|  |  |
| --- | --- |
| MOCKOWANIE | * Atrapy: Obiekt w teście, który jest nam potrzebny jako wypełnienie, a nie spełnia żadnego logicznego celu. Możemy w tym celu wykorzystać nawet pustą klasę. * Fake: Obiekt, który zawiera już logikę, ale nie taką jak prawdziwa implementacja. Na przykład tabelkaz danymi zamiast bazy danych. * Zaślepka(stub): Obiekt mający minimalną implementację potrzebnego przez nas interfejsu. Zazwyczaj funkcje zwracają dla jakiejś danej wejściowej predefiowaną daną wyjściową.  W praktyce tworzymy nowa klasę która implementuje stubbowany interfejs, w niej tworzymy wersje metod interfejsu które np. zwrócą nam gotowy, znany nam zestaw danych. * Mock: Obiekt, który poza predeniowaną daną wyjściową śledzi jeszcze wszystkie interakcje z interfejsem. Zazwyczaj bardziej skomplikowane klasy. Najlepiej skorzystać już z dostępnych wchodzących w skład bibliotek. |
| MOCKITO | * Jeden z frameworków do Mockowania * Jeżeli w teście potrzebuję sztucznego obiektu np. serwisu do udawania współpracy z baza danych:  //mock będzie działał po statycznym imporcie  private static DatabaseService dbService =   mock(DatabaseService.class); * Przez to że mock nie ma w sobie logiki musimy mu hardcodować to co ma zwrócić:  when(mockName.foo(args)).thenReturn(oczekiwanyReturn); np. hardkodujemy co repozytorium ma zwrócić przy danych argumentach. Najczęściej chcemy żeby zwróciło jakiegoś stuba. * Mockujemy np. metody z repozytoriów, które będą wywoływane przez serwisy * Z mocków najczęsciej dummy obiekty. Są to, znowu, hardcodowane obiekty które będziemy wrzucać do mocków. * Teraz już test nie będzie mi rzucał wyjątków, że nie mógł znaleźć tego obiektu. Taki obiekt jest wydmuszką, nie ma w sobie wbudowanej żadnej logiki, nawet jeżeli klasa, z której jest tworzony ma logikę biznesową w sobie. * Właściwą klasę, której metody testujemy, musimy zainicjalizować, żeby mieć dostęp do logiki w niej zawartej, inaczej nie moglibyśmy wywołać metody do testowania. |
| ANNOTACJE | Aby aktywować annotacje w Javie:  // LUB przed klasą testową  @RunWith(MockitoJUnitRunner.class)  // LUB  @Before /\* or @BeforeEach \*/  public void initMocks() {  MockitoAnnotations.initMocks(this);  }  - @Mock  Foo foo;  Tworzy mocka klasy Foo pod referencją foo. |
| WHEN-thenRETURN | * Jeżeli chcemy powiedzieć mockito jaki wynik powinnismy otrzymać z mocka:   when(createdMock.foo(1,2)).thenReturn(3);  Wtedy test może przejść.  Inny zapis, zgodny z BDD (behaviour driven development)  given(createdMock.foo(1,2)).willReturn(3);   * Jeżeli metodę wywoływać będziemy do asercji kilka razy to możemy powiedzieć mockito jakiego zwrotu oczekujemy za każdym razem:   // za pierwszym razem: true, drugim razem: false, trzecim razem: true  given(createdMock.foo()).willReturn(true, false, true);   * Rzucanie wyjątków:   given(fooMock().bar).willThrow(RuntimeException.class);   * Obsługa metod które zwracają void:   doNothing().when(Foo).bar();  // lub BDD:  willDoNothing().given(Foo).bar();   * Można to chainować, jeżeli chcemy żeby np było doNothing() przy pierwszy wywołaniu metody, a przy drugim rzcuenie wyjątku:   willDoNothing()  .willThrow(RuntimeException.class)  .given(Foo).bar();   * Używanie argumentów przekazanych do mocka:   doAnswer(invocationOnMock -> {  Object parameter = invocationOnMock.getArgument(0);  // wyrażenie lambda  }).when(Foo).bar(arg1, arg2);  // or  when(Foo.bar(arg1, arg2)).then(i -> {  //wyrażenie lambda jak wyżej  });  // BDD:  willAnswer(invocationOnMock -> {  Object parameter = invocationOnMock.getArgument(0);  // wyrażenie lambda  }).given(Foo).bar(arg1, arg2);  // or  given(Foo.bar(arg1, arg2)).will(i -> {  //wyrażenie lambda jak wyżej  });  invocationOnMock przechowuje wszystkie argumenty przekazane do metody i można pobrać całą ich tablicę, albo tylko konkretny obiekt. |

