Sławomir Sobótka

Domain Driven Design krok po kroku

Część II: Zaawansowane modelowanie DDD – techniki strategiczne: konteksty i architektura zdarzeniowa

Modele nietrywialnych domen wymagają struktury. Struktury, która pomaga nam okiełznać chaos. W zestawie technik DDD znajdują się podejścia strukturyzacji systemu na każdym poziomie abstrakcji. Techniki DDD stanowią również „rusztowania mentalne", które prowadzą procesy myślowe modelarza w kierunku bardziej adekwatnych modeli. W artykule zostaną przedstawione techniki destylacji domen, wydzielania kontekstów, komunikacji pomiędzy kontekstami, porządkowania złożonych kontekstów oraz na poziomie mikro – efektywnego modelowania Agregatów.

Plan serii

Niniejszy tekst jest drugim z serii artykułów mających na celu szczegółowe przedstawienie kompletnego zestawu technik modelowania oraz nakreślenie kompletnej architektury aplikacji wspierającej DDD.
Część 1: Podstawowe Building Blocks DDD;
Część 2: Zaawansowane modelowanie DDD – techniki strategiczne: konteksty i architektura zdarzeniowa;
Część 3: Szczegóły implementacji aplikacji wykorzystującej DDD na platformie Java (dwa podejścia: Spring i EJB 3.1 oraz JPA);
Część 4: Skalowalne systemu w kontekście DDD - architektura CqRS;
Część 5: Kompleksowe testowanie aplikacji opartej o DDD;
Część 6: Behavior Driven Development - Agile drugiej generacji

STRATEGICZNA STRUKTURYZACJA – RÓŻNE SKALE, JEDEN CEL

„Wszystko jest na miejscu i wszystko ma swoje miejsce" - cytat Benjamina Franklina jest chyba najlepszym podsumowaniem strategicznych technik DDD.

Pierwszą decyzją, jaką podejmujemy podczas modelowania z wykorzystaniem DDD, jest określenie tych miejsce w całej „rozciągłości" systemu, w których będziemy stosować techniki DDD.

Kolejnym poziomem strukturyzacji jest wydzielenie jasnych granic Bounded Context – kontekstów, w których obowiązują osobne modele.

Separacja modeli jest ze wszech miar pożyteczna, jednak zwykle rodzi problem: domeny muszą ze sobą współpracować. Zatem zastanowimy się, jak technicznie dokonać mapowania kontekstów. Poszczególne modele mogą być wciąż złożone, dlatego nadamy im głębszą - warstwową strukturę, która separuje elementy modelu wg ich podatności na zmiany.

Na zakończenie naszych rozważań powrócimy do omawianych w poprzedniej części Agregatów i zastanowimy się nad ich granicami. Poznamy techniki efektywnego modelowania granic Agregatów oraz typowe błędy.

DESTYLACJA CORE DOMAIN - KIEDY NIE STOSUJEMY DDD

Stosowanie technik DDD wnosi pewien narzut w procesie tworzenia oprogramowania. Narzut ten jest inwestycją, która jednak nie zwraca się w każdym przypadku. Dlatego na wstępie modelowania DDD określamy miejsca w systemie, w których stosujemy techniki DDD, oraz miejsca, w których ich nie stosujemy.

Supporting Domain – świadomy dług techniczny

Relatywnie proste domeny nie wymagają technik DDD – ich modelowanie z definicji nie wymaga złożonego procesu mentalnego. Możemy je z powodzeniem zaimplementować w uproszczonej (1,2 – warstwowej) architekturze. Zwykle okazuje się, że nad prostymi domenami wykonujemy jedynie operacje CRUD (Create, Read, Update, Delete), zatem warto skorzystać z mnogości wyboru framework'ów pozwalających generować niemal automatycznie aplikacje klasy „przeglądarka do bazy danych".

Te relatywnie proste domeny nazywamy Supporting Domain. W tych miejscach systemu decydujemy się na kompromisy jakościowe (jakość modelu, jakość kodu) – świadomie zaciągamy dług techniczny.

Przykładowo w naszym systemie możemy zdecydować się na wprowadzenie systemu komentarzy. Zakładając, że jego działanie nie jest krytyczne dla powodzenia całego przedsięwzięcia biznesowego, klasyfikujemy ten podsystem jako Supporting Domain. Warto zwrócić uwagę,

że w innym kontekście podsystem komentarzy może być krytyczny (ponieważ np. buduje zaufania) – wówczas traktujemy go jako Core Domain.

