Receptury projektowe – niezbędnik początkującego architekta

Część I: Cztery smaki odwracania (i utraty) kontroli: Dependency Injection, Events, Aspect Oriented Programming, Framework

Paradygmat Inversion of Control dla jednych programistów (np. skupionych wokół technologii Java) jest standardem i "naturalnym porządkiem świata", a dla innych pojawiającą się nowinką, która obiecuje rozwiązanie wszystkich problemów. W pierwszej części naszej serii przyjrzymy się czterem poziomom odwracania kontroli: Dependency Injection, Events, Aspect Oriented Programming, Framework pod kątem: problemów, jakie rozwiązują, motywacji - czyli kontekstu opłacalności ich stosowania oraz technikom implementacji.

PARADYGMAT INVERSION OF CONTROL

Standardowa (nieodwrócona) kontrola

Mówiąc o odwracaniu kontroli, mamy na myśli odwrócenie sterowania wykonaniem instrukcji programu.

Zanim jednak przejdziemy do omawiania kolejnych podejść do odwracania kontroli, musimy sobie uświadomić, czym jest normalna – nieodwrócona kontrola w językach obiektowych. Jak to zwykle bywa z pojęciami podstawowymi, są one raczej intuicyjne, więc ich definiowanie wydaje się sztuczne i siłowe. Zatem trywialny pseudokod z Listingu 1 będzie towarzyszył nam podczas całego artykułu jako rama ilustrująca kolejne poziomy odwracania kontroli.

Listing 1. Pseudokod ilustrujący kontrolę nieodwróconą

```
public class MojaKlasa{
   // KlasaWspółpracująca może być również interfejsem
   private KlasaWspółpracująca obiektWspółpracujący =
         new KonkretnaKlasaWspółpracująca();
   public void metoda(){
      //kolejne instrukcje
      obiektWspółpracujący.współpracuj();
   }
}
```

Pseudokod z Listingu 1 zawiera charakterystyczne dla nieodwróconej kontroli konstrukcje:

- obiekt tworzy potrzebne do swej pracy obiekty współpracujące, znając ich ilość
- podczas tworzenia obiekt decyduje o konkretnym typie obiektów współpracujących
- instrukcje są wykonywane kolejno i synchronicznie
- znamy wynik wykonania instrukcji obiektów współpracujących

W jakim celu odwracamy kontrolę

Generalnie kontrolę odwracamy po to, aby rozluźnić zależności pomiędzy obiektami (Coupling) lub wręcz się ich pozbyć. Na wyższych poziomach pozwala nam to na:

- tworzenie kodu poddającego się testowaniu automatycznemu
- tworzenie programów, których działanie zmienia się dynamicznie
- tworzenie programów otwartych na rozbudowę poprzez pluginy (różnego rodzaju)
- tworzenie systemów charakteryzujących się modularnością, gdzie moduł rozumiemy jako niezależną część, którą możemy wymieniać
- tworzenie systemów dających się skalować oraz działających asynchronicznie

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 radzenia sobie z wyzwaniami niezależnymi od konkretnej technologii, wraz z pełną świadomością konsekwencji płynących z podejmowanych decyzji projektowych.

TRZY POZIOMY ODWRACANIA KONTROLI

Reifikacje: IoC to nie to samo co DI

W niektórych źródłach (np. blogi) używa się wymiennie terminu Inversion of Control (IoC) oraz Dependency Injection (DI). IoC jest ogólną ideą, natomiast DI jest jedną z reifikacji – czyli konkretyzacją idei, ale jeszcze nie implementacją (implementacje DI to np. konkretne kontenery wstrzykiwania zależności).

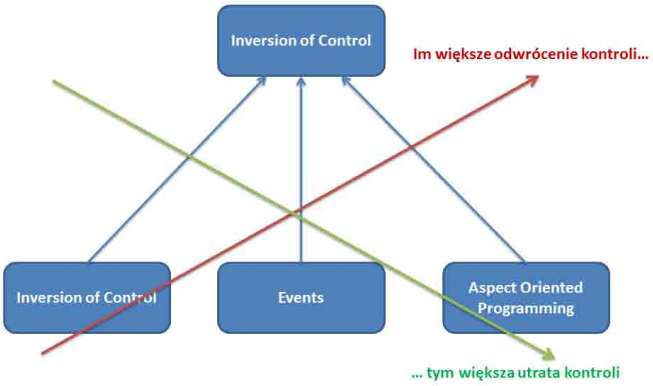
Ramka „Reifikacje Inversion of Control" podsumowuje kolejne poziomy odwracania kontroli, które będziemy w szczegółach omawiać w niniejszym artykule.

Reifikacje Inversion of Control– kolejne poziomy odwracania kontroli

- Dependency Injection – obiekt nie tworzy obiektów współpracujących (nie decyduje o ich typie ani zależnościach)
- Events – obiekt nie tylko nie zna typu obiektów współpracujących, ale nie zna również: ich ilości, kolejności wykonania, rezultatu ich pracy i ew. błędów, czasu wykonania (ew. asynchroniczność)
- Aspect Oriented Programming – obiekt praktycznie nic nie wie o dodatkowych operacjach
- Monady – konstrukcja z języków funkcyjnych, poza zakresem niniejszego artykułu
- Frameworki – ogólny przepływ jest sterowany przez framework, natomiast w istotnych z punktu widzenia specyfiki problemu momentach sterowanie trafia do naszego obiektu

Im bardziej odwracasz kontrolę, tym bardziej tracisz kontrolę

Techniki odwracania kontroli na Rysunku 1 zostały uszeregowane wg siły, z jaką odwracają kontrolę. Najpierw pozbywamy się kontroli nad tworzeniem obiektów współpracujących, następnie nad ich ilością, wynikiem działania i czasem aż po całkowitą utratę kontroli.



