Autor

Receptury projektowe – niezbędnik początkującego architekta

Część III: Zarządzenie złożonością przez trójpodział logiki – Open/closed principle w praktyce

W modelu złożonego problemu zwykle możemy wydzielić strukturę wyższego rzędu charakteryzującą się różną podatnością kodu źródłowego na zmiany. W niniejszym artykule zostanie przedstawiony meta-model, którym możemy posiłkować się w zmaganiach ze złożoną logiką biznesową. Meta-model będzie praktyczną realizacją drugiej zasady SOLID: Open/closed principle, która pozwala tworzyć rozwiązania otwarte na rozbudowę (rozbudowa to nie to samo co zmiana).

O serii „Receptury projektowe"

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

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

SOLIDNE PROGRAMOWANIE I PROJEKTOWANIE

Zasady SOLID (zobacz ramkę „W sieci") dostarczają nam ogólnych wytycznych w zakresie projektowania obiektowego. Ich ogólność pozwala na interpretację na różnych poziomach, zarówno na poziomie Object Oriented Design, jak i wyżej – na poziomie architektury aplikacyjnej i systemowej.

Jednak jak to przy takiej ogólności bywa, pojawia się pytanie: jak w praktyce podążać za tymi wytycznymi?

OPEN/CLOSED PRINCIPLE – ALE JAK TO ZAIMPLEMENTOWAĆ?

W niniejszym artykule zajmiemy się jedną z zasad SOLID: Open/closed principle, która brzmi: „kod/projekt powinien być otwarty na rozbudowę, ale zamknięty na zmiany".

Zmiana to modyfikacja istniejącego kodu źródłowego, a co za tym idzie konieczność przeprowadzenia (jeżeli nie istnieją automatyczne) testów oraz konieczność przeprowadzenia przeglądu kodu (no, chyba że przeglądy nie należą do procesu produkcji:).

Rozbudowa polega natomiast na dodaniu nowych klas bez modyfikacji istniejących.

W językach obiektowych chodzi oczywiście o „wpinanie" nowej implementacji interfejsu (na zasadzie pluginu). Początkujący projektanci, widząc takie rozwiązania, komentują go zwykle w ten sposób: „tak, technicznie to łatwe, ale jak mamy wpaść na pomysł takiego designu". Starsi koledzy robią wówczas minę mędrca i odpowiadają metaforycznie: „kod powinien być otwarty na rozbudowę niczym kwiat lotosu o świcie i zamknięty na zmiany niczym kwiat lotosu o zmierzchu".

My jednak podejdziemy do problemu metodycznie, stosując niemal „mechaniczną procedurę" tworzenia tego typu designu.

META-MODEL – TRZY RODZAJE LOGIKI

Analizę złożonych domen warto zacząć od wydzielania w modelu części charakteryzujących się różną podatnością na zmiany – zmiany w sensie zmian w kodzie źródłowym, a nie w sensie zmian wartości (np. w bazie danych).

W wielu problemach można wydzielić trzy osobne sfery logiki: logikę stabilną, domknięcia stabilnej logiki oraz wybór domknięcia. Pomocna na tym etapie może być prosta wizualizacja modelu: dzielimy diagram na 3 części (Rysunek 2) i zastanawiamy się, w której części umieścić kolejne odpowiedzialności.

Logika Stabilna

Logika stabilna relatywnie rzadko podlega zmianom. Są to główne operacje – klasy wyższego rzędu lub po prostu główne serwisy (procedury) w podejściu proceduralnym (duża część kodu stworzonego w językach obiektowych charakteryzuje się proceduralną naturą).

Natomiast zmienność „przykrywamy" stabilnym interfejsem i delegujemy na zewnątrz stabilnych struktur. Dzięki temu „chronimy" stabilny kod przed zmianami wynikającymi ze zmian wymagań w miejscach „niestabilnych".