Generic domain – modele domen bazowych

Pewne domeny są generyczne w stosunku do naszej głównej domeny. Przykładowo tworząc system służący do handlu zawartością multimedialną, nie będziemy skupiać wysiłku mentalnego, pracując nad modułem fakturowania. Fakturowanie jest krytyczne (zależy nam na jakości tego modułu), ale a) nie stanowi ono głównej wartości, oraz b) tego typu systemy istnieją na rynku (zarówno Open Source, jak i komercyjnym). Dlatego przykładowe fakturowanie możemy potraktować jako Generic Domain. Z podsystemami zaliczanymi do Generic Domains integrujemy się w taki sposób, aby ich modele nie „przeciekały" do naszego głównego modelu.

Innym przykładem Generic Domain może być biblioteka do operowania na Grafach. Być może będziemy ją wykorzystywać w Module Magazynowym, aby sugerować najkrótsze ścieżki, jakimi należy się poruszać pomiędzy regałami w magazynach, aby jak najszybciej skompletować zamówienie. W tym przypadku nadbudowujemy domenę Magazynu (regały, półki, korytarze) nad generyczną domeną grafów (węzły, ścieżki, algorytmy wyszukiwania ścieżek).

Core Domain – miejsce dla DDD

Omówiliśmy rodzaje domen, dla których nie stosujemy technik DDD. Techniki DDD mają zastosowanie (jako inwestycja) w Core Domain. Core Domain to sfera logiki biznesowej, która jest głównym powodem, dla którego system powstaje. Być może model z Core Domain stanowi przewagę systemu (klienta zamawiającego system) nad konkurencją. W Core Domain inwestujemy czas naszych najlepszych ludzi – Developerów oraz Ekspertów Domenowych. W DDD zakładamy, że modelowanie Core Domain stanowi największe wyzwanie w cyklu produkcji systemu. Zakładamy również, że słaby model stanowi wysokie ryzyko niepowodzenia przedsięwzięcia.

Podstawową techniką destylacji Core Domain jest spisanie Dokumentu Wizji. Dokument ten powinien być krótki (np. 1/2 A4), dzięki temu skłoni jego twórcę do skupienia się na głównych (Core) założeniach, które zwykle dokładnie mapują się na czynniki przewagi, czyli Core Domain.

GRANICE BOUNDED CONTEXT – GRANICE WSPÓLNEGO JĘZYKA

Drugim poziomem strukturyzacji modelu jest wydzielanie Bounded Context – kontekstów, do których jest przypisany Ubiquitous Language. Przypomnijmy w tym miejscu: istnienie „wspólnego języka", którym posługują się wszyscy uczestnicy projektu – od Eksperta Domenowego po Developerów - jest jednym z głównych założeń DDD.

Modele odseparowane przez Bounded Context nie mogą odnosić się do siebie wprost. Są to niezależne byty, które mogą być rozwijane przez osobne zespoły. Omawiane w dalszej części Wzorce Architektoniczne pozwalają na hermetyzację modeli, która redukuje problemy związane ze zmianami i pozwala na niezależne rozwijanie modeli przez zespoły, które nie „wchodzą sobie w drogę".

Projekt referencyjny (link w ramce „w sieci") zawiera 3 przykładowe Bounded Context: Sales, CRM, Shipping. W przypadku tego projektu BC odpowiadają modułom systemu.

Granice kompetencji Ekspertów domenowych

W systemach o większej skali mamy często do czynienia z sytuacją, gdzie korzystamy z wiedzy wielu Ekspertów Domenowych. Eksperci ci specjalizują się w hermetycznych domenach (sferach) wiedzy. Możliwość komunikacji merytorycznej pomiędzy ekspertami jest mocno ograniczona.

Przykładowo w kontekście naszego przykładowego modelu: Pojęcie „produktu" w Module Sprzedaży oznacza coś, co ma charakterystykę handlową, pewne atrybuty promocyjne, być może jest powiązane z profilami behawioralnymi klientów.

Natomiast w Module Magazynowym jest to przedmiot, którego główną cechą jest przykładowo to, że zajmuje określoną przestrzeń na regałach i być może wymaga specjalnego traktowania podczas transportu.

Jedynie ubóstwo naszego języka powoduje, że eksperci domenowi socjalizujący się w sprzedaży i eksperci specjalizujący się w domenie magazynów używają podobnej fali akustycznej, która brzmi „produkt". Jednak na poziomie znaczenia to brzmienie budzi zupełnie różne skojarzenia – skojarzenia z pojęciami pojawiające się w umyśle jednego eksperta są w ogóle niedostępne dla umysłu innego eksperta.

