Sławomir Sobótka

Receptury projektowe – niezbędnik początkującego architekta

Część II: Mock czy Stub? Command-query Separation prawdę ci powie.

Testując jednostkowo sieć powiązanych obiektów, dążymy do ich testowania w separacji. Separację osiągamy dzięki stosowaniu różnego rodzaju „dublerów". Często bez zastanowienia stosujemy dublery typu Mock. Mocki są relatywnie pracochłonną techniką, która nie zawsze jest uzasadniona. W niniejszym artykule zostanie przedstawiona pragmatyczna „reguła kciuka" oparta o paradygmat Command-query Separation, która daje prostą odpowiedź co do typu dublera, jakiego potrzebujemy w teście jednostkowym.

AUTOMATYCZNE TESTOWANIE AUTOMATYCZNEJ ZMIANY BIEGÓW

Artykuł rozpoczniemy od krótkiego wyjaśnienia gry słownej z powyższego tytułu. Od pierwszego numeru magazynu *Programista* publikowałem artykuły osadzone w przykładach z domeny biznesowej. Tym razem, dla odmiany, rzecz będzie o automatycznym testowaniu sterownika do automatycznej skrzyni biegów. Czyli domena, której znajomość może przydać się każdemu programiście z dwóch powodów: a) sprzyjająca programistom koniunktura na rynku pracy, oraz b) w niektórych markach super-samochodów montuje się już tylko takie przekładnie.

Bardzo szybki kurs mechaniki

Generalnie automatyczne skrzynie biegów składają się z dwóch modułów: sterującego, który podejmuje decyzje o zachowaniu się skrzyni, oraz wykonawczego, który zajmuje się mechaniką i docelowo przekazaniem momentu obrotowego.

Model obiektowy

Klasy przedstawione na Rysunku 1 próbują oddać taki właśnie model – w tym miejscu pragnę przeprosić znawców motoryzacji za brutalne uproszczenie działania rzeczywistej maszyny, ale dokonano go na potrzeby artykułu o testowaniu automatycznym. Ewentualnie możemy przyjąć założenie, że nasz model będzie wykorzystany w prostej grze typu arcade.

Omówmy zatem krótko przykładowy kod źródłowy, który będzie poddany rozważaniom na temat dublerów w testach jednostkowych.

Listing 1 zawiera testowaną klasę **`AutomaticTransmissionController`** będącą na pewnym poziomie rozwoju kodu źródłowego. Klasa odpowiada za sterowanie automatyczną skrzynią biegów i modeluje odpowiedzialności:

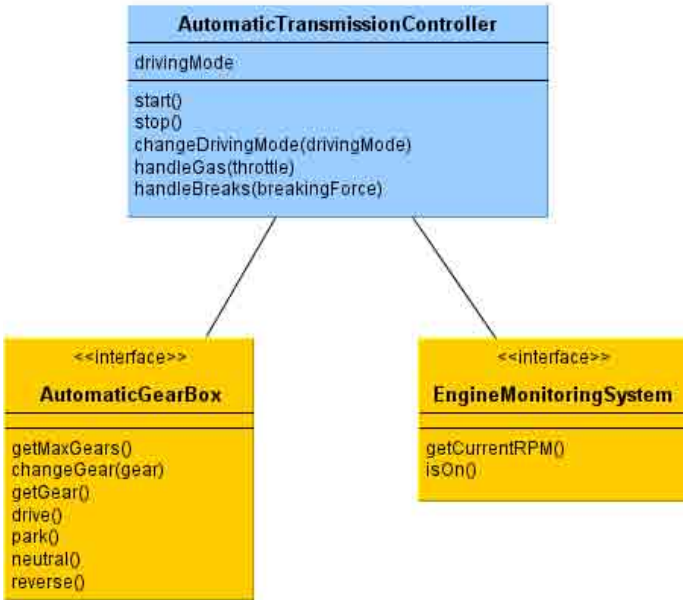
- włączenia i wyłączenia całego podsystemu przeniesienia napędu
- zmiany trybu jazdy (ekonomiczny, komfortowy, sportowy) - od wybranego przez kierowcę trybu będzie zależała charakterystyka pracy skrzyni – w praktyce tryby jazdy wyznaczają obroty silnika, przy których skrzynia dokonuje zmiany biegów. Przykładowo, w trybie sportowym będziemy się spodziewać, że sterownik będzie szybko dążyć do uzyskania maksymalnego momentu obrotowego
- obsługę wciśnięcia pedału gazu (pewien poziom wciśnięcia) – od sterownika będziemy się spodziewać, że „domyśli" się, iż kierowca „wyraża intencję" zapotrzebowania na moment obrotowy i sterownik może podjąć decyzję o redukcji biegu

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.