Rysunek 1. Odwrócenie kontroli wiąże się z utratą kontroli

Tracimy kontrolę, nie ze względu na niedoskonałość danego podejścia. Lepszą metaforą może być pozbycie się kontroli – świadome, po to, aby jednocześnie pozbyć się zależności i otworzyć projekt na rozbudowę.

POZIOM 1: DEPENDENCY INJECTION

Wstrzykiwanie zależności jest najbardziej podstawową techniką odwracania kontroli, a zarazem wiąże się z najmniejszą utratą kontroli – jeżeli kontroli potrzebujemy.

Problem: Nadużywanie operatora new

Wstrzykiwanie zależności rozwiązuje generalny problem cyklu życia obiektu, oraz decyzji o sposobie jego utworzenia.

Jeżeli w jakimś miejscu kodu stosujemy operator new, to wówczas:

- podejmujemy decyzję o konkretnym typie tego obiektu – wiążemy się z konkretną implementacją
- przekazujemy parametry do konstruktora – musimy w danym miejscu posiadać wiedzę o tych wartościach oraz generujemy kolejny Coupling
- i co najważniejsze: podejmujemy decyzję o utworzeniu kolejnej instancji - „szastamy aktem tworzenia" z tego tylko powodu, że potrzebujemy referencji do danego interfejsu

Powyższe konsekwencje rodzą problemy związane z rozbudową (zmianą konkretnego typu w wielu miejscach) oraz zarządzaniem siecią współpracujących obiektów.

Idea: Nie wiem kim jesteś

Listing 2 najlepiej oddaje ideę odwrócenia kontroli przez wstrzykiwanie zależności. Kod naszej klasy nie zajmuje się tworzeniem obiektów współpracujących, zamiast tego są mu one „wstrzyknięte" poprzez konstruktor, metodę lub refleksję (w językach wspierających ten mechanizm).

Listing 2. Pseudokod ilustrujący odwrócenie kontroli poprzez Wstrzykiwanie Zależności

```
public class MojaKlasa{
   // KlasaWspółpracująca może być również interfejsem
   //nie tworzymy
   private KlasaWspółpracująca obiektWspółpracujący;

   //wstrzykiwanie przez konstruktor - odpowiednie do
   //zależności wymaganych ponieważ wyraźnie wskazuje intencję
   public  MojaKlasa(KlasaWspółpracująca obiektWspółpracujący)
{
      this.obiektWspółpracujący = obiektWspółpracujący;
   }

   //wstrzykiwanie przez metodę - odpowiednie do
   //zależności opcjonalnych
   public setObiektWspółpracujący (KlasaWspółpracująca
                                   obiektWspółpracujący){
      this.obiektWspółpracujący = obiektWspółpracujący;
   }

   public void metoda(){
      //kolejne instrukcje
      obiektWspółpracujący.współpracuj();
   }
}
```

Warto zwrócić uwagę, że w powyższym pseudokodzie nie posiłkujemy się ServiceLocatorem w celu uzyskania dostępu do obiektu Współpracującego. Dzięki temu nasz kod jest wolny od zależności technicznych, a co za tym idzie łatwiej poddaje się testowaniu automatycznemu – możemy na czas testów wstrzykiwać zaślepki (Fake/Mock/Stub) obiektów zależnych.

Motywacja: Możesz wstrzykiwać nie tylko DAO – naprawdę!

Książkowe przykłady ilustrują zwykle wstrzykiwanie zależności na przykładzie wstrzykiwania Data Access Objects (lub Repozytoriów tudzież kontekstu persystencji) do Serwisów Aplikacyjnych. Tego typu przykłady nie są szczególnie motywujące, ponieważ:

- niezwykle rzadko dokujemy wymiany obiektów warstwy dostępu do danych
- potencjalnie możemy wymieniać obiekty dostępu do danych w testach jednostkowych, ale zwykle Serwisy Aplikacyjne nie są poddawane tego typu testom

Zatem cała „ceremonia" związana z Dependency Injection wydaje się zbędna. Jedynym zyskiem może być zintegrowanie jej z Aspect Oriented Programming – o czym w dalszej części.

Możemy natomiast wstrzykiwać do obiektów wykonujących np. logikę biznesową innego obiekty wykonujące logikę biznesową niższego rzędu.

Zastosowanie: Dostrajanie systemu per wdrożenie – Wzorzec Policy

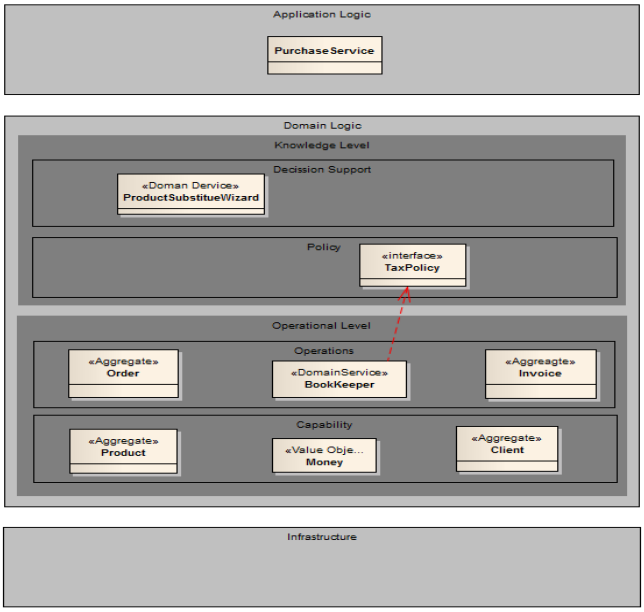
Wstrzykiwanie zależności pozwala tworzyć w łatwy sposób systemy oparte o architekturę pluginów typu „zróbmy pewną czynność inaczej". Listing 3 ilustruje przykład serwisu, który wystawia fakturę na podstawie zamówienia.

