# Domain Driven Design

## Sposób na projektowanie złożonych modeli biznesowych

Czy zastanawiałeś się co jest przyczyną rozkładu średnich i dużych systemów? Czy jest on nieunikniony i jest jedynie kwestią czasu? Być może istnieje jakiś sposób na utrzymanie entropii w ryzach? Jednak czy pomocny może być nowy język, nowa specyfikacja serwera, nowy framework webowy, nowy kolor karteczek przyklejanych na tablicy, nowe fancy-japońskie słówko?
Na pewno w jakimś stopniu, ale czy w wystarczającym?

W niniejszym artykule przedstawię w jaki sposób Domain Driven Design pomaga w okiełznaniu chaosu na poziomie modelu. Modelu, który jest sercem złożoności w większości systemów klasy biznesowej.
Zagadnienie DDD jest bardzo obszerne, dlatego po krótkim wstępie koncepcyjnym skupimy się na technicznych aspektach implementacji wybranych technik.

Dowiesz się:

- czym jest DDD i kiedy wnosi wartość,
- w jakim kontekście warto aplikować podejście DDD – do jakich klas problemów go stosować
- poznasz podstawowe założenia i techniki DDD: Ubiquitous Language, Bounded Context, Strategic Design, Architektura z rozwarstwieniem logiki.
- poznasz na przykładach kodu standardowe „klocki" (Building Blocks), których używamy do projektowania modeli biznesowych

Powinieneś wiedzieć:

- podstawy projektowania obiektowego
- architektury warstwowe
- znajomość podstaw Javy (lub innego języka składniowo wywodzącego się z C++) na poziomie czytania

### Projektowanie ukierunkowane na domenę

Rzecz będzie o projektowaniu – modelowaniu. Projektowaniu ukierunkowanym, czyli o skupieniu na pewnych aspektach, o wyborach, o ustaleniu tych a nie innych priorytetów. Domena, czyli pewna sfera wiedzy i aktywności, obszar tematyczny, do którego użytkownik aplikuje używane oprogramowanie. Na przykład domena fakturowania, rozliczania pacjentów, obiegu dokumentów, itd.

Domain Driven Design nie jest technologią ani metodyką. Jest to sposób myślenia, ustalania uwagi i priorytetów mających na celu przyśpieszenie (lub w ogóle umożliwienie powstania) projektów zmagających się ze złożoną domeną.

DDD jako nazwa zyskało popularność w 2003 roku po publikacji książki Erica Evansa „Domain-driven Design: Tackling Complexity in the Heart of Software". Od tej pory podlega nieustannemu rozwojowi zarówno od strony koncepcyjnej jak i – w o wiele większym stopniu – od strony technicznych sposobów na implementację założeń.

### Realia projektowe

Pracując nad projektami – szczególnie rozległymi aplikacjami biznesowymi, możemy spotkać się z pewnymi typowymi, powtarzalnymi symptomami. Aplikacja z czasem staje się mało podatna na zmiany. Jej struktura jest krucha, tak więc każda zmiana pociąga za sobą katastrofę o mniejszym lub większym zasięgu. Kod charakteryzuje się strukturą spaghetti, dlatego jego zrozumienie i przełożenie nowych wymagań na implementację wiąże się z dużym wysiłkiem umysłowym.

Przewrotnie można uznać takie stan za naturalny, wszak II Prawo Termodynamiki mówi, że entropia nie maleje...

## What drives design?

Mogłoby się wydawać, że ukierunkowanie projektowania na domenę jest naturalne w projektach wspierających działalność biznesu, jednak czasem bywa zgoła inaczej. Czasem skupiamy się na tak zwanym onanizmie technicznym szukając rozwiązania problemów w takich miejscach jak:

- Kolejny framework webowy – jeszcze kilka lat temu w świecie Javy co miesiąc wychodził nowy framework webowy będący wariacją na temat MVC. Natomiast aktualnie pojawiają się wariacje na temat zaokrąglania rogów na stronach lub na temat generatora aplikacji klasy CRUD.
- Nowa specyfikacja platformy lub serwera
- Poszukujemy nowego języka programowania, którego składania może nie jest do końca elegancka (ba, wygląda jak efekt skakania kota po klawiaturze) ale za to pozwala nam za zapisanie iteracji po kolekcji w jednej linijce lub zwalnia z konieczności pisania getterów i setterów – tak, to wiele zmienia...
- Próbujemy przykryć chaos zdalnymi procedurami – SOA
- Wreszcie w akcie rozpaczy udajemy się do szamana, który przy pomocy modnego akurat w tym miesiącu japońskiego słówka sprzeda nam sposób na tworzenie listy TODO za pomocą karteczek przylepianych na tablicy, tudzież bardziej przewidywalnego dostarczania kodu spaghetti przy pomocy liczb Fibonacciego.

W popularno-naukowych książkach z zakresu psychologii powtarza się ta sama anegdota: środek nocy, rozjuszony mężczyzna grasuje po kuchni; żona pyta go: Co robisz? On odpowiada: szukam kluczy! Żona: gdzie je ostatnio miałeś? Mąż: na podjeździe do garażu. Żona: to dlaczego szukasz w kuchni? Mąż: bo tu jest więcej światła.

Nie jest to dowcip z wachlarza żartów Korola Strasburgera, lecz anegdota pokazująca jak działa ludzki mózg – skłania się on do poszukiwań w znanych obszarach, tak aby nie wychlać się ze „strefy komfortu".

Poznanie nowego języka lub frameworka webowego jest relatywnie proste w stosunku do zmiany wyuczonego podejścia do problemu.

## Sedno problemu

Zanim przejdziemy dalej, musimy zatrzymać się na krótki akapit teorii. Warto uświadomić sobie, że istnieją dwa rodzaje złożoności.

Esencjonalna – zależy od problemu, wynika z jego natury i jest nieunikniona. Przykładowo problem wystawienia faktury, to (w zależności od klasy systemu) kilka do kilkuset kroków, których nie da się uprościć bez ograniczania funkcjonalności.

