Sławomir Sobótka

Domain Driven Design krok po kroku
Część I: Podstawowe Building Blocks DDD

W niniejszym artykule zostaną przedstawione podstawowe Building Blocks DDD wraz z przykładami ich implementacji w Javie z wykorzystaniem Spring i JPA.

Domain Driven Design jest zestawem technik i koncepcji służących do projektowania złożonych modeli biznesowych i ich dosłownej implementacji. Niektórzy – tak jak Martin Fowler – preferują stosowanie DDD również w prostszych przypadkach z uwagi na jego elegancję i implikacje techniczne w aspektach testowalności i rozszerzalności.
Techniki DDD zyskały swą popularność w roku 2003, wraz z publikacją książki Erica Evansa. Mimo upływu lat koncepcja DDD jest wciąż żywo rozwijana i wzbogacana o nowe techniki – do których sięgniemy w kolejnych odsłonach serii.

Artykuł jest pierwszym z serii tekstów mających na celu szczegółowe przedstawienie kompletnego zestawu technik modelowania oraz nakreślenie kompletnej architektury aplikacji wspierającej DDD.

Przed dalszą lekturą zachęcam do zapoznania się z artykułem stanowiącym wstęp do DDD „Domain Driven Design – Sposób na projektowanie złożonych modeli biznesowych" – link do tekstu znajduje się w ramce „w sieci". Tekst ten zawiera opis kluczowych koncepcji DDD, których poznanie jest krytyczne dla zrozumienia intencji rozwiązań technicznych oraz podejść, które będą prezentowane w naszej serii „krok po kroku".

W kolejnych częściach będziemy poznawać techniki pokrywające większość kluczowych artefaktów i aktywności procesu wytwórczego, pozwalające na przeprowadzenie kompletnego projektu opartego o DDD:

- **Część 2**: Zaawansowane modelowanie DDD, 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 2.0.

PROJEKT REFERENCYJNY

Wszystkich tych czytelników, którzy chcieliby zapoznać się z kolejnymi zagadnieniami serii, zapraszam do strony projektu „DDD&CqRS Laven", której adres znajduje się w ramce „w sieci". Znajdziecie tam kompletną implementację projektu, zawierającą przykłady rozwiązań zawarte w niniejszym tekście, jak i kolejnych częściach serii.

Motto projektu brzmi: „Więcej niż jedynie przykład, ale zdecydowanie nie jest to kolejny frameowork... zaczyn – coś, z czego robi się dobry chleb".

Laven to autorska koncepcja projektu, który zawiera rozwiązania niemal gotowe do stosowania w systemach produkcyjnych, jednak nie hermetyzuje ich w klasycznej formie frameworka. Z założenia wszelkie kastomizacje mogą być dokonywane na „żywym" kodzie, zamiast zmagania się z konfiguracją sztywnych struktur frameworka.

Założenia projektu referencyjnego:

- Prezentacja wszystkich Building Blocks DDD w niestrywializowany sposób;
- Prezentacja technik DDD (np. Bounded Context);
- Prezentacja rzeczywistych technik implementacji, gotowych do wdrożenia w kodzie produkcyjnym;
- Prezentacja pragmatycznego podejścia do implementacji CqRS;
- Dostarczenie rzetelnie wykonanego, wzorcowego kodu źródłowego;
- Przyjęto nieinwazyjną filozofię - ograniczenie wpływu technologii na kształt projektu;
- Opracowany styl architektoniczny jest przenośny na inne frameworki i platformy;

Przykłady podejścia do testowania jednostkowego i akceptacyjnego - z wykorzystaniem Behavior Driven Development (JBehave, Selenium, model Agentów);

Projekt jest całkowicie darmowy – zarówno kod, jak i dokumentacja.

Tłem technicznym projektu jest Java. Wszystkie techniki zostały zilustrowane na dwóch stosach technologii. Pierwszym z nich jest Spring (w tym Spring MVC) i JPA. Drugi to Java EE 6 (w tym JSF 2, EJB 3.1, JPA). W tekstach artykułów będziemy posługiwać się pierwszym stosem.

Warto podkreślić, że na poziomie koncepcyjnym będziemy niezależni od technologii. Duża część przykładów kodu jest niezależna od specyfiki platformy i powinna być zrozumiała dla każdego, kto posiada podstawową umiejętność czytania kodu w językach wywodzących się z C++.

OPIS DOMENY

Przykłady zostały osadzone w ramach bardzo uproszczonego systemu klasy ERP. Wybraliśmy kilka stosunkowo dobrze (w rozumieniu: intuicyjnie) znanych domen ERP i zaimplementowaliśmy ich namiastki. Każdy z nas posiada pewną intuicję odnośnie klientów, zamówień, produktów itp., tak więc nie będziemy tracić czasu na ich szczegółowe wyjaśnianie.

