Sławomir Sobótka

Domain Driven Design krok po kroku

Część III: Szczegóły implementacji aplikacji wykorzystującej DDD na platformie Java – Spring Framework i Hibernate

Artykuł ma na celu omówienie szczegółów technicznych związanych z odwracaniem kontroli przy pomocy kontenera Spring, projektowaniu architektury otwartej na rozbudowę i rozszerzenia oraz przedstawienie przykładów wykorzystania frameworków wychodzących poza standardowe tutoriale.

Pod dwóch wprowadzających w koncepcję DDD częściach, nadszedł czas na przyjrzenie się szczegółom implementacji projektu. Jako tło techniczne posłuży nam popularny w świecie Javy stos frameworków: Spring i Hibernate.

W części poświęconej Hibernate skupimy się na racjonalnym wykorzystaniu funkcjonalności frameworka oraz aspektach wydajności. Uprzedzając fakty: w kolejnej części serii zastanowimy się, kiedy nie należy korzystać z JPA jako rozwiązania wręcz szkodliwego w pewnych kontekstach.

Prezentowane w artykule podejścia są z założenia ogólne i w dużym stopniu niezależne od konkretnych frameworków. Podejście architektoniczne prezentowane w tekście można z powodzeniem wykorzystać na innych platformach.

Czytelników zainteresowanych implementacją w EJB 3.1 odsyłamy do strony projektu (ramka „W sieci"), gdzie można znaleźć kompletny projekt oparty EJB będący lustrzanym odbiciem projektu opartego o Spring.

Plan serii

Niniejszy tekst jest trzecim z serii artykułów mających na celu szczegółowe przedstawienie kompletnego zestawu technik modelowania oraz nakreślenie kompletnej architektury aplikacji wspierającej DDD.
Część 1: Podstawowe Building Blocks DDD;
Część 2: Zaawansowane modelowanie DDD – techniki strategiczne: konteksty i architektura zdarzeniowa;
Część 3: Szczegóły implementacji aplikacji wykorzystującej DDD na platformie Java – Spring Framework
Część 4: Skalowalne systemy w kontekście DDD – architektura CqRS;
Część 5: Kompleksowe testowanie aplikacji opartej o DDD;
Część 6: Behavior Driven Development – Agile drugiej generacji

PROJEKT REFERENCYJNY

Wszystkich tych Czytelników, którzy już teraz chcieliby zapoznać się z kolejnymi zagadnieniami naszej serii, zapraszam do odwiedzenia strony projektu „DDD&CqRS Laven", której adres znajduje się w ramce „W sieci".

TRZY SMAKI ODWRACANIA KONTROLI

Spring oferuje trzy mechanizmy Inversion of Control: Dependency Injection, Zdarzenia i Aspect Oriented Programming. W kolejnych rozdziałach zostaną przedstawione przykłady wykorzystania każdej z tych technik w celu stworzenia architektury poziomu aplikacji oraz systemu, która będzie otwarta na rozbudowę oraz będzie wspierać testowalność kodu.

W literaturze można spotkać wymienne stosowanie pojęć Inversion of Control oraz Dependency Injection – co jest błędem. DI jest szczególnym przypadkiem IoC.

WŁASNE ADNOTACJE

W kolejnych przykładach będziemy posługiwać się własnymi adnotacjami służącymi do definiowania archetypów komponentów Springa. Spring oferuje ogólną adnotację @Component oraz trzy specyficzne adnotacje archetypowe:

Listing 1: Adnotacja archetypowa przeznaczona dla serwisów warstwy aplikacji.

```
@Service
@Transactional
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ApplicationService {
  String value() default "";
}
```

Listing 1 ilustruje przykładową adnotację, która będzie tagowała serwisy aplikacyjne zapewniając im poprzez użycie adnotacji Springa transakcyjność.

PODSTAWOWE ODWRÓCENIE KONTROLI: DEPENDENCY INJECTION

Wstrzykiwanie zależności weszło niejako do kultury programistów Javy i nie sposób wyobrazić sobie framework, który nie wspiera tej podstawowej techniki Inversion of Control.