Przypadkowa – zależy do rozwiązania, wynika z wybranego podejścia. Można oczywiście dowolnie jest rozbudowywać i komplikować.

Wszyscy chyba intuicyjnie czujemy, że moment, w którym złożoność wymyka się z rąk jest momentem, w którym projekt zaczyna upadać.

DDD skupia się na modelowaniu złożoności. Jeżeli jakaś domena jest złożona, to należy podejść do niej z należytą uwagą i modelować tę złożoność, zamiast pozostawiać ją ślepemu losowi. Jeżeli nie modelujemy złożoności, to pozostawiamy ją pokrętnym kłębowiskom Ifów, schowanych gdzieś w linijkach o numerze 4000 do 6000 w serwisie – ośmiotysięczniku.

## Problemy z modelem

DDD adresuje problemy o złożonym modelu. Oczywiście nie każdy system charakteryzuje się wysoką złożonością modelu. Przykładowo platforma do publikowania blogów ma relatywnie prosty model (posty, komentarze, autorzy, targi, itd.), po prostu w tej klasie problemów wyzwaniem jest np. skalowanie lub ergonomia.

### Niespójny żargon

Jeżeli ekspert domenowy (osoba/osoby, które rozumieją domenę) posługują się innym żargonem niż zespół projektowy, to zachodzi potrzeba translacji. Psychologia komunikacji mówi jasno, że to nie może się udać. Wszyscy znamy zabawę z dzieciństwa w „głuchy telefon". Z tym, że w tej zabawie było o tyle łatwo, że nie występowały pojęcia ze specjalistycznych domen.

### Brak modelu lub model tylko na początku

Jeżeli nie mamy modelu, to dodajemy po prostu kolejne features do aplikacji. grawitacja jest nieubłagana i kula błota (słynna Big Ball of Mud), będzie się toczyć w dół nieuchronnie się powiększając.

Czasem bywa tak, że mamy model w początkowej fazie projektu (wtedy gdy tak naprawdę jeszcze nic nie rozumieliśmy), ale nie utrzymujemy go na dalszych etapach. Prowadzi to do istnienia mylącej, niespójnej i nieadekwatnej dokumentacji. Oczywiście zgodnie z zasadą Eisenhowera: „na wojnie posiadanie planów jest bezużyteczne, ale planowanie jest nieodzowne" dobre jest i to.

### Model czysto analityczny

W projektach prowadzonych metodyką, w której istnieje rola analityka, czasem bywa tak, że opracowany model

jest nieimplementowalny. Dochodzi do sytuacji, w której po dostarczeniu przez analityka artefaktów analitycznych, team developerski musi dokonać na nich niejako ponownego procesu analizy. Przyczyn może być wile (np. analiza na zasadzie „dyktafonu") ale nie będziemy ich w tym tekście poruszać. Warto zadać sobie pytanie po co właściwie przeprowadzamy tą analizę. Czy celem nie jest powstanie kodu pracującego np. na serwerze, który to kod robi lub oszczędza pieniądze? Wszystkie wysiłki umysłowe powinny dążyć do tego celu – powstania działającego kodu – później będą jedynie „odpadem produkcyjnym".

### Ewolucja modelu

Model powinien oczywiście ewoluować w czasie. Jednak musimy mieć na uwadze słowa eksperta od ewolucji – Richarda Dawkinsa: „procesy ewolucyjne są ślepe, dążą do lokalnego maximum, z którego czasem nie mogą się wydostać".

Zbyt naiwny model, pozostawiony sam sobie, zabrnie w ślepą uliczkę - zgodnie z prawem ewolucji.

## Model w ujęciu DDD

### Model wymaga opieki

Model w ujęciu DDD jest niczym drzewko bonzai. Wymaga nieustannej opieki i pielęgnacji. DDD aplikuje się do projektów, w których model:

- jest powodem tworzenia projektu
- jest sercem systemu - jego największą wartością
- dlatego, że np. daje klientowi (jego organizacji) przewagę nad konkurencją
- a modelowanie jest największym wyzwaniem w projekcie

W poprzedniej wspomniałem, że tylko niektóre klasy problemów spełniają te warunki – do pozostałych DDD się po prostu nie aplikuje.

### Bada danych nie jest modelem

Gdy wchodzimy do złożonego projektu zaczynamy od zapoznania się z modelem. Zwykle wówczas na prośbę o przedstawianie modelu dostajemy schemat bazy danych z kilkuset/kilku tysiącami tabelek.

W podejściu DDD schemat bazy nie jest modelem ponieważ jest to struktura statyczna, w której nie widać reguł działania i zachowania systemu. Z punktu widzenia DDD baza jest tylko repozytorium na stan Obiektów Biznesowych.

Dodatkowo schemat bazy powstaje zwykle zbyt wcześnie, na etapie gdy jeszcze nic nie wiemy o projektowanej domenie i jedynie wydaje się nam, że ją rozumiemy. Kolejne miesiące projektu upływają nam na hackowaniu błędnego schematu bazy, obchodzeniu niepoprawnych modeli.

### Do czego służy model

Model w ujęciu DDD służy do nieustannego zbieranie wiedzy o regułach i zachowaniu. Model zawiera wspólny żargon. DDD jest wspiera techniki zwinne, dlatego wiedzę „kruszymy" (strategia knowledge crunching) w kolejnych iteracjach. Model w DDD jest przygotowany do rozbudowy i uszczegóławiania.

Model w DDD powinien stanowić wspólną płaszczyznę porozumienia pomiędzy Ekspertem (ew. ekspertami) Domenowym a zespołem projektowym. Dosyć dobrą metaforą jest papierowa makieta budynku lub gliniany prototyp karoserii samochodu. Pozwalają one wypowiedzieć się każdemu uczestnikowi projektu na temat „domeny" ponieważ są zrozumiałe dla każdego z nich.

### Czym jest model

Model w DDD jest bazą wiedzy o regułach i zachowaniu. Nie jest to model rzeczywistego świata a jedynie uproszczenie na bieżące potrzeby, bez nieistotnych szczegółów i technikaliów. Unikamy przeładowania mentalnego.

