Autor

Receptury projektowe – niezbędnik początkującego architekta

Część VII: Building Blocks dla Twojej lewej półkuli: połączenia podejścia obiektowego, proceduralnego, funkcyjnego w codziennej pracy z kodem.

Wiele czasu i energii spędzamy na dyskusjach o wyższości jednego paradygmatu programowania nad innym, o wyższości jednego języka programowania nad innym. W niniejszym artykule będę chciał przekonać czytelników do tego, aby obok siebie, równorzędnie stosować zarówno paradygmat obiektowy, jak i funkcyjny oraz nie zapominać o proceduralnym.

PRZYKŁADOWY PROJEKT

Wszystkie przykłady kodu pochodzą z referencyjnego projektu DDD-Leaven v2.0, którego źródła na github można znaleźć w ramce „W sieci".

PODSTAWY

Podstawy są trudne. W każdej dziedzinie. Podstawy rozumiemy dopiero z czasem. Na początku zadowala nas sam fakt, że coś działa. Dopiero po latach następuje refleksja nad tym, dlaczego podstawowe zasady są takie a nie inne i dlaczego jest to ważne.

BUILDING BLOCKS DLA TWOJEJ LEWEJ PÓŁKULI

Jeden z bardzo prostych modeli działania naszego mózgu zakłada, że lewa półkula działa precyzyjnie, logicznie i posługuje się abstrakcyjnymi symbolami. Działa ona również indukcyjnie, czyli operuje wewnątrz znanej sobie puli symboli (tytułowe building blocks) wykorzystując je do modelowania otaczającego świata.

W codziennej praktyce przekłada się to na znane zapewne wszystkim zjawisko: jeżeli jedyne building blocks, jakie znasz, to Serwis i anemiczna encja, to jesteś w stanie cały świat opisać serwisem lub encją (de facto procedurą i rekordem z Pascala:) Jeżeli natomiast Twój ulubiony building block to Hashmapa, to jesteś w stanie cały świat opisać Hashmapą Hashmap.

Zatem jako architekt powinieneś/powinnaś dążyć do poszerzania „wachlarza" building blocks, aby otworzyć sobie szersze możliwości konstruowania modeli programistycznych problemów, jakie rozwiązujesz w codziennej pracy.

PRZEGLĄD PODSTAWOWYCH BUILDING BLOCKS

W typowych mainstreamowych technologiach i frameworkach służących do tworzenia aplikacji możemy zauważyć cztery generalne building blocks. Każdy z nich charakteryzuje się odpornością na innego rodzaju zmiany i służy do innego rodzaju zadań.

O serii „Receptury projektowe"

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

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

OBIEKTY

Obiekty eksponują zachowania. Służą do tego, aby wysyłać do nich sygnały. Natomiast ich wewnętrzne struktury powinny zostać hermetyczne.

Zastosowanie obiektów

Przy takim podejściu obiekty będą sprawdzać się w sytuacjach, gdy „API" obiektu jest stabilne, a spodziewamy się zmian w ich wewnętrznej strukturze. Dzięki hermetyzacji niestabilnej struktury podczas zmian jej kodu katastrofa jest lokalna, bo ograniczona do zakresu klasy.

Listing 1 Przykład obiektu i zasad: Abstrakcja, Enkapsulacja, CqS, Information Expert

```
@Entity
@AggregateRoot
public class Client extends BaseAggregateRoot{
  private String name;
  //...
  public ClientData generateSnapshot(){
    return new ClientData(aggregateId, name);
  }
  public boolean canAfford(Money amount) {
    //...
  }
  public void charge(Money amount) {
    //...
  }
}
```

Przykładowo na Listingu 1 widzimy obiekt modelujący klienta w pewnej domenie biznesowej. W tej domenie postrzegamy klienta bezosobowo, jako „torbę z pieniędzmi". Dlatego obiekt ten zawiera odpowiedzialność obciążania klienta należnością i sprawdzenia, czy klient może pozwolić sobie na wydatek pewnej kwoty.

Projektowanie obiektów

Projektując obiekty, przeprowadzamy następujący tok myślowy:

- co obiekt robi (metody)
- jakie reguły rządzą tymi operacjami
- jakie dane są potrzebne do wykonania tych operacji zgodnie z założonymi regułami

Łatwa rozbudowa obiektów

Przykładowo nasz obiekt może sprawdzać, czy klient, którego on reprezentuje, może sobie pozwolić na wydatek jeżeli:

- posiada dostateczną ilość pieniędzy
- jeżeli ich nie posiada, to być może udzielimy mu kredytu, pod warunkiem że:
  - ma status VIP
  - nie przekroczył jeszcze limitu kredytu

Tak więc z iteracji na iterację możemy dodawać kolejne pola do obiektu oraz kod do metod tak, że „api" obiektu będzie wciąż stabilne, zmiany będą hermetyczne.

