

Testy jednostkowe trudnego kodu w Javie

Testowanie metod statycznych i konstruktorów

Istnieje szereg konstrukcji w kodzie Javy, które skutecznie utrudniają pisanie testów jednostkowych. Należą do nich wywołania konstruktorów i metod statycznych. Celem tego artykułu jest przedstawienie różnorodnych technik, pomocnych w pisaniu testów dla kodu, w którym takie konstrukcje występują.

Dowiesz się:

- jak przetestować jednostkowo każdy kod w Javie,
- jakie są silne i słabe strony różnych technik testowania metod statycznych i konstruktorów,
- do czego mogą przydać się: refaktoring, aspekty i instrumentacja.

Powinieneś:

- czym testy jednostkowe różnią się od innych rodzajów testów,
- umieć pisać testy jednostkowe (z wykorzystaniem TestNG i/lub JUnit),
- umieć korzystać z bibliotek mocków (np. EasyMock, jMock),
- mieć pojęcie o AspectJ.

Poziom trudności



Artykuł podzielony jest na trzy części. W pierwszej przedstawimy problem i poznamy fragment kodu, z którym przyjdzie się nam zmierzyć. W drugiej, najobszerniejszej, poznamy cztery różne techniki, które umożliwią przetestowanie takiego kodu, i omówimy ich silne i słabe strony. W części trzeciej pokusimy się o podsumowanie całości zagadnienia, dając ogólne zalecenia co do postępowania z trudnym do testowania kodem.

W sieci znaleźć można liczne artykuły wyczerpująco omawiające testy jednostkowe i nie ma potrzeby byśmy powtarzali zawarte w nich informacje. Do naszych celów wystarczy taka oto maksymalnie okrojona definicja testu jednostkowego:

Test jednostkowy testuje możliwie mały fragment systemu (najczęściej pojedynczą klasę) w izolacji od reszty systemu.

Testowany fragment kodu izolujemy poprzez zastąpienie wszelkich obiektów współpracujących specjalnie przygotowanymi obiektami mock, które jedynie symulują ich działanie we wskazany przez nas sposób. Dzięki temu, w

przypadku niepowodzenia (tj. gdy test *nie przejdzie*) możemy bardzo precyzyjnie zlokalizować wadliwy fragment kodu i co za tym idzie, bardzo szybko dokonać stosownej poprawki.

Tworzenie mocków ułatwiają wyspecjalizowane biblioteki – ich najbardziej znanymi przedstawicielami w świecie Javy są jMock i EasyMock. Cechą obu wymienionych bibliotek jest to, że tworzą one mocki w oparciu o mechanizm proxy.

Korzystając z wyżej wymienionych bibliotek, w klasie testowej tworzymy obiekty proxy, a następnie podsuwamy je testowanemu obiektowi zamiast prawdziwych obiektów współpracujących. Możemy tego dokonać, przekazując proxy poprzez setter, jako argument konstruktora lub jako argument testowanej metody – pod warunkiem, że klasa testowanego obiektu dostarcza taką możliwość. I na tym właśnie polega ograniczenie tej techniki.

Pewne powszechnie stosowane konstrukcje, jak np. tworzenie instancji obiektu przez wywołanie konstruktora lub niektóre ze wzorców projektowych (np. wzorzec singletonu lub fabryki), nie poddają się testowaniu przy pomocy mechanizmu proxy. Nie ma bowiem możliwości zastąpienia takich obiektów współpracujących stworzonymi obiektami proxy.

Skoro nie potrafimy tych konstrukcji przetestować, to ich nie stosujemy uznali developerzy i gre-

mialnie zwrócili się w stronę frameworków IOC, które eliminują konieczność użycia poprzednio wymienionych konstrukcji i czynią kod łatwiejszym do testowania. Jednak nie w każdym projekcie użycie Springa czy Guice ma sens i nadal istnieje wiele sytuacji, gdzie najbardziej naturalnym rozwiązaniem jest bezpośrednie wywołanie konstruktora czy też użycie wzorca fabryki (ang. factory). Z tego powodu, nawet mimo popularności IOC, wciąż istnieje potrzeba testowania kodu, który z takich konstrukcji korzysta.

