Autor

Domain Driven Design krok po kroku

Część V: Kompendium testowania aplikacji opartej o DDD – problemy, strategie, taktyki i techniki

W jaki sposób zapewnić jakość systemu? Czy pokrycie 80% kodu testami jest wystarczające? A może wystarczy 20%, ale „zainwestowane" racjonalnie. Czy praca z Javą lub .NET musi być mozolna z uwagi na redeploy? Na wszystkie te pytania odpowiemy w przedostatniej części naszej serii.

Celem artykułu będzie zaproponowanie strategii testowania automatycznego dopasowanej do projektów opartych o DDD. Strategia odpowie na pytanie: „co, kiedy i w jaki sposób testować", aby osiągnąć korzystny współczynnik poniesionego na testy kosztu w stosunku do jakości.

W artykule zostaną przedstawione również konkretne taktyki testowania poszczególnych Building Blocks DDD.

Proponowane taktyki zwiększają również produktywność zespołów pracujących z kompilowanymi językami dzięki temu, że ogranicza w dużej mierze czas marnowany na redeploy testowanego systemu.

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". Dostępne wersje: Java – Spring, Java – EJB 3.1 i .NET – C#.

SYSTEMATYZACJA POJĘĆ

Zanim przejdziemy do omawiania konkretnych technik, chciałbym na wstępie uzgodnić wspólny zakres pojęć. Jest to o tyle ważne, że w nomenklaturze testowania automatycznego mamy kilka podejść do kategoryzacji testów i często dochodzi do sytuacji, gdzie dwie osoby, używając tego samego słowa, mają na myśli inny rodzaj testu.

W niniejszym artykule będziemy posługiwać się pojęciami z jednej z najnowszych książek „Growing Object-Oriented Software, Guided by Tests" Steva Freemana.

Spojrzymy na testy w ujęciu 2-wymiarowym: pod kątem zakresu i roli. Oba wymiary przecinają się, tak więc testy o danym zakresie mogą potencjalnie grać różne role.

Zakres testów

Z punktu widzenia zakresu testów będziemy zajmować się testami jednostkowymi, testami End2End oraz integracyjnymi.

Testów jednostkowych będziemy używać do testowania zachowania poszczególnych Building Blocks DDD. BB będą testowane w izolacji, tak więc będziemy posługiwać się różnego rodzaju zaślepkami (Fake, Mock, Stub). Testy jednostkowe posłużą nam do sprawdzenia, czy logika działa dokładnie tak, jak założono.

Testów End2End będziemy używać w celu testowania poprzez warstwy – począwszy od warstwy logiki aplikacyjnej (lub ew. UI), aż po warstwę infrastruktury. Z uwagi na możliwą ilość kombinacji nie będziemy w stanie sprawdzić wszystkich przypadków, zatem nie będziemy stosować testów o tym zakresie do sprawdzenia perfekcji.

Testy End2End mogą mieś zakres komponentu lub całego systemu. W naszym przypadku skupimy się na zakresie komponentu rozumianego jako Serwis w stylu SOA – czyli nasze Serwisy Aplikacyjne.

Testami integracyjnymi nazywa się czasem testy, które my nazwaliśmy End2End. Natomiast w niniejszym artykule (zgodnie z nową nomenklaturą) testami integracyjnymi będziemy nazywać takie testy, które sprawdzają poprawność integracji z innymi komponentami (np. innymi modułami systemu lub usługami technicznymi). W przypadku projektu opartego o 4-warstwową architekturę DDD testami integracyjnymi pokrylibyśmy wybrane abstrakcje z Warstwy Infrastruktury. Ze względu na specyfikę tych testów charakterystyczną dla danego typu infrastruktury pominiemy ten rodzaj testów.

Rola testów

Większość testów będziemy wykorzystywać, aby upewnić się, czy kod działa dokładnie zgodnie z wymaganiami.

Inne testy będą służyły nam jako ogólny cel biznesowy, wyznaczający zadania i kryteria akceptacji.

Oba rodzaje testów zwykle przechodzą do roli regresyjnych, gdzie sprawdzają, czy kod po zmianach **wciąż** działa tak jak założono.