### Wszędobylski język

Jeżeli ekspert domenowy nie rozumie modelu, wówczas projekt od samego początku jest „marszem ku klęsce" (zobacz: Edward Yourdon „Death march"). DDD opisuje szereg technik służących do osiągnięcia tak zwanego Ubiquitous Language (wszędobylskiego języka), którym posługują się wszyscy uczestnicy projektu. Nie trzymajmy się UMLa, jeżeli nie przemawia on do Eksperta Domenowego. Nie obawiajmy się wprowadzać własnych symboli. Jeden z Ekspertów Domenowych, z którym miałem przyjemność niegdyś pracować zaczął „czuć wspólny model" dopiero wówczas, gdy sam rysował elementy bardzo uproszczonej notacji UML w Excelu (np. klasy i powiązania pomiędzy nimi dało się osiągnąć jako odpowiedni styl border dla grupy komórek).

Pamiętajmy też, że szeregowi użytkownicy systemu myślą raczej historyjkami niż abstrakcyjnymi modelami. Stąd też w Behavior Driven Development (o którym w zakończeniu) nastawienie na User Story.

Ważna jest również relacja odwrotna - aby Ekspert Domenowy znał i rozumiał szanse oraz ograniczenia jakie niesie technologia.

### Wspólny poziom abstrakcji

Wspólny (wszędobylski) język niesie ze sobą istotną implikację. W projekcie powinien istnieć jeden, wspólny model domeny. Bez rozróżnienia na różne poziomy abstrakcji (model analityczny, projektowy, develoeprski). W DDD istnieje jeden, wspólny model obiekty domeny.

Oczywiście utrzymanie wspólnego modelu na poziomie analizy i developmentu wymaga specjalnego podejścia. Pamiętajmy, że DDD jako model traktuje coś, co wyraża nie tylko strukturę, ale przede wszystkim dynamikę. Dlatego mówiąc o modelu z punktu widzenia developer-

skeigo skupiamy się jedynie na jednej warstwie - tej, która jest wolna od szczegółów technicznych. Dzięki temu kod z tej warstwy może mieć dosłowne przełożenie na modele analityczne.

## Architektura aplikacji

DDD wprowadza modyfikację do klasycznej architektury warstwowej. W tym miejscu należy uściślić, co rozumiemy pod pojęciem warstwy (ang. *Layer*). Warstwy służą do logicznej separacji kodu na obszary odpowiedzialne za ściśle określone zadania. W klasycznej architekturze warstwowej mamy warstwy: prezentacji, logiki oraz dostępu do danych. Natomiast pod pojęciem poziomu technicznego (ang. *Tier*) rozumiemy np. klienta (przeglądarkę, telefon), serwer aplikacji, serwer bazodanowy, itp. Oba pojęcia Layer i Tier bywają czasem mylone, dlatego przypominamy je dla porządku.

Modyfikacja wprowadzona przez DDD polega na rozwarstwieniu warstwy logiki na dwie wyspecjalizowane warstwy: logikę aplikacji oraz logikę biznesową. Taka decyzja architektoniczna jest podyktowana czystym pragmatyzmem, ponieważ w realnych projektach utrzymanych w klasycznej architekturze warstwowej z jedną warstwą logiki, przybiera ona z czasem formę ośmiotysięczników – nieutrzymywanych serwisów typu „god class" charakteryzujących się powielaniem kodu, wysokim couplingiem oraz strukturą spaghetti.

Ponadto dzięki takiemu podejściu warstwa logiki biznesowej może być skupiona jedynie wokół modelu biznesu i nie musi zawierać technikaliów zależnych platformy, serwera lub frameworka. Z technicznego punktu widzenia czyni ją to przenośną oraz co ważne – testowanlą poza ciężkim środowiskiem serwerowym.

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

### Logika aplikacji

Warstwy logiki aplikacji jest cienką warstwą serwisów lub rzadziej stanowych obiektów. Odpowiada za szczegóły techniczne takie jak bezpieczeństwo czy transakcje oraz co najważniejsze – orkiestruje obiekty domenowe. Orkiestruje, czyli układa ich wywołania w kroki Use Case lub User Story. Czasem spotyka się podejście, gdzie zwrot UseCase występuje w nazwie klasy zamiast ApplicationService. Czasem stosowne jest aby taka klasa była stanowa a jej obiekty żyły w sesji albo konwersacji.

Listing 1 przedstawia przykładowy serwis aplikacyjny realizujący dwie funkcjonalności: dodanie produktu do zamówienia oraz zatwierdzanie zamówienia.

Na marginesie dodam, że mamy tutaj do czynienia z klasycznym projektem – jakimś rodzajem handlu elektronicznego. Domena jest o tyle wdzięczna, że każdy mniej więcej intuicyjnie ją rozumie, dzięki czemu uniknie-my kilku stron wprowadzania w meandry działania banku, szpitala lub fabryki. Niestety z punktu widzenia DDD może wydawać się nieco zbyt prosta aby uwypuklić zasadność jak i korzyści płynące ze stosowania technik modelowania DDD. Liczę jednak na wyobraźnię czytelników, którzy dokonają paraleli omawianego przykładu do złożonych problemów, z którymi na co dzień się zmagają.

Serwis został adnotowany adnotacjami frameworka Spring. Analogicznie możemy postąpić w przypadku zastosowania EJB 3.x lub np. frameworka Seam.

Adnotacje nakazują nałożenie aspektu transakcji oraz bezpieczeństwa na każdą metodę. Oczywiście dobrą praktyką jest stworzenie własnej adnotacji @ApplicationService, która będzie nieść ze sobą szereg standardowych adnotacji, ukrywając szczegóły poszczególnych adnotacji technicznych.

Serwis posiada wstrzyknięte repozytoria zarządzające persystencją obiektów domenowych. Repozytoria zostaną omówione szczegółowo w dalszej części artykułu, póki co wystarczy wiedzieć, że służą one do pobierania oraz zapisywania obiektów – zwykle na podstawie bazy danych.