Zatem próba uogólnienia i „zsumowania" kilku modeli wynikających z wiedzy osób o rozłącznych kompetencjach jest zajęciem karkołomnym. W DDD takie podejście jest zaliczane do anty-wzorców i nazywa się Corporate Model Anti-Pattern.

Jeżeli eksperci domenowi znający na wskroś swe specjalizacje nie są w stanie się porozumieć, to w jaki sposób możemy sobie wyobrazić sytuację, w której developerzy – nie znający zwykle żadnej z tych domen – mogą być w stanie stworzyć wspólny model kilku domen...?

Granice słownictwa Ekspertów Domenowych z Ubiquitous Language wyznaczają granice modeli - Bounded Context.

Techniki uwspólniania modeli

Część pojęć jest wspólna dla wszystkich Bounded Context. Przykładowo klasa Money z poprzedniego artykułu modeluje pojęcie, co do którego wszyscy Eksperci Domenowi zgadzają się w każdym szczególe. Wspólne pojęcia umieszczamy w tak zwanym Shared Kernel.

Inną techniką uwspólniania modeli jest podejście nazywane w DDD: Conformist. Polega na tym, że jeden

model jest zależnością do drugiego. Czyli zmiany w jednym modelu powodują „katastrofę” w drugim. Z tego względu jest to podejście, którego rozważenie zaleca się w przypadków projektów wspieranych przez zespoły outsourcingowe.

WSPÓŁPRACA KONTEKSTÓW – W KIERUNKU ARCHITEKTURY ZDARZENIOWEJ

Modele domen wyznaczone przez Bounded Context są hermetyczne – oznacza to, że pojęcia z jednego modelu nie „wyciekają” do innych modeli. Jednak systemy mają to do siebie, że ich składowe muszą współpracować.

W poprzedniej części wprowadziliśmy wstępny szkic warstwowej architektury aplikacji. Nad warstwą logiki domenowej umieszczamy logikę aplikacji, modelującą Use Case/Sser Story/Serwisy SAO. Poniżej domeny umieszczamy warstwę infrastruktury, która zawiera między innymi implementacje repozytoriów.

Zobaczmy, jak na bazie tych założeń możemy podejść do komunikacji pomiędzy Bounded Contexts.