Przyjrzyjmy się fragmentowi kodu, na którym będziemy testować różne podejścia. Listing 1. przedstawia fragment klasy dokonującej pewnych operacji biznesowych (przykład zaczerpnięty z tutoriala biblioteki jMockit). Kod ten zawiera zarówno wywołanie konstruktora (`new ServiceB()`) jak i metod statycznych (`Database.find(...)`, `Database.save(...)`).

Możemy zgłosić pod adresem kodu z Listingu 1. zarzut, który najłatwiej wyrazić jako *to powinno się było napisać inaczej!* (i dalej rozwodzić się nad tym, czemu singletony są niedobre, nad zaletami kodowania do interfejsu, a nie do implementacji, dependency injection itp. itd.). Zgadzać się, tak nie powinniśmy kodować. Jednak celem tego artykułu jest przetestowanie właśnie takiego, dalekiego od ideału kodu. Stąd też przejdziemy do porządku dziennego nad zgłoszoną wątpliwością, skupiając się jedynie na technicznej stronie zagadnienia.

Zastanówmy się, co warto przetestować w powyższym fragmencie kodu. Wydaje się, że są to dwie rzeczy:

- czy pole `total` obiektu `data` jest wypełnione wartością wyliczoną przez `ServiceB` na podstawie danych zwróconych przez zapytanie skierowane do `Database`,
- czy obiekt `data` został zapisany do bazy danych.

Rozwiązania

Możliwe rozwiązania podzielimy na dwie grupy. Pierwsza z nich umożliwia napisanie testów bez jakiegokolwiek ingerencji w kod źródłowy klasy. Druga wymaga dokonania pewnych zmian w kodzie testowanej klasy, przekształcając ją w taki sposób, by dalej można już było skorzystać z *tradycyjnych* bibliotek mocków (tj. opartych o mechanizm proxy). Każde z opisywanych rozwiązań rozważamy, wskazując na jego silne i słabe strony.

Metody niewymagające zmian w kodzie testowanej klasy

Poniżej przedstawiam dwa rozwiązania oparte odpowiednio na bibliotece JEasyTest i jMockit.

Ograniczymy do minimum opis obu bibliotek, skupiając się tylko na cechach istotnych z punktu widzenia wykorzystania ich do przetestowania interesującego nas fragmentu kodu (jednocześnie zachęcam do przyjrzenia się wszystkim możliwościom przez nie oferowanym).

Nim przejdziemy do szczegółów technicznych, pozwolę sobie przytoczyć fragmenty z tutoriala JEasyTest, w którym autor przestrzega przed zachłystaniem się możliwościami, jakie niesie ze sobą ta biblioteka (tłumaczenie własne):

- Nigdy nie powinniśmy zapominać o koncepcjach takich jak luźne powiązanie komponentów, zapewnienie wysokiej wewnętrznej spójności komponentów i wyraźne określenie ich obowiązków, modularyzacja, kodowanie raczej do interfejsów niż do implementacji, zasada żądaj, a nie prosz, dependency injection, i wielu innych.
- [...] fakt, że JEasyTest umożliwia np. mockowanie metod statycznych, nie jest powodem byśmy ignorowali zasady programowania obiektowego. Używajmy JEasyTest w rozważny sposób.

Mechanizm proxy a testy jednostkowe

Mechanizm proxy stanowi część pakietu `java.lang.reflect` i jest dostępny od wersji JDK 1.3. Umożliwia tworzenie obiektów proxy, implementujących jeden lub wiele interfejsów, których zadaniem jest przyjmowanie żądań i przekazywanie ich dalej do obiektu, który w rzeczywistości je obsługuje. Mechanizm proxy jest powszechnie wykorzystywany przez biblioteki tworzące mocki (EasyMock, jMock) aby:

- przechwycić żądanie skierowane do obiektu, który zastąpiliśmy mockiem,
- zapamiętać wywołania metod,
- zwrócić żądane wartości.

