat jako
    //zbiór funkcji operujących na wspólnym stanie)
    public Money calculateTotalCost() {
       //..
    }
}
```

Strategia: Wyznaczanie granicy poprzez niezmienniki

Najprostszą i najbardziej efektywną techniką wyznaczania granicy Agregatu jest enkapsulacja niezmienników nałożonych na obiekty. W naszym przykładowym Agregacie mamy następujące niezmienniki:

- nie można modyfikować zatwierdzonych zamówień
- nie można zatwierdzać zamówień pustych (niezawierających pozycji)
- nie może istnieć więcej niż jedna pozycja na dany produkt, jeżeli dodajemy produkt już dodany, to następuje zwiększenie ilości na danej pozycji
- po każdej modyfikacji (dodanie/usunięcie produktu) zamówienie powinno posiadać aktualną cenę
- ceny w zamówieniu obowiązują z momentu dodanie produktu do zamówienia

Na podstawie tych niezmienników określamy granicę spójności (czyli granicę Agregatu) tak jak na Rysunku 1. Przykładowo obiekt Product jest poza tą granicą, ponieważ jego ceny mogą ulegać zmianie. Gdybyśmy chcieli opierać logikę na najświeższych cenach produktu, wówczas model musiałby mieć zupełnie inną postać.



Rysunek 1. Diagram ilustrujący granicę Agregatu (granicę spójności operacji)

Taktyka: Kompozycja

W modelowaniu obiektowym możemy łączyć obiekty na 2 sposoby:

- agregacja – obiekty mogą istnieć niezależnie
- kompozycja – obiekty zawierane nie mogą istnieć bez obiektu, który je zawiera

Jeżeli określimy dobrze granicę Agregatu, wówczas możemy jasno stwierdzić, w jakiej relacji istnieją nasze obiekty.

Jest to kluczowa kwestia, ponieważ wynikają z niej w sposób jednoznaczny techniki, jakie zastosujemy na poziomie ORM:

- Lazy Loading
- Operacje kaskadowe
- Sposoby mapowania na poziomie bazy danych

Decyzje te zostaną opisane w kolejnych sekcjach artykułu.

W naszym przykładzie Order zawiera listę pozycji (OrderItem) w silny sposób – kompozycja. Innymi słowy pozycje nie mogą istnieć bez korzenia Agregatu (Order).

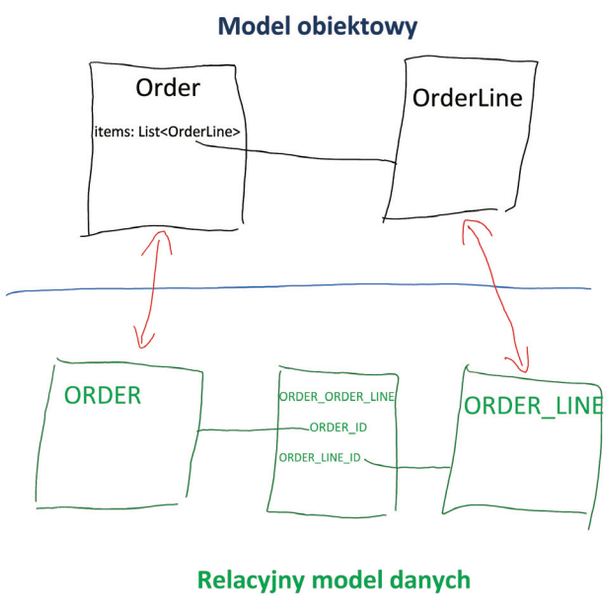
Technika: Unikaj tabel linkujących

Na poziomie bazy danych mamy relację jeden-do-wielu pomiędzy tabelami odpowiadającymi naszym klasom. Jedno zamówienie może zawierać wiele pozycji – adnotacja @OneToMany na Listingu 1.