Załóżmy, że ogólny algorytm fakturowania w każdym kraju jest generalnie taki sam, jednak w różnych krajach różni się ona sposobem obliczania podatku.

Listing 3 Przykład wstrzyknięcia przez konstruktor Polityki Obliczania Podatku do Serwisu Księgowania

```
public interface TaxPolicy{
   public Tax  calculateTax(ProductType productType,
      Money netPrice)
}

public class BookkeeperService {
   private TaxPolicy  taxPolicy;

   public BookkeeperService(TaxPolicy taxPolicy){
      this. TaxPolicy =  taxPolicy;
   }

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

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

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

      return invoice;
   }
}
```

Zmienną część możemy wynieść poza stabilny interfejs TaxPolicy. Teraz np. podczas wdrażania systemu w konkretnym kraju będziemy sterować tym, jaka konkretna implementacja TaxPolicy ma zostać wstrzyknięta do BookkeeperService. O tym, w jaki sposób możemy sterować wstrzyknięciem, traktują sekcje „Technika wstrzykiwania".



Rysunek 2. Miejsce na wstrzykiwane polityki w architekturze aplikacji

Polityki są jednym z Building Blocks w Domain Driven Design. Building Blocki to standardowe elementy, z których budujemy modele domen biznesowych. Jednym z takich klocków jest polityka, która modeluje wariację pewnej operacji biznesowej. 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 głównych operacji, które są stabilne.

Posługiwanie się wzorcem Polityki pozwala już na etapie modelowania odnaleźć w projekcie miejsca, które są rodzajem pluginu, oraz podjąć decyzję o konieczności ich wstrzyknięcia np. na podstawie zewnętrznej konfiguracji.

Zastosowanie: Testy automatyczne

Przykładowa klasa BookkeeperService charakteryzuje się wysoką podatnością na testowanie jednostkowe. Możemy bowiem w łatwy sposób na potrzeby testów jednostkowych przekazać do jej konstruktora zaślepkę (Fake/Stub/Mock) TaxPolicy. Jest to niejako „skutek uboczny" projektowania sterowanego przez Dependency Injection.

Klasę BookkeeperService możemy również testować w ujęciu komponentowym, gdzie będzie ona tworzona przez kontener wstrzykiwania zależności (np. Spring). Mamy wówczas możliwość przygotowania na potrzeby testów komponentowych specjalnej konfiguracji wstrzykiwania zależności, która spowoduje wstrzyknięcie zaślepki TaxPolicy – osiągniemy w ten sposób stan taki sam jak w przypadki testów jednostkowych.

Technika wstrzykiwania: Fabryki wstrzykujące

Analizując kod z Listingu 3, możemy dojść do wniosku, że skoro konstruktor klasy BookkeeperService wymaga TaxPolicy, to przesuwamy problem wywołania operatora new dla konkretnej implementacji polityki podatkowej do warstwy wyższej. Zapewne nie chcielibyśmy, aby wyższe warstwy (a ostatecznie warstwa prezentacji) znały szczegóły wyboru implementacji i zajmowały się składaniem obiektów.

Najprostszym rozwiązaniem jest posłużenie się idiomem Fabryki. W naszym przypadku mamy dwie możliwości przedstawione na Listingu 4: fabrykowanie BookkeeperService gotowego do pracy (wraz z TaxPolicy) albo fabrykowanie jedynie konkretnej implementacji TaxPolicy. Podejście pierwsze ukrywa szczegóły i przez to jest zwykle lepszym wyborem.

Listing 4. Podejścia do fabrykowania obiektów

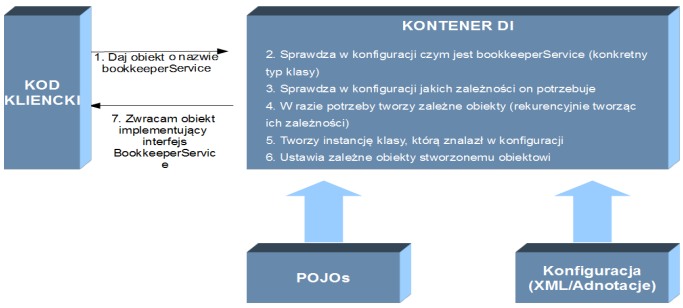
```
//fabrykowanie gotowego obiektu
BookkeeperService bookkeepr = bookkeeperFactory.create(...)
//fabrykowanie jedynie polityki
TaxPolicy taxPolicy = taxPolicyFactory.create(...)
BookkeeperService bookkeepr = new BookkeeperService(taxPolicy)
```

Z czasem możemy dojść do sytuacji, gdzie nasze fabryki będą opierać się na konfiguracji, np.: zewnętrzne pliki XML lub properties, baza danych itd.