STRUKTURY DANYCH

Gdyby klasa `Client` zamiast metod `canAfford` i `charge` zawierała metody `getMoney` i `setMoney`, to wówczas stałaby się strukturą danych i przestała być obiektem.

Typowy błąd

Często tego typu struktury danych (tak zwane anemiczne encje) są mylone z obiektami – mylące jest słówko kluczowe class, które występuje w niektórych językach, których na poziomie składni nie rozróżnia się obiektów od struktur – rozróżnienie jest koncepcyjne.

Zastosowanie struktur danych

Struktury w odróżnieniu od obiektów nie posiadają zachowania i w odróżnieniu od obiektów eksponują swe wewnętrzne struktury – struktury są po to, aby móc je przeglądać.

Stosujemy zatem struktury danych w przypadku gdy spodziewamy niestabilności kodu operacji – dlatego operacje to są wyniesione poza kod struktury – do procedur. Natomiast same struktury są stabilne, dlatego możemy je swobodnie eksponować, nie narażając reszty kodu na zmiany.

Do tanga trzeba dwojga

Na Listingu 1 mamy przykład, w którym klient posiada metodę `generateSnapshot`, która zwraca strukturę danych ClientData. Dane klienta są chwilową migawką stanu obiektu klient. Struktura ClientData może być przykładowo potrzeba jako składowa obiektu Purchase – tak jak ilustruje to kod na Listingu 2. Idea jest taka, że w obiekcie Purchase interesuje mnie stan obiektu klient z momentu, w jakim klient tworzył zakup. Dlatego Purchase przechowuje w swoim wnętrzu strukturę danych `ClientData` a nie obiekt klacy `Client`.

Listing 2 Tworzenie zakupu (Purchase) na podstawie struktury danych wygenerowanej z obiektu

```
public class PurchaseFactory {
  public Purchase create(AggregateId orderId,
       Client client, Offer offer){
    if (! canPurchse(client, offer.getAvailabeItems()))
      throw new DomainOperationException(
        client.getAggregateId(),
        „client can not purchase");

    //...
    Purchase purchase = new Purchase(orderId, client.generateSnapshot(),
        items, new Date(), false, purchaseTotlCost);

    return purchase;
  }
  //...
}
```

PROCEDURY

Procedury są wciąż nieodzownym Building Blockiem, nie odeszły wraz z Turbo Pascalem, zostały w postaci Serwisów – bezstanowych klas, czyli paczek procedur. Bynajmniej nie piszę tego sarkastycznie. Procedury są idealnym Building Blockiem do projektowania API znajdującego się na styku modułów lub podsystemów.

Procedury są sekwencją kroków do wykonania. Procedury mają szereg efektów ubocznych (modyfikują stan systemu, np. bazy) i posiadają wiele zależności oraz zwykle szeroką odpowiedzialność.

Oczywiście mamy tutaj klasyczne techniki dekompozycji procedur na pod-procedury znane z I roku studiów.

Procedura może zwracać wynik działania – sense status operacji lub wyjątek. Jeżeli nie jest to zgodne z przyjętymi zasadami OO, to nie przejmujemy się tym, ponieważ procedury to nie obiekty, zatem nie obowiązują nas tutaj zasady OO.

Procedury jako model kroków Use Case/User Story i API modułu

Przykładowa paczka procedur (Serwis) z Listingu 3 modeluje Use Casey procesu rezerwacji i potwierdzania oferty. Procedury w swym wnętrzu orkiestruje obiekty, pobierając je z magazynu danych, wywołując ich metody w celu uzyskania przepływu scenariusza i utrwala te obiekty.