DRAŻLIWY TEMAT: POKRYCIE KODU TESTAMI

Wiele projektów czy nawet całych organizacji przyjmuje metrykę zakładającą, że 80% kodu powinno być pokryte testami. Liczba 80 jest popularna zapewne z uwagi na to, że często pojawia się jako przykład w literaturze. Oczywiście w projektach rozwiązujących pewne klasy problemów 80% to stanowczo zbyt mało, ale w oprogramowaniu „biznesowym" jest to powszechnie przyjęta poprzeczka.

Z drugiej zaś strony, gdy zapytać doświadczonych programistów o ich intuicyjne odczucia co do procenta kodu, który warto pokryć testami z uwagi na zawiłość lub potrzebę siatki bezpieczeństwa wynikającą z częstych zmian, to zwykle dla systemów „biznesowych" pada odpowiedź: 20%.

Tak się paradoksalnie składa, że 20% + 80% = 100%. Tak więc przy odrobinie „szczęścia" może się okazać, że będziemy mieć pokrycie kodu testami w 80%, ale testy trafią akurat w „mniej istotne" 80% kodu.

Zaczyna być to prawdopodobne, gdy połączymy bezkontekstową metrykę z zespołem, który nie ma jeszcze wyrobionej intuicji lub jest niezaangażowany emocjonalnie w projekt....

A czy nie lepiej byłoby mieć pokrycie 20% krytycznego kodu na poziomie bliskim 100%? Czyli innymi słowy czy nie lepiej zainwestować wysiłek tam, gdzie zwrot będzie najbardziej korzystny?

W DDD wiemy dokładnie, gdzie znajduje się większość krytycznego kodu: w Core Domain w warstwie logiki domenowej. Pojęcie Core Domain zostało opisane szerzej w drugiej części naszej serii, a warstwy w pierwszej.

Problemy, strategie, taktyki i techniki Testowania

Problemy:

- Eksplozja kombinatoryczna przypadków – w górnych warstwa multiplikują się możliwe stany obiektów domenowych
- Koszt stworzenia i utrzymania testów – zaślepki (Fake/Mock/Stub), dane testowe np. w bazie danych

Strategie:

- Mapowanie piramidy testów na warstwy aplikacji
- Perfekcja domeny, ogólna zgodność wymagań
- Obniżanie kosztów poprzez unikanie zbędnej pracy

Taktyki:

- Warstwa Aplikacji – Testy End2end – Komponentowe
- Warstwa Domenowa – Testy jednostkowe
- Abstrakcje Warstwy Infrastruktury – Testy integracyjne
- Unikaj pracy z serwerem

Techniki:

- Testowanie niezmienników Agregatów
- Assembler Agregatów
- Testowanie Serwisów Domenowych w stylu funkcyjnym
- Jak zaślepiać: Command-Mock, Query-Stub

PROBLEMY Z TESTAMI

Problem: Eksplozja kombinatoryczna przypadków

W naszym przykładowym projekcie w warstwie logiki domenowej mamy 3 przykładowe klasy modelujące koncepcje biznesowe: Order, Invoice i InvoicingService.

Załóżmy, że aby przetestować w 100% każdą z nich, musimy stworzyć po 100 testów jednostkowych, czyli 100+100+100=300 testów.

Gdybyśmy natomiast chcieli przetestować działanie przez wyższe warstwy (logikę aplikacyjną lub UI), wówczas możemy spodziewać się nie sumowania, a multiplikacji przypadków, czyli 100*100*100=1000000 przypadków. Oczywiście nie wszystkie kombinacje są możliwe, być może tylko 1% procent. Jeden procent z miliona to sto tysięcy. Natomiast trudne pytanie brzmi: „który jeden procent".

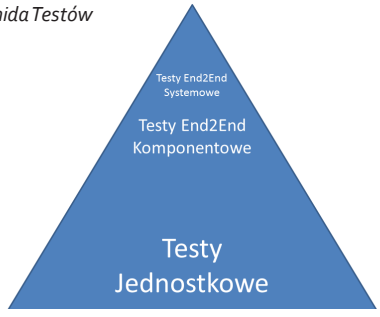
Problem: Koszt stworzenia i utrzymania testów

Z technicznego punktu widzenia największy koszt stworzenia i utrzymania testów to przygotowanie S.U.T. (System Under Test). W przypadku testów jednostkowych są to zaślepki interfejsów, które chcemy odizolować, a w przypadku testów End2End są to dane testowe lub zaślepki innych modułów. Na poziomie kodu testów jest to sekcja GIVEN.