Przykładem może być klasa BookKeeper z Listingu 1 (patrz sekcja „Przykłady), która modeluje ogólny algorytm księgowania; specyfika obliczania podatków (charakterystyczna dla danego kraju) została wyniesiona poza interfejs TaxPolicy.

Wyzwaniem podczas projektowania stabilnej logiki jest strategiczne opracowanie „punktów rozszerzeń", które będą pozwalać na rozbudowę (a nie zmianę) podczas kolejnych zmian wymagań.

Typowe Building Blocks DDD będące kandydatami do zawierania stabilnej logiki:

- Agregaty
- Serwisy Domenowe
- Serwisy Aplikacyjne

Domknięcia logiki

Implementacje interfejsów możemy potraktować jako domknięcia logiki nadrzędnej. W przykładzie z Listingu 1 mamy główny (nadrzędny) algorytm księgowania, natomiast implementacje interfejsu TaxPolicy pozwalają na „domknięcie" całości o szczegóły (szczegóły z punktu widzenia logiki nadrzędnej, bo obliczanie podatku nie jest problemem trywialnym).

Wyzwaniem podczas projektowania domknięć jest opracowanie stabilnego interfejsu – czyli takiego, który będzie na tyle ogólny, aby przetrwać kolejne zmiany wymagań, ale jednocześnie nie będzie łamał czwartej zasady SOLID: Interface Segregation.

Typowe Building Blocks DDD będące kandydatami do zawierania logiki domknięć:

- Polityki
- Specyfikacje

Wybór domknięcia

Mamy zatem sytuację, w której rozdzieliśmy logikę na stabilny (w sensie zmian kodu) „core" oraz wiele możliwych domknięć tej logiki. Pojawia się pytanie: jak wybrać odpowiednie domknięcie?

Być może jest to prosty problem, który możemy rozwiązać na kilka sposobów:

- wybór domknięcia na GUI przez użytkownika
- pobranie domknięcia z konfiguracji
- wstrzyknięcie domknięcia przy pomocy kontenera Inversion of Control i techniki Dependency Injection

Jeżeli jednak wybór domknięcia sam w sobie wymaga stworzenia modelu i zaprogramowania algorytmu wyboru, wówczas może pojawić się trzeci rodzaj logiki: wybór domknięć. Logikę tę hermetyzujemy w Fabrykach.

W naszym przykładzie mamy księgowego, który jest domknięty sposobem obliczania podatku. Jeżeli tworzymy system, który nie jest jedynie magazynem i prostym transformatorem danych, a zarządza on również autonomicznym podejmowaniem decyzji, to może pojawić się wymaganie: jeżeli klienci prowadzą działalność w różnych krajach, to system powinien dobrać politykę obliczania podatku tak, aby było to najbardziej korzystne dla klienta.

Generalnie jeżeli pojawi się trzeci rodzaj logiki biznesowej – wybór domknięcia – to warto odseparować go od pozostałych rodzajów logiki i hermetyzować w serwisach/fabrykach. Logika tego rodzaju podlega osobnym zmianom niż logika główna i logika domknięć. Być może nie uda się nam stworzyć generycznego modelu parametryzowanej fabryki i będziemy zmuszeni dokonywać zmian w kodzie źródłowym fabryki. Ale przynajmniej będą to zmiany ograniczone do konkretnego miejsca, bez potrzeby „brudzenia" zmianami (testowanie i przeglądy) kodu pozostałych dwóch rodzajów logiki.

Typowe Building Blocks DDD będące kandydatami do zawierania logiki wyboru domknięcia:

- Serwisy Domenowe
- Fabryki
- Repozytoria

Co fabrykować?

Możemy fabrykować obiekty stabilne, które będą miały już odpowiednio ustawione zależności do wynikających z logiki domknięć. Czyli w naszym przypadku możemy produkować w fabryce księgowego, z odpowiednią polityką obliczania podatków. Dzięki temu wyższe warstwy nie muszą znać takich szczegółów jak istnienie domknięć.