Problem tej klasy został już rozwiązany – są to kontenery wstrzykiwania zależności (co opisuje kolejny rozdział).

W pewnych przypadkach nie możemy oddać kontroli nad tworzeniem obiektów kontenerowi DI. Przykładowo obiekty zarządzane przez maper relacyjno-obiektowy nie mogą być już zarządzane przez kontener DI. Wówczas możemy zastosować technikę polegającą na tworzeniu tych obiektów przez własnoręcznie napisane fabryki wstrzykujące. Oczywiście fabryki te mogą być zarządzane przez kontener DI.

Technika wstrzykiwania: Kontenery wstrzykiwania zależności



Rysunek 3. Zasada działania wstrzykiwania zależności

Rysunek 3 przedstawia ogólną zasadę działania kontenera wstrzykiwania zależności. Kod kliencki (kod, który chce używać obiektu) pobiera z kontenera żądany obiekt – zwykle po interfejsie. Kontener na podstawie konfiguracji buduje obiekt. Być może aby zbudować obiekt, należy wstrzyknąć do niego jego obiekty zależne (które mogę mieć swoje dalsze zależności).

Kontener dokonuje wstrzyknięć na podstawie konfiguracji:

- adnotacje (Java), atrybuty (C#)
- XML
- kod metod fabrykujących

Niektóre kontenery (takie jak Spring, Seam, CDI) są nie tylko kontenerami wstrzykiwania zależności, ale również kontenerami Inversion of Control – kontenerami o szerszym zastosowaniu. Obiekty przez nie zwracane mają zapewnione wsparcie dla innych technik IoC takich jak zdarzenia i AOP.

Kontenery potrafią zarządzać całym cyklem życia obiektów:

- tworzenie
- wstrzykiwanie zależności
- inicjowanie
- długość życia (skojarzona np. z żądaniem i sesją HTTP)
- niszczeniem obiektów
- ilością obiektów (singletony, pule obiektów)
- wzbogacaniem o porady Aspektowe
- mechanizm Interceptorów

Listing 5 Przykład wstrzyknięcia przez konstruktor obiektu zarządzanego przez kontener (w tym wypadku Spring) do innego obiektu zarządzanego

```
@Component
public class OrderingService{
   private OrderDao orderDao;
   private BookkeeperService bookkeeper;

   @Autowired
   public OrderService(OrderDao orderDao, BookkeeperService
         bookkeeper){
      this. orderDao = orderDao;
      this.bookkeeper = bookkeeper;
   }
   @Transactional
   public void finish(UUID orderNo){
      //…
      Order order = ordaerDao.load(orderNo);
      order.confirm()
      Invoice invoice = bookkeepre.issue(order);
      //..
   }
}
```

Pytanie stare jak kontenery IoC: wstrzykiwać przez adnotacje (Java)/atrybuty (C#) czy przez XML?

Istnieje wiele sporów na temat tego, czy konfigurację obiektów powinniśmy przechowywać na zewnątrz w plikach XML czy w kodzie klas przy pomocy adnotacji/atrybutów danego kontenera.

Z jednej strony pliki XML stają się zmorą, ze względu na ich wielkość oraz przeładowanie mentalne związane z koniecznością pracy z wieloma plikami źródłowymi (kod i xml). Z drugiej zaś strony adnotacje usztywniają kod – zmiana wstrzyknięcia wymaga edycji kodu źródłowego oraz wg purystów „brudzą go".

Jak zwykle każdy nietrywialny problem wymaga głębszego zastanowienia i użycia różnych narzędzi. Z własnego doświadczenia mogę zasugerować następującą praktykę:

- Obiekty „corowe" (w naszym przykładzie OrderingService i BookkeeperService), których implementacja zmienia się relatywnie rzadko, warto konfigurować przez adnotacje. Kod jest samo-dokumentujący się, pliki XML nie zawierają „szumu informacyjnego", nie mamy potrzeby zmiany konfiguracji
- Obiekty „dostrajające" (w naszym przykładzie implementacje TaxPolicy) warto konfigurować przez XML, co pozwala nam wynieść konfigurację na zewnątrz i zmieniać działanie systemu przez edycję konfiguracji (np. podczas wdrożenia systemu u nowego klienta) bez modyfikacji źródeł.

Niektóre kontenery oferują możliwość tworzenia metod fabrykujących, które będą uruchamiane w celu skonstruowania zależności.

Kiedy nie stosować wstrzykiwania zależności

Oczywiście nie wszystkie obiekty powinny być zarządzane przez Kontener DI. Wstrzykiwanie jest pożyteczną techniką tylko wówczas, gdy:

- chcemy zapewnić możliwość wymiany zależności
- nie potrzebujemy wymiany, jednak oddajemy zarządzenie obiektami kontenerowi po to, aby wzbogacić je o Aspekty (np. transakcyjność, autoryzacja)

Czasem zależność pomiędzy obiektami jest naturalna i wynika z modelu, wówczas nie potrzebujemy ich wstrzykiwać.

POZIOM 2: EVENTS

Kolejną wg siły odwracania kontroli techniką są Zdarzenia. Pamiętajmy jednak, że odwrócenie kontroli wiąże się z jej utratą, dlatego decyzja o zastosowaniu silnika zdarzeń w architekturze aplikacji lub systemu powinna być związana ze świadomością jej konsekwencji. Mam nadzieję, że po lekturze tego rozdziału czytelnicy wyrobią sobie intuicję podpowiadającą, kiedy zdarzenia będą grać na ich korzyść, a kiedy mogą być szkodliwe.