Słabością proxy z JDK jest fakt, że umożliwia on wyłącznie tworzenie obiektów proxy dla interfejsów. Z tego powodu wyżej wymienione biblioteki posiłkują się biblioteką CGLib zdolną tworzyć obiekty proxy również dla konkretnych klas.

Aspekty – JEasyTest

Pierwsze z proponowanych rozwiązań wykorzystuje bibliotekę JEasyTest, która działa w oparciu o AspectJ. Jej autor - Roberto Malagigi, stworzył ją z myślą o użyciu jako plugin do Eclipse, w rzeczywistości można ją również z powodzeniem zastosować do naszych celów.

Spójrzmy na listing 2., który przedstawia kluczowe fragmenty klasy testującej napisanej z użyciem JEasyTest.

W kodzie na Listingu 2. widać dwie specyficzne dla JEasyTest adnotacje – `@ClassUnderTest` i `@JEasyTest`. Adnotacja `@ClassUnderTest` wskazuje bibliotecę JEasyTest, jaka klasa jest testowana – dzięki temu tworzone są aspekty zawierające pointcuty wskazujące na odpowiednie miejsca w kodzie testowanej klasy (po skompilowaniu aspekty te znajdują się w katalogu `target/jeasytest/generatedAspects` – warto się im przyjrzeć, by mieć lepsze pojęcie o tym, co robi JEasyTest). Adnotacja `@JEasyTest` ozna-

cza po prostu metodę testową, którą JEasyTest powinien się zainteresować.

Najciekawsze są jednak fragmenty wykorzystujące statyczne metody z pakietu `org.jeasytest.external.Util` – fragment `on(Database.class).expectStaticNonVoidMethod("save").(...)` oraz `on(ServiceB.class).(...)`. Przedstawiają one oczekiwania wobec tego, co ma się stać. Taki fragment kodu:

```
on(Database.class).expectStaticNonVoidMethod
    ("find").with(arg("select item from
        EntityY item where
        item.someProperty=?"),
        arg("abc")).andReturn(Collections.
            EMPTY_LIST);
```

możemy przeczytać jako oczekiwanie, że *oto na klasie Database zostanie wywołana statyczna metoda find z dwoma argumentami - select item from EntityY item where item.someProper-*

Listing 1. Kod do przetestowania

```
public final class ServiceA {
    public void doBusinessOperationXyz(EntityX data) throws InvalidItemStatus {
        List items = Database.find("select item from EntityY item
            where item.someProperty=?",
            data.getSomeProperty());
        BigDecimal total = new ServiceB().computeTotal(items);
        data.setTotal(total);
        Database.save(data);
    }
}
```

Listing 2. Test napisany z użyciem biblioteki JEasyTest

```
@ClassUnderTest(ServiceA.class)
public class ServiceATest extends TestCase {

    private final static BigDecimal TOTAL = new BigDecimal("125.40");

    @JEasyTest
    public void testBusinessOperation() throws InvalidItemStatus {
        EntityX entity = new EntityX(1, "entityName", "entityCode");
        ServiceA serviceA = new ServiceA();
        ServiceB servB = createMock(ServiceB.class); // EasyMock

        // oczekiwania wywołań metod statycznych i konstruktora
        // z użyciem JEasyTest
        on(Database.class).expectStaticNonVoidMethod("find").with(
            arg("select item from EntityY item where item.someProperty=?"),
            arg("abc")).andReturn(Collections.EMPTY_LIST);
        on(ServiceB.class).expectEmptyConstructor().andReturn(servB);
        on(Database.class).expectStaticVoidMethod("save").with(arg(entity));

        // oczekiwanie wywołania metody z użyciem EasyMock
        expect(servB.computeTotal(Collections.EMPTY_LIST)).andReturn(TOTAL);
        replay(servB);
        serviceA.doBusinessOperationXyz(entity);
        verify(servB);
        assertEquals(TOTAL, entity.getTotal());
    }
}
```

