Mock Objects

# Uwagi ogólne

W procesie tworzenia oprogramowania, czymś naturalnym jest pisanie kodu, który będzie składał się z zależności w postaci łańcucha wywołań wielu różnych obiektów. Co zrobić w przypadku gdy chcemy przetestować funkcjonalność ze środka tego łańcucha zależności? Jednym z rozwiązań jest tzw. Mockowanie obiektów. Jest to technika za pomocą której zamienia się konkretną implementację obiektu sztucznym tworem, który jest w stanie zachować się dokładnie tak jak tego oczekujemy. Dzięki temu możemy przetestować wybrany scenariusz bez przejmowania że nasz obiekt zależy od innych obiektów.

W ramach tej listy zadań będziemy testować mały projekt odpowiedzialny za obsługę zamówień. W kolejnych zadaniach będziemy pisać testy do implementacji OrderService. Do tworzenia mock’ów użyjemy frameworka Mockito. Projekt jest dołączony do listy.

Żeby włączyć testy należy otworzyć projekt w IDE (np Netbeans, Eclipse) i wybudować projekt. Alternatywnie można wykonać następujące polecenie w folderze ./mocks\_homework1:

-mvn clean install

Zadania należy rozwiązywać w przygotowanej klasie testowej OrderServiceTest znajdującej się w folderze ./ homework1/src/test/java/pl/luciow/warehouse/.

Dokumentacja do mockito: <http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html>

# Zadania do wykonania

Zadanie 1. Przetestuj funkcję fillOrder() w przypadku gdy wywołanie removeItems na warehouse nie rzuci wyjątku. Żeby utworzyć mock’a warehouse piszemy:

Warehouse warehouseMock = Mockito.mock(Warehouse.class);

Następnie musimy zaprogramować mock żeby zachował się tak jak tego oczekujemy.

Mockito.when(warehouse.removeItems(Mockito.any(List.class))).thenReturn(null);

Zadanie 2. Przetestuj funkcję fillOrder() w przypadku gdy wywołanie removeItems na warehouse rzuci wyjątek. Możemy to zaprogramować w mocku następująco

Mockito.when(…).thenThrow (new NotEnoughItemsException);

Zadanie 3. Przetestuj funkcję cancelOrder() w WarehouseImpl ustawiając mock tak, żeby w wywołaniu addItems() wywołał oryginalną funkcję. Żeby zaprogramować funkcję zwracającą void, musimy użyć trochę innej składni niż w poprzednim zadaniu:

Mockito.doCallRealMethod().when(warehouseMock).addItems(Mockito.any(List.class));

Zadanie 4. Przetestuj funkcję processPayment() ustawiając mock PaymentService tak, żeby przy wywołaniu metody processPayment() rzucił wyjątek. Zweryfikuj czy do metody sendMail() na mock’u MailService trafia mail z treścią „Error occured”. W tym celu musimy stworzyć klasę dziedziczącą po klasie ArgumentMatcher i nadpisać metodę matches(Object o); W tej metodzie musimy sprawdzić czy obiekt ‘o’ spełnia nasze wymagania. Po wywołaniu testu musimy sprawdzić czy wszystko jest ok. W tym celu:

Mockito.verify(mailServiceMock).sendMail((Mail)Mockito.argThat(new MyArgumentMatcher()));

Zadanie 5. Przetestuj funkcję processPayment() ustawiając mock PaymentService tak, żeby przy wywołaniu metody processPayment() zwrócił id. Zweryfikuj czy do metody sendMail() na mock’u MailService trafia mail z treścią „Success”.

Zadanie 6. Przetestuj funkcję processPayment() w klasie PaymentServiceImpl weryfikując czy w przypadku rzucenia wyjątku przez ValidatorUtils płatność **nie** jest przekazywana do dalszego przetwarzania. Ten test napisz w klasie PaymentServiceImplTest, w pakiecie pl.luciow.warehouse.impl. Żeby wykonać to zadanie będzie trzeba skorzystać z mechanizmu programowania metod za pomocą obiektów Answer, ustawiając mock PaymentValidatora tak, żeby w przypadku wywołania metody validate dodał String do otrzymanej listy błędów. Przykład użycia obiektu Answer można znaleźć w podanej dokumentacji. Zweryfikować ilość wywołań metody można następująco:

Mockito.verify(mock, Mockito.times(1)).method().;