Innym podejściem jest produkowanie w fabrykach jedynie domknięć. Daje nam to możliwość zdecydowania w wyższych warstwach czy będziemy korzystać z fabryk, czy może otworzymy możliwość wyboru domknięcia np. poprzez GUI.

Oba podejścia zostały zilustrowane na Listingu 1.

STRUKTURA A TREŚĆ

Do tej pory omawialiśmy strukturę problemu – podział na 3 rodzaje logiki pod względem podatności na zmiany. Jest to swego rodzaju meta-model, który może być stosowany jako framework mentalny. Framework mentalny, czyli narzędzie, które będzie nam służyło do „porządkowania" myśli.

Możemy wyobrazić go sobie jako foremki, do których wsypujemy mokry piasek, aby nadać mu kształt. Im więcej foremek mamy w swojej „skrzynce z narzędziami", tym więcej możliwości mamy w fazie projektowania. Przykładowo projektant, który zna jedynie dwie foremki: serwis i encja, albo jedną: hashmapa hashmap, potrafi każdy problem zamodelować przy pomocy tych foremek.

Przeprowadzając analizę danego problemu – uzbrojeni w framework – możemy zacząć zadawać pytania:

- co jest stabilne, co relatywnie rzadko podlega zmianom, czym jest core?
- co się zmienia, jaki jest stabilny kontrakt tych zmian?
- jak podjąć decyzję o wyborze domknięcia, jakie reguły tym rządzą?

Zatem struktura zamiast wyłaniać się przypadkowo z chaosu może być nałożona na wstępie, aby destylować design w momencie analizy problemu.

Oczywiście przyjęcie nieodpowiedniego frameworka mentalnego na wstępie może być pomyłką. Jednak framework mentalny prezentowany w tym artykule jest na tyle ogólny, że pomyłka nie będzie trudna do „odwrócenia" ani nawet szkodliwa.

W kolejnym rozdziale przedstawię metafory wizualne, które pomagają zarządzać strukturą. Następny rozdział będzie zawierał trzy przykłady, których treść wpasowuje się w przedstawioną strukturę.

METAFORY (WIZUALNE)

W ciągu roku pracuję średnio z 30 zespołami, co daje mi w sumie obraz kilkuset różnych podejść do komunikowania pomysłów i rozwiązań. Jeden z wniosków, jaki się wyłania z moich obserwacji, jest następujący: brak komunikacji jest przyczyną wielu problemów; a próby opisowego komunikowania złożonych koncepcji są, delikatnie mówiąc, nieefektywne.

Ciekawe są również obserwacje strategii komunikacji wizualnej i jej niedopasowania pomiędzy poszczególnymi członkami zespołu. Z uwagi na różne wewnętrzne modele reprezentowania złożonych struktury w niniejszym tekście przedstawię trzy przykładowe sposoby wizualizowania omawianego frameworka mentalnego.

Technikom zwinnego dokumentowania architektury poświęciłem artykuł w poprzednim numerze (do pobrania, patrz ramka „W sieci"), natomiast w tym miejscu skupimy się tylko na wizualizacji meta-modelu trzech rodzajów logiki.

Metafora linearna

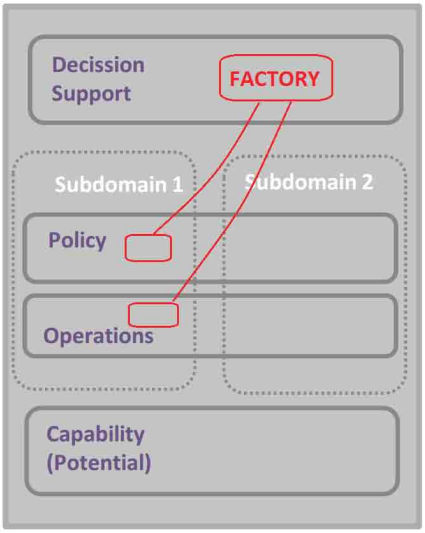
W artykule „Zaawansowane modelowanie DDD – techniki strategiczne: konteksty i architektura zdarzeniowa" (do pobrania, patrz ramka „W sieci") przedstawiłem linearną (w sensie układania rodzajów logiki na stosie) wersję opisywanego frameworka mentalnego. W tym miejscu jedynie przypomnę ogólne założenia. Model domenowy możemy podzielić na 4 poziomy (od dołu do góry):

