laboratoria 12—13

Testowanie jednostkowe operacji klas

Cel zajęć: Weryfikacja poprawności działania tworzonego oprogramowania przy pomocy testów jednostkowych.

Źródła wiedzy: Wykład IO: *Testowanie oprogramowania* (dostępny też w *ePortalu Pwr*).
Przykład: *Miniprojekt Biblioteka* (dostępny też w *ePortalu PWr*).
Tutorial do JUnit6: *JUnit User Guide*.
Tutorial do Mockito: *Class Mockito*.

Zawartość instrukcji: 3 zadania – ich wykonanie jest oceniane.
Spis rzeczy, które należy umieścić w sprawozdaniu z wykonania zadań.
Spis błędów i braków mogących obniżyć ocenę.

Zadania

Zadanie 1

Weryfikacja poprawności działania tworzonego oprogramowania przy pomocy testów jednostkowych – testy operacji niezależnych lub zależnych tylko od operacji przetestowanych

Należy sprawdzić poprawność działania wykonanego oprogramowania, wykonując jednostkowe testy działania operacji realizujących bardziej skomplikowane zadania biznesowe bez użycia symulacji („mockowania”). Testy należy wykonać w odpowiedniej kolejności, zaczynając od testów operacji niezależnych, następnie wykonując testy operacji od nich zależnych itd. Kolejność testowania operacji można wywnioskować z diagramów klas i sekwencji, wykonanych na wcześniejszych etapach laboratoriów, ponieważ pokazują zależności między klasami i między operacjami.

Testy należy wykonać w wybranym środowisku IDE przy pomocy odpowiedniej biblioteki (dla języka *Java* zaleca się użycie *JUnit6*).

Każda klasa testów (np. *TestKsiążka.java*) powinna:

– testować operacje klasy, której nazwa jest częścią jej nazwy (np. *Książka.java*);
– określać nazwę klasy testów i nazwę testu (adnotacja *@DisplayName* w *JUnit6*);
– określać kolejność wykonania testów operacji (adnotacje *@Order, @TestMethodOrder* w *JUnit6*);
– opisywać etapy testów komentarzami (*jeśli*, *gdy*, *wtedy*).

Nie ma minimalnej liczby operacji do przetestowania, ale w sumie klasy testów powinny:

– przygotować dane do testu (operacje *setUp() i setUpBeforeClass()* i adnotacje *@BeforeEach, @BeforeAll* w *JUnit6*);
– posprzątać dane po teście (operacje *tearDown() i tearDownAfterClass()* i adnotacje *@AfterEach, @AfterAll* w *JUnit6*);
– wykorzystać przynajmniej 3 różne asercje (*assertEquals(), assertNotEquals()*, *assertIterableEquals()*, *assertSame(), assertNull()*, *assertNotNull()*, *assertTrue()*, *assertFalse()*, *assertThrows()* i inne w *JUnit6*);
– wykorzystać przynajmniej 2 różne sposoby parametryzowania testu, aby wykonać go dla różnych wartości parametrów wejściowych (przy pomocy adnotacji *@ParameterizedTest*, *@Parameter*, *@ValueSource*, *@CsvSource*, *@MethodSource*, *@FieldSource* w *JUnit6*).

Powyższe przykłady operacji i adnotacji nie wyczerpują możliwości *JUnit6*.

Jednostkowy test operacji:

Sprawdza, czy pojedyncza operacja działa zgodnie z oczekiwaniem. Poprawność działania operacji i poprawność jej wyniku sprawdzana jest w izolacji od innych, zwłaszcza nieprzetestowanych części kodu (obiektów, klas, lub operacji).

Test jednostkowy polega zwykle na utworzeniu obiektu testowanej klasy i wywołniu jego testowanej operacji z określonymi parametrami operacji (jeśli je posiada).

Rzeczywisty wynik testu porównuje się z oczekiwanym wynikiem (przy pomocy asercji) w celu wykrycia błędów kodu w wyniku lub efektach ubocznych (np. w zmianie stanu testowanego lub innego obiektu) działania testowanej operacji.

Etapy testu:

1) *Jeśli* (ang. *given*): utworzenie obiektów będących parametrami wejściowymi testowanej operacji i obiektów będących spodziewanym wynikiem tej operacji (do porównania z jej rzeczywistym wynikiem).
2) *Gdy* (ang. *when*): wykonanie testowanej operacji.
3) *Wtedy* (ang. *then*): porównanie rzeczywistego wyniku operacji ze spodziewanym wynikiem – asercje.

Kolejność wykonania jednostkowych testów:

Najpierw testuje się operacje niezależne, następnie operacje zależne od tych już przetestowanych itd. Zależna operacja to taka, która w parametrze lub w ciele zawiera wykonanie innej operacji z tej samej lub innej klasy, albo wykorzystanie atrybutu z tej samej lub innej klasy.

W architekturze n-warstwowej zaczyna się od testowania klas warstwy niezależnej (zwykle jest to warstwa encji), następnie testuje się klasy warstw bezpośrednio od niej zależnych (np. warstwy kontroli), następnie klasy warstw bezpośrednio od nich zależnych itd.

Przykładowa kolejność testów jednostkowych dla warstw encji (*model* w MVC) i kontroli (*kontroler* w MVC):

1) Testy klas modelujących encje danych w warstwie encji.
2) Testy klas modelujących zadania biznesowe encji (np. tworzenie i komponowanie encji danych) w warstwie encji.
3) Testy klas modelujących elementarne usługi biznesowe (np. krok realizacji przypadku użycia) w warstwie kontroli.
4) Testy klas modelujących złożone usługi biznesowe (np. realizację całego przypadku użycia) w warstwie kontroli.