Bounded Context

Techniką Bounded Context zajmiemy się szczegółowo w drugiej części naszej serii, natomiast na tym etapie zakładamy, że istnieją trzy moduły:

- **Sprzedaż** - obsługuje zamawianie produktów, obliczania rabatów, fakturowanie, analizę trendów sprzedaży.
- **CRM** - obsługuje zarządzanie relacjami z klientem. Klient w tym module jest modelowany jako inny artefakt (innego ograniczonego kontekstu) niż w module Sprzedaż.
- **Magazyn** – obsługuje przechowywanie, pakowanie i wysyłkę zamówień.

Zakładamy, że wiedza domenowa odnośnie wymagań dla każdego z modułów znajduje się w umyśle innego Eksperta Domenowego. Eksperci ci niekoniecznie rozumieją się nawzajem, mimo używania słów, które brzmią tak samo, mają na myśli inne pojęcia.

Zastrzegamy, że przedstawiony model jest mocno uproszczony na potrzeby edukacyjne. Jeżeli są Państwo zainteresowani tematyką modelowania omawianych domen, to odsyłamy do zasobów modeli i archetypów analitycznych.

MODYFIKACJA WARSTWOWEJ ARCHITEKTURY APLIKACJI

Każdy z modułów naszego systemu będzie zaprojektowany wg tego samego stylu architektonicznego – w rozumieniu architektury aplikacji.

W najprostszym ujęciu architektury warstwowej wyróżniamy 3 warstwy: prezentacji, logiki i dostępu do danych. Na marginesie: pamiętajmy, że warstwy (Layers) służą do porządkowania kodu i nie są tym co Poziomy (Tiers) porządkujące infrastrukturę techniczną (klienty, serwery, bazy danych).

Co prawda DDD abstrahuje od prezentacji i persystencji, jednak my zajmiemy się tymi zagadnieniami w kolejnych częściach serii z uwagi na wpływ, jaki wywierają na wydajność i skalowanie.

Modyfikacja wprowadzona przez DDD polega na rozwarstwieniu warstwy logiki na dwie wyspecjalizowane warstwy: logikę aplikacji oraz logikę biznesową.

Logika aplikacji

Logika aplikacji jest cienką warstwą serwisów (rzadziej obiektów stanowych) odpowiadających między innymi za szczegóły techniczne takie jak bezpieczeństwo i transakcje. Najważniejszą odpowiedzialnością tej warstwy jest jednak modelowanie kroków Use Case lub User Story. Scenariusz kroku polega na orkiestracji obiektów domenowych z niższej warstwy.

Serwisy aplikacyjne stanowią niejako API serwera, uruchamiane z klientów webowych, mobilnych lub publikowane jako WebServisy. Serwisy z tej warstwy są nazywane również Operation Script (w odróżnieniu od proceduralnego Transaction Script).

Klasy z tej warstwy zwykle testujemy w podejściu end-to-end, ponieważ mnogość możliwych scenariuszy przejścia praktycznie uniemożliwia wysokie pokrycie testami jednostkowymi – stosunek włożonej pracy do poziomu redukcji ryzyka jest zwykle niekorzystny. Dodatkowym kosztem testów jednostkowych w tej warstwie jest konieczność zaślepiania (fake/stub/mock) dużej ilości zależności. Zagadnieniem testowania lub specyfikowania tej warstwy zajmiemy się w części 5 i 6.

Logika domenowa

Warstwa logiki domenowej modeluje zachowania i reguły obiektów biznesowych. DDD skupia większość technik i uwagi na modelowaniu tej właśnie warstwy. Odpowiedzialność warstwy logiki biznesowej skupia się jedynie wokół modelu biznesu i nie powinna zawierać technikaliów zależnych platformy, serwera lub frameworka. Z technicznego punktu widzenia czyni to ją przenośną oraz, co ważne – testowalną poza ciężkim środowiskiem serwerowym.

Natomiast z analitycznego punktu widzenia kod warstwy logiki domenowej może mieć dosłowne przełożenia na model analityczny – czyli zachowujemy podstawowe założenie DDD: Ubiquitous Language.

Klasy z tej warstwy testujemy jednostkowo. Jest to relatywnie tanie podejście z uwagi na mniejszą ilość zależności. Dążymy do wysokiego pokrycia testami kodu z tej warstwy z uwagi na to, że modeluje on złożone odpowiedzialności biznesowe.