- decission support – mechanizmy analityczne, rodzaj „inteligencji"
- policy – domknięcia operacji, model reguł i ograniczeń
- operations – konkretne operacje zbudowane
- capability – bazowy model, ogólny potencjał biznesu

Mamy tutaj do czynienia z:

- klasami *capability* i *operations*, które są relatywnie stabilne
- interfejsami *policy*, które są domknięciami operacji
- fabrykami mogącymi zawierać „inteligencję" (np. sugerowanie kraju do opodatkowania) rezydującymi na poziomie *decission support*

Autor



Rysunek 1. Meta-model trzech rodzajów logiki – podejście linearne

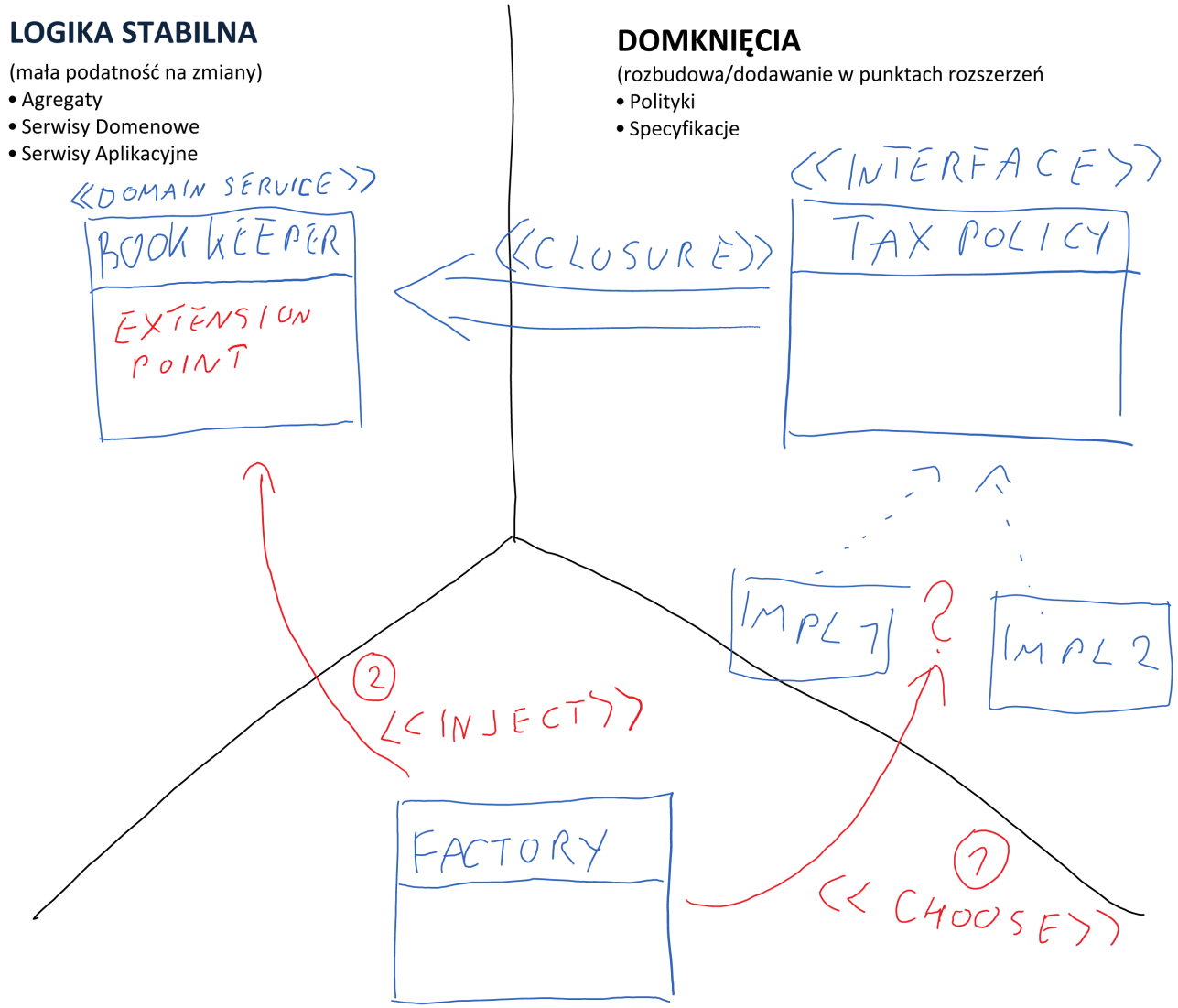
Metafora 2D

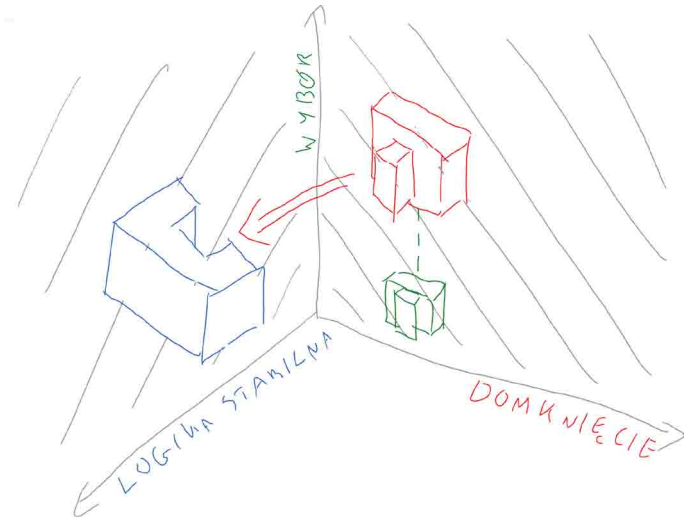
Podejście linearne może być ograniczające dla strategii wyobrażeniowych konkretnych osób, dlatego warto posłużyć się płaszczyzną, na której mamy trzy obszary logiki (zob. Rysunek 2).

Metafora 3D

Część osób wyobraża sobie modele programistyczne jako trójwymiarowe przestrzenie zawierające trójwymiarowe wizualizacje obiektów (czasem są to ruchome „maszyny"). Tego typu rysunki są bardziej pracochłonne, ale dla złożonych problemów mogą zaowocować efektem „Acha!" (zob. Rysunek 3).

Rysunek 2. Metamodel trzech rodzajów logiki – podejście dwuwymiarowe