Pierwsza z metod dodaje do zamówienia o danym ID produkt o danym ID w zadanej ilości. Warto zwrócić uwagę na styl kontraktu jaki „publikuje" warstwa aplikacji. Posługujemy się identyfikatorami, dzięki czemu klient nie imputuje nam Produktu wraz z jego ceną. Cenę produktu pobieramy sobie w bezpieczny sposób z repozytorium.

Scenariusz biznesowy metody addProductToOrder jest bardzo prosty: pobiera ona z repozytoriów dwóch „aktorów" biznesowych (zamówienie oraz produkt), po czym orkiestruje główny scenariusz biznesowy – dodaje produkt do zamówienia. Warto zwrócić uwagę, na fakt, że na poziomie metod modelu domenowego zwykle posługujemy się już obiektami domenowymi a nie ich identyfikatorami.

Druga metoda `submitOrder` orkiestruje już nieco bardziej złożony scenariusz: zatwierdzenie zamówienia z jednoczesnym wygenerowaniem faktury. Serwis pobiera aktora domenowego – zamówienie, po czym wysyła do niego sygnał aby „się zatwierdził". W kolejnym kroku używany jest serwis biznesowy (czyli obiekt z warstwy modelu domenowego, w odróżnieniu od serwisu aplikacyjnego), który zajmuje się transformacją zamówienia do faktury.

Obiekty z warstwy domenowej będą kolejna omawiane w następnym rozdziale.

## Pattern Language dla logiki domenowej

DDD definiuje szereg standardowy „klocków" (Building Blocks), z których modelujemy domenę. W standardowym podejściu jedynym środkiem wyrazu modelu są anemiczne encje, czyli struktury danych, które nie posiadają żadnej aktywności. Natomiast DDD

dostarcza dużo szerszego wachlarza środków wyrazu, które operują na bardziej odpowiednim poziomie abstrakcji.

Koncepcja języka języka wzorców jakim są Building Blocks pochodzi z prac Rebeki Wirfs-Brock poświęconych Responsibility Driven Design. Ogólna koncepcja RDD zakłada, że należy zdefiniować Role jakie mogą grać w systemie obiekty, a do każdej Roli przypisane są odpowiedzialności. Tworząc klasę musimy określić jaką ma grać rolę, czyli jaka jest jej odpowiedzialność. Przykładowo obiekty grające role transformatorów odpowiadają za transformację pewnych obiektów w inne.

Building Blocks to nic innego jak ustalenie standardowych ról jakie mogą być grane przez obiekty wchodzące w skład modelu domenowego.

### Make explicit what is implicit

Ogólna idea jaka przyświeca istnieniu wzorców Building Blocks jest następująca: uwypuklaj i ujawniaj istotne koncepcje modelu domeny zamiast ukrywać je głęboko w implementacji. Modeluj złożoność zamiast zamiatać ją pod dywan i udawać, że nie istnieje – złożoność zamieciona pod dywan „wychodzi bokiem" w jeszcze gorszej postaci. Czyli innymi słowy wszystkie istotne wyrażenia z naszego Ubiquitous Language powinny znaleźć się w modelu.

W kolejnych sekcjach omówimy poszczególne Building Blocks z uwzględnieniem ich siły wyrazu.

### Entity

Encje są tym co znamy z maperów relacyjno-obiektowych – obiektem, który jest identyfikowalny po jakimś kluczu. W DDD encje posiadają jednak metody biznesowe. Oczywiście nie chodzi o to aby całą logikę biznesową „rozsmarować" na encjach. Encje powinny posiadać tylko charakterystyczne dla siebie odpowiedzialności wynikające z ich natury. W naszym przykładzie z Listingu 1 encją jest Produkt.

**Listing 1.** *Serwis Aplikacyjny*

```
@Transactional
@Service
@Secured...
public class OrderApplicationService{
  @Autowired
  private OrdersRepository ordersRepository;
  @Autowired
  private ProductsRepository productsRepository;
  @Autowired
  private InvoicesRepository invoicesRepository;

  public void  addProductToOrder(Long orderId, Long productId, int quantity){
    Order order = ordersRepository.load(orderId);
    Product product = productssRepository.load(productId);

    order.addProduct(product, quantity);

    ordersRepository.save(order);
  }

  public void  submitOrder(Long orderId, Payment payment){
    Order order = ordersRepository.get(orderId);
    order.confirm(payment);

    InvoicingService invoicing = new InvoicingService();//TODO wstrzyknąć aby zwiększyć testability
    Invoice invoice invoicing.generateInvoice(order);

    ordersRepository.save(order);
    invoicesRepository.save(invoice)
  }
}
```

### Agregate

Agregaty to z technicznego punktu widzenia grafy encji, w których jedna encja jest główna – jest tak zwanym Aggregate Root i zawiera metody biznesowe operujące na sobie i swej zawartości. Agregat jest jednostką pracy – zwykle pobieramy go z Repository (lub tworzymy nowy w Facotry), wykonujemy na nim operacje biznesowe, po czym utrwalamy przez Repository.

Natomiast koncepcyjnie Agregate to główne jednostki modelu i pracy w DDD. Po prostu Encje są zbyt ziarniste. Warto podkreślić, że Agregat powinien dążyć do maksymalnej enkaspsulacji swej implementacji. Zatem nie powinien zawierać on setterów (chyba, że jest to uzasadnione) ponieważ wszelkie modyfikacje powinny zachodzić przez metody biznesowe. Gettery pojawiają się również tylko wówczas gdy jest to sensowne. Warto pamiętać, że gettery i settery nie są wymagane dla JPA – JPA może działać refleksyjnie na polach prywatnych i bywa to nawet szybsze w niektórych implementacjach.

Enkapsulacja staje się szczególnie istotna gdy spojrzymy na DDD jako na technikę agilną, która w każdej iteracji „kruszy" coraz więcej wiedzy na temat modelu – uszczegóławia go (technika knowledge crunching).