BUILDING BLOCKS DLA WARSTWY LOGIKI DOMENOWEJ

Naszą podróż po krainie DDD odbędziemy w stylu Bottom-Up, rozpoczynając od strony technicznej, tak aby w pierwszej kolejności poznać standardowe „klocki" służące do modelowania domeny. Na bazie tej wiedzy, w części drugiej przejdziemy do zagadnień strategicznego modelowania.

Czytelników wywodzących się ze świata Java EE czeka na wstępie mały szok poznawczy, który może przerodzić się w Dysonans Kognitywny. Buildng Blocks służące do modelowania Domeny to nie tylko Encje – do tego nie będą to encje anemiczne, czyli struktury danych.

Building Blocks DDD są swego rodzaju językiem wzorców, które stanowią „rusztowanie mentalne" dla modelarza. Każdy ze standardowych klocków ma za zadanie uwypuklenie pewnej koncepcji, zgodnie z główną zasadą DDD: „make explicit what is implicit". Czyli modelujmy jawnie i wprost złożone reguły i koncepcje biznesowe.

Przykład Agregatu: Order

```
@Entity
@Table(name = "Orders")
@DomainAggregateRoot
public class Order extends BaseAggregateRoot
{
  public enum OrderStatus {
    DRAFT, SUBMITTED, ARCHIVED
  }

  @ManyToOne
  private Client client;

  /**
  * Sample of Value Object usage
  */
  @Embedded
  private Money totalCost;

  /**
  * Sample of encapsulation - this structure
is hidden
  */
  @OneToMany(cascade = CascadeType.ALL, fetch
= FetchType.EAGER)
  private List<OrderLine> items;

  private Timestamp submitDate;

  @Enumerated(EnumType.STRING)
  private OrderStatus status;

  /**
  * Sample of Policy usage (Strategy Design
Pattern)<br>
  * Policy is injected by Factory/Repo but
can be also changed in business
  * method
  */
  // To be injected by Factory/Repository
  @Transient
  private RebatePolicy rebatePolicy;

  protected Order() {
  }

  /**
  * Meant to be used by factory<br>
  * Notice that Policy is set via setter
because Policy need to be initialised
  * also in Repository
  */
  Order(Client client, Money initialCost,
OrderStatus initialStatus) {
    this.client = client;
    totalCost = initialCost;
    status = initialStatus;

    items = new ArrayList<OrderLine>();
  }

  /**
  * Sample business method that:
  * <ul>
  * <li>hides internal state - OrderLine
  * <li>can veto - if order is not in DRAFT
status
  * <li>not only modifies structure (list) but
also performs logic -
  * calculates total cost
  * </ul>
  */
  public void addProduct(Product product, int
quantity) {
    checkIfDraft();

    OrderLine line = find(product);

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

    recalculate();
  }

  /**
  * Sample business method. Fires Domain
Event
  */
  public void submit() {
    checkIfDraft();

    status = OrderStatus.SUBMITTED;
    submitDate = new Timestamp(System.
currentTimeMillis());

    eventPublisher.publish(new
OrderSubmittedEvent(getEntityId()));
  }

  public void archive() {
    status = OrderStatus.ARCHIVED;
  }

  private void checkIfDraft() {
    if (status != OrderStatus.DRAFT)
      throw new
OrderOperationException("Operation allowed
only in DRAFT status", client.getEntityId(),
getEntityId());
  }

  private void recalculate() {
    totalCost = Money.ZERO;
    for (OrderLine line : items) {
      line.recalculate(rebatePolicy);
      totalCost = totalCost.add(line.
getEffectiveCost());
    }
```

```
  }

  private OrderLine find(Product product) {
    for (OrderLine line : items) {
      if (product.equals(line.getProduct()))
        return line;
    }
    return null;
  }
  /**
  * Sample encapsulation of unstable internal
implementation - assumption:
  * this impl may vary in time. So we use
projection of the internal state of
  * this Aggregate<br>
  * <br>
  * Projection hides internal structure using
Value Objects. Projection is
  * also unmodifiable.
  *
  * @return
  */
  public List<OrderedProduct>
getOrderedProducts() {
    List<OrderedProduct> result = new
ArrayList<OrderedProduct>(items.size());

    for (OrderLine line : items) {
      result.add(new OrderedProduct(line.
getProduct().getEntityId(), line.
getProduct().getName(), line.getQuantity(),
          line.getEffectiveCost(), line.
getRegularCost()));
    }
    return Collections.
unmodifiableList(result);
  }
  /**
  * Sample access to the internal state
(immutable Value Object) - <b>remember to
allow such access
  * only if makes sense</b>, don't do that by
default!<br>
  * <br>
  * Notice that there is no setter!
  *
  * @return
  */
  public Money getTotalCost() {
    return totalCost;
  }
}
```

Reguły i koncepcje, które w klasycznym podejściu znikają głęboko w tysiącach linii kodu gigantycznych serwisów projektowanych zgodnie z najlepszym anty-wzorcem, czyli „Boską klasą".