Rysunek 1. Diagram klas omawianych w przykładach



- obsługę wciśnięcia hamulca (pewien poziom wciśnięcia)
- pomijamy wiele innych „bodźców", na jakie reaguje sterownik (obroty silnika, prędkość kątowa pojazdu, predykcje trajektorii na podstawie nawigacji GPS itd.)
- pomijamy mechanizm uczenia się przez sterownik stylu jazdy kierowcy

Nasz sterownik w obecnej wersji współpracuje z dwoma podsystemami:

- system monitorowania stanu silnika (interfejs **EngineMonitoringSystem** z Listingu 2), który udziela informacji o aktualnej ilości obrotów silnika na minutę
- moduł wykonawczy skrzyni (interfejs **AutomaticGearBox** z Listingu 3), który odpowiada za podstawowe operacje na fizycznej maszynie, takie jak załączanie konkretnego biegu
- pomijamy szereg innych sensorów, jakie mają wpływ na działanie sterownika

Generalnie mamy tutaj do czynienia z klasycznym podziałem odpowiedzialności: **AutomaticGearBox** „wie", jak wykonać operację, natomiast **AutomaticTransmissionController** „wie", kiedy i dlaczego ją wykonać.

Listing 1. Klasa AutomaticTransmissionController, będąca na wczesnym etapie rozwoju, którą będziemy poddawać testom automatycznym

```
/**
 * Controlling module of the gear box
 */
public class AutomaticTransmissionController {

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

   private AutomaticGearBox gearBox;

   private EngineMonitoringSystem engineMonitoringSystem;

   private DrivingMode drivingMode;

   public AutomaticTransmissionController(
            DrivingMode initialDrivingMode,
            AutomaticGearBox gearBox,
            EngineMonitoringSystem engineMonitoringSystem){
      this.drivingMode = initialDrivingMode;
      this.gearBox = gearBox;
      this.engineMonitoringSystem = engineMonitoringSystem;
   }

   public void start(){
      //TODO check engine and start gearbox
   }

   public void stop(){
      //TODO park gearbox
   }

   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){
      //TODO check if we are actually started (Driving Mode)

      int gear = calculateGear(throttle, breakingForce);
      if(gear != gearBox.getGear()){
         this.gearBox.changeGear(gear);
      }
   }

   /**
    * determines 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>
    * @return optimal gear
    */
   private int calculateGear(double throttle,
            double breakingForce) {
      int currentRPM = engineMonitoringSystem.getCurrentRPM();
      // TODO maybe use Strategy Design Pattern/Template Method
      // Pattern to handle different driving modes
      return 0;
   }
}
```

Listing 2. Interfejs podsystemu monitorowania silnika

```
/**
 * Engine's monitoring module
 */
public interface EngineMonitoringSystem {
   /**
    * @return engine's revolutions per minute
    */
   public int getCurrentRPM();

   public boolean isOn();

}
```

Listing 3. Interfejs modułu wykonawczego – sterowanie maszyną

```
/**
 * Executive module of the gear box
 */
public interface AutomaticGearBox {

   /**
    * @return maximum forward gears supported by this model
    */
   public int getMaxGears();

   /**
    * changes forward gear
    * @param gear
    */
   public void changeGear(int gear);

   /**
    * @return current gear
    */
   public int getGear();

   /**
    * changes gear box mode to D (drive)
    */
   public void drive();

   /**
    * changes gear box mode to P (park)
    */
   public void park();

   /**
    * changes gear box mode to N (neutral)
    */
   public void neutral();

   /**
    * changes gear box mode to R (reverse)
    */
   public void reverse();
}
```

Ale...

Biegli programiści obiektowi (nie mylić z programistami stosującymi słowo kluczowe **class**) mają zapewne kilka uwag, szczególnie do Listingu 1.

Nie jest to ostateczna wersja modelu i z czasem dokonamy jego przebudowy. Jednak w obecnej postaci ilustruje on kilka typowych błędów i problemów, jakie spotykamy na co dzień.

GDZIE JEST DUBLER?