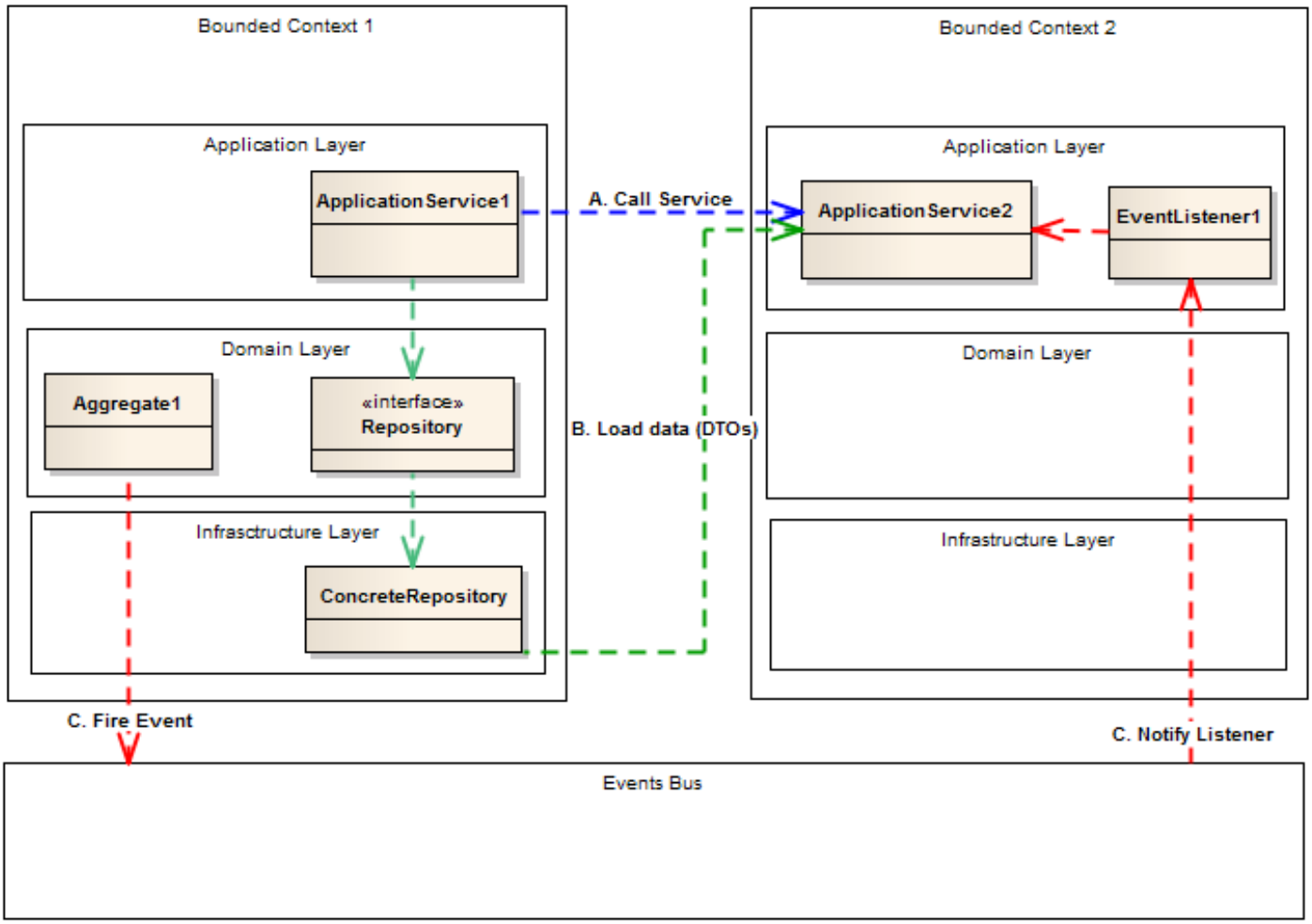
Współpraca poprzez dane

W najbardziej podstawowym przypadku jeden BC potrzebuje pewnych danych z innego BC. W takim wypadku komunikację możemy modelować poprzez Repozytoria. Implementacja repozytorium w BC1 wywołuje Serwis Aplikacyjny (de facto API) BC2. Innymi słowy API BC2 staje się źródłem danych w BC1. Pamiętajmy, że z założenia model domenowy nie „wycieka” ponad warstwę Serwisów Aplikacyjnych. Zatem komunikacja musi następować poprzez Obiekty Transferowe.

Dzięki takiemu podejściu uzyskujemy hermetyzację modeli. Model domeny BC2 nie jest znany w domenie BC1. Zespół pracujący nad BC2, publikując API, „podpisuje kontrakt” ze „światem zewnętrznym” i od tej pory szczegółem implementacyjnym jest sposób, w jaki kontrakt ten jest spełniony. Dzięki temu model domeny BC2 może rozwijać się niezależnie i we własnym tempie.

Podejście to różni się w zasadniczy sposób od popularnego podejścia, w którym moduły „gmerają sobie nawzajem” w modelach danych.

Rysunek 1. Podejścia architektoniczne do współpracy pomiędzy Bounded Context



Współpraca poprzez operacje

Innym przypadkiem współpracy BC jest wywołanie API w celu wykonania pewnych operacji. Czyli nie operujemy wprost na modelu domenowym innego BC, a wywołujemy zbudowane nad nim API. Jest to klasyczna praktyka, ale wspominamy o niej dla zachowania kompletności.

Współpraca poprzez sygnały – Zdarzenia biznesowe

Wywoływanie serwisów z innych BC wprowadza silny Coupling. W przypadku większej ilości współpracujących ze sobą modułów skończymy z Architekturą o strukturze klasycznego Spaghetti.

Klasyczną techniką Decouplingu jest wprowadzenie modelu zdarzeń. Zdarzenia są jedną z technik Inversion of Control – obok Dependency Injection i Aaspect Oriented Programming. Musimy pamiętać jednak, że odwracając kontrolę, jednocześnie ją tracimy. Zatem zdarzeń używamy wówczas, gdy chcemy modelować współpracę, gdzie nie zależy nam na kontroli kolejności, czasu ani rezultatu wykonania.

W DDD przy pomocy zdarzeń modelujemy istotne z biznesowego punktu widzenia fakty, które zaszły w cyklu życia Agregatu.

Listing 1. Zgłoszenie zdarzenia w Bounded Context CRM w Agregacie Customer

```
@Entity
public class Customer extends BaseAggregateRoot{

   public enum CustomerStatus{
      STANDARD, VIP, PLATINUM
   }

   @Enumerated(EnumType.STRING)
   private CustomerStatus status;

   public void changeStatus(CustomerStatus status){
      if (status.equals(this.status))
         return;
      this.status = status;
      eventPublisher.publish(
         new CustomerStatusChangedEvent(
            getEntityId(), status));
   }
}
```

Listing 1 ilustruje przykład zgłoszenia zdarzenia przez Agregat Customer w momencie zmiany jego statusu. Klasa CustomerStatusChangedEvent jest nośnikiem informacji o tym, który klient (ID) zmienił status. Modelując zdarzenia, musimy podjąć decyzję o tym, jakie informacje umieszczamy w klasach zdarzeń – w naszym przykładzie jest to techniczne ID – być może lepszą decyzją byłoby przeniesienie informacji o biznesowym numerze klienta...