Testowanie Oprogramowania

Listing 3. Test napisany z użyciem biblioteki jMockit

```

@Test
public class ServiceATest extends Expectations {
    // wersja klasy Database, którą zastąpimy prawdziwą klasę Database
    public static class MockDatabase {
        static int findMethodCallCount;
        static int saveMethodCallCount;

        public static void save(Object o) {
            saveMethodCallCount++;
        }

        public static List find(String ql, Object arg1) {
            findMethodCallCount++;
            return Collections.EMPTY_LIST;
        }
    }

    public void testDoBusinessOperationXyz() throws Exception {
        // podmieniamy definicję klasy Database
        Mockito.redefineMethods(Database.class, MockDatabase.class);
        EntityX data = new EntityX(5, "abc", "5453-1");

        final BigDecimal total = new BigDecimal("125.40");
        // podmieniamy klasę ServiceB
        Mockito.redefineMethods(ServiceB.class, new Object() {
            public BigDecimal computeTotal(List items) {
                return total;
            }
        });

        new ServiceA().doBusinessOperationXyz(data);
        // sprawdzenie wyników i wartości zapamiętanych przez test-spy
        assert total.equals(data.getTotal());
        assert MockDatabase.findMethodCallCount == 1;
        assert MockDatabase.saveMethodCallCount == 1;
    }
}

```

Listing 4. Testowana klasa po refaktoringu extract method

```

public class ServiceA
{
    public void doBusinessOperationXyz(EntityX data) throws InvalidItemStatus {
        List items = find(data);
        ServiceB serviceB = getServiceB();
        BigDecimal total = serviceB.computeTotal(items);
        data.setTotal(total);
        save(data);
    }

    void save(EntityX data) {
        Database.save(data);
    }

    ServiceB getServiceB() { // uwaga - domyślny identyfikator dostępu !
        return new ServiceB();
    }

    List find(EntityX data) { // uwaga - domyślny identyfikator dostępu !
        return Database.find(
            "select item frcm EntityY item where item.someProperty=?", data
                .getSomeProperty());
    }
}

```

ty=? oraz abc; w odpowiedzi na co, klasa Database zwróci pustą listę.

Taki zapis przetwarzany jest przez JEasyTest na odpowiednie konstrukcje języka AspectJ, a następnie w trakcie budowania projektu kompilator AspectJ wplata je w kod źródłowy testowanej klasy.

Każdy, kto korzysta z frameworków typu EasyMock, z łatwością odnajdzie się wśród metod udostępnianych przez JEasyTest. Zauważmy, że w przypadku EasyMock (zakładając, że jesteśmy w stanie podmienić obiekt reprezentujący bazę danych przez mock), analogiczna linia kodu wygląda podobnie (choć zwęższej):

```

expect(databaseStub.find("select item from
    EntityY item where item.someProperty=?",
    "abc")).andReturn(Collections.
    EMPTY_LIST);

```

W omawianym fragmencie kodu z listingu 2. używamy JEasyTest wyłącznie do testowania wywołań metod statycznych i konstruktorów – tam gdzie to możliwe, korzystamy ze standardowego rozwiązania z użyciem biblioteki EasyMock.

Korzystając z JEasyTest, należy zmodyfikować sposób budowania projektu – przed odpaleniem testów odpowiednie aspekty muszą zostać stworzone i wplecione w kod. Nie wymaga to od programisty żadnego wysiłku – JEasyTest dostarcza stosownych tasków ANT-a do wykonania tych zadań; istnieje też plugin do mavena oferujący analogiczną funkcjonalność.

Zalety:

- nie musimy dokonywać żadnych zmian w kodzie testowanej klasy,
- kod klasy testowej jest bardzo czytelny (dla osób, którym nie jest obca składnia stosowana powszechnie w bibliotekach mocków).