DI odwraca kontrolę w najbardziej podstawowym stopniu – zwalania obiekt z tworzenia konkretnych instancji obiektów w nim agregowanych. Dzięki temu, że

obiekty nie decydują o konkretnych typach obiektów zawieranych, charakteryzują się one mniejszym Couplingiem (marę zależności pomiędzy klasami).

WSTRZYKUJEMY NIE TYLKO DAO

Różnego rodzaju tutoriale ilustrują wstrzykiwanie zależności na przykładzie Data Access Objects (DAO) wstrzykiwanych do Serwisów. Odwrócenie kontroli na tym poziomie nie jest krytyczne i może sprawiać wrażenie mało istotnego triku technicznego.

Jeżeli stoimy przed problemem zaprojektowania systemu, który będzie działał w nieco inny sposób w różnych wdrożeniach, możemy podejść do problemu na kilka sposobów:

- stworzyć Branch w Repozytorium kodu – programiści, który dokonywali scalania gałęzi kodu wiedzą dlaczego nie należy podchodzić w ten sposób do problemu
- dodać w kodzie niezliczoną ilość magicznych parametrów, które są rozpatrywane instrukcjami if (lub switch w przypadku Senior Developerów)
- niektóre języki pozwalają na wprowadzanie dyrektyw kompilacji, co pozwala na budowanie systemu per wdrożenie

Możemy również zaprojektować model domenowy, w którym zmienne zachowania są przykryte stabilnym interfejsem a konkretna implementacja jest wstrzykiwana przez kontener IoC.

Uważny czytelnik na pewno skojarzył już omawianą technikę z Building Blockiem DDD o nazwie Policy.

Listing 2 ilustruje technikę wstrzyknięcia obiektu (w tym wypadku jest to Polityka obliczania podatków). Decyzja o konkretnym typie polityki (konkretnej implementacji interfejsu TaxPolicy) została przeniesiona do konfiguracji Springa (kontenera IoC w ogólności).

Dzięki temu na zewnątrz (np. w pliku XML specyficznym dla wdrożenia) możemy zdecydować o tym w jaki sposób nasz system będzie obliczał podatki.

Listing 2. Wstrzyknięcie TaxPolicy do serwisu aplikacyjnego PurchaseApplicationService.

```
@ApplicationService
public class PurchaseApplicationService {
  @Inject
  private BookKeeper bookKeeper;
  @Inject
  private TaxPolicy taxPolicy;

  public void approveOrder(Long orderId) {
    Order order = orderRepository.load(orderId);
    // Domain logic
    order.submit();
    // Domain service
    Invoice invoice = bookKeepr.issuance(order, taxPolicy);

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

Listing 3. Serwis domenowy BookKeeper „domknięty" polityką obliczania podatku.

```
public class BookKeeper {
  public Invoice issuance(Order order, TaxPolicy taxPolicy){
    Invoice invoice = new Invoice(order.getClient());