PIRAMIDA TESTÓW – TEORIA A PRAKTYKA

Piramida testów jest ogólnym kanonem, mówiącym o proporcjach testów. Duża podstawa piramidy jest mantrą: „dużo testów jednostkowych", natomiast zwężający się czubek jest mantrą: „mało testów end2end". Piramida testów pozbawiona kontekstu architektury aplikacji i systemu nie mówi nic o tym, kiedy i gdzie stosować testy danego typu.

Rysunek 1. Piramida Testów

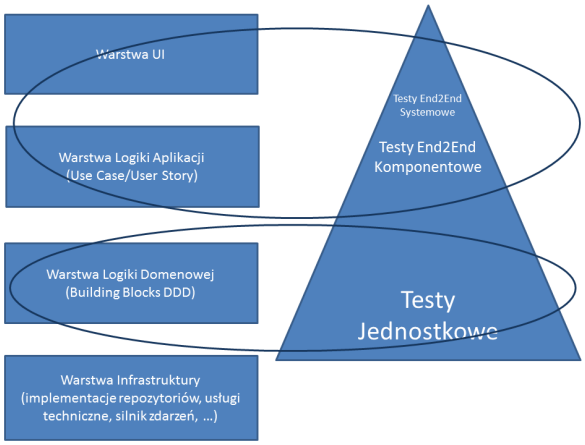


Natomiast w DDD mamy bardzo silne założenia architektoniczne, np. podział logiki na aplikacyjną (model Use Case/User Story) i domenową. Dzięki temu możemy „czytać" piramidę dosłownie...

STRATEGIA: MAPOWANIE PIRAMIDY NA WARSTWY APLIKACJI

Podstawowa strategia testowania, jaką rekomendujemy, polega na dosłownym przełożeniu Piramidy Testów (zarówno zakresu, jak i roli testów) na warstwy logiki:

- górne warstwy – logika aplikacji (ew. UI): testy End2End, akceptacyjne, relatywnie mało
- dolna warstwa – logika domenowa: testy jednostkowe, dążenie do maksymalnego pokrycia



Rysunek 2. Mapowanie Piramidy Testów na warstwy DDD

Strategia ta wynika z dwóch przesłanek:

- złożoności każdej z warstw
- ilości możliwych kombinacji przypadków

Złożonością zajmiemy się w kolejnych rozdziałach, natomiast teraz przedyskutujmy ilość możliwych kombinacji przypadków.

STRATEGIA: PERFEKCJA DOMENY, OGÓLNA ZGODNOŚĆ WYMAGAŃ

Ze względu na mnogość przypadków nie jesteśmy w praktyce w stanie testować perfekcji działania poprzez warstwę logiki aplikacyjnej (API).

Perfekcję będziemy sprawdzać przy pomocy testów jednostkowych działających na modelu w warstwie logiki domenowej.

Natomiast poprzez warstwę logiki aplikacyjnej będziemy uruchamiać testy akceptacyjne – komponentowe oraz (w kolejnej części naszej serii) scenariusze akceptacyjne BDD i wykonywalne specyfikacje. Ich celem nie jest sprawdzenie perfekcji – ze względu na ilość przypadków. Celem tych testów będzie prowadzenie procesu tworzenia oprogramowania, skupianie uwagi programistów na zadaniach i mierzenie postępu pracy (np. Scrum Velocity).

STRATEGIA: OBNIŻANIE KOSZTÓW POPRZEZ UNIKANIE ZBĘDNEJ PRACY

Strategia obniżania kosztów tworzenia testów wspiera dodatkowo dwie powyższe strategie. Kod serwisów aplikacyjnych (Listing 1) ma następujące cechy:

- są to procedury (zwane w językach obiektowych Serwisami), więc są relatywnie mało skomplikowane – jest to po prostu sekwencja czynności do

wykonania. Jeżeli pojawia się tutaj złożoność, to zwykle jest to symptom tego, że brakuje dla niej modelu w niższej warstwie
- posiadają wiele zależności, np. wstrzyknięte repozytoria i ogólnie usługi infrastruktury

Z tych dwóch cech wynika strategia polegająca na unikaniu testów jednostkowych (izolowanych) w tej warstwie, ponieważ:

- ryzyko błędu w procedurze jest relatywnie nieduże (jednak to zależy od natury systemu)
- ilość zaślepek (Mock/Stub/Fake), jakie należałoby stworzyć, generuje wysokie koszty stworzenia i utrzymania testów jednostkowych na tym poziomie

Zatem rachunek kosztów i ryzyka sugeruje nam, aby warstwy aplikacyjnej nie testować jednostkowo.