Moduł wykonawczy – implementacje interfejsu **`AutomaticGearBox`** posiadają jedynie bardzo podstawowy „instynkt samozachowawczy", który czyni je bezbronnymi przed błędami modułu sterującego – klasy **`AutomaticTransmissionController`**. Zakładamy, że błędy kontrolera mogą doprowadzić dosłownie do zmielenia zębatek wykonawczych i dlatego chcemy testować kontroler w izolacji od modułu wykonawczego. Użyjemy do tego celu testów jednostkowych, które będą grały rolę weryfikatora perfekcji działania kontrolera.

W systemach klasy Enterprise mamy do czynienia z problemami tego samego typu – przykładowo: serwisy, które oprócz algorytmów biznesowych dokonują wysłania maili, komunikatów na szyny zdarzeń, interakcji z innymi systemami itd.

MOTYWACJA

Zakładamy ogólną znajomość zagadnienia testowania automatycznego, rodzajów testów, technik i narzędzi. Czytelników zainteresowanych problematyką testowania (problemy, strategie, taktyki, techniki, narzędzia) odsyłam do dwóch ostatnich artykułów z serii „Domain Driven Design krok po kroku" – dostępne za darmo, adres w ramce „W sieci".

Poniżej szybkie przypomnienie motywów stojących za wprowadzeniem dublerów.

Szybkość

Testy, które testują izolowany fragment kodu, wykonują się szybciej, ponieważ znika opóźnienie wynikające np. z dostępu do baz danych, web serwisów itd.

Perfekcja

Im mniejszy fragment kodu, tym mniej możliwych kombinacji stanów początkowych i końcowych. Możemy zatem relatywnie tanim kosztem osiągnąć wysokie pokrycie kodu testami. Oczywiście, jeżeli poprawnie działają małe fragmenty, to nie znaczy, że większa całość również działa poprawnie.

Precyzja

Testy takie precyzyjnie wskazują lokalizację błędu, unikamy stwierdzeń typu „błąd znajduje się gdzieś w klasie X lub jej przyległościach"

Skupienie uwagi

Im mniejszy fragment kodu, jaki musimy „ogarniać" umysłem, tym szybszy czas tak zwanego „załadowania kontekstu" - czyli odbudowania stanu pamięci (prospektywnej, ważności, skojarzeniowej, epizodycznej i konceptualnej – zobacz ramka „W sieci") potrzebnej do wykonania pracy umysłowej.

Brak dostępu do stanu – „pure" OO lub serwisy

Z punktu widzenia kodu, najbardziej będzie interesował nas problem braku dostępu do stanu obiektów współpracujących.

Listing 4. Interakcja z bezstanowym obiektem

```
public class OrderService {
   public void confirm(Order order){
      //...
      mailSender.send(new OrderConfirmedMessage(order))
   }
}
```

Listing 4 ilustruje sytuację, w której testowana klasa **`OrderService`** współpracuje z obiektem **`mailSender`**, który jest serwisem, czyli obiektem bezstanowym. Testując działanie metody **`confirm`**, nie możemy zatem sprawdzić stanu obiektu **`mailSender`**. Podobnie ma się rzecz z dobrze zaprojektowanym (hermetycznym) modelem obiektowym. Nie mamy wówczas dostępu do całego stanu obiektu przez gettery. Jedyny sposób to sprawdzenie interakcji **`OrderService.confirm()`** z **`mailSender`**. Możemy np. zweryfikować, czy aby na pewno wysłano jedno i tylko jedno powiadomienie do tego o konkretnej treści.

W tym celu będziemy potrzebować dublera.

TYPY DUBLERÓW

Listing 5 prezentuje jeden z wielu testów kontrolera. Test ten należy do grupy testów trybu sportowego (z uwagi na mnogość testów w każdym trybie zostały one rozdzielone).

Test sprawdza następujący scenariusz: poruszamy się na niskich obrotach silnika (1500 RMP) na piątym biegu. Czy aby na pewno podczas gwałtownego wciśnięcia pedału gazu (0.7 głębokości wciśnięcia) dojdzie do decyzji sterownika o redukcji biegu z 5 na 4?

Kod źródłowy testu wykorzystuje bibliotekę Mockito (ramka „W sieci"). Zainteresowanych odsyłam do doskonałej dokumentacji autorstwa twórcy biblioteki – Szczepana Fabera. Na tym etapie wystarczy wiedzieć, że metody: **`mock, when i verify`** są statycznymi metodami z klasy **`org.mockito.Mockito`**.