Natomiast domyślnie Hibernate (jedna z implementacji standardu JPA) stworzy tabelę linkującą pomiędzy tabelą Order i OrderItem (Rysunek 2). Tak, struktura danych będzie wyglądać tak jak w przypadku relacji many-to-many. To domyślne zachowania wynika z następujących przesłanek:

- tabela linkująca będzie zawierać klucze obce do tabeli Order i OrderItem
- istnieje możliwość „przepięcia" pozycji do innego zamówienia bez zmiany w tabeli OrderItem (tabela OrderItem nie zawiera klucza obcego do tabeli Order, zmiana danych nastąpi jedynie w tabeli linkującej)
- dzięki temu „przepięcie" pozycji nie będzie wiązało się z „pobrudzeniem" danych, czyli ORM nie będzie stosował mechanizmu Optimistic Locking.

Jednak przy założeniu, że OrderItem jest integralną częścią Order, scenariusz powyższy nie ma sensu. Możemy zatem śmiało założyć, że tabela linkująca nie jest nam potrzeba. Należy zatem wymusić takie zachowanie poprzez adnotację @JoinColuml przy kolekcji pozycji (Listing 1 – technika D), która spowoduje, że na poziomie bazy danych tabela OrderItem będzie posiadać klucz obcy do tabeli Order, a co za tym idzie tabela linkująca będzie zbędna.



Rysunek 2. Diagram ilustrujący model danych, który jest domyślny w Hibernate dla powiązania one-to-many

Technika: Agregat jest dokumentem

Projektanci systemów stosujący rozwiązania noSQL zapewne już zauważyli, że klasa Order wraz z jej skomponowanymi pozycjami (i logicznymi niezmiennikami) jest typowym „dokumentem" w sensie baz danych dokumentowych.

Oczywiście zmiana bazy danych nie wchodzi w grę w większości projektów, dlatego pochylimy się nad kolejnymi problemami wynikającymi z niezgodności paradygmatów relacyjnego i obiektowego.

RACJONALNE POBIERANIE AGREGATÓW

Mamy za sobą określanie granicy Agregatu, jak zatem teraz pobrać taki Agregat? Czy naiwne wywołanie metody find(java.lang.Class<T> entityClass, java.lang.Object primaryKey) jest racjonalne i bezpieczne?

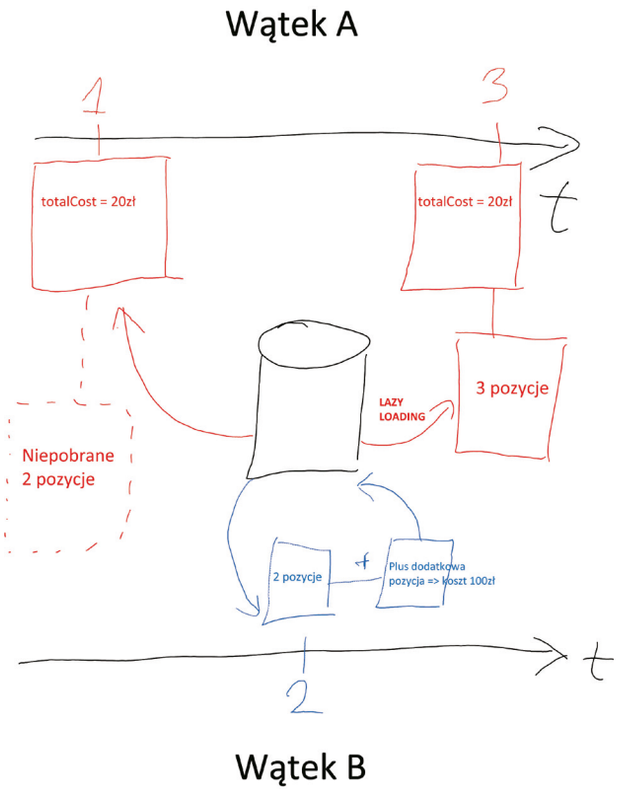
Strategia: Unikaj pułapki czasu – pobieraj całe Agregaty

Załóżmy, że mamy serwis aplikacyjny, który pobiera zamówienie z bazy danych do pamięci, aby wykonać pewne przetwarzanie.