Rysunek 3. Metamodel trzech rodzajów logiki – podejście trójwymiarowe

Podejście funkcyjne

Programiści używający języków funkcyjnych zapewne od razu skojarzyli nasz framework mentalny z pojęciami takimi jak *higher order functions* i *closures*. Po raz kolejny sprawdza się powiedzenie „język, jakim mówisz, determinuje sposób, w jaki myślisz".

… a w podejściu obiektowym

Natomiast dla programistów obiektowych mogę zasugerować pewną sztuczkę mentalną. Często programowanie obiektowe jest nauczane w naiwny (i błędny) sposób jako separację rzeczowników (klasy) i czasowników (metody).

W opisywanym podejściu czynność (obliczenie podatku – Listing 1, obliczenie rabatu – Listing 2, obliczenie odpowiedniego przełożenia dla skrzyni biegów – Listing 3) urosła do rangi „first class citizen" w języku obiektowym, czyli do rangi obiektu. Warianty wykonania czynności zostały zamknięta do obiektów implementujących wspólny interfejs (emulowanie funkcji). Spoglądając z innej strony, jest to klasyczny wzorzec: Strategy Design Pattern. Strategie możemy następnie dekorować (Decorator Design Pattern), nadawać im strukturę łańcucha (Chain of Responsibility Design Pattern), drzewa (Specification Design Pattern) etc.