**Listing 2.** *Agregat*

```
@Entity
public class Order extends AggregateRoot{
  @OneToMany
  private List<OrderItem> items;

  @Embeded
  private Money sum;
  private OrderStatus status;
  private Date createDate;

  private RebatePolicy rebatePolicy;

  public Order(RebatePolicy rebatePolicy){
    this.rebatePolicy = rebatePolicy;
    sum = new Money(0.0);
    //...
  }

  public void add(Product p, int quantity){
    OrderItem oi = OrderItemFactory.build(p, quantity, rebatePolicy);
    items.add(oi);
    sum = sum.add(oi.getCost());
  }

  public void confirm(Payment payment){
    if (status == CONFIRMED)
      throw new InvalidStateException();

    status = CONFIRMED;
    createDate = new Date();

    fireEvent(orderEventsFactory.orderSubmitted(getId()));
  }

  //TODO można zastanowić się nad ukryciem OrderItem i zwracać listę Value Objectów
  public List<OrderItem> getOrderItems(){
    return unmodifiableList(items);
  }
}
```

Dzięki enkapsulacji możemy z większą swobodą modyfikować wnętrze agregatu.

Listing 2 przedstawia przykładowy Agregat. Nasze zamówienie z technicznego punktu widzenia jest encją JPA i dziedziczy po klasie bazowej stworzonej dla wszystkich agregatów. Klasa bazowa zapewnia możliwość polimorficznego wykonywania operacji na agregatach (zatem mamy zgodność z Zasadą Podstawienia Liskov) oraz dostarcza wspólnych cech, takich jak ID lub możliwość publikowania zdarzeń.

Agregat zawiera pola reprezentujące jego stan, takie jak status, data utworzenia czy lista pozycji na zamówieniu.

Warto zwrócić uwagę na fakt, że lista pozycji (pole items) nie posiada gettera. Metoda getOrderItems nie jest getterem a jedynie pozwala przejrzeć zawartość zamówienia zwracając kopię listy. Dzięki temu nie istnieje możliwość zmiany stanu agregatu (dodania/usunięcia produktu) inaczej niż przez metody biznesowe.

Nasz przykładowy agregat posiada dwie metody biznesowe. Metoda add dodaje produkt w zadanej ilości. Warto zwrócić uwagę, że metoda ta ukrywa fakt istnienia klasy OrderItem – jest to szczegół implementacji agregatu. Dodatkowo metoda ta przeprowadza logikę biznesową – oblicza całkowity koszt zamówienia biorąc pod uwagę rabatowanie.

Druga metoda biznesowa confirm zatwierdza zmówienie. Zamówienie może zgłosić veto – rzucić wyjątek. Metoda wykonuje logikę biznesową – zmienia stan zamówienia oraz generuje zdarzenie biznesowe, które będzie opisane w dalszej części tekstu.

### Value Object

Na listingu 2. widzimy pole sum, które jest całkowitym kosztem zamówienia. Oczywiście wszyscy wiemy, że aby dokładnie liczyć pieniądze typ Double jest niewystarczający, dlatego należy je liczyć na typie BigDecimal. Natomiast w naszym modelu zastosowaliśmy specjalną klasę, która uwypukla w naszym modelu domenowym koncepcję pieniędzy – klasę Money. Po raz kolejny mamy zastosowanie zasady „make explicit what is implicit". Klasa Money ukrywa implementację (w środku może to być BigDecimal albo int z przesunięciem kilku zer w celu zwiększenia wydajności) oraz dostarcza wygodnych metod operujących na kwotach pieniędzy co jeszcze mocniej zwiększa ekspresję modelu.

Nasza klasa Money może zawierać dodatkowo oprócz kwoty również: walutę oraz datę pobrania kursu w stosunku do waluty bazowej.

Warto zwrócić uwagę na łatwość techniczną z jaką wprowadzamy Value Object do Agreagatu. Jest to typ zagnieżdżony (embeded) JPA dzięki czemu nie musimy wprowadzać nowej tabeli. Po prostu podczas mapowania nasza struktura obiektów zostanie spłaszczona do jednej tabeli.

Value Objects są często niedoceniane jako środek wyrazu, ale warto stosować je wszędzie tam gdzie mamy do czynienia z liczbami bądź łańcuchami, które podlegają obostrzeniom, np.:

- numer telefonu – Wprowadzenie wygodnej klasy `PhoneNumer` z metodami pobierającymi np. kierunkowy zwalnia nas z pisania Utilsów
- liczby będące miarami w jakiś jednostkach – operacje przeliczające jednostki, sprawdzające zakres

Rozważmy prosty przykład z listingu 3: obliczanie prawdopodobieństwa zdarzenia polegającego na zajściu dwóch zdarzeń.

Co jeżeli p1 lub p2 jest null lub liczbą spoza przedziału <0;1>?

Co jeżeli ktoś myśląc 80% podstawił pod p1 wartość 80 zamiast 0.8?

Aby obsłużyć te przypadki musimy wprowadzić kod defensywny, które jest sprawdzi i odpowiednio zareaguje. Następnie metodą Copiego-Pasty dublujemy ten kod w 150 miejscach w projekcie. Po pewny czasie główny programista zarządza wprowadzanie standardowego rozwiązania – refaktoryzację do Utilsa z dwoma parametrami.

Gdyby wprowadzić Value Object o nazwie Probability, wówczas kod stałby się bardziej wyrazisty, a model bardziej oddawałby intencję.

### Service

Na listingu 1 w metodzie submitOrder widzimy użycie klasy InvoicingService.

Jest to serwis biznesowy – z warstwy modelu domenowego – w odróżnieniu od serwisu aplikacyjnego z warstwy wyższej.

Obiekt tej klasy służy do wygenerowania faktury na podstawie zamówienia. Klasa `InvoicingService` została wprowadzona w celu odciążenia klasy Order z funkcjonalności tworzenia faktur. Być może klasa order wydaje się naturalnym miejscem dla metody generateInvoice ale nie chcemy aby klasa Order stała się tak zwaną „boską klasą", która wszystko wie, wszystkich zna i wszystko potrafi. Boskie klasy utrudniają utrzymanie kodu i zrozumienie oraz niemal uniemożliwiają testowanie jednostkowe z powodu dużej ilości zależności.