Problem: Coupling z materią i czasem

Podejście DI pozwoliło nam odciąć się od konkretnej implementacji klas współpracujących, co w efekcie pozwala tworzyć pluginy typu „zróbmy coś inaczej".

Zdarzenia idą o kilka kroków dalej, pozwalając abstrahować nie tylko od typu współpracownika, ale również od ilości współpracowników, kolejności ich wykonania, wyniku ich działania oraz od strzałki czasu.

Pozwoli nam to m.in. na wprowadzenie pluginów typu „zróbmy coś jeszcze".

Idea: Nie wiem ilu Was jest, kiedy i w jakiej kolejności wykonacie pracę oraz jaki będzie jej wynik

Listing 6. Pseudokod ilustrujący odwrócenie kontroli poprzez Zdarzenia

```
public class MojaKlasa{
   //brak współpracownika
   public void metoda(){
   //brak sterowania kolejnością instrukcji
   eventsPublisher.publish(new NastąpiłoMojeZdarzenie(parametry));
   //brak sprawdzania wyniku (również oczekiwania
   //na wykonanie)
   }
}
```

Listing 6 ilustruje ideę odwracania kontroli poprzez zdarzenia. Klasa MojaKlasa nie ma już zależności do żadnej klasy współpracującej. Zamiast tego posiada zależność do silnika zdarzeń (pole eventsPublisher), do którego wysyła sygnały o wystąpieniu zdarzenia.

Silnik zdarzeń powiadamia wszystkie zarejestrowane Listenery. Z poziomu obiektu emitującego zdarzenie nie mamy wiedzy o ilości Listenerów, kolejności powiadomienia ani czasie ich wykonania.

Obiekt silnika zdarzeń może być singletonem (co utrudnia testowanie automatyczne) bądź może być wstrzyknięty techniką DI.

Warto zwrócić uwagę, że obiekt zdarzenia posiada nazwę oznajmującą. Jest to stwierdzenie faktu, a nie wysłanie rozkazu. Jest to istotne z punktu widzenia semantyki tej techniki odwracania kontroli.

Motywacja: Nie tylko onClickListener dla Buttonów

Niemal każdy programista otarł się o środowiska tworzenia graficznych interfejsów użytkownika, a w nich o zdarzenia generowane przez kontrolki graficzne w reakcji na poczynania użytkowników.

Warto zwrócić uwagę, że nowoczesne środowiska (takie jak Android) odchodzą od mechanizmów Listenerów na rzecz Handlerów (jeden handler per zdarzenie techniczne). Po prostu nienaturalne jest połączenie więcej niż jednego Listenera np. do przycisku, a jeżeli pojawia się taka sytuacji, to często jest symptomem nieodpowiedniej struktury sterowania- np. braku Mediatora.

Jednak zdarzeń możemy używać również do innych celów, np. na poziomie modelu domenowego.

Listing 7. Zgłoszenie zdarzenia w Agregacie Customer

```
@Entity
public class Customer extends BaseAggregateRoot{

   public enum CustomerStatus{
      STANDARD, VIP, PLATINUM
   }

   @Enumerated(EnumType.STRING)
   private CustomerStatus status;

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

Dla przykładu w Domain Driven Design przy pomocy zdarzeń modelujemy istotne z biznesowego punktu widzenia fakty, które zaszły w cyklu życia Agregatu.

Listing 7 ilustruje przykład zgłoszenia zdarzenia przez Agregat Customer w momencie zmiany jego statusu. Klasa **`CustomerStatusChangedEvent`** jest nośnikiem informacji o tym, który klient (ID) zmienił status. Modelując zdarzenia, musimy podjąć decyzję o tym, jakie informacje umieszczamy w klasach zdarzeń – w naszym przykładzie jest to techniczne ID – być może lepszą decyzją byłoby przeniesienie informacji o biznesowym numerze klienta.

Listing 8. Listener zdarzenia

```
@EventListeners
public class CustomerStatusChangedListener{

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

Listing 8 przedstawia kod jednego z Listenerów naszego przykładowego zdarzenia. Listener ten wprowadza następującą funkcjonalność: jeżeli w module CRM klient stanie się VIPem, wówczas w module Sales nadajemy mu 10% rabatu na wszystkie niezatwierdzone zamówienia.

Zastosowanie: Pluginy typu „zróbmy coś jeszcze"

Zdarzenie modeluje fakt, który miał miejsce i nie można go zawetować – możemy co najwyżej zareagować w dodatkowy sposób na zaistniały fakt.

Wpinając kolejne Listenery, rozbudowujemy system o nowe funkcjonalności na zasadzie pluginów. Pluginy możemy załączać np. tylko wówczas, gdy klient wykupi daną funkcjonalność, lub mogą być tworzone przez podmioty trzecie.

Warto zwrócić uwagę na fakt iż intencja tych pluginów jest inna niż w przypadku wstrzykiwania polityk – gdzie plugin działał na zasadzie „zróbmy coś inaczej".