Zdarzenie modeluje fakt, który miał miejsce i nie można go zawetować – możemy co najwyżej zareagować w dodatkowy sposób na zaistniały fakt. Aby osiągnąć taki kontrakt na poziomie implementacji technicznej musimy posłużyć się transakcyjnym silnikiem zdarzeń, który zagwarantuje nam, że zdarzenie zostanie propagowane dopiero wówczas, gdy transakcja, w której go wygenerowano, powiodła się. Przykładem standardu wspierającego komunikaty transakcyjne jest Java Message Service.

Listing 2. Listener zdarzenia w Bounded Context Sales – listener znajduje się na poziomie aplikacji

```
@EventListeners
public class CustomerStatusChangedListener{

   @EventListener(asynchronous=true)
   public void handle(CustomerStatusChangedEvent
   event) {
      if (event.getStatus() == CustomerStatus.VIP){
         calculateReabteForAllDraftOrders(
            event.getCustomerId(), 10);
      }
   }

   private void calculateReabteForAllDraftOrders(
      Long customerId, int i) {
   // TODO Auto-generated method stub
   }
}
```

Listing 2 przedstawia kod jednego z Listenerów naszego przykładowego zdarzenia. Listener ten wprowadza następującą funkcjonalność: jeżeli w module CRM klient stanie się VIPem, wówczas w module Sales nadajemy mu 10% rabatu na wszystkie niezatwierdzone zamówienia.

Warto zwrócić uwagę na fakt, iż nasz przykładowy Listener posiada jedynie logikę filtrowania zdarzeń, natomiast całą pracę deleguje do Serwisu Aplikacyjnego (API) swojego modułu. Jest to zatem jedynie kod „klejący". Jak widzimy na Rysunku 1, nasz przykładowy Listener jest klientem do API na takim samym poziomie jak np. klienty GUI. Jest to zatem jeden z wielu aktorów, którzy mogą uruchamiać API. Przykładowo rabaty na zamówienia mogą być naliczane nie tylko automatycznie podczas promowania klienta do statusu VIP, ale również ręcznie, poprzez GUI np. panelu administracyjnego.

Zdarzenia wprowadzamy do architektury systemu z kilku powodów:

- decoupling modeli – opisany przykład
- asynchroniczne wykonanie dodatkowych operacji – przykładowo w module CRM możemy wysyłać maile do klientów, których statusy zmieniono; wysyłka maili to czynność z jednej strony dodatkowa, a z drugiej potencjalnie długotrwała
- otwarcie architektury na pluginy – „wpinanie" kolejnych listenerów można potraktować jako dodawanie pluginów
- rejestrowanie zdarzeń w celu ich przetwarzania i analizy - np. z wykorzystaniem silników CEP (Complex Events Processing) pozwalających na deklarowanie kwerend filtrujących określone wzorce zdarzeń w czasie
- składowanie zdarzeń jako modelu danych alternatywnego dla bazy relacyjnej – Event Sourcing

Saga – model czasu w procesie biznesowym

Rozszerzeniem modelu zdarzeń jest Saga biznesowa. Technicznie saga jest persystentnym multi-listenerem. Oznacza to, że obiekty Sagi nasłuchują wielu zdarzeń oraz ich stan jest utrwalany pomiędzy kolejnymi zdarzeniami – z uwagi na potencjalnie długi czas upływający pomiędzy kolejnymi zdarzeniami.

Natomiast na poziomie koncepcyjnym Saga modeluje de facto czas.

Listing 3 zawiera przykładowy kod Sagi, która modeluje przepływ zamówienia. Saga reaguje na zdarzenia: stworzenia zamówienia, zatwierdzenia zamówienia, nastąpienia wysyłki oraz dostarczenia zamówienia. Zakładamy, że nie możemy przewidzieć kolejności pojawienia się tych zdarzeń, oraz czas pomiędzy ich wystąpieniem może sięgać miesięcy.

Każda metoda nasłuchująca konkretnego zdarzenia w Sadze modyfikuje jej wewnętrzny stan (wzorzec projektowy Memento) oraz sprawdza, czy warunki biznesowe potrzebne do zakończenia Sagi zostały spełnione.

Więcej o Sagach na stronie projektu Levan oraz na stronie szyny NServiceBus, która wspiera model Sag poprzez API silnika.