Agregaty

Agregaty są główną jednostką modelowania w DDD. Technicznie jest to graf obiektów, natomiast koncepcyjnie spójna jednostka pracy/zmiany.

Implementując agregaty, zwracamy uwagę na ich hermetyczność – jeden z głównych wyznaczników kodu obiektowego.

Najważniejszym zagadnieniem podczas projektowania Agregatu jest określenie jego granicy. Modelowanie zbyt dużych agregatów jest główną przyczyną niepowodzeń modeli opartych o DDD. Dodatkowo zbyt duże agregaty powodują implikacje wydajnościowe na poziomie Mapera Relacyjno-obiektowego oraz problemy związane z równoległym dostępem do danych. Strategiami określania granicy agregatu zajmiemy się w kolejnej części naszej serii. Zweryfikujemy wówczas nasz model, poddając pod dyskusję wielkość Agregatu Order (listing obok).

Listing klasy Order ilustruje szereg opisanych poniżej technik DDD, które zostały zastosowane w celu literalnego oddania wiedzy Eksperta Domenowego, utrzymana Ubiquitous Language oraz przygotowania modelu na rozbudowę przy minimalnym impakcie na kod innych klas.

Enkapsulacja Agregatu

Warto zwrócić uwagę na fakt, iż Agregat nie pozwala na modyfikację wszystkich swych pól przez settery. Tego typu modyfikacje mogą mieć owszem sens, ale tylko w niewielu przypadkach.

Stan Agregatu zmieniany jest za pomocą metod biznesowych oddających słownictwo Eksperta Domenowego. Niektóre metody biznesowe mogą nie być dozwolone w pewnym stanie Agregatu (np. Order.submit ()) - dlatego rzucają wyjątki (błędy) domenowe.

Poza tym nie wszystkie gettery mają sens – część pól jest szczegółem implementacyjnym.

Zauważmy, że metoda naszego Aggregate: addProduct przyjmuje jako parametr encję Product i ukrywa istnienie lasy OrderLine, która jest szczegółem wewnętrznej implementacji.

W tym konkretnym przykładzie zakładamy, że model Zamówienia jest niestabilny - oczekujemy zmian w najbliższej przyszłości. Załóżmy, że Ekspert domenowy nie jest pewien modelu pozycji na zamówieniu. Dlatego chcemy zmniejszyć obszar katastrofy wynikającej ze zmian Agregatu do niego samego.

Dlatego nie ujawniamy poza Agregat Order modelu zawartego w klasie OrdelLine. Wprowadzamy ValueObject o nazwie OrderedProduct jako „adapter", który niejako rzutuje wewnętrzny model Agregatu na świat zewnętrzny.

Zauważmy, że metoda Order.getOrderedProducts dokonuje transformacji wewnętrznego (z założenia niestabilnego) modelu pozycji do „zewnętrznego interfejsu" OrderedProduct. Metoda zwraca niemodyfikowalną listę, ponieważ operacje na kolekcji będącej „projekcją" nie mają żadnego sensu.

Inne odpowiedzialności Agregatów

W kolejnych częściach naszej serii zapoznamy się z zaawansowanymi odpowiedzialnościami Agregatów takimi jak generowanie Zdarzeń Biznesowych. Pociągnie to za sobą konieczność wstrzykiwania zależności do wnętrza Agregatów.

Przykład Value Object: Money

```
@SuppressWarnings("serial")
@Embeddable
public class Money implements Serializable {
  public static final Currency DEFAULT_
CURRENCY = Currency.getInstance("EUR");

  public static final Money ZERO = new
Money(BigDecimal.ZERO, DEFAULT_CURRENCY);

  private BigDecimal value;

  private String currencyCode;

  public Money(double value, Currency
currency) {
    this(new BigDecimal(value), currency.
getCurrencyCode());
  }

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof Money) {
      Money money = (Money) obj;
      return compatibleCurrency(money) &&
value.equals(money.value);
    }
    return false;
  }

  public Money multiplyBy(double multiplier)
{
    return multiplyBy(new
BigDecimal(multiplier));
  }

  public Money add(Money money) {
    if (!compatibleCurrency(money)) {
      throw new
IllegalArgumentException("Currency
mismatch");
    }

    return new Money(value.add(money.value),
determineCurrencyCode(money));
  }

  public String getCurrencyCode() {
    return currencyCode;
  }

  public boolean greaterThan(Money other) {
    return value.compareTo(other.value) > 0;
  }

  @Override
  public String toString() {
    return String.format("%0$.2f %s", value,
getCurrency().getSymbol());
  }
}
```