Wyobraźmy sobie scenariusz współbieżnego dostępu do danych:

- Stan w bazie danych jest następujący: zamówienie zawiera dwie pozycje, każda z nich kosztuje 10zł, zatem sumaryczny koszt to 20 zł.
- Serwis aplikacyjny jest uruchomiony przez dwa wątki służące obsłudze dwóch żądań użytkowników
- Wątek A pobrał zamówienie (sumaryczny koszt 20 zł), ale jeszcze nie pobrał pozycji – będą pobrane przy pomocy Lazy Lodingu, gdy będą potrzebne
- Wątek B zmienił zamówienie, dodając do niego jeszcze jedną pozycję, co spowodowało, że sumaryczny koszt zamówienia wzrósł do 100zł
- Wątek A wreszcie dokonuje operacji, która wymaga pobrania pozycji, zatem pobierane są przy pomocy Lazy Loadingu 3 pozycje.

Wątek A widzi zatem stan, który narusza niezmiennik założony we wcześniejszej części artykułu. Koszt całkowity nie odpowiada ilości pozycji.



Rysunek 3. Diagram ilustrujący problem opóźnionego pobierania danych, które mogą ulec modyfikacji

Autor

Sytuacja zostanie wykryta przez mechanizm Optimistic Locking (adnotacja @Version) tylko wówczas, gdy zamówienie będzie zapisywane w bazie danych.

A co jeżeli nasz serwis nie modyfikuje zamówień (nie będziemy ich zapisywać), a jedynie używa stanu, aby dokonać innych operacji poza Agregatem zamówienia (np. wystawia fakturę)?

Jednym z rozwiązań jest pobranie zamówienia z blokadą optymistyczną do zapisu: http://docs.oracle.com/javaee/6/api/javax/persistence/LockModeType.html#OPTIMISTIC

Strategia: Unikaj stanu - podejście funkcyjne

Innym rozwiązaniem może być zmiana strategii modelowania z obiektowego na funkcyjne. Zamiast zwracać stan obiektu getTotalCost() możemy wywołać operację calculateTotalCost(), która za każdym razem pobierze najświeższą wersję pozycji i przeliczy aktualny koszy.

Co prawda Java nie jest językiem funkcyjnym, ale możemy myśleć o tej metodzie jak o funkcji.

Impedance mismatch

Mamy tutaj do czynienia z problemem nieco szerszym – niezgodnością paradygmatów obiektowego i relacyjnego (zobacz ramka „w sieci").

W podejściu relacyjnym istnieje miejsce „do którego należy pójść", aby otrzymać dane. W podejściu obiektowym dane są „na miejscu – w obiekcie".

Wywołanie „funkcji" jest niejako ukłonem w stronę świata relacyjnego i wprost modeluje fakt, że oto „idziemy do miejsca, gdzie są dane".

Taktyka: Eager zamiast Lazy Loadingu

Jednak czasem ze względów wydajnościowych takie rozwiązanie (przeliczanie od nowa wartości) może być niedopuszczalne.

Możemy zatem przyjąć taktykę polegającą na tym, że zawsze pobieramy cały Agregat chciwie/łapczywie/gorliwie (trzy piękne tłumaczenia słowa Eager). Mapowanie Eager znajduje się na Listingu 1 (technika C). Część czytelników zapewne obruszyła się w tym momencie „Jak to, mamy pozbyć się Lazy Loadingu?!?"