POZIOMY MODELU – OKIEŁZNAĆ CHAOS

Mając do czynienia ze złożonymi modelami, możemy potrzebować dodatkowego rusztowania „rozpinającego" strukturę modelu. Z czasem, gdy nasze rozumienie modelu pogłębia się, zauważamy, że pewne jego elementy są ogólnym fundamentem, na którym opierają się specyficzne czynności. Specyficzne czynności mają znowuż swe wariacje.

Listing 3. Saga modelująca przepływ zamówienia

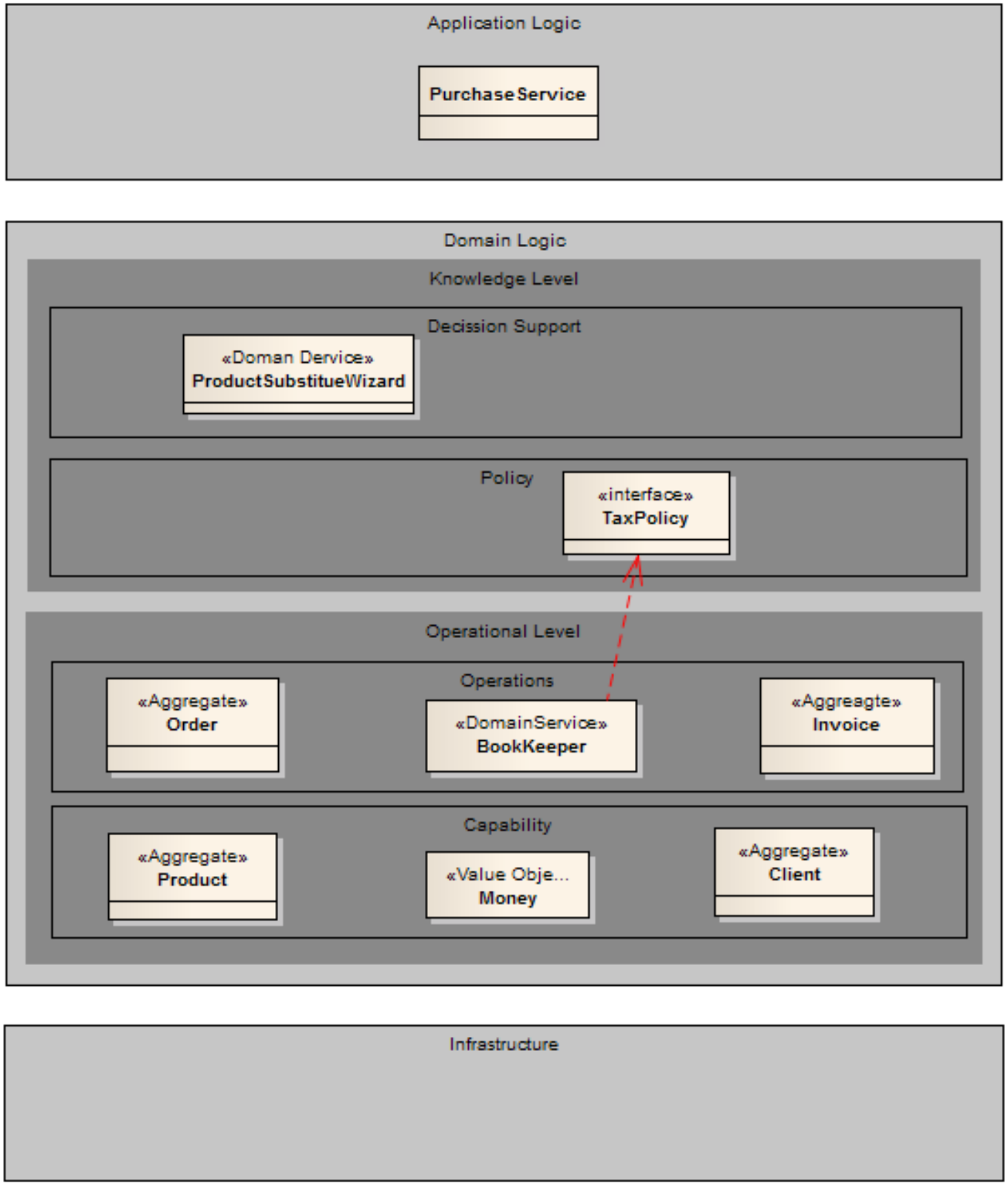
```
@Saga
public class OrderShipmentStatusTrackerSaga extends
     SagaInstance<OrderShipmentStatusTrackerData> {

   @Inject
   private OrderRepository orderRepository;

   @SagaAction
      public void handleOrderCreated(OrderCreatedEvent event) {
      data.setOrderId(event.getOrderId());
      completeIfPossible();
   }

   @SagaAction
   public void handleOrderSubmitted(OrderSubmittedEvent event) {
      data.setOrderId(event.getOrderId());
      // do some business
      completeIfPossible();
   }

   @SagaAction
      public void orderShipped(OrderShippedEvent event) {
      data.setOrderId(event.getOrderId());
      data.setShipmentId(event.getShipmentId());
      completeIfPossible();
   }

   @SagaAction
   public void shipmentDelivered(ShipmentDeliveredEvent event) {
      data.setShipmentId(event.getShipmentId());
      data.setShipmentReceived(true);
      completeIfPossible();
   }

   private void completeIfPossible() {
      if (data.getOrderId() != null
         && data.getShipmentId() != null
         && data.getShipmentReceived()) {
         Order shippedOrder = orderRepository.load(data.getOrderId());
         shippedOrder.archive();
         orderRepository.save(shippedOrder);
         markAsCompleted();
      }
   }
}
```