Agregat jako Maszyna Stanów

Możemy rozszerzyć model Agregatu o wprowadzenie Wzorca Projektowego State Machine. Jest to silna technika służąca do modelowania problemów o następującej charakterystyce:

- obiekt może być w wielu stanach (w systemach biznesowych są zwykle związane z pewnymi statusami domeny);
- obiekt wykonuje operacje, których sposób wykonania zależy od aktualnego stanu;
- oczekujemy wprowadzenia w przyszłości nowych stanów - otwieramy nasz projekt na rozbudowę;
- nie spodziewamy się, wprowadzając wielu nowych operacji - ponieważ utrzymanie stanu może być kłopotliwe.

Przykład uzasadnionego wprowadzenia Maszyny Stanów: w module Magazynowym mamy Agregat modelujący Paczkę. Paczka zawiera szereg metod biznesowych związanych z dodawaniem do niej zawartości, wysyłką i odbiorem. Zakładamy, że zakres odpowiedzialności tegoż Agregatu jest raczej stabilny i nie spodziewamy się wielu nowych metod.

Paczka może istnieć w wielu stanach (zlecona, pakowana, wysłana, odebrana,...) i spodziewamy się, że zbiór możliwych stanów jest różny, w zależności od wdrożenia systemu u klientów.

Value Objects

Value Objects są zwykle niedocenianym elementem modelu domenowego. Warto z nich korzystać, ponieważ mimo prostoty wnoszą dużą wartość do modelu – mianowicie zwiększają siłę wyrazu Ubiquotus Language, pomagając w enkapsulacji detali (listing obok).

Listing klasy Money przedstawia przykłady Value Object, którego zadaniem jest opakowanie typów „technicznych" i nadanie im znaczenia (metod) biznesowego. Przykładowo bez stosowania VO technicznie możliwe jest pomnożenie 5zł przez 2zł, ale w wyniku otrzymamy bezsensowną wartość 10zł w kwadracie. Zwróćmy uwagę, że klasa Money pozwala dodać Money, ale mnożenie jest możliwe przez wartość bezwymiarową. Jak rozumieć zwiększenie kwoty pieniędzy w przypadku wartości ujemnych? Decyduje o tym Ekspert Domenowy, a model oddaje te reguły w swych metodach.

VO dodatkowo dokonują walidacji wartości, które są przez nie opakowywane. Przykładowo wartość prawdopodobieństwa spoza zakresu <0, 1> nie ma sensu.

VO z technicznego punktu widzenia są immutable. Dlatego mogą być bezpiecznie zwracane z wnętrza agregatów jako wartości do odczytu. Na listingu klasy Order mieliśmy do czynienia z przykładem wykorzystania VO OrderedProduct jako nośnika danych z wnętrza Agregatu.

Z punktu widzenia implementacji VO są pomocne w redukcji code smell o nazwie „primitive obsession". Smell ten polega na posługiwaniu się zbyt niskim poziomem abstrakcji (typami technicznymi), co prowadzi do pojawiania się

długich list parametrów i tak zwanych „Data clumps". Przykładowo dodanie do siebie dwóch kwot, w dwóch walutach po kursach obowiązujących w danym przedziale dat może wyglądać w następujący sposób: add(BigDecimal, String, Date, Date, BigDecimal, String, Date, Date).

Fabryki

Fabryki odgrywają w DDD rolę obiektów domenowych, ponieważ spoczywa na nich część logiki biznesowej. Ich zadaniem jest budowanie Agregatów na podstawie „półproduktów". Fabryki dokonują również walidacji półproduktów, dzięki czemu zakładamy, że istniejące w pamięci Agregaty są w poprawnym stanie (listing poniżej).

Przykład Fabryki

```
@DomainFactory
public class OrderFactory {
  @Inject
  private RebatePolicyFactory
rebatePolicyFactory;
  @Inject
  private InjectorHelper injector;

  public Order crateOrder(Client client)
throws OrderCreationException {
    checkIfclientCanPerformPurchase(client);

    Order order = new Order(client, Money.
ZERO, OrderStatus.DRAFT);
    injector.injectDependencies(order);

    RebatePolicy rebatePolicy =
rebatePolicyFactory.createRebatePolicy();
    order.setRebatePolicy(rebatePolicy);

    addGratis(order, client);

    return order;
  }

  private void checkIfclientCanPer
formPurchase(Client client) throws
OrderCreationException {
    if (client.getEntityStatus() !=
EntityStatus.ACTIVE)
      throw new OrderCreationException("Can
not perform purchase, because of the Client
status: "
        + client.getEntityStatus(), client.
getEntityId());
  }
```

Listing klasy OrderFactory przedstawia przykładową Fabrykę, która tworzy zamówienie dla Klienta, sprawdzając uprzednio, czy klient ten może dokonywać zakupów.