== ARGUMENT MATCHERS ==

- Jeżeli metoda, którą wywołujemy z mocka przyjmuje jakieś argumenty,

możemy albo wpisać ich wartości na szytwno, albo dopuścić do używania dowolnych wartości użwayjac Argument matcherów

- Nie można mieszać Argument Matcherów z prawdziwymi wartościami

- Przykład:

//mockujemy klasę Foo zawierającą metodę bar:

// int bar(int a, int b){

// return a+b;

// }

// aby sprawić żeby metoda dla każdej wartości int

// zwracała null:

given(fooMock.bar(anyInt(), anyInt())).willReturn(null);

- O KURDE JAKI BAJER Matcher Lambda

zamiast predefiniowanego argument matchera możemy wpisać jakby swój,

używając argThat(/\*wyrażenieLambda\*/) i wyrażenia lambda

np.

//zwróć null kiedy drugi argument jest większy od 5

given(fooMock.bar(anyInt(), argThat(i -> i > 5 ))).willReturn(null);

== ARGUMENT CAPTOR ==

- Służy do przechwycenia i sprawdzenia jaki dokładnie argument został przekazany metodzie.

- Składnia:

ArgumentCaptor<Foo> argumentCaptor = ArgumentCaptor.forClass(Foo.class);

- Przechwytywanie parametrów odbywa się w czasie verify

- W kodzie verify (albo then) używany wtedy zamiast argumentu

stworzony przez nas argumentCaptor.

verify(Bar).barFoo(argumentCaptor.capture());

// lub

then(Bar).should().barFoo(argumentCaptor.capture());

- Dzięki temu mamy w zmiennej argumentCaptor przechwycone to co przekazaliśmy w argumencie metody.

Mozemy się tym teraz posługiwać używając argumentCaptor.getValue() żeby wyciągnąć obiekt z captora, np:

//argumetnCaptor.getValue() zwraca obiekt włącznie z logiką w nim

assertThat(argumentCaptor.getValue().bar(), is(euqalTo(5)));

- Jeżeli metoda przyjmuje więcej niż jeden argument to dla każdego musimy utworzyć oddzielny captor

== VERIFY ==

- Aby sprawdzić czy nasz test użył jakiejś metody ze stworzonego przez nas mocka:

verify(createdMock).foo();

// dodatkowe możliwości

// sprawdzi czy metoda wywoła się conajmniej raz

verify(createdMock, atLeastOnce()).foo();

// sprawdzi czy metoda wywoła się 2 razy

verify(createdMock, atLeast(2)).foo();

// sprawdzi czy metoda nigdy się nie wywołała

verify(createdMock, never()).foo();

// lub

then(createdMock).should(never()).foo();

\* Składnia BDD:

then(createdMock).should().foo();

- Sprawdzenie kolejności wykonania metod:

//sprawdzi czy foo() wykona się przed bar()

InOrder inOrder = inOrder(createdMock);

inOrder.verify(createdMock).foo();

inOrder.verify(createdMock).bar();

- Jeżeli w zamockowanej klasie jest metoda, która zwraca jakiś obiekt, np. kolekcję, mockito postara się nie zwracać null, tylko pusty element danego typu, np. pustą listę, lub 0 dla Integera, false dla boolean.

== SPY ==

\*\* NIE UŻYWA SIĘ ICH NAJCZĘSCIEJ \*\*

- Spy definiuje się tak samo jak Mock tylko że pisze się Spy.

Foo fooSpy = spy(Foo.class);

- Spy zachowuje logikę zawartą w klasie

- Z obiektu Spy cały czas możemy stubowac metody:

// wszystkie metody oprócz bar() będą działać zgodnie z implementacją w klasie Foo.

when(fooSpy.bar()).thenReturn(null);

- Wszystkie metody z klasy fooSpy możemy sprawdzać jak w zwykłym mocku:

verify(fooSpy).bar();

when(fooMock.bar(anyInt(),anyInt())).thenReturn(null);