Różne części modelu charakteryzują się różną podatnością na zmiany. Czytelnicy zaznajomieni z Archetypami Modeli Biznesowych znają pojęcie dwóch poziomów modelu: Operational Level i Knowledge Level. Natomiast w DDD nadajemy modelowi jeszcze głębszą strukturę i wyłaniamy w nim aż cztery poziomy...

Rozwarstwienie logiki to dopiero początek

W poprzedniej części dokonaliśmy rozwarstwienia logiki na warstwę aplikacji i warstwę logiki domenowej. Warstwa aplikacji modelu (czylu Use Case/User Story/Serwisy SAO) jest odpowiedzialna za:

Rysunek 2. Poziomy modelu domenowego (wraz z mapowaniem na Archetypowe Operational i Knowlege Level)



- orkiestrację domeny – scenariusz sterowania obiektami domenowymi
- dodatkową logikę typową dla tej konkretnej aplikacji
- technikalia takie jak transakcje i bezpieczeństwo

Warstwa logiki domenowej to miejsce, w którym modelujemy przy pomocy Building Blocks logikę ograniczoną przez Bounded Context.

W tym rozdziale zajmiemy się dalszą strukturyzacją złożonych modeli w warstwie logiki domenowej.

Capability

Poziom Capability zawiera klasy modelujące potencjalne możliwości, jakie oferuje nasz system. W naszym przykładowym systemie umieścimy tutaj agregaty Product i Client oraz Value Object Money. Posiadając produkty, użytkowników i pieniądze, możemy potencjalnie dalej modelować: handel, usługi, reklamacje itd. Na tym poziomie zmiany są relatywnie niezbyt częste.

Operations

Poziom Operations zawiera klasy modelujące konkretne operacje, jakie aktualnie wspiera nasz system. W naszym przykładowym systemie umieścimy tutaj agregaty Order, Invoice oraz Serwis Biznesowy BookKeeper. Składanie zamówień oraz wystawianie na ich podstawie przez księgowego faktur to konkretne operacje, jakie zbudowaliśmy nad modelem potencjału (produktami, klientami i pieniędzmi). Ten poziom jest średnio podatny na zmiany.

Policy

Poziom polityk niejako „dostraja" poziom Operacji. Przykładowo Księgowy (Serwis Domenowy BookKeeper) z poziomu Operacji nalicza podatki na różne sposoby – w zależności od wdrożenia systemu w różnych krajach lub w zależności od tego, kto jest klientem na fakturze. Zwróćmy uwagę, że sposób wybrania polityki (konfiguracja wdrożenia bądź decyzja w runtime) to funkcjonalność aplikacji.

Obiekty na poziomie polityk modelują wariacje operacji biznesowych. Model ten jest mocno podatny na zmiany.

Warto zauważyć, że polityki na poziomie technicznym to technicznie rzecz biorąc domknięcia Operacji. Domknięcia w sensie popularnych od pewnego czasu języków funkcyjnych. Natomiast w starszych językach implementujemy je poprzez Wzorzec Projektowy Strategii (zob. część I).

Decission Support

Niektóre systemy posiadają rodzaj „sztucznej inteligencji" - mniej lub bardziej wyrafinowane mechaniczny analityczne wspierające lub wręcz podejmujące decyzje.

W naszym przykładowym systemie mógłby być to model sugerowania zamienników produktów w razie ich braku – sugestie na postawie analizy behawioralnej klienta, jego znajomych lub wszystkich klientów (polityka!). Innym przykładem jest model dobierania rabatu (jeżeli klientowi przysługuje wiele rabatów, np z uwagi na: pozycję zamówienia, zawartość całego zamówienia, historię zamówień klienta oraz dodatkowe rabaty: dla VIPów, z okazji zimy itd.). Model dobierania rabatów może być dostrojony (policy!) tak, aby działał na korzyść klienta lub właściciela systemu...