Listing 1. Serwis Aplikacyjny pl.com.bottega.erp.sales.application.services.PurchaseApplicationService — charakteryzujący się: a) relatywnie niską komplikacją, oraz b) dużą ilością zależności

```
@ApplicationService
public class PurchaseApplicationService {
   @Inject
   private OrderRepository orderRepository;
   @Inject
   private OrderFactory orderFactory;
   @Inject
   private ProductRepository productRepository;
   @Inject
   private InvoiceRepository invoiceRepository;
   @Inject
   private InvoicingService invoicingService;
   @Inject
   private SystemUser systemUser;
   @Inject
   private ApplicationEventPublisher eventPublisher;
   public void approveOrder(Long orderId) {
      Order order = orderRepository.load(orderId);

      Specification<Order> orderSpecification =
         generateSpecification(systemUser);
      if (!orderSpecification.isSatisfiedBy(order))
         throw new OrderOperationException(„Order does not
            meet specification“, order.getEntityId());

      order.submit();

      Invoice invoice = invoicingService.issuance(order,
         generateTaxPolicy(systemUser));

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

TAKTYKA: WARSTWA APLIKACJI – TESTY END2END – KOMPONENTOWE/SYSTEMOWE

Serwisy Warstwy aplikacji testujemy testami o zakresie End2End komponentowym lub systemowym. Z uwagi na ilości zależności nie inwestujemy czasu w tworzenie zaślepek, zatem nie będą to testy jednostkowe. Tworzenie zaślepek np. Repozytoriów jest żmudne i kosztowne, a z drugiej strony ryzyko wystąpienia błędu w tej warstwie jest relatywnie małe.

Z uwagi na ilość możliwych przypadków nie testujemy perfekcji działania, a jedynie ogólną zgodność z wymaganiami (np. Use Case lub User Story).

Testy Systemowe

Przedstawiony na Listingu 2 test o zakresie systemowym łączy się ze zdalną usługą serwera (w przypadku Spring można wykorzystać SpringRemoting, w przypadku EJB: Remote Interface).

Listing 2. Przykładowy test Serwisu Aplikacyjnego pl.com.bottega.erp.sales.ProductsOrderingFunctionalTest oparty o Spring Test i Spring Remoting

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(„classpath:/functionalTestsContext.xml“)
@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
public class ProductsOrderingFunctionalTest {

   @Inject
   @Rule
   public ExpectedEvents events;

   @Inject
   private ProductFinder productFinder;

   @Inject
   private Gate gate;

   @Inject
   private OrderFinder orderFinder;

   @Test
   public void
      shouldAddAlreadyAddedProdcyByIncreasingQuantity(){
   //given
   Long existingProductId = anyProduct().getProductId();
   //when
   Long orderId = createOrderWithProduct(existingProductId);
   addProduct(existingProductId, orderId);
   //then
    assertEquals(1, orderFinder.getClientOrderDetails(orderId).
      getOrderedProducts().size());
   }
```

Testy Komponetowe

Innym podejściem może być testowanie o zakresie komponentowym, gdzie test uruchamia kontekst Spring, z którego pobiera testowany komponent i operuje na nim w swej pamięci.

W tego typu testach izolujemy komponent - np. Serwis Aplikacyjny lub cały moduł w rozumieniu Bounded Context. Izolacja może polegać na zaślepieniu np. silnika zdarzeń i sprawdzaniu, czy w razie wykonania pewnych operacji na API, na interfejsie zdarzeń pojawiają się oczekiwane komunikaty.

TAKTYKA: WARSTWA DOMENOWA – TESTY JEDNOSTKOWE

W warstwie domenowej przyjmiemy zupełnie inną taktykę niż w warstwie aplikacyjnej. Klasy domenowe charakteryzują się:

- dużą złożonością, z której wynika ryzyko błędu
- zawierają newralgiczny kod, z czego wynika wysoka dotkliwość błędu
- relatywnie często podlegają zmianom, z czego wynika potrzeba siatki bezpieczeństwa
- niewielką ilością zależności technicznych, z czego wynika niski koszt stworzenia zaślepek

Klasy z tej warstwy będziemy zatem pokrywać testami o zakresie jednostkowym w celu sprawdzenia perfekcji działania kodu.