Kod źródłowy testu zawiera szereg „zapachów" (code smell), które rokują pewne problemy z jego utrzymaniem. Jednak jego refaktoryzacją zajmiemy się w jednym z kolejnych artykułów, który będzie poświęcony wzorcom i najlepszym praktykom tworzenia utrzymywanych testów jednostkowych.

Listing 5. Jeden z wielu testów kontrolera

```
import org.junit.Test;
import static org.mockito.Mockito.*;

import pl.com.bottega.automatictransmission.
AutomaticTransmissionController.DrivingMode;

public class AutomaticTransmissionControllerSportModeTest {

   private DrivingMode MODE = DrivingMode.SPORT;

   private double RAPID_ACCELERATION = 0.7;

   private int LOW_RPM = 1500;

   @Test
   public void shouldReduceGear_
         whenAcceleratingRapidlyOnLowRPM(){
      //given
      int currentGear = 5;

      //----Stub----
      EngineMonitoringSystem engineMonitoringSystem =
         mock(EngineMonitoringSystem.class);
      when(engineMonitoringSystem.getCurrentRPM())
      .thenReturn(LOW_RPM);

      //----Mock----
      AutomaticGearBox gearBox = mock(AutomaticGearBox.class);
      when(gearBox.getGear()).thenReturn(currentGear);

      AutomaticTransmissionController controller =
         new AutomaticTransmissionController(MODE,
            gearBox, engineMonitoringSystem);

      //when
      controller.handleGas(RAPID_ACCELERATION);

      //then
      int desiredGear = currentGear - 1;
      verify(gearBox, times(1)).changeGear(desiredGear);
      verifyNoMoreInteractions(gearBox);
   }
}
```

Listing 5 zawiera przykłady dwóch rodzajów dublerów, natomiast dla zachowania kompletności omówimy wszystkie ich rodzaje.

Dummy – zaślepka

Nieużywane parametry, których istnienie jest wymuszone przez kompilator.

Fake – uproszczona implementacja

Uproszczone implementacje obiektów zależnych, np. baza danych in-memory.

Stub – zachowuj się!

Na Listingu 5 w sekcji *Stub* widzimy przykład dublera Podsystemu Monitorowania Silnika, którego jedynym celem jest zwracanie aktualnej ilości obrotów na minutę. Gwoli wyjaśnienia: w Mockito *Stub* tworzymy przy pomocy metody **`mock()`**. Dubler został stworzony na potrzeby bardzo prostego scenariusza testowego i zwraca zawsze tę samą wartość, natomiast w ogólności mógłby zwracać sekwencję odczytów.

Generalnie idea Stubów polega na tym, że „uczymy" je, jak mają się zachowywać, ale nie weryfikujemy interakcji odbytej z nimi. Czyli skupiamy się na ich wewnętrznym stanie.

Mock – czy odbyto z tobą interakcję?

Listing 5 w sekcji *Mock* ilustruje przykład dublera skrzyni biegów. Dubler ten zostaje dodatkowo „odpytany" w sekcji *then* o interakcję, jaką z nim odbyto. Intencja testu jest następująca: chcemy się upewnić, czy aby na pewno sterownik zredukuje bieg, czyli wywoła dokładnie jeden raz metodę **`changeGear`** z parametrem o wartości 4 i nie wywoła żadnej innej metody skrzyni biegów.

Mock pozwala bardzo precyzyjnie zweryfikować interakcję pomiędzy obiektami i jest użyteczny, gdy nie mamy dostępu do stanu obiektu (hermetyzacja lub bezstanowy serwis), a jedyny sposób na weryfikację logiki to sprawdzenie interakcji pomiędzy obiektami (nazwy wywołanych metod, ich parametry, ilość wywołań).

Czy jednak zawsze potrzebujemy tak precyzyjnej weryfikacji? Zbyt precyzyjna weryfikacja prowadzi do powstania testów typu „white-box". W testach tego typu opieramy się na wiedzy o wewnętrznej implementacji testowanych obiektów, co prowadzi do powstania tak zwanych kruchych testów (*fragile test smell*) – wszelka zmiana implementacji kodu testowanego wymaga zmiany kodu testów.

Na pytanie o potrzebę weryfikacji odpowiemy w dalszej części artykułu.

NARZĘDZIE CZY „RĘCZNA ROBOTA"?