Zastanówmy się jednak przez chwilę nad naszym modelem. Jeżeli założyliśmy, że:

- operujemy na obiektach, których stan jest spójny
- modelujemy niezmienniki (reguły biznesowe), które wyznaczają granicę Agregatu
- to wynika z tego, co następuje:
- Agregaty są niewielkie
- za każdym razem (każde wywołanie metody Agregatu) potrzebujemy niemal wszystkich pól Agregatu, aby sprawdzić niezmienniki

Możemy zatem przyjąć, że racjonalnie będzie pobrać do pamięci od razu cały Agregat. Redukuje to prawdopodobieństwo przeplecenia się wątków w taki sposób, że Agregat straci spójność swych wewnętrznych obiektów. Aby wyeliminować ryzyko, możemy:

- włączyć silniejszą separację transakcji, podczas gdy ORM wysyła kilka kwerend do bazy danych, aby pobrać cały Agregat
- wymusić (opcja Hibernate) pobieranie jednym zapytaniem z podzapytaniami

BEZPIECZNE ZAPISYWANIE AGREGATÓW

Skoro Agregat został pobrany, to przychodzi czas na jego zapis. ORM zwykle posiada mechanizm brudzenia (Dirty checking), który wykrywa zmiany w polach obiektów i podczas zamykania sesji persystencji utrwala te zmiany.

ORM zapewnia również zwykle mechanizm Optmistic Locking, który wykrywa współbieżną modyfikację obiektów przez inne wątki. Mechanizm ten

jest oparty o specjalne pole, które jest inkrementowane podczas każdego zapisu obiektu.

Technika: Blokuj optymistycznie całe Agregaty

A co jeżeli nasz Agregat uległ zmianie, ale nie na poziomie korzenia?

Przykładowo: jeden wątek zmienia adres dostawy, a inny wątek zmienia klienta. Technicznie korzeń Agregatu (Order) nie uległ zmianie. Uległy zmianie obiekty zawierane. Zatem obiekt order nie będzie zapisywany w bazie, a więc jego pole adnotowane @Version (Listing 1 – technika A), które zapewnia nam optymistyczne blokowanie, nie będzie inkrementowane.

Ale co jeżeli logicznie uznamy, że zamówienie uległo zmianie, bo np. istnieje niezmiennik, mówiący o tym, że pewnym klientom nie wysyłamy zamówień pod pewne adresy?

W Hibernate należy wówczas założyć blokadę następującą na korzeń Agregatu: http://docs.oracle.com/javaee/6/api/javax/persistence/LockModeType.html#OPTIMISTIC_FORCE_INCREMENT

Blokada ta działa w następujący sposób: podczas synchronizacji L1 cache z bazą danych (czyli zapisu obiektów) zostanie zawsze sprawdzone i zwiększone pole @Version na obiekcie przekazanym do metody lock(). Nie jest ważne, czy obiekt ten będzie zapisywany w bazie, czy będą zapisywane jedynie inne obiekty.

Możemy zatem traktować ten mechanizm blokowania jak zakładanie optymistycznej blokady na cały Agregat. Ponieważ zablokowanie korzenia Agregatu będzie w praktyce skutkowało zablokowaniem całego Agregatu.

Całość możemy enkapsulować w Repozytorium zamówień (Listing 2). Podczas zapisu zakładamy rzeczoną blokadę. Oczywiście jeżeli ORM posiada mechanizm Dirty Checking, wówczas nie musimy zapisywać wprost obiektów, „brudne" obiekty zostaną zapisane automatycznie. Natomiast wprowadzamy idiom repozytorium po to, aby ukryć w nim tego typu szczegóły jak optymistyczne blokowanie z wymuszeniem inkrementacji wersji.

Zakładanie blokady podczas odczytu byłoby zbyt naiwnym założeniem.