Fabryka bierze na siebie również odpowiedzialność wstrzykiwania zależności do Agregatu. W naszym przykładzie Agregat potrzebuje do pracy Polityki Rabatowania. Fabryka wylicza konkretny typ Polityki na podstawie reguł biznesowych biorących pod uwagę historię zamówień klienta. Dzięki takiemu podejście zmniejszamy coupling Agregatu Order. Klasa Order posiada jedynie zależności do domenowych obiektów współpracujących i nie musi być zależna od obiektów technicznych. Zależności techniczne przechodzą na klasę Fabryki.

W teorii couplingu mamy 3 poziomy zależności: create, contain, call. Fabryka bierze na siebie (odciążając Agregat) najbardziej szkodliwą zależność: create oraz w niektórych przypadkach contain.

W rezultacie znacznie zwiększamy testability klasy Order.

Zagadnienia zwiększania testowalności Agregatów przez wprowadzanie ich fabryk zostaną szerzej poruszone w jednej z kolejnych części poświęconej testowaniu automatycznemu, natomiast zagadnienia wstrzykiwania poruszmy w części poświęconej architekturze aplikacji.

Repozytoria

Repozytoria są abstrakcją persystencji dla Agregatów. Repozytorium odpowiada za pobranie i utrwalenie konkretnego Agregatu. Repozytoria, w odróżnieniu od DAO, nie są obarczone dziesiątkami metod wyszukujących, tworzonych na potrzeby ekranów. Repozytorium może zawierać metody wyszukujące (OrderRepository.findUncorfirmed), ale tworzone jedynie na potrzeby logiki biznesowej. Natomiast wyszukiwania na potrzeby ekranów należą do dedykowanych serwisów wyszukujących, którymi zajmiemy się w części poświęconej architekturze aplikacji.

```
@DomainRepository public interface OrderReposi-
tory {
     public Order save(Order order);
     public Order load(Long orderId);
     public List<Order>findUncorfirmed(Client
client)
}
```

Listing interfejsu OrderRepository ilustruje przykładowy kontrakt Repozytorium. Przykładową implementację z wykorzystaniem JPA oraz opartą o idiom Generic Repository i klasę Bazowego Agregatu przedstawimy w części poświęconej szczegółom implementacji na platformie Java EE. Warto dodać, że w systemach budowanych jako wartość dodana ponad istniejącymi systemami może dojść do sytuacji, w której Repozytorium danego Agregatu pobiera jego części z kilku DAO (kilka systemów, baz, web serwisów) i dokonuje ich złożenia w nowy koncept biznesowy.

Serwisy Domenowe

Paradygmat obiektowy nie jest adekwatny do każdego modelu. Zwykle część modelu domenowego to procedury mające na celu np. transformację jednych Agregatów w inne.

Z tego powodu w DDD istnieje Building Block modelujący tego typu operacje – są to Serwisy Domenowe. W odróżnieniu od Serwisów Aplikacyjnych (o których za chwilę) Serwisy Domenowe operują na poziomie logiki biznesowej.

Przykład Serwisu Domenowego

```
@DomainService
public class InvoicingService {
  @Inject
  private ProductRepository
productRepository;

  public Invoice issuance(Order order,
TaxPolicy taxPolicy){
    //TODO refactor to InvoiceFactory
    Invoice invoice = new Invoice(order.
getClient());

    for (OrderedProduct orderedProduct :
order.getOrderedProducts()){
    Product product = productRepository.
load(orderedProduct.getProductId());

    Money net = orderedProduct.
getEffectiveCost();
  Tax tax = taxPolicy.calculateTax(product.
getType(), net);

    InvoiceLine invoiceLine = new
InvoiceLine(product, orderedProduct.
getQuantity(), net, tax);
    invoice.addItem(invoiceLine);
    }
    return invoice;
    }
}
```

Listing klasy InvoicingService jest modelem „księgowego" w domenie zamówień. Jego zadaniem jest wygenerowanie Faktury na podstawie Zamówienia, biorąc pod uwagę Politykę Podatkową.

Zwróćmy uwagę na fakt, iż nasz księgowy nie jest obiektem persystentnym. Mimo tego traktujemy go jako obiekt domenowy.

Księgowy enkapsuluje reguły związane z fakturowaniem; część z reguł jest specyficzna dla wdrożenia w danym kraju, dlatego została wyniesiona do Polityk Podatkowych.