Uwypuklamy w modelu czynność generowania faktury w osobnym serwisie – Transformatorze w nomenklaturze

**Listing 3.** *Kod operujący na typach podstawowych*

```
Double probability = p1 * p2;
```

RDD, który tworzy obiekty na podstawie innych obiektów.

Prosta reguła dla początkujących modelarzy może być następująca: encje/agregaty/vo posiadają te metody, które są dla nich charakterystyczne i wielokrotnie używane w różnych use case/user story. Natomiast specyficzne metody, używane raz lub tylko kilkakrotnie warto umieszczać w osobnych serwisach.

Serwis podczas swej pracy może posiłkować się innymi obiektami domenowymi, takimi jak opisane poniżej polityki.

### Policy

Polityki służą do eksponowania w modelu wariacji zachowań (operacji biznesowych). Polityka z technicznego punktu widzenia to nic innego jak Wzorzec Projektowy Strategii. Na listingu 2 widzimy jak interfejs RebatePolicy jest przekazywany do konstruowania OrderOtem. Następnie OrderItem będzie z niego korzystał obliczając swą cenę.

Listing 5 przedstawia użycie polityki – jej polimorficzne wywołanie po interfejsie. Natomiast Listing 6 to przykładowy interfejs polityki.

W naszym modelu może istnieć wiele polityk (implementacji interfejsu), np. rabat dla VIPa, rabat zimowy i dziesiątki innych. Dalsze modelowanie może zakładać np. składanie rabatów przy pomocy Wzorca Projektowego Dekorator.

Wprowadzenie polityki jest jedną z technik „supple design", które zwiększają giętkość modelu, otwierając go na przyszłą rozbudowę. Rozbudowa będzie polegała na tworzeniu implementacji interfejsu RebatePolicy bez potrzeby modyfikowania Core naszego modelu. Kolejnym wzorcem zwiększającym giętkość modelu jest Specification, którego jednak nie będziemy omawiać na łamach tego artykułu, ze względu na złożoność o obszerność kodu. Zainteresowanych odsyłam do katalogu wzorców projektowych lub do książki Evansa.

Oczywiście uważny czytelnik zastanawia się zapewne skąd Agregat/Encja dowiaduje się o tym z jaką polityką pracuje. Czyli innymi słowy to odpowiada za przekazanie konkretnej polityki do konstruktora Agregatu?

### Factory

Fabryki w DDD służą do tworzenia złożonych agregatów. W naszym przykładzie Agregat Order potrzebuje do pracy RebatePolicy natomiast w ogólności może wymagać przekazania mu innych współpracowników biznesowych. Tego typu odpowiedzialność przenosimy na fabryki, które nie są obiektami czysty technicznymi, lecz zawierają w sobie część logiki. W naszym przypadku logiki wyliczającej jaką implementację RebatePolicy przekazać do tworzonego zamówienia.

Dodatkowo Fabryki dbają o poprawność tworzonych obiektów. Fabryka składa obiekt biznesowy na podstawie przekazanych do metody fabryki paramentów. Logika tej metody musi zapewnić walidację „składników", ponieważ zgodnie z zasadą DDD nie mogą istnieć w systemie (nawet w pamięci) obiekty w niepoprawnym/niedozwolonym stanie.

Fabryki powinny fabrykować całe Agregaty. Ale czasem w momencie tworzenia obiektu nie mamy jeszcze całej wiedzy o tym jakie powinny być jego składowe. Przykładowo tworząc zamówienie możemy nie mieć jeszcze danych potrzebnych do wyliczenia konkretnego typu rabatu. Wówczas będziemy potrzebować fabryki tworzącej Politykę rabatową – fabryki, która będzie tworzyć produkt w odpowiednim momencie.

Z technicznego punktu widzenia fabryki mają również znaczny wpływ na testowalność kodu, czyli jego podatność na bycie testowanym. Dzięki temu, że fabryki odpowiadają za składanie złożonych agregatów, to znacznie zmniejszają ilość operatorów new w kodzie agregatu. Czyli agregat nie tworzy większości zawieranych w sobie obiektów przez operator new a ma je przekazane prze konstruktor. Pozwala na to testowanie agregatu z użyciem technik mock/stub, co z kolei przekłada się na możliwość testowania kodu poza ciężkim środowiskiem serwera, czyli w rezultacie na produktywność tworzenia kodu.

**Listing 4.** *Value Object*

```
Probabilty p1 = Probabilty.fromPercentage(80);
Probability probability = p1.and(p2);
```

**Listing 5.** *Użycie Polityki*

```
public class OrderItem{
  private Money cost;
  public OrderItem(RebatePolicy rebatePolicy){
     //Money cost = logika biznesowa
    Money rebate = rebatePolicy.calculateRebate(t
                   his);
    cost = cost.substract(rebate);
  }
  public Money getCost(){
    return cost;
  }
}
```

**Listing 6.** *Interfejs Polityki*

```
public interface RebatePolicy{
  public Money calculateRebate(OrderItem oi);
}
```

### Repository

Repozytorium – zarządza trwałością agregatów danego typu. Czyli jest w stanie pobrać agregat po ID oraz zapisać go. Repozytorium różni się od DAO tym, że nie powinno zawierać setek metod wyszukujących dane potrzebne dla tabelek na GUI. Oczywiście repozytorium może wyszukiwać obiekty na potrzeby operacji biznesowych, np. niezatwierdzone zamówienia użytkownika.

Repozytorium analogicznie jak fabryka może wstrzykiwać do Agregatów obiekty współpracujące.

W implementacji warto skorzystać z typowych idiomów (re-używalnych kawałków kodu), takich jak Generic Repository dla JPA, które są dostępne w sieci.

### Events

Zdarzenia są istotnym elementem rozległych modeli domenowych. Na listingu 2 w metodzie confirm widzimy jak Agregat powiadamia system o zajściu zdarzenia biznesowego: „zatwierdzono zamówienie od danym ID".