Inne zastosowania

Zdarzenia wprowadzamy do architektury systemu z kilku powodów:

- decoupling modułów systemu – integracja niezależnych od siebie funkcjonalności (przykład z Listingu 7 i 8)
- asynchroniczne wykonanie dodatkowych operacji – przykładowo w module CRM możemy wysyłać maile do klientów, których statusy zmieniono; wysyłka maili to czynność z jednej strony dodatkowa, a z drugiej potencjalnie długotrwała
- rejestrowanie zdarzeń w celu ich przetwarzania i analizy - np. z wykorzystaniem silników CEP (Complex Events Processing) pozwalających na deklarowanie kwerend filtrujących określone wzorce zdarzeń w czasie
- składowanie zdarzeń jako modelu danych alternatywnego dla bazy relacyjnej – Event Sourcing

Technika: Observer Design Pattern

Najprostsza implementacja silnika zdarzeń może być oparta o Observer Design Pattern – Listing 9.

Listing 9. Pseudokod ilustrujący mechanikę działania Observer Design Pattern

```
public class Observable{
   private List<Listener> listeners;

   public void addListener(Listener listener){
      listeners.add(listener)
   }

   public void doSth(){
      //...
      //klonowanie aby powiadomić słuchaczy zarejstrowanych
      //do momentu zajścia zdarzenia, zarejestrowani po fakcie
      //będą dodani do oryginalnej listy (należy dodać sekcję
      //krytyczną aby przygotować kod na działanie wielowątkowe)
      for (Listener listener : listeners.clone())
         listener.handleEvent(new MyEvent(this))
   }
}

public interface Listener{
   public void handleEvent(MyEvent event);
}
```

W podejściu tym nadawca zdarzenia posiada referencje do wszystkich słuchaczy (nie znając jednak ich konkretnych typów). Słuchacze posiadają referencję do źródła zdarzenia – przeniesioną w obiekcie klasy **`MyEvent`**. Mamy zatem powiązanie pomiędzy nadawcą a odbiorcami zdarzenia.

Od strony technicznej rozwiązanie ma ograniczenia zawężające jego stosowalność do jednej wirtualnej maszyny.

Natomiast na poziomie architektury systemowej musimy posłużyć się rozwiązaniem opartym o wzorzec Events Broker.

Technika: Events Broker lub Events Bus

Na poziomie architektury systemowej, gdzie silnika zdarzeń używamy do integracji modułów rozmieszczonych na niezależnych maszynach działających asynchronicznie, musimy posłużyć się rozwiązaniem architektonicznym typu Events Broker lub Events Bus.

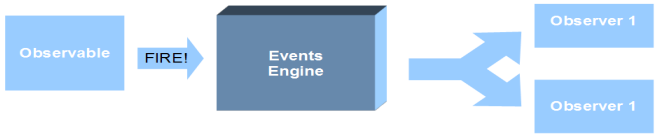
Idea brokera zdarzeń została przedstawiona na Rysunku 4. Nadawca zdarzenia nie zna słuchaczy. Zamiast tego komunikuje się z Brokerem. Słuchacze zdarzeń nie znają nadawcy zdarzenia, rejestrują się u Brokera zorientowani na konkretne zdarzenie, a nie na nadawcę.

Broker zapewnia:

- zdalny dostęp
- transakcyjność – zdarzenia są wycofywane w razie awarii po stronie nadawcy lub powtarzane w razie awarii po stronie odbiorców
- trwałość zdarzeń – zdarzenia po przyjęciu przez brokera są zapisywane w jego bazie danych, dzięki czemu w razie awarii brokera lub słuchaczy zostaną wzowione

Przykładem standardu wspierającego komunikaty transakcyjne jest Java Message Service.

Architektura systemu oparta o Events Broker wiąże się z problemem Single Point of Failure, który polega na tym, że w momencie awarii Brokera cały system przestaje działać. Rozwiązaniem tego problemu jest architektura oparta o Events Bus – jej omówienie wykracza poza ramy tego artykułu.



Rysunek 4. Zasada działania Events Broker

Technika: Saga

Odwracania kontroli poprzez wprowadzenie zdarzeń doprowadza do utraty kontroli nad wynikiem działania słuchaczy. W przypadku gdy potrzebujemy kontroli nad procesem składającym się z wieli zdarzeń, możemy posłużyć się wzorcem Sagi.

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

Natomiast na poziomie koncepcyjnym Saga modeluje de facto czas.

Listing 10. Przykładowa Saga modelująca przepływ zamówienia

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

    @Inject
    private OrderRepository orderRepository;

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

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

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

Listing 10 zawiera przykładowy kod Sagi, która modeluje przepływ zamówienia. Saga reaguje na zdarzenia: zatwierdzenia zamówienia i nastąpienia wysyłki. Zakładamy, że nie możemy przewidzieć kolejności pojawienia się tych zdarzeń, oraz czas pomiędzy ich wystąpieniem może sięgać miesięcy.

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

Kiedy nie stosować zdarzeń

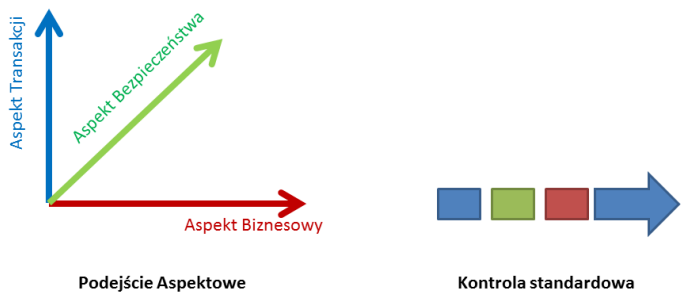
„Nie używamy już zdarzeń, ponieważ nie mamy nad nimi kontroli" - tego typu argumentację można usłyszeć czasem z ust architektów. Zdarzenia są techniką odwracania kontroli, więc prowadzą do utraty kontroli – ot taka niespodzianka.