W przykładzie dublerów z Listingu 5 posłużyliśmy się biblioteką do automatycznego generowania Mocków i Stubów. Wyobraźmy sobie jednak takie kryterium: sterownik działa poprawnie, jeżeli na x kilometrów zdecyduje o zmianie biegu o 2 w górę nie więcej niż 3 razy.

Przy pewnej komplikacji warunków, może się okazać, że stworzenie kodu Mocka własnoręcznie będzie prostsze i bardziej czytelne niż w przypadku zastosowania karkołomnych konstrukcji z bibliotek.

PARADYGMAT COMMAND-QUERY SEPARATION

Command-query separation jest zasadą programowania imperatywnego, która w kontekście programowania obiektowego sprowadza się do prostej reguły: każda metoda klasy powinna być zaliczona do kategorii command albo query – czyli nigdy do obu tych kategorii.

Metody typu command

Metody typu command są sygnałami wysyłanymi do obiektów lub innymi słowy rozkazami wykonania operacji.

Metody typu query

Metody typu query służą do odpytania obiektu o jego stan. Oczywiście obiekt może hermetyzować implementację i zwrócić pewną projekcję stanu. Jednak wielokrotne zadawanie pytań nie powinno mieć wpływu na odpowiedź. Pozwala to np. uniknąć Heisen-bugów, czyli błędów zachowujących się niczym w świecie kwantowym, gdzie obserwacja stanu zmienia stan.

Naruszenie paradygmatu

Listing 6 ilustruje prosty przykład naruszenia CQS. Aby poznać stan, musimy zmienić stan, wykonując operację, otrzymujemy stan.

Listing 6. Naruszenie CQS

```
public class Quant {
   int state;
   public int doSth(){
      state++;
      return state;
   }
}
```

Listing 7 ilustruje natomiast typowy dla aplikacji klasy Enterprise przykład naruszenia CQS: zatwierdzamy zamówienie, a w wyniku otrzymujemy listę zamówień. Metoda powstała zapewne na potrzeby przepływu ekranów, gdzie na jednym ekranie zatwierdzamy zamówienie i jeżeli operacja się powiedzie, to przechodzimy na ekran z listą własnych zamówień. Co jednak gdy operacja zakończy się niepowodzeniem, zmieni się flow ekranów lub pojawi się inny rodzaj aplikacji klienckiej (np. mobilna), która obsługuje inną logikę po zatwierdzeniu zamówienia (niepotrzebnie przesyłamy listę zamówień do zdlanej maszyny)?

Listing 7. Typowe naruszenie CQS w aplikacjach klasy Enterprise

```
public class OrderingService {
   public List<Order> confirm(String orderNumber){
      //...
   }
}
```

Oczywistym rozwiązaniem jest stworzenie 2 serwisów: jeden zajmujący się zatwierdzaniem zamówień, a drugi wyszukiwaniem zamówień. Wyższa warstwa (np. kontrolery MVC z warstwy prezentacji) powinna wywołać kolejno oba te serwisy. Natomiast jeżeli mamy przypadek zdalnego klienta i zależy nam na redukcji komunikacji sieciowej, oba te serwisy możemy „przykryć" fasadą.

Na marginesie: CQS to nie CQRS

W poprzedniej serii poświęconej Domain Driven Design poświęciłem wiele akapitów architekturze Command-query Responsibility Segregation. Dla porządku rozdzielimy pojęcia: CQS jest zasadą programowania obiektowego, natomiast CQRS jest rozwinięciem architektury warstwowej polegającym na wprowadzeniu 2 stosów warstw odpowiednich do 2 klas żądań obsługiwanych przez API serwera.

Tak więc CQRS jest przeniesieniem zasady CQS kilka poziomów wyżej, do skali architektury aplikacji i systemu.

SUMMA SUMMARUM: PROSTA RECEPTURA

Jeżeli zaczniemy przestrzegać zasady CQS i postrzegać każdą z metod jako sygnał rozkazujący wykonanie operacji albo zapytanie o stan, to wybór pomiędzy dublerem typu Mock albo Stub stanie się oczywisty.

Mock dla metod Command

Jeżeli nasz kod testowany wywołuje metodę obiektu współpracującego, która należy do kategorii command, to wówczas powinniśmy użyć dublera typu Mock.

Zwykle jest to krytyczne, aby upewnić się, czy aby na pewno rozkazy zostały wysłane w zamierzony przez nas sposób.