Wady:

- prosty refaktoring (*RenameMethod*) może uszkodzić test (np. zmiana nazwy metody *find* w klasie *Database*),
- dłuższa kompilacja projektu (związana z generacją aspektów i ich wplataniem w bytecode, czym zajmuje się kompilator AspectJ) a co za tym idzie wolniejsze odpalanie zestawu testów,
- nie radzi sobie z klasami oznaczonymi jako *final*.

Instrumentacja

(mechanizm redefinicji klas) – jMockit

W drugim rozwiązaniu skorzystamy z biblioteki jMockit, która opiera się na dostępnym w JDK od wersji 1.5 mechanizmie redefinicji klas (pakiet `java.lang.instrument`) i używa biblioteki ASM.

Listing 3. przedstawia fragment kodu klasy testującej. W kodzie przedstawionym na Listingu 3. wprowadziliśmy nową klasę `MockDatabase`, którą wykorzystamy jako *test-spy* (do zliczania wywo-

łań metod) oraz *stub* (do zwrócenia pustej kolekcji przy wywołaniu metody *find*) podczas testu.

Cała tajemnica jMockit opiera się na metodzie *redefineMethods*, która w powyższym przypadku umożliwia:

- zastąpienie oryginalnej klasy *Database* przez klasę *MockDatabase*,
- podmianę metody *computeTotal* w klasie *ServiceB* na implementację zwracającą wcześniej ustaloną wartość.

Podczas uruchamiania testów należy uaktywnić agenta jMockit poprzez przekazanie do

JVM parametru `-javaagent:jmockit.jar`. Jako dodatkowy zysk ze stosowania jMockit należy podkreślić możliwość podmiany klas oznaczonych jako *final* (w powyższym przykładzie fakt, że klasa *ServiceB* jest oznaczona jako *final* nie stanowi przeszkody w testowaniu) – jest to nieosiągalne przy pomocy innych opisywanych tu technik.

Zalety:

- nie musimy dokonywać żadnych zmian w kodzie testowanej klasy,
- działa również dla klas oznaczonych jako *final*,

- nie ma problemów z mierzaniem pokrycia kodu (ang. *code coverage*) (warto tu zaznaczyć, że jMockit dostarcza własne narzędzie do mierzania pokrycia kodu).

Wady:

- musimy ręcznie stworzyć klasę/klasę *test-spy*, co znacznie zwiększa rozmiary klasy testu i (zazwyczaj) wprowadza pewną dozę logiki do klasy testu.

Metody wykorzystujące przekształcenia kodu testowanej klasy

W odróżnieniu od zaprezentowanych wcześniej technik, sposoby opisywane w tym rozdziale opierają się na takim przekształceniu kodu klasy testowanej, by mógł on podlegać testowaniu przy pomocy tradycyjnych, tj. opartych na mechanizmie proxy, bibliotek *mocków*.

Refaktoring + dziedziczenie

Skorzystajmy z refaktoringu *extract method*, by wydzielić do osobnych metod niewygodne fragmenty kodu (czyli te zawierające odwołania do singletonów lub wywołania konstruktorów) – sens tego kroku stanie się jasny już niżej. Listing 4. przedstawia klasę *ServiceA* po dokonaniu zmian (proszę porównać z oryginalną wersją widoczną na Listingu 1.).

Dzięki domyślnym modyfikatorom dostępu w nowo utworzonych metodach, możemy w klasie testowej stworzyć klasę dziedziczącą z klasy testowanej i nadpisać te metody. Na Listingu 5. widać, że w klasie testowej stworzyliśmy podklasę testowanej klasy *ServiceA*. Jej implementacja nadpisuje trzy nowo utworzone metody – *getService*, *find*, *save* – które zwracają wartości wygodne z punktu widzenia testu. W ten sposób niejako stworzyliśmy obiekty współpracujące (w postaci nadpisanych metod) tam gdzie ich wcześniej nie było.