    for (OrderedProduct product : order.
getOrderedProducts()){
      taxPolicy.calculateTax(product.getType(), net);
      //...
    }
    return invoice;
  }
}
```

STYL FUNKCYJNY

Przyjrzyjmy się klasie modelującej księgowego przedstawionej na Listingu 3.

Model księgowego zakłada, że jest operacja księgowania jest „domknięta" operacją obliczania podatku. Warto zwrócić uwagę na fakt, że BookKeeper nie decyduje o sposobie obliczania podatku, polityka obliczania podatku jest przekazana z warstwy wyższej – warstwy Aplikacji. BookKeeper jako model jest otwarty na domknięcia i pozostawia tą kwestię warstwie wyższej. Dzięki temu jest otwarty na rozbudowę, niezależny od frameworka a co za tym idzie łatwo testowalny. Na potrzeby testów jednostkowych może przekazać mu zaślepkę polityki podatkowej.

Na Listingu 2 warstwa aplikacji (serwis PurchaseApplicationService) przekazuje księgowemu politykę, którą do niej wstrzyknięto. Jednak jest to jedna z wielu możliwości. Aplikacja może równie dobrze pobrać politykę z preferencji zalogowanego użytkownika lub wywnioskować z adresu klienta. Generalnie to aplikacja decyduje o tym w jaki sposób stwierdzić typ konkretnej polityki. Model domenowy (BookKeeper) jest otwarty na różne możliwości.

POLITYKI JAKO PLUGINY – WSTRZYKIWANIE DOSTROJEŃ

W części drugiej naszej serii opisany został podział modelu domenowego na 4 poziomy (od góry):

- Decission Support
- Policy
- Operations
- Capability

W przykładzie z Listingu 1 wykorzystaliśmy Dependency Injection w celu złożenia Operacji (BookKeeper) z jej Polityką (TaxPolicy). Kontener IoC pozwala na separację tego typu konfiguracji do zewnętrznych artefaktów (np. plików XML).

KIEDY XML TO ZBYT MAŁO

Pliki XML są w pewnych sytuacjach ograniczeniem, ponieważ zawierają „zaszyte na sztywno" ustawienia. Jeżeli potrzebujemy możliwości decydowania w czasie działania programu o konkretnym typie obiektu zarządzanego przez kontener IoC możemy posłużyć się metodami fabrykującymi Springa.

Listing 4. Metoda fabrykująca Spring

```
@Configuration
public class TaxPolicyFactory {
  @Bean
  @Scope(„prototype")
  public TaxPolicy create(){
    return ...;
  }
}
```

Kod przedstawiony na Listingu 4 pozwala na zdecydowanie o konkretnym typie za każdym razem gdy potrzeba wstrzyknięcia (zasięg prototype).

Spring wywoła metodę create (nazwa nie jest istotna a jedynie zwracany typ) gdy zajdzie potrzeba wstrzyknięcia polityki podatkowej.

Warto zwrócić uwagę na fakt, iż obiekty @Configuration są same w sobie komponentami Springa. Zatem możemy wstrzykiwać do nich inne obiekty, których praca jest potrzebna do podjęcia decyzji o konkretnym typie polityki podatkowej – np.: aktualnie zalogowany użytkownik (w celu sprawdzenia jego preferencji) lub obiekt pobierający dane z bazy danych (w celu odczytania konfiguracji).

RĘCZNE WSTRZYKIWANIE W FABRYKACH DOMENOWYCH

Wstrzykiwania zależności możemy użyć w obiektach, które są zarządzane przez kontener IoC. Co zrobić w przypadku obiektów nie zarządzanych przez kontener Springa - np. Agregatów,którymi zarządza Hibernate?

Możemy posłużyć się mechanizmem instrumentalizacji ByteCode oferowanym przez Spring – adnotacja @Configurable.

Możemy również wstrzykiwać „ręcznie". Agregaty rozpoczynają swoje życie w pamięci w jednym z dwóch miejsc: Fabrykach Domenowych (nowe Agregaty) lub Repozytoriach (agregaty już istniejące w systemie).

Listing 5 jest przykładem Fabryki Domenowej, która sama w sobie jest komponentem zarządzanym przez kontener IoC – posiada własną adnotację archetypową @DomainFactory.

Fabryka to może być wstrzykiwana do miejsc, które tworzą nowe zamówienia, np. do Serwisów Aplikacyjnych.

Listing 5. Fabryka Domenowa wstrzykująca zależności do Agregatu.

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

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

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

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

    addGratis(order, client);

    return order;
  }
}
```

Fabryka ta sama w sobie posiada wstrzyknięte zależności – fabrykę polityk rabatowych. Wygenerowana polityka rabatowa jest wstrzykiwana ręcznie (metoda setRebatePolicy) do Agrgeatu Order.