Listing 3 Procedura w postaci Serwisu, która zarządza obiektami i strukturami danych

```
public class OrderingServiceImpl implements OrderingService {
  @Inject private SystemUser systemUser;
  @Inject private ClientRepository clientRepository;
  //...
  public AggregateId createOrder() {
    //...
  }

  public void addProduct(AggregateId orderId, AggregateId productId,
      int quantity) {
    //...
  }

  public Offer calculateOffer(AggregateId orderId) {
    //...
  }

  @Override
  @Transactional(isolation = Isolation.SERIALIZABLE)
  public void confirm(AggregateId orderId,
       OrderDetailsCommand orderDetailsCommand, Offer seenOffer)
          throws OfferChangedExcpetion {
    Reservation reservation = reservationRepository.load(orderId);
    if (reservation.isClosed())
      throw new DomainOperationException(reservation.getAggregateId(),
        „reservation is already closed");

    Offer newOffer = reservation.calculateOffer(discountFactory.
    create(loadClient()));

    if (! newOffer.sameAs(seenOffer, 5))//TODO load delta from conf.
      throw new OfferChangedExcpetion(reservation.getAggregateId(),
        seenOffer, newOffer);

    Client client = loadClient();
    Purchase purchase = purchaseFactory.create(
      reservation.getAggregateId(), client, seenOffer);
    if (! client.canAfford(purchase.getTotalCost()))
      throw new DomainOperationException(client.getAggregateId(),
        „client has insufficent money");
```

```
    purchaseRepository.save(purchase);

    Payment payment = client.charge(purchase.getTotalCost());
    paymentRepository.save(payment);

    purchase.confirm();
    reservation.close();

    reservationRepository.save(reservation);
    clientRepository.save(client);

  }

  private Client loadClient() {
    return clientRepository.load(systemUser.getDomainUserId());
  }
}
```

Procedury w architekturze warstwowej

Procedury doskonale nadają się do modelowania API systemu, występując na styku modułów. Dlatego w 4-warstwowej architekturze (prezentacja, aplikacja, domena, infrastruktura) stosujemy je w warstwie aplikacji. Zainteresowanych odsyłam do ramki „W Sieci" i artykułu poświęconego architekturze DDD.

FUNKCJE

Niniejszy artykuł skupia się wokół mainstreamowych technologii, dlatego dotkniemy podejścia funkcyjnego na takim poziomie, na jakim wspiera go składnia języka Java. Tak więc póki co w Javie nie mamy funkcji jako takich, więc emulujemy je różnego rodzaju idiomami i wzorcami.

Przyjmiemy tutaj uproszczoną definicję funkcji: funkcje przetwarzają dane wejście w dane wyjściowe, zawsze wygenerują takie same wyjście dla danego wejścia i nie powodują efektów ubocznych. Siła wykorzystania funkcji ujawnia się dopiero, gdy zaczniemy je ze sobą składać.

Listing 4 Funkcja w postaci metody obiektu, która zwraca wartość – strukturę danych (Value Object). Funkcja generująca ofertę jest domknięta inną funkcją obliczającą rabat

```
@Entity
public class Reservation extends BaseAggregateRoot{
  @OneToMany(cascade = CascadeType.ALL,
    fetch=FetchType.EAGER, orphanRemoval = true)
  @JoinColumn(name = „reservation")
  @Fetch(FetchMode.JOIN)
  private List<ReservationItem> items;
  @Embedded
  private ClientData clientData;

  public void add(Product product, int quantity){
    //...
  }

  @Function
  public Offer calculateOffer(DiscountPolicy discountPolicy) {
    for (ReservationItem item : items) {
      if (item.getProduct().isAvailabe()){
        Discount discount = discountPolicy.applyDiscount(...);
        OfferItem offerItem =
          new OfferItem(item,discount);

        availabeItems.add(offerItem);
      }
    }

    return new Offer(availabeItems,);
  }
```

Listing 5 Funkcja w postaci serwisu, która przetwarza wartość w obiekt. Funkcja generująca fakturę na jest domknięta inną funkcją liczącą podatek

```
public class BookKeeper {
  public Invoice issuance(InvoiceRequest invoiceRequest, TaxPolicy
  taxPolicy){
    //...
    for (RequestItem item : invoiceRequest.getItems()){
      Money net = item.getTotalCost();
      Tax tax = taxPolicy.calculateTax(
        item.getProductData().getType(), net);
      InvoiceLine invoiceLine = new InvoiceLine(
        item.getProductData(), item.getQuantity(),
        net, tax);
```

Autor

```
      invoice.addItem(invoiceLine);
    }

    return invoice;
  }
}
```

Funkcje bez funkcji

Na Listingu 4 widzimy metodę `generateOffer`. W rozumieniu składni Javy jest to metoda, natomiast koncepcyjnie mamy tutaj do czynienia z funkcją – jej wejściem jest stan obiektu, a wartością w tym przypadku struktura danych.