GRANICE AGREGATÓW – MODELUJEMY NIEZMIENNIKI

Na zakończenie omawiania zaawansowanego modelowania DDD poruszymy zagadnienie określania granicy Agregatów. Jest to najważniejszy aspekt modelowania DDD, który decyduje o powodzeniu lub klęsce modelowania. Nieodpowiednio „zakreślone" granice Agregatów powodują powstanie modeli słabo podatnych na zmiany oraz nieefektywnych z wydajnościowego punktu widzenia.

W części pierwszej przyjęliśmy dosyć intuicyjne granice naszych Agregatów, natomiast teraz zastanowimy się bardziej świadomie nad ich modelem.

Czytelnicy doświadczeni w modelowaniu DDD mogli zwrócić uwagę na przykładowy Agregat Order, którego granica była zakreślona dosyć „chciwie", co mogło potencjalnie powodować problemy z utrzymaniem i wydajnością modelu.

Agregat w definicji DDD jest spójną jednostką zmiany (pracy). Od strony praktycznej Agregat powinien modelować i enkapsulować w swym wnętrzu niezmienniki. Przykładowo, jeżeli model domenowy zakłada, że zawsze:

$$a + b = c$$

to wówczas Agregat powinien enkapsulować a, b, c oraz zapewniać, że po wywołaniu każdej metody Agregatu niezmiennik jest zachowany. W naszym przykładowym Agregacie Order modelujemy niezmienniki: 1) dodatnie/usunięcie produktu powoduje przeliczenie ceny po rabatach, 2) dodatnie produktu już istniejącego nie powoduje pojawienia się nowej pozycji zamówienia, a jedynie zwiększenie ilości na już istniejącej pozycji.

Technika analizy przypadków użycia

Dosyć łatwo możemy doprowadzić do nieodpowiedniego modelowania granicy Agregatów, jeżeli zastosujemy klasyczne podejście polegające na grupowaniu rzeczowników w „worki". Przykładowo, jeżeli odnajdziemy w modelu rzeczownik Zamówienie, to „wrzucamy" do niego kolejne, „mniejsze" rzeczowniki, tworząc zbyt duży agregat.

Techniką, która sprawdza się lepiej w DDD, jest powstrzymanie się od wstępnego grupowania pojęć w tego typu „worki" do momentu analizy przypadków użycia/operacji domenowych i grupowania ich pod kątem spójnych jednostek zmiany.

PODSUMOWANIE – O CZYM NIE POWIEDZIELIŚMY

Omówiliśmy kolejne poziomy strategicznego modelowania domeny, począwszy od Destylacji Core Domen, czyli miejsca, gdzie stosujemy techniki DDD, poprzez wyłanianie na podstawie Ubiquitous Language granic modelu – Bounded Context, aż po granice poszczególnych Agregatów.

Poruszyliśmy również zagadnienie struktury wielkoskalowej modelu, gdzie separujemy potencjał modelu od konkretnych operacji i modeli wspierających decyzje oraz dostrajających je polityk. Kluczem nadawania tej struktury jest zakres odpowiedzialności oraz podatność na zmiany.

Warto zawrócić uwagę, iż zbyt chciwie zakreślone Agregaty powodują następujące problemy:

- efektywne pobieranie danych – Lazy Loading rozwiązuje jedynie część z nich
- równoległym dostępem do danych – im więcej danych w agregacie, tym większe prawdopodobieństwo, że wielu użytkowników będzie miało interes w jego równoległej modyfikacji; Optimistic Locking jedynie chroni nas przed konsekwencjami, nie rozwiązując problemu
- skalowanie – jesteśmy zmuszeni, aby cały Agregat persystować w tym samym modelu danych oraz na tej samej maszynie (zakładając, że rozproszone transakcje są błędem w sztuce projektowania skalowalnych systemów)

Zainteresowanych tą tematyką odsyłam do pierwszej pozycji w ramce „w sieci".

W SIECI:

- strategiczne modelowanie agregatów: http://dddcommunity.org/library/vernon_2011
- oficjalna strona DDD http://domaindrivendesign.org
- wstępny artykuł poświęcony DDD http://bottega.com.pl/pdf/materialy/sdj-ddd.pdf
- przykładowy projekt: http://code.google.com/p/ddd-cqrs-sample/
- sagi w NServiceBus http://www.nservicebus.com/sagas.aspx

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).