Technika: Testowanie Value Objects

Jeżeli pokryjemy testami w 100% VO, to wówczas nie musimy ich zaślepiać w testach innych Building Blocks, takich jak Agregaty i Serwisy Domenowe. Zapewne każdy z nas widział w swej karierze testy jednostkowe, które dokonywały zaślepiania struktur danych. Nie jest to ani produktywne, ani nie wnosi wartości.

Listing 3. Przykładowy test jednostkowy Value Object pl.com.bottega.ddd.domain.sharedkernel.MoneyTest

```
public class MoneyTest {

   private static final Currency USD = Currency.getInstance(„USD“);
   private static final Currency EUR = Currency.getInstance(„EUR“);
```

```
@Test
public void shouldIncreasenegaiveAmountByDiving(){
//given
Money money = new Money(-10);
//when
Money result = money.increaseBy(2);
//then
assertEquals(new Money(-5), result);
}
```

Technika: Testowanie niezmienników Agregatów

W drugiej części naszej serii opisaliśmy jedną z technik modelowania granicy Agregatów, polegającą na hermetyzacji niezmienników. Przykładowo jeżeli pomiędzy obiektami a,b, c zachodzi niezmiennik a+c=c, to wówczas modelujemy Agregat, który zawiera a,b,c oraz posiada metody, które operują na a,b,c, zapewniając niezmiennik.

Testy jednostkowe Agregatów powinny sprawdzać wszystkie metody pod kątem zapewniania niezmienników. Pozwala to na zwiększenie czytelności testów (ich intencja jest jasna) oraz na zmniejszenie kosztu utrzymania testów – skupiamy testowane hipotezy wokół niezmienników, które są sensem istnienia Agregatów.

Listing 4. Przykładowy test jednostkowy Agregatu pl.com.bottega.erp.sales.domain.OrderTest

```
public class OrderTest {
   private Order order;
   private DomainEventPublisher eventPublisherMock;
   private RebatePolicy rebatePolicyMock;

   @Before
   public void beforeEachMethod() throws Exception {
      rebatePolicyMock = mock(RebatePolicy.class);
      eventPublisherMock = mock(DomainEventPublisher.class);
      applyRebate(Money.ZERO);
      order = new Order(new Client(), Money.ZERO, OrderStatus.DRAFT);
      order.setRebatePolicy(rebatePolicyMock);
      order.setEventPubslisher(eventPublisherMock);
   }

   @Test
   public void shouldBeDraftWhenCreated() throws Exception {
      assertEquals(OrderStatus.DRAFT, order.getStatus());
   }
   @Test
   public void shouldSubmitOrderWithProduct() throws Exception
{
      // when
      order.addProduct(anyProduct(), 1);
      order.submit();
      // then
      expectEvent(OrderSubmittedEvent.class);
   }

   @Test
   public void shouldCountProductsNumberCorrectly() throws Exception {
      // given
      Product product = anyProduct();
      // when
      order.addProduct(product, 1);
      order.addProduct(product, 1);
      // then
      OrderedProduct ordered = getOnlyElement(order.
         getOrderedProducts());
      assertEquals(product.getName(), ordered.getName());
      assertEquals(2, ordered.getQuantity());
   }
```

Technika: W testach Agregatów zaślepiamy Polityki i silnik zdarzeń

Test przedstawiony na Listingu 4 tworzy zaślepki (w tym wypadku zaślepki typu Stub) dla silnika zdarzeń i polityki rabatowej.

Pozwala to odizolować na czas testów Agregat od infrastruktury oraz od kodu biznesowego obliczania rabatów. Jest to szczególnie istotne z powodu:

- precyzyjnej lokalizacji błędów,
- zmniejszenia ilości możliwych kombinacji,
- uniknięcia długotrwałych operacji,
- izolacji od kodu, który może podlegać zmianom – zmiana algorytmu rabatowania wpłynęłaby na asercje Agregatu Order.

Technika: Assembler Agregatów

Wprowadzenie Agregatów w odpowiedni stan przed testem może być złożonym procesem. Zakładając, że chcemy osiągnąć pokrycie Agregatów testami bliskie 100% oraz że chcemy testować zgodnie z dobrą praktyką „jeden test – jedna hipoteza biznesowa (niezmiennik metody)", ilość testów będzie bardzo duża.