Natomiast na Listingu 5 funkcja `issuance` występuje w postaci osobnego serwisu.

Są to swego rodzaju protezy, które emulują funkcje w składni języka, który ich nie posiada.

Domknięcia

Funkcję `generateOffer` możemy nazwać „higher order function", ponieważ została ona domknięta (closure) funkcją obliczania rabatu. Mamy zatem generalny algorytm generowania oferty (który w tym przekładzie odrzuca niedostępne produkty) i „dostrajamy" go szczególną operacją rabatowania.

Podobnie funkcja `issuance` zawiera generalny algorytm generowania faktury i jest domknięta funkcją liczącą podatek (zależne od danego kraju).

Dalej możemy stosować kolejne wzorce projektowe: Chain of Responsibility lub Decorator w celu bardziej wyrafinowanego składania funkcji.

PROBLEM SKŁADNI

Przykładowo w języku Java wszystkie cztery podstawowe Building Blocks tworzymy przy pomocy słówka kluczowego class. Na poziomie języka nie mamy żadnego wsparcia służącego do rozróżniania rodzaju konstrukcji.

Osobista dyscyplina projektanta/developera jest jedynym środkiem do osiągnięcia porządku w kodzie.

Przykładowy błąd: obiekt klient posiada odpowiedzialność: canAfford i charge, które rządzą się określonymi regułami domenowymi. Gdy dodamy jednak do tej klasy konstrukcję ze struktur danych: setMoney, to wówczas pozwolimy na to, aby niejako „bokiem" zmienić stan obiektu, nie respektując reguł domenych zawartych w canAfford i charge. Nie trzeba wiele czasu, aby super-junior-consultant wykorzystał tę „furtkę" do puszczenia bokiem brzydkiego hacka i linijce numer 17865. Od teraz przez lata tuziny smutnych ludzi będą zastanawiać się nad tym, gdzie coś podmienia im wartość posiadanych pieniędzy, bo w skali miesiąca przekłada się na realne koszty roboczo-godzin spędzonych na „dekodowanie" „zaszyfrowanej" „intencji".

OBIEKTY

Po ogólnym przeglądzie Building Blocks wrócimy jeszcze do typowych błędów w programowaniu obiektowym oraz do zasad analizy i projektowania obiektowego.

Paradygmat obiektowy

Paradygmat obiektowy składa się z czterech zasad:

- abstrakcja – istnieje agent odbierający sygnały i wykonujący operacje – na Listingu 1 klasa `Client` jest agentem modelującym zasobność klientów,
- enkapsulacja – klasy nie ujawniają implementacji – na Listingu 1 widzimy jedynie „api", nie wnikając w wewnętrzne struktury implementacyjne, przykładowo `Client` może być fasadą do podsystemu płatności i modułu pożyczek,
- polimorfizm – zmienne zachowanie obsługujemy przez specyficzne klasy, ważne jest, że mówimy tutaj o zachowaniu, a nie o strukturach danych
- dziedziczenie – technika wspierająca polimorfizm, czyli dziedziczymy ze względu na wspólne zachowanie, a nie ze względu na wspólne struktury

danych (wrócimy do tego problemu w sekcji poświęconej SOLID)

Command-query Separation

CqS jest zasadą programowania imperatywnego, która dla języków obiektowych może brzmieć następująco: metoda w klasie może być jednym z dwóch:

- command – sygnał wysyłany do obiektu, który zmienia stan obiektu
- query – zwraca stan obiektu, nie zmieniając stanu (oczywiście nie musi być to naiwny getter, a może być to np. projekcja zwracająca snapshot w postaci struktury danych)

Przykład na Listingu 1 ilustruje tę zasadę: metoda `charge` jest rozkazem, a `canAfford` kwerendą. Wielokrotne uruchamianie kwerend nie zmienia stanu.

CqS ma ogromny impakt na techniki tworzenia zaślepek (test double) w testach jednostkowych. Dla command tworzymy mocki, a dla query tworzymy stuby. Zagadnienie to jest szerzej opisane w jednym z poprzednich artykułów z serii – ramka „W sieci".

SOLID i GRASP

Nie będziemy w szczegółach omawiać tutaj zasad GRASP i SOLID; zainteresowanych odsyłam do ramki „W sieci". Jedna z zasad SOLID: Open-closed została omówiona w jednym z poprzedni artykułów, dostępnym do pobrania – link we wspomnianej ramce.