Zalety:

- żadnej *magii* opartej na aspektach czy instrumentacji – mniej doświadczeni developerzy bez trudu połączą się o co chodzi,
- kod metody testowej `testBusinessOperation` jest bardzo prosty do zrozumienia,
- bezproblemowy *code coverage*,
- API testowanej klasy pozostaje niezmienione.

Wady:

- nie jesteśmy w stanie stwierdzić:
 - czy wywołano metodę *save* na obiekcie *Database*,
 - czy wywołano metodę *computeTotal* na obiekcie klasy *ServiceB*.
- wymaga dużo pracy,
- musimy dokonać znacznej ingerencji w kod klasy – przy dużej ilości wywołań metod statycznych lub konstruktorów prowadzi to do bardzo dużych zmian,

Listing 5. Test dla niefaktorowanej klasy

```
public class ServiceATest extends TestCase {

    private static final BigDecimal TOTAL = BigDecimal.TEN;

    class MyServiceA extends ServiceA {
        private boolean saved = false;
        private ServiceB serviceB;

        public void setServiceB(ServiceB serviceB) {
            this.serviceB = serviceB;
        }

        @Override
        ServiceB getServiceB() {
            return serviceB;
        }

        @Override
        List find(EntityX data) {
            return Collections.EMPTY_LIST;
        }

        @Override
        void save(EntityX data) {
            this.saved = true;
        }

        public boolean isSaved() {
            return saved;
        }
    };

    public void testBusinessOperation() throws InvalidItemStatus {
        MyServiceA serviceA = new MyServiceA();
        ServiceB serviceB = createMock(ServiceB.class);
        serviceA.setServiceB(serviceB);
        EntityX entity = new EntityX(1, "abc", "def");
        expect(serviceB.computeTotal(Collections.EMPTY_LIST)).andReturn(TOTAL);
        replay(serviceB);
        serviceA.doBusinessOperationXyz(entity);
        verify(serviceB);
        assertEquals(TOTAL, entity.getTotal());
        assertTrue(serviceA.isSaved());
    }
}
```

Testowanie Oprogramowania

- zmieniliśmy znacznie kod testowanej klasy, w klasie testowej testowaliśmy jej potomka, a nie tę klasę... to, co właściwie przetestowaliśmy?
- klasa testowa urosła do nieprzyzwoitych rozmiarów i zawiera bardzo dużo kodu, którego zadaniem jest przygotowanie gruntu pod właściwy test,
- by móc w testach dokonać sprawdzenia różnych scenariuszy, będziemy musieli w klasie testowej albo stworzyć kilka klas dziedziczących z testowanej klasy (co oznacza sporo pracy) lub stworzyć jedną klasę i wyposażać ją w pewną dozę logiki (co również oznacza sporo pracy i dodatkowo możliwość wprowadzenia błędów w samej klasie testowej!),
- czy wprowadzając zmiany do testowanej klasy polepszyliśmy jej design, czy też popsuliśmy go? – jeżeli to drugie, to doszło do paradoksalnej sytuacji – testy, które ze swojej natury mają stać na straży jakości kodu, doprowadziły do pogorszenia pewnych cech tego kodu.

Zmiana API

Ostatnia z prezentowanych technik jest bardzo prosta (przynajmniej w teorii) i opiera się na prostym założeniu, że wszelkie niewygodne z punktu widzenia testów konstrukcje należy wyeliminować. Słusznie, ale w przypadku omawianego kodu, takie podejście oznacza poważne zmiany nie tylko w API testowanej klasy, ale i w całym projekcie:

- np. singleton `Database` należy zastąpić zwykłą klasą (o jej *singletonowości* zadba np. Spring),
- klasa `ServiceA` powinna otrzymywać obiekty współpracujące jako:
 - argumenty konstruktora,
 - poprzez settery,

- jako argumenty metody `doBusinessOperationXyz`.