Warto zwrócić uwagę na decyzję projektową polegającą na stworzeniu komponentu Księgowego. Bardzo złym pomysłem byłoby obarczanie Agregatu Order metodą issueInvoice() i podobnymi, ponieważ z czasem zamówienie stałoby się „Boską klasą".

Księgowy operuje na „projekcji" z wnętrza Agregatu Order, czyli na VO OrderedProduct, dzięki czemu nie jest wrażliwy na zmiany w implementacji Agregatu.

Polityki

Z technicznego puntu widzenia polityki są implementowane jako Strategy Design Pattern, wnosząc pierwiastek funkcyjności do języków obiektowych.

Koncepcyjnie Polityki są „domknięciem" lub „dostrojeniem" modelu. Pozwalają na rozbudowę modelu bez modyfikacji corowych Agregatów. W dotychczasowych przykładach spotkaliśmy się z Polityką Rabatowania (RebatePolicy) używaną przez Agregat Order oraz Polityką Obliczania Podatku (TaxPolicy) używaną przez księgowego (InvoicingService).

```
@DomainPolicy
public interface TaxPolicy {
    /**
    * calculates tax per product type based on
net value
    */
    public Tax calculateTax(ProductType product-
Type, Money net);
```

Listing interfejsu TaxPolicy ilustruje kontrakt, jaki jest zawierany pomiędzy księgowym (InvoicingService) a implementacjami obliczania podatku dla różnych krajów.

Z punktu widzenia modelarza Polityka jest kolejnym Building Blockiem, który wspiera mantrę „make explicit what is implicit". Obliczanie podatku jest co prawda czynnością, ale tak istotną z puntu widzenia modelu, że znajduje w nim swoje miejsce jako „first class citizen".

Pozostałe Building Blocks

W kolejnych częściach naszej serii omówimy zastosowanie zaawansowanych „klocków" DDD takich jak: Zdarzenia Domenowe, Sagi i Specyfikacje.

MODELOWANIE SCENARIUSZY

Po zapoznaniu się z Building Blocks warstwy logiki domenowej przejdziemy o jedno „piętro" wyżej, do warstwy logiki aplikacji. Jak już wspomniano we wstępie, warstwa ta modeluje kroki Use Case lub User Story. W niniejszym artykule przyjrzymy się implementacji Serwisu Aplikacyjnego w sposób poglądowy. Natomiast w kolejnych częściach serii skupimy się na implementacji (transakcje, bezpieczeństwo), testowaniu end-to-end (również BDD i JBehave) oraz podejdziemy nieco inaczej do implementacji tej warstwy (CommandHandlers i architektura CqRS).

Listing klasy PurchaseApplicationService (obok) przedstawia typowy model kroków Use Case/User Story operujący na modelu domenowym: stworzenie zamówienia, dodanie produktu do zamówienia, zatwierdzenie zamówienia.

Serwis Aplikacyjny:

- integruje wiele zależności (repozytoria, fabryki, serwisy pomocnicze);
- zapewnia transakcyjność i bezpieczeństwo (w tym wypadku poprzez adnotacje i AOP);
- integruje komponenty aplikacyjne (w tym wypadu pracujący w sesji, zalogowany użytkownik);
- orkiestruje obiekty domenowe.

Przykładem orkiestracji wszystkich obiektów domenowych jest metoda approveOrder(id), która kolejno:

- pobiera Agregat Zamówienia z Repozytorium;