PRZYKŁADY

Spójrzmy na kilka przykładów, które „wypełniają treścią" strukturę naszego meta-modelu. Przypomnę, że w każdym z nich dyskusyjne jest, czy chcemy fabrykować gotowe do pracy obiekty stabilne (z wstrzykniętymi domknięciami) czy jedynie domknięcia (pozwalając wyższej warstwie na orkiestrację całości i ew. użycie innych domknięć – np. przekazanych za pomocą GUI).

Podatki

Przykład opracowany na potrzeby serii artykułów poświęconych DDD. Projekt i artykuły dostępne w ramce „W sieci".

Struktura:

- stabilna: klasa BookKeeper odpowiedzialna za wystawienie faktury (Invoice) na podstawie zamówienia (Order)
- domknięcie: punktem rozszerzenia jest wywołanie po interfejsie polityki obliczania podatku (TaxPolicy), która jest charakterystyczna dla danego kraju
- wybór domknięcia: BookKeeperFactory decyduje o tym, wg zasad jakiego kraju naliczyć podatek; zwraca BookKeeper gotowego do pracy (z wstrzykniętą polityką)

Listing 1. Logika stabilna: BookKeeper, domknięcia: TaxPolicy

```
public class BookKeeper {
   @Inject
   private TaxPolicy  taxPolicy;

   public Invoice issuance(Order order){
      Invoice invoice = new Invoice(order.getClient());

      for (OrderedProduct product : order.getOrderedProducts()){
         Money net = product.getEffectiveCost();
         Tax tax = taxPolicy.calculateTax(product.getType(), net);

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

      return invoice;
   }

}

public interface TaxPolicy {
   public Tax calculateTax(ProductType productType, Money net);
}

public class BookKeeperFactory {
   public BookKeeper create(...){...}
}
```

Rabaty

Struktura

- stabilna: model Rezerwacji (Reservation) produktów, która odpowiada za wyliczenie planu cenowego (PricingPlan)
- domknięcie: strategia obliczania rabatów (DiscountStrategy), będąca parametrem metody obliczającej plan cenowy
- wybór domknięcia: w tym modelu Rezerwacja nie zawiera strategii rabatowej, jest ona parametrem przekazywanym przez wyższą warstwę; aczkolwiek wciąż możemy korzystać z Fabryki rabatów, która zawiera logikę budowania rabatów najmniej korzystnych dla klienta

Listing 2. Logika stabilna: Reservation, domknięcia: DiscountStrategy

```
public class Reservation {
   private List<Product> products;

   public PricingPlan generatePricingPlan(DiscountStrategy
   discountStrategy){
      PricingPlan pricingPlan = new PricingPlan(client);

      for (Product product : products){
         pricingPlan.addItem(product,
            discountStrategy.calculate(product));
      }

      return invoice;
   }

}

public interface DiscountStrategy {
   public Discount calculate(Product product);
}

public class DiscountFactory {
   public DiscountStrategy create(...){...}
}
```

Moment obrotowy

Przykład przygotowany na potrzeby artykułu *„Mock czy Stub? Command-query Separation prawdę ci powie"* (do pobrania, patrz ramka „W sieci").

Struktura:

- stabilna: model sterownika automatycznej skrzyni biegów
- domknięcie: strategia wyliczania odpowiedniego biegu
- wybór strategii: ustawienie sterownika w jeden z trybów: eco/comfort/sport/sport+