Listing 2. Przykładowe repozytorium, które zakłada blokadę na korzeń Agregatu podczas jego zapisu

```
public class JpaOrderRepository implements OrderRepository {

   @PersistenceContext
   protected EntityManager entityManager;

   public Order load(Long id) {
      //założenie optymistycznej blokady odczytu
      //(zmiana innych Agregatów spowoduje sprawdzenie wersji
      danego Agregatu)
      return entityManager.find(Order.class,
         id, LockModeType.OPTIMISTIC);
   }

   public void save(Order order) {
      //założenie optymistycznej blokady zapisu całego
      Agregatu
      //(zmiana wnętrza Agregatu spowoduje sprawdzenie i
      zwiększenie wersji)
      entityManager.lock(order,
         LockModeType.OPTIMISTIC_FORCE_INCREMENT);
   }
}
```

Technika: Zapisuj kaskadowo całe Agregaty

Jeżeli przyjmiemy, że nasze Agregaty to grafy obiektów połączonych relacją kompozycji (obiekt zawierany nie może istnieć bez zawieranego), to śmiało możemy zdecydować się na mapowanie wszystkich operacji kaskadowych na wszystkie obiekty zawierane przez Agregaty (Listing 1). Operacje kaskadowe polegają tym, że jeżeli wykonujemy jakąś operację (zapis, odświeżenie, usunięcie) na korzeniu Agregatu, to te same operacje będą wykonywane na obiektach zawieranych.

POBIERANIE DANYCH PRZEKROJOWYCH

Czytelników zainteresowanych problemem pobierania danych przekrojowych (problem niedostosowania encji do potrzeb wyświetlania danych) odsyłam do jednego z poprzednich artykułów z serii DDD „Część IVb: Skalowalne systemy w kontekście DDD – architektura Command-query Responsibility Segregation (stos Read)", więcej informacji na ten temat znajduje się w ramce „w sieci".

PROSTE REGUŁY WYNIKAJĄ Z PRZEMYŚLANYCH ZAŁOŻEŃ

Początkujący użytkownicy ORM zastanawiają się zwykle nad regułami rządzącymi stosowaniem mechanizmów ORM: Lazy Loadingiem, operacjami kaskadowymi, blokowaniem i wprowadzaniem tabel linkujących.

Jedyną odpowiedzią, jaką uzyskują od swych doświadczonych kolegów, jest słynne „to zależy".

Jeżeli natomiast przyjmiemy pewne założenie co do kontekstu, w jakim stosujemy ORM, np. modelujemy spójne grafy obiektów o dobrze określonej granicy (np. stosując zasady DDD), to automatycznie znikają punkty swobody i nic już nie „zależy". Reguły są proste i jasne – zobacz ramkę „dobre praktyki".

Dobre praktyki

- Unikaj Lazy Loadingu – obiekt o dobrze określonej granicy enkapsulacji nie potrzebuje tego mechanizmu
- Pobieraj chciwie/łapczywie/gorliwie (ang. eager) całe Agregaty – wynika z powyższego
- Mapuj Agregaty bez tabel linkujących – nie potrzebujesz ich przy relacji kompozycji
- Zapisuj kaskadowo całe Agregaty – wynika z relacji kompozycji
- Optimistic Locking to nie tylko atrybut @Version
- Blokuj optymistycznie całe Agregaty poprzez nakładanie blokady na korzeń Agregatu

W sieci

- Niezgodności pomiędzy paradygmatem obiektowym i relacyjnym: http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch
- Agregacja a kompozycja: http://pl.wikipedia.org/wiki/Agregacja_(programowanie_obiektowe)
- Seria artykułów poświęcona Domain Driven Design: http://bottega.com.pl/artykuly-i-prezentacje#ddd
- Przykładowy projekt, z którego zaczerpnięto przykłady: http://bottega.com.pl/ddd-cqrs-sample-project

Sławomir Sobótka slawomir.sobotka@bottega.com.pl

Programujący architekt aplikacji specjalizujący się w technologiach Java i efektywnym wykorzystaniu zdobyczy inżynierii oprogramowania. Trener i doradca w firmie Bottega IT Solutions. W wolnych chwilach działa w community jako: prezes Stowarzyszenia Software Engineering Professionals Polska (http://ssepp.pl), publicysta w prasie branżowej i blogger (http://art-of-software.blogspot.com).