Przykład Serwisu Aplikacyjnego

```
@ApplicationService
public class PurchaseApplicationService {
  @Inject
  private OrderRepository orderRepository;
  @Inject
  private OrderFactory orderFactory;
  @Inject
  private ProductRepository
productRepository;
  @Inject
  private InvoiceRepository
invoiceRepository;
  @Inject
  private InvoicingService invoicingService;
  @Inject
  private SystemUser systemUser;
  @Inject
  private ApplicationEventPublisher
eventPublisher;

  /**
  * Sample usage of factory and repository
  *
  * @throws OrderCreationException
  */
  public void createNewOrder() throws
OrderCreationException {
    Client client = loadClient(systemUser.
getUserId());
    Order order = orderFactory.
crateOrder(client);
    orderRepository.persist(order);
  }

  /**
  * Sample call of the domain logic<br>
  * Sample publishing Application (not
Domain) Event
  */
  public void addProductToOrder(Long
productId, Long orderId, int quantity) {
    Order order = orderRepository.
load(orderId);
    Product product = productRepository.
load(productId);

    // Domain logic
    order.addProduct(product, quantity);

    orderRepository.save(order);

    // if we want to Spy Clients:)
    eventPublisher.publish(new
ProductAddedToOrderEvent(product.
getEntityId(), systemUser.getUserId(),
quantity));
  }

  /**
  * Sample of the separation of domain logic
in aggregate and domain logic in
  * domain service
  *
  * @param orderId
  */
  public void approveOrder(Long orderId) {
  Order order = orderRepository.
load(orderId);

    Specification<Order> orderSpecification =
generateSpecification(systemUser);
    if (!orderSpecification.
isSatisfiedBy(order))
      throw new
OrderOperationException("Order does not meet
specification", order.getEntityId());

    // Domain logic
    order.submit();
    // Domain service
    Invoice invoice =
invoicingService.issuance(order,
generateTaxPolicy(systemUser));

    invoiceRepository.save(invoice);
    orderRepository.save(order);
  }

  /**
  * Assembling Spec contains Business Logic,
therefore it may be moved to
  * domain Building Block -
OrderSpecificationFactory
  *
  * @param systemUser
  * @return
  */
  @SuppressWarnings("unchecked")
  private Specification<Order> generateSpecific
ation(SystemUser systemUser) {
    Specification<Order> specification = new Co
njunctionSpecification<Order>(//
      new DestinationSpecification(Locale.
CHINA).not(),// do not send to China
       new ItemsCountSpecification(100),//
max 100 items
       new DebtorSpecification()// not debts
or max 1000 => debtors can
  // buy for max 1000
         .or(new TotalCostSpecification(new
Money(1000.0))));
    // vip can buy some nice stuff
    if (!isVip(systemUser)) {
      specification = specification.and(new
RestrictedProductsSpecification());
    }
    return specification;
  }
}
```

DDD Building Blocks

- Entity – identyfikowalne obiekty zawierające odpowiedzialność biznesową;
- Aggregate – hermetyczne grafy obiektów, z jedną encją będącą „korzeniem agregatu", która stanowi API całości. Agregat jest główną jednostką logiki domenowej w DDD;
- Value Object – wrapper dla typów prostych, nadający im znaczenie biznesowe oraz wygodny interfejs;
- Service – specyficzne operacje, które nie pasują do żądnego agregatu;
- Policy – model wariacji operacji biznesowych, Strategy Design Pattern;
- Specification – model złożonych warunków biznesowych, wywodzi się z Composite Design pattern;
- Event – model wydarzeń biznesowych, może służyć do przetwarzania równoległego lub komunikacji pomiędzy Bounded Context;
- Saga – model złożonego procesu, który stan jest trwały oraz zależy od wielu zdarzeń;
- Factory – tworzy nowe instancje złożonych Agregatów, dbając o ich poprawność. Zwiększa testowalność, biorąc niejako na siebie operatory new;
- Repository – zarządza trwałością Agregatu/Encji.

W sieci

- http://domaindrivendesign.org
  oficjalna strona DDD
- http://bottega.com.pl/pdf/materialy/sdj-ddd.pdf
  wstępny artykuł poświęcony DDD
- http://code.google.com/p/ddd-cqrs-sample/
  przykładowy projekt

- weryfikuje go przy pomocy Specyfikacji Domenowej;
- wykonuje za Zamówieniu operację biznesową submit();
- zapisuje Agregat Zamówienia do Repozytorium;
- generuje Fakturę przy pomocy Księgowego, decydując o użytej przez niego Polityce Podatkowej na podstawie danych o Zalogowanym Użytkowniku;
- zapisuje Fakturę do Repozytorium.

Uogólniając, mamy do czynienia ze stworzeniem/pobraniem „aktorów" biznesowych, wykonaniem przez nich (pomiędzy nimi) scenariusza, utrwaleniem stanu aktorów.

PODSUMOWANIE

Niniejszy artykuł miał na celu zapoznanie czytelników z podstawowymi Building Blocks DDD oraz podstawowymi koncepcjami stojącymi za technikami modelowania DDD. Doświadczeni czytelnicy zapewne zauważyli, iż wszystkie wymienione techniki należą do dobrych praktyk projektowania obiektowego (SOLID, GRASP, RDD). Przykładowy model został zaprojektowany w nieco defensywnym, hermetycznym stylu, tak aby przyszłe modyfikacje nie naruszały zbyt drastycznie całej struktury, a miały jedynie lokalny impakt.

W kolejnych częściach wprowadzimy zaawansowane Building Blocks oraz zaawansowane techniki strategicznego modelowania, co skłoni nas do modyfikacji aktualnego „kształtu" modelu.

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. Do jego zainteresowań należy szeroko pojęta inżynieria oprogramowania, modelowanie, wzorce, zwinne procesy wytwórcze. Hobbystycznie interesuje się psychologią i kognitywistyką. 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).