Stosujemy je zatem wówczas, gdy nie potrzebujemy kontroli nad procesem. Jeżeli proces pozwala na traktowanie modułów jak czarnych skrzynek, które emitują zdarzenia lub są pobudzane zdarzeniami z zewnątrz i nie zależy nam nad kontrolą kolejności ani czasu wykonania, to wówczas dokonaliśmy dobrej decyzji projektowej. Jeżeli natomiast potrzebujemy kontroli nad przepływem procesu na poziomie większym niż daje nam nawet Saga, to zdarzenia nie są odpowiednie w tym kontekście.

POZIOM 3: ASPECT ORIENTED PROGRAMMING

Problem: Ortogonalne Aspekty

AOP jest trzecim i najsilniejszym mechanizmem Odwracania Kontroli. AOP używamy w celu dodania dodatkowych Aspektów do Aspektu głównego. W aplikacjach biznesowych aspektem głównym jest zwykle logika biznesowa, natomiast aspekty dodatkowe to zagadnienia ortogonalne (nie przecinające się z logiką biznesową) takie jak transakcyjność, bezpieczeństwo, audit logi itd.



Rysunek 5. Idea Aspect Oriented Programming

Rysunek 5 ilustruje problem ortogonalnych aspektów – w naszym przypadku aspektów: logiki biznesowej, transakcyjności i bezpieczeństwa. W podejściu klasycznym kod metody realizującej wymaganie biznesowe musi być przeplatany kodem odpowiedzialnym za zarządzanie transakcjami i zapewnienie autoryzacji wykonania operacji. Natomiast koncepcyjnie są to zagadnienie ortogonalne (na Rysunku 5 posłużono się metaforą płaszczyzn ortogonalnych).

Motywacja: Nie chcę znać się na aspektach technicznych

Podejście AOP jest właściwie standardem w przypadku systemów korporacyjnych opartych o platformę Java (zarówno Java EE, jak i np. Spring). Twórcy platformy wychodzą z założenia, że należy odciążyć programistów zajmujących się problemami biznesowymi od zagadnień technicznych takich

jak transakcyjność baz danych czy bezpieczeństwo. Dlatego Serwery i frameworki biorą na siebie obsługę aspektów technicznych, zakładając że programista biznesowy może nawet nie posiadać odpowiednich kompetencji, aby zająć się tymi zagadnieniami.

Idea: Nic nie wiem i nie chcę tego wiedzieć

Listing 11. Pseudokod ilustrujący kontrolę odwróconą przy pomocy AOP.

```
public class MojaKlasa{
   public void metoda(){
   }
}
```

Jak widzimy na Listingu 11, kod aspektu głównego (zwykle biznesowego) nie posiada żadnych zależności do innych aspektów (zwykle technicznych).

Dodatkowe funkcjonalności (takie jak np. transakcje i bezpieczeństwo) są na niego „nakładane" jako porady aspektowe (Advice).

Porady to osobne klasy, które są aplikowane przez silnik AOP na punkty złączenia (Join Point). Punktem złączenia może być:

- przystąpienie do wywołania metody
- zakończenie wykonania metody z powodzeniem
- zakończenie wykonania metody niezależnie od powodzenia
- okalanie wywołania metody (przed i po wywołaniu)
- wyrzucenie wyjątku

Zastosowanie: Aspekty techniczne

Przykładowo Porada Transakcyjna może „doradzać", aby przed wykonaniem metody `MojaKlasa.metoda()` rozpocząć transakcję bazodanową, a po zakończeniu wykonania metody zatwierdzić tę transakcję, natomiast w razie wyjątku odpowiedniego typu wycofać transakcję.

Kontynuując przykład: Porada bezpieczeństwa może „doradzić", aby przed wykonaniem metody `MojaKlasa.metoda()` sprawdzić, czy pracujący w sesji użytkownik posiada pewne uprawnienia, a po wykonaniu metody zapisać audit log.

Porady mogą wchodzić w skład standardowych porad oferowanych przez dany framework lub możemy tworzyć własne.

Zastosowanie: Zmiana działania kodu, do którego źródeł nie mam dostępu

AOP powala wpłynąć na działanie kodu, do którego źródeł nie mamy dostępu. Możemy np. dostać się do parametrów wejściowych metod, do zwracanych wyników i rzucanych wyjątków. Mamy również możliwość zdecydowania o tym, czy dana metoda zostanie w ogóle wywołana.

Może być to użyteczne w przypadku, gdy musimy zmodyfikować działanie kodu, którego nie możemy po prostu przeprogramować.

Wyobraźmy sobie taką oto hipotetyczną sytuację: kod dostarczony przez firmę trzecią jako zamknięta biblioteka zwraca listę pacjentów. Biblioteka nie została przykryta fasadą i jest wykorzystywana w wielu miejscach naszego systemu. Pojawia się wymaganie, aby nie wyświetlać pacjentów o statusie VIP użytkownikom nie posiadającym odpowiednich uprawnień.

Brudnym, ale szybkim rozwiązaniem o globalnym zasięgu, jest stworzenie aspektu przechwytującego wywołania metody zwracającej pacjentów i modyfikację wyniku zwracanego przez nią.

Technika: Kontener wspierający AOP

AOP może być wspierany na poziomie składni języka (np. AspctJ będący rozszerzeniem Javy), jednak najbardziej popularne podejście to integracja silnika AOP z kontenerem DI.