Aby zredukować wysiłek i uodpornić kod testów na zmiany, możemy wprowadzić do kodu testów wzorzec Assemblera (trywializacja wzorce Builder) – przykład na Listingu 5.

Listing 5. Przykładowy Assembler Agregatu wraz z techniką użycia, która nadaje testom styl DSL, zwiększając ich czytelność.

```
@Test
public void correctOrderWithOneProductCanBeSubmitted() throws
Exception {
   givenOrder().with(ProductAssembler.product(„Gizmo").withPrice(2.33));
   whenOrder().submit();
   thenOrder().isSubmitted().hasProductCosting(„Gizmo", 2.33);
}

private OrderAssembler givenOrder() { return orderAssembler; }

private Order whenOrder() {
   order = orderAssembler.build();
   return order;
}
private OrderAssert thenOrder() { return new
OrderAssert(order); }
```

Technika: Testowanie Serwisów Domenowych w stylu funkcyjnym

Dobrze zaprojektowane Serwisy Domenowe powinny być utrzymane w stylu funkcyjnym, tj: posiadają wejście i wyjścia oraz nie komunikują się z infrastrukturą. Na Listingu 1 widzimy przykład użycia InvoicingService, gdzie na wejściu podajemy zamówienie oraz politykę rabatową, a na wyjściu otrzymujemy fakturę.

InvoicingService jest funkcją, która modeluje algorytm wystawiania faktury na zamówienie i jest domknięta funkcją obliczania podatku. Dzięki takiemu stylowi (brak efektów ubocznych oraz brak komunikacji z infrastrukturą) nasze „funkcje" możemy relatywnie łatwo i tanio (bez tworzenia zaślepek i przygotowywania danych) testować jednostkowo pod kątem poprawności algorytmu.

Technika: Testowanie Fabryk

Fabryki grają istotną rolę w modelu DDD – są modelem reguł rządzących tworzeniem nowego Agregatu (wymagane obiekty i ich walidacja, wstrzykiwanie, ustawienie parametrów początkowych). Od strony technicznej Fabryki biorą na siebie większość Couplingu (miara powiązania), dzięki czemu tworzone przez nie Agregaty charakteryzują się mniejszym Couplingiem. Znaczenie Fabryk omawialiśmy w pierwszej części serii, dlatego w tym miejscu nie będziemy powtarzać informacji na ich temat.

Natomiast z perspektywy testowania trzeba jasno powiedzieć, że testowanie jednostkowe Fabryk jest kosztowne – z uwagi na Coupling (często techniczny, np. zależność od Repozytoriów), czyli konieczność tworzenia wielu zaślepek.

Warto zatem przyjąć regułę: po prostu nie testujemy jednostkowo Fabryk. Jeżeli Fabryki zawierają złożoną logikę, którą należałoby poddać testom, to wówczas wydzielamy ją do Serwisów Domenowych projektowanych w stylu funkcyjnym i testujemy zgodnie z taktyką z poprzedniego rozdziału.

Technika: Testowanie Polityk i Specyfikacji

Podczas testowania Agregatów i Serwisów Domenowych izolujemy je od domknięć, czyli Polityk i Specyfikacji. Dzięki temu precyzyjnie lokalizujemy błędy oraz odcinamy się od zmian w Politykach i Specyfikacjach. Izolacji dokonujemy poprzez wprowadzeniu zaślepek typu Mock albo Stub. O tym, jaki rodzaj zaślepki dobrać, traktuje następny rozdział.

Natomiast konkretne implementacje Polityk i Specyfikacji testujemy osobnymi, odpowiednimi dla nich testami jednostkowymi, osiągając tanim kosztem wysokie pokrycie kodu. Przykładowo Serwis Domenowy Invoicng Service użyty w Listingu 1 pokrywamy testami algorytmu generowania faktury dla różnych zamówień, natomiast polityki obliczania podatku (zgodne z prawem polskim i niemieckim) testujemy testami algorytmów podatkowych.

Technika: Jak zaślepiać: Command-Mock, Query-Stub

W testowaniu automatycznym możemy stosować 4 rodzaje zaślepek:

- Dummy – zaślepki nieistotnych parametrów,
- Fake – własne implementacje (np. bazy danych),
- Stub – obiekty, które „uczymy", jak powinny się zachować podczas interakcji z nimi,
- Mock – Stub, które dodatkowo weryfikujemy, czy odbyto z nimi założoną interakcję.