Warto przytoczyć tutaj trzecią zasadę SOLID: Liskov Substitution Principle, która w pragmatycznym ujęciu może być interpretowana następująco: używaj dziedziczenia, tylko gdy będziesz używać polimorfizmu. Zasada ta wyklucza naiwne książkowe modele typu: Pracownik i Klient dziedziczą po wspólnej klasie Użytkownik. Dziedziczą tylko dlatego, że „wyciągamy przed nawias" wspólne dane (imię, nazwisko, konto na Facebooku), ale nie mamy wspólnych zachowań. Czyli wbrew pierwszemu i czwartemu punktowi paradygmatu OO.

SZCZEGÓLNE BUILDING BLOCKS

Ramka „Podstawowe rodzaje Building Blocks" przedstawia bardziej szczegółowy podział bazowych Building Blocks. Są tam wyróżnione podtypy charakterystyczne dla Domain Driven Design.

DODATKOWE BB

Omówiliśmy bazowe Building Blocks, które pojawiają się na co dzień w każdym kodzie. W bardziej specyficznych zastosowaniach pojawią się:

- Komunikaty: zdarzenia, rozkazy, intencje
- Porady aspektowe
- Konstrukcje specyficzne dla frameworków: kontrolery, presentery, view-modele itd.
- itd.

Warto znać charakterystykę i odpowiedzialność każdego z nich. Poszerza to wachlarz „klocków", jakich używa Twoja lewa półkula do projektowania architektury aplikacji, pozwala na stosowanie ich ze świadomością konsekwencji, ułatwia komunikację z innymi członkami zespołu i przyspiesza uczenie się technologii, które bazują na Building Blocks.

SZERSZY KONTEKST

Zainteresowanych tematem Building Blocks zachęcam do zapoznania się ze spójnymi metodykami opartami o standaryzowane Building Blocks: Responsibility Driven Design i Domain Driven Design.

Podstawowe rodzaje Building Blocks

Obiekty	Aggregate	Są rozróżnianie po jakiegoś rodzaju identyfikatorach Eksponują zachowanie Ukrywają struktury Posiadają dobrze określoną granicę (nie można trawersować ich getterami) Modelują reguły biznesowe obowiązujące na składowych agregatu.
	Value Object	Są rozróżnianie po wartościach Są niemodyfikowalne (modyfikacja zwraca kopię)
Struktury	DTO	Eksponują struktury Nie posiadają zachowania (zachowanie przeniesione do procedur)
	Anemiczne encje	Rodzaj DTO – mapowane na wiersze w bazie danych przez ORM
Procedury	Service	Sekwencja kroków do wykonania Orkiestrują obiekty i inne procedury Mogą zwracać wynik działania (np. status) Mają zależności do innych serwisów Powodują efekty uboczne Wynik zależy od stanu systemu (obiektów zależnych)
	Factory	Produkują obiekty Walidują półprodukty, z których budowane są obiekty
	Repository	Abstrakcja nad magazynem danych (bp. Bazą danych) Zwracają i zapisują obiekty
Funkcje	Service	Przetwarzają jedne obiekty/struktury w inne Nie mają zależności Nie powodują efektów ubocznych Mogą być domykane Wynik zawsze taki sam dla danego zestawu danych wejściowych Rodzaj „higher order function"
	Policy	jw. Najprostsza implementacja: Strategy Design Pattern Rodzaj „domknięcia" (closure)
	Metody	Metoda, która zmienia stanu obiektu, jedynie przetwarza stan w wynik Analogiczna do serowisu, którego pierwszym parametrem byłby dany obiekt (ale w tym przypadku obiekt pozostaje hermetyczny) Również może być domykana

W sieci

- Projekt referencyjny https://github.com/BottegaIT/ddd-leaven-v2
- Poprzednie części serii: http://www.bottega.com.pl/artykuly-i-prezentacje#receptury
- SOLID http://en.wikipedia.org/wiki/Solid_(object-oriented_design)
- GRASP http://en.wikipedia.org/wiki/GRASP_(object-oriented_design)
- CqS http://en.wikipedia.org/wiki/Command-query_separation
- Wykorzystanie CqS w testowaniu automatycznym: http://bottega.com.pl/pdf/materialy/receptury/cqs-mock.pdf
- Wykorzystanie Open-close Principle http://bottega.com.pl/pdf/materialy/receptury/ocp.pdf
- Building Blocks DDD: http://www.bottega.com.pl/pdf/materialy/ddd/ddd1.pdf

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).