Listing 3. Logika stabilna: AutomaticTransmissionControler, domknięcia: GearCalculator

```
public class AutomaticTransmissionController {

   public enum DrivingMode {
      ECO, COMFORT, SPORT, SPORT_PLUS
   }

   private AutomaticGearBox gearBox;
   private EngineMonitoringSystem engineMonitoringSystem;
   private DrivingMode drivingMode;

   public void changeMode(DrivingMode drivingMode){
      this.drivingMode = drivingMode;
      handleChange(0, 0);
   }

   public void handleGas(double throttle){
      handleChange(throttle, 0);
   }

   public void handleBreaks(double breakingForce){
      handleChange(0, breakingForce);
   }

   /**
    * maintains gear that is proper for current:
    * driving mode, engine's RPM
    * and given gas throttle and breaking force
    *
    * @param throttle gas throttle <0,1>
    * @param breakingForce breaking force <0,1>
    */
   private void handleChange(double throttle,
            double breakingForce){
      GearCalculator calculator = createGearCalculator(drivingMode)
      int gear = calculator.calculate(egineMinitoringSystem.
      getCurrentRpm(),
            throttle, breakingForce);

      if(gear != gearBox.getGear()){
         this.gearBox.changeGear(gear);
      }
   }
}
public interface GearCalculator {
   public int calculate(int currentRPM, double throttle,
                  double breakingForce);
}
```

WPŁYW NA TESTABILITY

Wprowadzenie struktury tego meta-modelu pozwala na drastyczne zwiększenie testability – czyli podatności kodu na testowanie automatyczne.

Każdy z trzech rodzajów logiki pokrywamy osobnymi testami:

- logika stabilna – testy jednostkowe operują na Mockach naszych domknięć
- domknięcia – osobne testy jednostkowe
- fabryki – mogą być kłopotliwe w testowaniu automatycznym z uwagi na ilość zależności (jednak pozostałe części kodu są dzięki temu relatywnie łatwe w testowaniu)

Takie podejście redukuje ilość przypadków testowych. Załóżmy, że przykładowa klasa BookKeeper wymaga 100 testów hipotez biznesowych, polityka obliczania podatku wg reguł polskich kolejnych 100 testów i polityka obliczania podatku wg reguł niemieckich również 100 testów. W sumie 300 przypadków testowych.

Gdyby jednak cały nasz kod miał strukturę monstrualnego ośmiotysięcznika (jedna klasa, setki instrukcji IF, 8kloc), wówczas mielibyśmy scenariusze księgowania z polskimi regułami: 100 * 100 oraz scenariusze księgowania z niemiecki regułami: również 100 * 100 – w sumie 20 000 przypadków testowych.

PODSUMOWANIE

- Celem artykułu było przedstawienie syntezy kilku podejść i technik:
- Zasady Open/closed principle (SOLID)
- Projektowania architektury otwartej na plugin
- Zwiększenia testability
- Wykorzystania kontenera Inversion of Control
- Wizualizowania designu w różnych formach przestrzennych
- Stosowania meta-modelu w fazie analizy jako frameworka mentalnego dostarczającego nowych „foremek" dla złożonej treści

Mam nadzieję, że całościowe spojrzenie na projektowanie obiektowe z różnych perspektyw przyczyni się do rozwoju Twojej kariery:)

W sieci

- Zasady SOLID: http://en.wikipedia.org/wiki/Solid_(object-oriented_design)
- Poprzednie artykuły, do których odnoszę się w tekście: http://bottega.com.pl/artykuly-i-prezentacje#receptury
- Przykładowy projekt, z którego zaczerpnięto przykłady: http://bottega.com.pl/ddd-cqrs-sample-project

Meta-model trzech rodzajów logiki

	Charakterystyka	Building Blocks DDD
Logika stabilna	kod źródłowy nie wymaga (częstych) zmian, zawiera punkty rozszerzeń	Agregat Serwis Domenowy Serwis Aplikacyjny
Domknięcia logiki	kod źródłowy jest rozbudowywany o nowe implementacje kontraktu domknięcia (interfejsu)	Polityka Specyfikacja
Wybór domknięcia	realizowany np. w Fabrykach, wymaga zmian, ew. może być oparty o generyczny mechanizm konfiguracji	Fabryka Repozytorium Serwis Domenowy Kontener Inversion of Control (Dependency Injection na podstawie: XML, adnotacje, metody fabrykujące działające w runtime)

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).