Kontener wstrzykujący zależności, który składa nasze obiekty, może niejako „przy okazji" wzbogacić te obiekty o kod wykonujący porady aspektowe. Jest to przeźroczyste z punktu widzenia programisty, ponieważ odbywa się na poziomie kontenera.

Technicznie AOP może działać na zasadzie:

- modyfikacji skompilowanego kodu (np. podczas ładowania go do pamięci)
- Proxy Design Pattern (klasy proxy są generowane przez kontener)

Inne techniki: Interceptory oraz Wzorce Projektowe Command i Decorator

Konstrukcja Interceptorów oferowana przez niektóre środowiska (np. EJB 3) może być wykorzystana jako namiastka AOP.

Jeżeli architektura aplikacji jest oparta o wzorzec Command, to wówczas zbudowanie własnego silnika AOP staje się trywialne – wystarczy wprowadzić wzorzec Decorator w celu dekorowania poleceń.

Kiedy nie stosować Aspektów

Podejście AOP sprawdza się doskonale, gdy nakładamy techniczne porady aspektowe na biznesowy aspekt główny.

Natomiast jeżeli zaprojektujemy aspekty biznesowe nakładane na inne aspekty biznesowe, wówczas możemy spodziewać się problemów ze zrozumiem logiki, jej utrzymaniem i wzrostem komplikacji. Przykładowo wyobraźmy sobie sytuację, w której na aspekt składania zamówienia nakładamy aspekt wystawienia faktury. W początkowej fazie rozwiązanie może kusić swym urokiem, a w praktyce zwykle nie zdarza się tak, że aspekty biznesowe są ortogonalne – czyli niezależne. Nawet jeżeli jest tak na początku, to z czasem zaczynają pojawiać się powiązania pomiędzy pewnymi przypadkami naszego przykładowego zatwierdzania zamówień z pewnymi specyficznymi warunkami fakturowania.

Generalnie jeżeli potrzebujemy kontroli, a mimo tego używamy rozwiązania, które tak silnie odbiera kontrolę, to możemy spodziewać się kumulacji złożoności przypadkowej (Accidental Complexity).

SMAK CZWARTY: FRAMEWORKI

Konstruowanie frameworków możemy umiejscowić niejako obok omawianych trzech technik odwracania kontroli. Frameworki rozwiązują nieco inny problem i mogą posiłkować się wszystkimi trzema omawianymi do tej pory technikami.

Problem: Powtarzalny Flow

Frameowrk zakłada pewien standardowy przepływ sterowania podczas obsługi danego problemu. Na pewnych etapach sterowanie nie jest standardowe i wymaga „wpięcia" własnej logiki.

Przykładowo frameowrki służące do tworzenia aplikacji web obsługują standardowy przepływ: począwszy od pojawienia się żądania HTTP, przez konwersję i walidację parametrów, po wykonanie logiki i renderowanie odpowiedzi. Wiele etapów tego cyklu można zautomatyzować, zostawiając programistom jedynie specyficzne fragmenty logiki.

Motywacja: Wielka Teoria Unifikacji

Framework powstaje zwykle po to, aby uogólnić i uwspólnić techniczne aspekty problemu. Idea jest szczytna, o ile rozpoznamy wszelkie przypadki szczególne i przewidzimy je w konstrukcji frameworka. W przeciwnym przypadku naiwne uproszczenia będą pożywką do pojawiania się wspomnianej już złożoności przypadkowej, gdzie proste problemy urastają do rangi katastrofy.

Autor

Idea: Nie dzwoń do nas, my zadzwonimy do Ciebie

Ogólna idea działania frameworka może być streszczona w postaci Zasady Hollywood: „Nie dzwoń do Nas, my zadzwonimy do Ciebie". Nasz kod jest traktowany niczym początkujący aktor żądny sławy – za każdym razem słyszy „oddzwonimy do Ciebie".

Framework zawiera szkielet ogólnego przepływu i w odpowiednich momentach wywołuje nasz kod, który obsługuje meritum problemu. Mamy zatem po raz kolejny odwrócenie kontroli. Przykładowo, tworząc aplikację webową w frameworku opartym o wzorce MVC, MVP, PAC, MVVM, piszemy kod kontrolerów. Nie sterujemy ich wywołaniem, kontrola jest przejęta przez framework, który wywoła nasz kod w specyficznym przypadku w odpowiednim momencie.

Framework wywołujący nasz kod może stosować dodatkowo techniki:

- Dependency Injection – wstrzykuje zależności (np. Context) do naszego kodu
- Events – generuje zdarzenia pozwalające na rozszerzenie działania frameworka
- AOP – zapewnia mechanizmy aspektowe

Zastosowanie: podbudowa Ego

Analizując architekturę wielu frameworków oraz ich wpływ na wzrost złożoności systemów, których powstawanie wspierają, można dojść do wniosku, że często główną ich funkcją jest rozbudowa Ego ich twórców:)

- Martin Fowler o Dependency Injection: http://martinfowler.com/articles/injection.html
- Martin Fowler o Zdarzeniach: http://martinfowler.com/eaaDev/EventNarrative.html
- Martin Fowler o Dekoratorach jako implementacji AOP http://martinfowler.com/bliki/DecoratedCommand.html
- Artykuły z serii DDD, do których odnosi się artykuł: http://bottega.com.pl/artykuly-i-prezentacje

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).