SILNIEJSZE ODWRÓCENIE KONTROLI: EVENTS

Zdarzenia są drugim wg siły odwracania kontroli mechanizmem DI oferowanych przez kontener Springa. W przypadku zdarzeń nie dość, że nie znamy konkretnego typu obiektu współpracującego (tak jak w przypadki DI), to również nie znamy:

- ilości obiektów współpracujących
- ich typów
- wyników ich pracy
- kolejności wykonania
- czasu, w którym wykonają pracę

Listing 6. Fragment Agregatu Order, który emituje zdarzenie

```
@Entity
@Table(name = "Orders")
public class Order extends BaseAggregateRoot {
  public void submit() {
    status = OrderStatus.SUBMITTED;
    eventPublisher.publish(new
OrderSubmittedEvent(getEntityId()));
  }
```

Zdarzeniom i ich szerokiemu zastosowaniu poświęciliśmy rozdział w poprzednim artykule, dlatego w tym miejscu ograniczymy się jedynie ilustracji kodu, który emituje zdarzenie do silnika zdarzeń zarządzanego przez Spring.

Warto zwrócić uwagę, że domyślny silnik zdarzeń wbudowany w kontener IoC Springa służy jednie do synchronicznej komunikacji pomiędzy komponentami.

Rozwiązania asynchroniczne powinny być oparte o persystentne kolejki zapewniające trakcyjność i trwałość komunikatów. Konfiguracja tego typu zasobów wykracza poza ramy niniejszego artykułu.

ODWRÓCENIE KONTROLI TOTALNE: ASPECT ORIENTED PROGRAMMING

AOP jest trzecim i najsilniejszym mechanizmem Odwracania Kontroli oferowanym przez Spring. W artykule skupimy się na omówieniu zastosowania AOP zakładając, że czytelnicy są zaznajomieni z zagadnieniem od strony teoretycznej.

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.

W przyjętej przez nas architekturze warstwowej rozdzielamy Logikę Aplikacyjną od leżącej niżej Logiki Domenowej. Aspekty ortogonalne (techniczne) będą znajdować się wokół Serwisów i Stanowych obiektów Aplikacyjnych.

Stąd też przedstawiona na Listingu 1 przykładowa adnotacja archetypowa nakłada transakcję na poziomie Logiki Aplikacji. Natomiast kod Serwisu Aplikacyjnego przedstawiony na Listingu 2 w swych dwóch ostatnich linijkach zapisuje agregaty Order i Invoice w jednej transakcji. Innymi słowy metody Serwisów Aplikacyjnych (modelujące kroki Use Case/User Story) traktujemy zwykle jako operacje atomowe.

PERSYSTENCJA DOMENY

Obiekty domenowe takie tak Agregaty, Encja oraz Value Objects muszą być utrwalane. Mapery Relacyjno-Obiektowe mają swoich zagorzałych zwolenników jak i przeciwników. Jednak jak każde narzędzie mają po prostu swój kontekst stosowalności a poza tym kontekstem stają się nieużyteczne lub wręcz szkodliwe.

W przypadku operacji na obiektach domenowych ORM sprawdza się jako produktywne narzędzie.

Natomiast w przypadku odczytu dużej ilości danych o naturze przekrojowej (różnego rodzaju tabelki w warstwie GUI, raporty) narzędzia ORM nie są odpowiednim rozwiązaniem do tej klasy problemów.

AGREAGTY

Typowy Serwis Aplikacyjny pobiera kilka Agregatów, uruchamia na nich metody biznesowe zmieniając ich stan oraz utrwala zmiany. W tego typu scenariuszach sprawdzają się funkcjonalności mapera takie jak Lazy Loading i Operacje Kaskadowe. Zatem każdy Agregat DDD jest Encją w rozumienia Hibernate (adnotacja @Entity).

Lazy Loading pozwala odłożyć pobieranie danych do momentu gdy będą potrzebne w celu wykonania operacji na wnętrzu Agregatu. Operacje kaskadowe pozwalają na utrwalenie całego grafu wewnętrznych obiektów Agregatu. Obie te funkcjonalności mają sens dla hermetycznych Ageratów o dobrze określonych granicach – zagadnienia te dyskutowaliśmy w poprzednim artykule.