Najbardziej rozsądna wydaje się być prosta „reguła kciuka": metody typu Command zaślepiamy przy pomocy Mock objectów, natomiast metody typu Query zaślepiamy przy pomocy Stub objectów.

Podział metod na Command i Query jest zdefiniowany przez paradygmat Command-query Separation (nie mylić z architekturą CqRS). Metody typu Command modyfikują stan obiekty, natomiast metody typu Query odpytują obiekt o jego stan, nie modyfikując go.

Zatem przykładowo: jeżeli nasz testowany Agregat lub Serwis Domenowy woła metodę innej klasy, to:

- jeżeli jest to wywołanie typu Query („podaj dane"), to wówczas wystarczy nam zaślepka typu Stub – uczymy zaślepkę jedynie zachowania, natomiast nie weryfikujemy, ile razu i z jakimi parametrami była ona wykonana,
- jeżeli to jest wywołanie typu Command („wykonaj operację/zmień się"), to wówczas użyjemy zaślepki typu Mock, ponieważ raczej będzie nam zależało na sprawdzeniu, ile razy polecanie zostało wydane i z jakimi parametrami.

TAKTYKA: WARSTWA INFRASTRUKTURY – TESTY INTEGRACYJNE

Jako uzupełnienie zakresu testowania nakreślimy jedynie podejścia do testów integracyjnych – takich, które sprawdzają, czy moduły (w sensie Bounded Context, jak i usług technicznych) całego systemu współpracują ze sobą.

W naszej architekturze będą to testy abstrakcji z warstwy infrastruktury.

Testy tego typu nie sprawdzają logiki biznesowej, a jedynie integrację technologii. W prostych systemach mogą być zbędne.

Najważniejsza zasada : w testach tego typu nie posługujemy się kodem biznesowym, ponieważ może się on zmienić, rujnując testy technologii. Na potrzeby tych testów najlepszym podejściem będzie stworzenie osobnej domeny, która nie będzie podlegać zmianom.

TAKTYKA: UNIKAJ PRACY Z SERWEREM

Czy zastanawiałeś się, ile czasu zajmuje Ci proces: od momentu naciśnięcia ctrl+s w edytorze kodu do momentu, gdy wiesz, czy w tym kodzie nie popełniłeś błędu?

Być może w tym procesie są takie czynności jak: redeply aplikacji (hot deploy znowu nie zadziałał), restart serwera (wyciek pamięci), wprowadzenie danych roboczych do bazy, przejście przez kilkanaście ekranów aplikacji, wpisując na każdym z nim wartości do kilku pól, kliknięcie finish na ostatnim formularzu.

Mija kilka/kilkanaście minut, a być może chodziło jedynie o zmianę jednego operatora w metodzie jednego Agregatu.

Mapując piramidę testów na warstwy aplikacji, doszliśmy do taktyki polegającej na pracy z kodem domenowym przy użyciu testów jednostkowych. W projektach DDD to w kodzie domenowym znajduje się największa złożoność systemu, zatem możemy znaczną część codziennej pracy z kodem wykonać jedynie poprzez testy jednostkowe bez potrzeby wykonywania żmudnych czynności.

PROCES AGILE: SCENARIUSZE BEHAWIORALNE I WYKONYWALNE SPECYFIKACJE

Na wyższym poziomie tworzymy:

- wykonywalne scenariusze akceptacyjne sprawdzające zgodność zachowania systemu z User Story na poziomie flow użytkownika,
- wykonywalne Specification by Example sprawdzające realizację celów biznesowych stojących ponad flow użytkownika.

W przypadku tych technik nie mówimy już o testach, a o wykonywalnej dokumentacji wymagań.

Kolejną część naszej serii poświęcimy omówieniu:

- kompletnego procesu modelowania DDD „Modeling Whirlpool",
- procesu wytwórczego Behavior Driven Development (zwanego Agile drugiej generacji)
- narzędzi wspierających BDD i Specification by Example: Jbehave i Selenium

- 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/
- Mock i Stub: http://martinfowler.com/articles/mocksArentStubs.html
- Mockito: http://code.google.com/p/mockito/

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).