Zdarzeń używamy kierując się różnymi intencjami:

- jeżeli chcemy dodawać nowe funkcjonalności uruchamiane po zajściu pewnych wydarzeń – architektura pluginowa
- dodatkowe czynności są od siebie niezależne
- jeżeli chcemy pewne czynności wykonywać asynchronicznie (zdarzenia możemy, ale nie musimy obsługiwać asynchronicznie w innej transakcji) – ponieważ obliczenia z nimi związane są długotrwałe i nie zależy nam na ich natychmiastowym wykonaniu
- jeżeli zależy nam na rozluźnieniu powiązań pomiędzy obiektami z różnych domen

### Bounded Context

Technika Bounded Context jest najbardziej wyrafinowaną i subtelną z technik DDD. Zakłada ona istnienie różnych modeli opisujących te same byty, używanych w różnych kontekstach. Zagadnienie Bounded Context wykracza daleko poza ramy niniejszego artykułu, ale warto wspomnieć o nim przy okazji zdarzeń.

**Listing 7.** *Złamanie Bounded Context*

```
public class Client extends AggregateRoot {
  //...
  public void changeStatus(ClientStatus status){
    this.status = status;
    //wiecej logiki biznesowej

      if (status == VIP){
        //wyszukanie niezrealizowanych zamówień
        //rabat na każde niezrealizowane
                    zamówienia
    }
  }
}
```

Rozważmy kod z listingu 7, które intencja jest następująca: istnieje możliwość zmiany statusu klienta (zmiana polega na pewnych operacjach na kliencie); w razie gdy klient otrzymał status VIP, wówczas naliczamy x% rabatu na wszystkie niezrealizowane zamówienia.

Umieszczenie dodatkowej logiki (naliczanie rabatu) w klasie klienta nie generalnie jest dobrym pomysłem, ponieważ powoduje, że klient staje się „boską klasą". Być może lepszym pomysłem jest wyniesienie tej logiki do osobnego serwisu i wywołanie w warstwie aplikacji.

Jednak w aplikacji klasy ERP (*Enterprise Resource Management*) również takie model upadnie dosyć szybko. Można założyć, że w tej klasie systemów istnieje moduł CRM (*Client Relationship Management*) służący do zarządzania klientami oraz moduł Sprzedażowy odpowiedzialny między innymi za rabaty.

W systemie klasy ERP model jest na tyle zawiły, że za pewnie nie ma jednego Eksperta Domenowego, który byłby w stanie opisać obie domeny. Dlatego oba moduły będą modelowane przez różne osoby oraz będą skupiać się na różnych aspektach. Tak więc w każdym z nich zapewne pojawi się jakaś metafora Klienta. Pojawi się wówczas „przypisany kontekst" do klienta. Klient w kontekście CRM jest zamodelowany inaczej niż w kontekście Sprzedaży Oczywiście na poziomie danych w bazie może (zależy od poziomu separacji modułów) być to ten sam rekord. I co najważniejsze jeden moduł (kontekst) nie powinien bezpośrednio wołać drugiego ze względu na pojawianie się kodu i zależności wyższego rzędu o strukturze Spaghetti.

Lepszym rozwiązaniem w tym przypadku będzie wygenerowanie zdarzenia biznesowego „klient od ID x zmienił status na y". Wówczas inne zainteresowane konteksty (moduły) moduły mogą reagować na to wydarzenie w przezroczysty sposób odciążając corową logikę modułu CRM.

### Saga

Saga jest najnowszym Building Blockiem DDD i służy do modelowania procesów, w których mamy do czynienia z kilkoma rozproszonymi zdarzeniami, które nabierają nowego znaczenia jeżeli pojawią się w zestawie.

Technicznie Saga jest listenerem kilku zdarzeń a jej stan jest utrwalany w bazie danych.

Koncepcyjnie służy do uwypuklenia w modelu orkiestracji kilku zdarzeń.

Listing 8 przedstawia przykładową Sagę w języku C# ponieważ w tym języki można łatwiej wyrazić pewne konstrukcje. W Javie nie mamy możliwości implementowania kilku interfejsów na tym samym typie generycznym (`CustomerBilledForOrder` i `OrderSubmitted` mają wspólną nad-klasę) z uwagi na fakt, że informacje

o typach generycznych nie są zapisywane w bytecode po kompilacji.

Zatem w Javie należałoby zaimplementować jedną metodę handle, która poprzez instanceof dowiaduje się z jakim zdarzeniem mam do czynienia. Potrzebne będą również własne adnotacje aby oznaczyć klasy informacją o tym jakimi zdarzeniami jest zainteresowana dana Saga.

Nasza przykładowa Saga jest zainteresowana dwoma zdarzeniami biznesowymi: zatwierdzeniem zamówienia oraz uiszczeniem opłaty.

W przypadku zajścia zdarzenia biznesowego, jeden z naszych technicznych listenerów sprawdza przy pomocy refleksji, które sagi są zainteresowane tym typem zdarzenia. Następnie pobiera po biznesowym ID (w naszym przypadku ID zamówienia) dane odpowiednich Sag z bazy i wywołuje metody nasłuchujące Sagi. Saga obsługując zdarzenie zmienia swój stan i być może kończy swój cykl życia dokonując odpowiednich obliczeń biznesowych.

Jeżeli Saga nie jest gotowa aby zakończyć swój cykl życia, wówczas jej stan jest po prostu utrwalany w bazie i Saga czeka na kolejne zdarzenia. Warto zwrócić uwagę, że stan Sagi jest zaimplementowany przy pomocy pomocy Wzorca Projektowego Memento – czyli „wysłania informacji do siebie samego w przyszłości"

Oczywiście ten sam efekt można osiągnąć na wiele innych sposobów, ale chodzi o wyrażanie modelu w odpowiednim paradygmacie oraz na odpowiednim poziomie abstrakcji. Jeżeli jakiś proces jest sterowany wieloma zdarzeniami, to warto w ten sposób go modelować, aby ujawniać intencję.

### State Machine

Maszyna stanów nie należy do oficjalnych Building Blocks DDD, ale warto się nią posiłkować w przypadku gdy obiekty biznesowe mogą znajdować się w różnych stanach a dla każdego stanu ich zachowanie jest inne oraz gdy istnieją reguły przejścia pomiędzy stanami.

Zainteresowanych odsyłamy do State Design Pattern.