Mapując wnętrze Agregatu przy pomocy JPA warto zwrócić uwagę na mapowanie wewnętrznych kolekcji:

użycie rozszerzonych kolekcji @LazyCollection(LazyCollectionOption.EXTRA), które pozwala na uniknięcie pobierania całych kolekcji gdy nie jest to potrzebne (np. operacje contains, size)

włączenie wszystkich operacji kaskadowych, w tym specyficznych dla Hibenrate DELETE_ORPHAN, które wykrywają odłączenia obiektów z kolekcji

zwrócenie uwagi na typ kolekcji: w Hibernate mamy do czynienia z: Set, Bag, List. Częstym błędem jest użycie Bag zamiast List co powoduje kasowanie kolekcji podczas każdej jej modyfikacji. Bag w Hibernate to List w Javie bez @IndexColulm

Listing 7. Value Object z adnotacją @Embeddable

```
@SuppressWarnings("serial")
@Embeddable
public class Money implements Serializable {
  public static final Currency DEFAULT_CURRENCY = Currency.getInstance("EUR");
  public static final Money ZERO = new Money(BigDecimal.ZERO);

  private BigDecimal value;

  private String currencyCode;

  protected Money() {
  }

  public Money(BigDecimal value, Currency currency) {
    this(value, currency.getCurrencyCode());
  }
  public Money multiplyBy(BigDecimal multiplier) {
    return new Money(value.multiply(multiplier), currencyCode);
  }

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

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

Listing 8. Agregat zawierający VO

```
@Entity
@Table(name = "Orders")
public class Order extends BaseAggregateRoot {
  @Embedded
  private Money totalCost;
```

W sieci

- oficjalna strona DDD
  http://domaindrivendesign.org
- wstępny artykuł poświęcony DDD
  http://bottega.com.pl/pdf/materialy/sdj-ddd.pdf
- przykładowy projekt:
  http://code.google.com/p/ddd-cqrs-sample/
- Sagi
  http://www.udidahan.com/2009/04/20/saga-persistence-and-event-driven-architectures/
- Inversion of Control
  http://martinfowler.com/articles/injection.html
- Generic DAO
  http://blog.xebia.com/2009/03/09/jpa-implementation-patterns-data-access-objects/
- Architektura CqRS
  http://martinfowler.com/bliki/CQRS.html

VALUE OBJECTS

VO to „małe zgrabne” klasy, które podnoszą poziom wyrazu modelu domenowego na bardziej odpowiedni (Listing 7).

Typy zagnieżdżone w JPA pozwalają nam mapować Agregaty zawierające w sobie VO w sposób „płaski” na poziomie bazy danych. Agregat zawierający VO używa adnotacji @Embeded co powoduje, że w na poziomie tabel mapujemy taki Agregat do jednej tabelki bez potrzeby wykonywania operacji JOIN w SQL podczas odczytu Agregatu.

Agregat Order z Listingu 8 będzie mapowany na jedną tabelę Orders, zawierającą kolumny totalCost_value oraz totalCost_currency.

REPOZYTORIA

Repozytoria zawierają wiele wspólnej logiki, dlatego warto oprzeć je o Wzorzec Generic DAO (ramka „W sieci”) oraz operować na bazowej klasie dla Agrgeatów.

Niektóre repozytoria muszą dokonać wspólnych operacji na Agregatach (np. wstrzyknięć), które można zaimplementować jako idiom metody haczykowej będącej trywializacją Wzorce Template Method.

Zainteresowanych szczegółami implementacji repozytoriów odsyłamy do projektu Leaven (ramka „w sieci”).

HIBERNATE DZIAŁA ZBYT WOLNO?

Optymalizacja ORM może polegać na włączeniu Cache Drugiego Poziomu (L2 cache) dla Encji i Agregatów wraz z kolekcjami.

Natomiast odczyty przekrojowych danych to problem, do którego ORM nie jest przystosowany – zalecamy wówczas skorzystanie z natywnego SQL uruchamianego na puli połączeń Read-only.

Kolejnym etapem optymalziacji jest przygotowane dedykowanych modeli do odczytu w architekturze Command-query Responsibility Segregation, której poświęcimy kolejny artykuł z naszej serii.

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).