Dlatego na Listingu 5 upewniamy się, w jaki sposób sterownik steruje skrzynią biegów.

Stub dla metod Query

Jeżeli nasz kod testowy wywołuje metodę obiektu współpracującego, która należy do kategorii query, to wówczas wystarczający jest dubler typu Stub. Nie weryfikujemy, ile razy dubler został odpytany o stan. Nie ma to znaczenia, ponieważ metody query nie zmieniają stanu – nie powodują efektów ubocznych. Dzięki temu nie uzależniamy testów od implementacji (testy white-box), a co za tym idzie kod testów jest mniej podatny na zmiany kodu testowanego.

Oczywiście możemy założyć, że niepotrzebne wywołania metod query mogą mieć znaczący wpływ na wydajność systemu (np. zbędne zapytania do bazy danych), ale jak mówi zasada nadrzędna: przedwczesna optymalizacja jest przyczyną całego zła na świecie.

THERE IS NO SPOON – PODEJŚCIE FUNKCYJNE

Znamy już prostą „zasadę kciuka" mówiącą czym kierować się podczas wyboru rodzaju dublera w testach jednostkowych. Jednak cała ceremonia związana z tworzeniem kodu dublerów jest żmudna i pracochłonna. Biblioteki typu Mockito znacznie ją redukują, jednak jak mawiali prastarzy programiści: najlepszy jest ten kod, którego nie ma – nie ma z nim problemów.

Będziemy zatem dążyć do unikania dublerów. Wymaga to jednak nieco innego podejścia do modelowania problemów. Kod testu z Listingu 5 wymaga stosowania dublerów, ponieważ kod poddawany testowaniu (Listing 1) opiera się na strukturze zagnieżdżonych obiektów.

Gdyby podejść do problemu tak jak na Listingu 8, wówczas będziemy testować „funkcję" (jej odpowiednikiem w języku obiektowym jest funktor). Funkcja posiada wejście oraz wyjście i nie posiada zależności, zatem nie musimy tworzyć dla nich dublerów.

W przykładzie aktualne obroty silnika są przekazywane w parametrze zamiast pobierania ich z podsystemu monitorowania silnika.

Konkretne implementacje interfejsu (Eco, Comport, Sport) będą wykonywać obliczenia wg własnych algorytmów.

Oczywiście będziemy potrzebować klas wyższego rzędu/warstwy, aby zaprogramować w nich orkiestrację – zdobycie danych wejściowych, wywołanie funktora i wykorzystanie danych wejściowych. Jednak możemy założyć, że z uwagi na mnogość przypadków nie będziemy wyższej warstwy testować jednostkowo. Zakładając, że cała komplikacja znajduje się w funktorze, możemy tylko na nim skupić precyzyjne testy jednostkowe.

Listing 8. Funkcja (funktor) odpowiedzialna za krytyczne obliczenia

```
public interface GearCalculator {
   public int calculate(int currentRPM, double throttle,
                        double breakingForce);
}
```

Rodzaje dublerów

- Dummy – pomijalny parametr wymagany np. przez kompilator
- Fake – alternatywna, uproszczona implementacja
- Stub – obiekt, który „uczymy", jak powinien się zachować
- Mock – obiekt, który odpytujemy o to, czy odbyto z nim określoną interakcję

Paradygmat Command-query Separation

- Command – metoda będąca sygnałem do wykonania pracy (zmiany stanu)
- Query – metoda odpytująca o stan (nie zmienia stanu)

Receptura

- Mock ↔ Command – dla metod typu command stosuj Mocki, aby zweryfikować, jakie rozkazy, z jakimi parametrami oraz ile razy wydano
- Stub ↔ Query – dla metod typu query nie ma potrzeby weryfikowania interakcji – test „black-box" (niezależne od implementacji kodu testowanego)

W sieci

- Martin Fowler o różnicy Mock vs Stub http://martinfowler.com/articles/mocksArentStubs.html
- Artykuły z serii DDD, wśród których znajdują się materiały poświęcone testowaniu automatycznemu w ujęciu: problemy, strategie, taktyki, techniki, narzędzia: http://bottega.com.pl/artykuly-i-prezentacje
- Rozproszenie uwagi i rodzaje ludzkiej pamięci: http://blog.ninlabs.com/2013/01/programmer-interrupted/
- Mockito: http://code.google.com/p/mockito/
- Cqs: http://en.wikipedia.org/wiki/Command-query_separation

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).