## Destylacja Domen

Oczywiście w każdym projekcie brakuje czasu (który jest zwykle funkcją budżetu) na to aby dokładnie i rzetelnie opracować model a następnie go pielęgnować.

DDD jest podejściem pragmatycznym i dostarcza odpowiedniej techniki do radzenia sobie z tym problemem. Przed przystąpieniem do modelowania należy dokonać destylacji domen.

Wyłaniamy Core Domain, która jest głównym komponentem systemu ponieważ dostarcza największą wartość i być może stanowi o przewadze nad konkurencją. Tylko w Core Domain stosujemy techniki DDD oraz inwestujemy w nią najlepszych ludzi. Zespół pracujący nad Core Domain powinien charakteryzować się skillsetem skła-

## 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

## Główne techniki DDD

- Ubiquitous Language – wspólny język modelu, którym posługuje sią cały zespół, począwszy od Eksperta Domenaowego po programistów warstwy logiki domenowej
- Building Blocks – język standardowych wzorców, z których budujemy modele
- Rozwarstwienie logiki – podział logiki na aplikacyjną (kroi use case/user story, technikalia) oraz domenową (model domeny, building blocks)
- Supple Design – techniki otwierania modelu na rozbudowę w kolejnych iteracjach
- Domains Distillation – wyodrębnianie corowej domeny, w której stosujemy techniki DDD oraz domen wspomagających i specyficznych
- Anti-corruption Layer – warstwa abstrakcji zapobiegająca przenikaniu modelu z innych domen (supporting, generic, inne integrowane systemy) do naszej corowej domeny.

**Listing 8.** *Saga*

```
public class OrderSaga : Saga<OrderSagaData>, IsagaStartedBy<CustomerBilledForOrder>, ISagaStartedBy<Orde
                  rSubmitted>{
  private OrderSagaData data;

  public void handle(CustomerBilledForOrder message){
    data.setCustomerHasBeenBilled(true);
    data.setCustomerId(message.getCustomerId());
    data.setOrderId(message.getOrderId());

    completeIfPossible();
  }

  public void handle(OrderSubmitted message){
    data.setProductIdsInOrder = message.ProductIdsInOrder;
    data.setCustomerId(message.getCustomerId());
    data.setOrderId(message.getOrderId());

    completeIfPossible();
  }

  private void completeIfPossible(){
    //if ( reguła biznesowa )
    eventsMananger.spread(...);
    markAsCompleted();
  }
  //metody wzorca Meneto z klasy bazowej
  public OrderSagaData saveToMemento(){
    return data;
  }

  public void loadFromMemento(OrderSagaData memento){
    data = memento;
  }
}
```

dającym się z modelowania OO oraz komunikatywności i rozumieniu potrzeb biznesowych.

Supporting Domain to funkcjonalności, które nie są krytyczne i możemy zaryzykować niższą jakość ich kodu. W dużych projektach mogą być outsourcowane.

Generic Domain to specyficzne pod-domeny, w przypadku których warto użyć gotowych rozwiązań integrując się z nimi poprzez specjalną warstwę anticorruption-layer, która odcina nasz model od specyfiki modeli Generic. Przykładem może być biblioteka operująca na grafach lub zakup pod-modułu księgowości.

## Czy DDD jest odpowiednie dla mojego projektu?

Techniki DDD są opłacalne dla projektów operujących na złożonych i nietrywialnych domenach. Oczywiście w dużych projektach zwykle tylko część modelu (Core Domain) będzie spełniać ten warunek.

## W Sieci

- *http://domaindrivendesign.org* – oficjalna strona DDD
- *http://prezi.com/mwtwdw2i7q4h/domain-driven-design-a-place-for-everything-and-everything-in-its-place* – prezentacja będąca ilustracją graficzną dla niniejszego tekstu
- *http://prezi.com/nywckem9elvc/command-query-responsibility-segregation-nowe-bardziej-racjonalne-podejscie-do-warstw* – prezentacja na temat architektury CqRS.

Kluczowym elementem procesu wykorzystującego techniki DDD jest dostęp do eksperta lub grupy ekspertów domenowych.

Ważne jest również iteracyjny proces „kruszenia wiedzy", w którym uszczegóławiamy model.

Natomiast na poziomie zespołu develoeprskiego ważnym aspektem są kompetencje członków: dobra znajomość technik OOD oraz raczej przewaga inteligencji werbalnej niż algorytmicznej.

## Poza DDD

Od kiedy DDD jako spójny zestaw technik pojawiło się w inżynierii oprogramowania minęło osiem lat. Od tego czasu pojawiło się kilka dodatkowych koncepcji:

### Behavior Driven Development

BDD to zwinny proces wytwarzania oprogramowania integrujący DDD jako zestaw technik modelowania, TDD (scenariusze akceptacyjne) jako technikę developemntu oraz Scrumowe User Strory.

### Archetypy modeli biznesowych

DDD jest zastawem technik służących do okrywania nieznanych nam domen. Natomiast Archetypy modeli biznesowych go gotowe opracowania domen dobrze znanych (takich jak struktury organizacyjne, modle produktów, crm itd), którymi warto się posiłkować w modelowaniu.

### Command-query Responsibility Segregation

CqRS jest architekturą, która wspiera modelowanie DDD oraz odpowiada na szereg wyzwań technicznych takich jak optymalizacja modelu pod kątem odczytu danych przekrojowych.

**SŁAWOMIR SOBÓTKA**

*Programujący architekt aplikacji specjalizujący się w technologiach Java i efektywnym wykorzystaniu zdobyczy inżynierii oprogramowania. Trener i konsultant w firmie Bottega IT Solutions. Entuzjasta Software Craftsmanship.*

*Do jego zainteresowań należy szeroko pojęta inżynieria oprogramowania: architektury wysokowydajnych systemów webowych (w szczególności CqRS), modelowanie (w szczególności DDD), 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), lider lubelskiego Java User Group, publicysta w prasie branżowej i blogger (http://art-of-software.blogspot.com).*

*Kontakt z autorem:*
*slawomir.sobotka@bottega.com.pl*