Zakładając, że wybierzemy najpopularniejszą z opcji (settery), kod testu wyglądać będzie tak:

Jak widać na Listingu 6., w przypadku gdy wszystkie obiekty współpracujące są wstrzeliwane zgodnie z DI, wówczas w teście wystarczy zastąpić je klasycznymi mockami.

Zalety:

- poprawiliśmy design testowanej klasy,
- (najprawdopodobniej) poprawiliśmy design całego projektu,
- testy stają się proste.

Wady:

- musieliśmy zmienić API klasy,
- (najprawdopodobniej) musieliśmy dokonać wielu zmian w całym projekcie.

Podsumowanie

Jak zwykle w przypadku istnienia wielu metod poradzenia sobie z problemem nie istnieje *najlepsze* rozwiązanie. Wszystko zależy od konkretnego przypadku. Jeżeli miałbym zaproponować ogólny sposób postępowania to

radziłbym następująco. Jeżeli czas pozwala, wówczas z pewnością warto *uzdrowić* kod kłopotliwych klas (metoda 4.) – zyski będą długoterminowe.

Jeżeli czasu brak (bo np. odziedziczyliśmy projekt, który w całości nadaje się do przepisanania, lub też korzystamy z licznych bibliotek zewnętrznych narzucających korzystanie z określonych konstrukcji), wówczas pozostają techniki oparte na aspektach albo instrumentacji.

Nie widzę tutaj wyraźnej przewagi żadnej z nich i wybór pozostawiam gustowi czytelnika.

Osobiście wybieram `jEasyTest`, bo odpowiada mi jego składnia (zbliżona do znanych mi frameworków), a w `jMockit` irytuje mnie konieczność ręcznej implementacji zastępowanej klasy. Ale jako że oba podejścia mają swoje wady i zalety, nie skreślam żadnego z nich.

Również w przypadku, jeżeli kłopotliwy do testowania kod występuje bardzo sporadycznie w projekcie, wówczas prawdopodobnie najłatwiej posłużyć się techniką 3. lub 4., nie włączając do procesu testowania dodatkowych bibliotek.

Listing 6. Test dla klasy o zmienionym API

```
private static final BigDecimal TOTAL = new BigDecimal(10);
private static final String SOME_PROPERTY = "some_property";
private ServiceA serviceA;
private EntityDAO entityDAO;
private ServiceB serviceB;
private Entity entity;
private List items;

@Override
protected void setUp() throws Exception {
    serviceA = new ServiceAImpl();
    serviceB = createMock(ServiceB.class);
    entityDAO = createMock(EntityDAO.class);
    entity = createMock(Entity.class);
    items = createMock(List.class);
    serviceA.setEntityDAO(entityDAO);
    serviceA.setServiceB(serviceB);
}

public void testBusinessOperation() throws InvalidItemStatus {
    expect(entityDAO.find("select item from EntityY item where item.someProperty=?",
        SOME_PROPERTY)).andReturn(items);
    expect(entity.getSomeProperty()).andReturn(SOME_PROPERTY);
    expect(serviceB.computeTotal(items)).andReturn(TOTAL);
    entity.setTotal(TOTAL);
    entityDAO.save(entity);
    replay(entityDAO, serviceB, entity, items);
    serviceA.doBusinessOperationXyz(entity);
    verify(entityDAO, serviceB, entity, items);
}
```

TOMASZ KACZANOWSKI

Jestem developerem Javy, pracuję w firmie *Software Mind S.A.* Od dłuższego czasu śledzę (i wypróbuję na własnym kodzie) wszystko co związane z testami – zwłaszcza jednostkowymi.

Kontakt z autorem: tkaczano@poczta.onet.pl

W Sieci

<https://jeasytest.dev.java.net/> – strona domowa projektu `JEasyTest`

<http://www.jmock.org> – strona domowa projektu `jMock`

- <http://www.easymock.org> – strona domowa projektu `EasyMock`
- <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html> – informacje o mechanizmie proxy
- <https://jmockit.dev.java.net/> – strona domowa projektu `jMockit`