Zadanie 2

Weryfikacja poprawności działania tworzonego oprogramowania przy pomocy testów jednostkowych – symulacja obiektów i operacji, od których zależy testowana operacja

Należy sprawdzić poprawność działania wykonanego oprogramowania, wykonując jednostkowe testy działania operacji realizujących bardziej skomplikowane zadania biznesowe z użyciem symulacji („mockowania”). Symulować należy te fragmenty kondu (obiekty, operacje), od których zależy testowana operacja. Te zależności można wywnioskować z diagramów klas i sekwencji, wykonanych na wcześniejszych etapach laboratoriów.

Testy należy wykonać zgodnie z wymaganiami zadania 1, w wybranym środowisku IDE przy pomocy odpowiednich bibliotek (dla języka *Java* zaleca się użycie *JUnit6* i *Mockito*).

Nie ma minimalnej liczby operacji do przetestowania, ale w sumie klasy testów powinny:

– tworzyć symulacje obiektu lub klasy (operacja *mock()* lub adnotacja *@Mock* oraz operacja *MockitoAnnotations.initMocks()* w *Mockito*);

– wstrzykiwać do symulacji inne symulacje, od których jest zależna (adnotacja *@InjectMocks* w *Mockito*);

– określać zachowanie symulacji operacji coś zwracających: co zwraca (operacja *when().thenReturn()* w *Mockito*) lub jaki wyjątek wyrzuca (operacja *when().thenThrow()* w *Mockito*);

– określać zachowanie symulacji operacji niczego niezwracających (void): co zwraca (operacja *doReturn().when()* w *Mockito*) lub jaki wyjątek wyrzuca (operacja *doThrow().when()* w *Mockito*) i innne (operacje *doNothing().when()* i *doCallRealMethod().when()* w *Mockito*).

Ponadto klasy testów mogą:

– określać kolejność użycia symulacji (klasa *inOrder* w *Mockito*);

– sprawdzać, czy, ile razy i z jakimi parametrami symulację użyto w teście (operacje *verify(), times(), never(), atLeast(), atLeastOnce(), atMost(), atMostOnce()* w *Mockito*).

Powyższe przykłady operacji i adnotacji nie wyczerpują możliwości *Mockito*.

Symulacja / atrapa / imitacja zależności testowanej operacji:

Jeśli izolacja testowanej operacji nie jest możliwa, a testowana operacja korzysta z działania lub wyniku innej, nieprzetestowanej, niedostępnej lub nieprzewidywalnie zachowującej się części kodu (obiektu, klasy, operacji lub jej wyniku), to tę inną część kodu lub jego zachowanie się symuluje, czyli imituje („mockuje”).

Test wykorzystujący symulację nie gwarantuje poprawności działania testowanej operacji w przypadku zastosowania rzeczywistej zależnej części kodu.

Symulacja w etapach testu:

1) *Jeśli* (ang. *given*): utworzenie i określenie użycia i zachowania symulacji (atrap).
2) *Gdy* (ang. *when*): wykonanie testowanej operacji z użyciem symulacji (atrapy).
3) *Wtedy* (ang. *then*): sprawdzenie użycia symulacji (atrapy) przed użyciem asercji.

Zadanie 3

Weryfikacja poprawności działania tworzonego oprogramowania przy pomocy testów jednostkowych – zestawy testów

Należy wykonać zestawy testów (pliki *.java* w przypadku użycia *JUnit6*) wykonanych w poprzednich zadaniach:

– zestaw testów klas warstwy encji, czyli znajdujących się w jej pakiecie;

– zestaw testów klas warstwy kontroli, czyli znajdujących się w jej pakiecie;

– przynajmniej 2 zestawy testów oznaczonych wybranymi tagami i nie oznaczonych innymi wybranymi tagami, które mają jakieś praktyczne zastosowanie jako taki zestaw.

Testy należy wykonać w wybranym środowisku IDE przy pomocy odpowiedniej biblioteki (dla języka *Java* zaleca się użycie *JUnit6*).

Nazwę klasy zestawu testów należy rozpocząć od słowa *Suite*.

W klasach testów należy testy użyte w zestawach oznaczyć odpowiednimi tagami opisującymi je tematycznie (adnotacja *@Tag()* w *JUnit6*).

W klasie zestawu testów należy:

– określić nazwę testu (adnotacja *@SuiteDisplayName* w *JUnit6*);

– określić zestaw testów (adnotacja *@Suite* w *JUnit6*);

– wybrać testy do zestawu na podstawie ich tagów (adnotacje *@IncludeTags*, *@ExcludeTags* w *JUnit6*);

– wybrać testy do zestawu na podstawie nazw lub pakietów testowanych klas (adnotacje *@SelectClasses*, *@SelectPackages, @IncludePackages, @ExcludePackages* w *JUnit6*).

Powyższe przykłady adnotacji nie wyczerpują możliwości *JUnit6*.

Sprawozdanie

Co powinno być na początku sprawozdania:

– Autorzy (imię, nazwisko, nr albumu).

– Nazwa tworzonego oprogramowania (zwięzły opis jego dziedziny).

– Temat tego etapu laboratoriów (*Testowanie jednostkowe operacji klas*).

Co powinno być w treści sprawozdania:

– Pełny kod klas testów i klas zestawów testów (tekst, a nie obraz ze zdjęciem ekranu), zawierający:
  - klasy w kolejności ich w pełni kwalifikowanych nazw (czyli wraz ze ścieżką pakietów, np. pakiet.klasa.java),
  - nazwę pliku z definicją klasy (nad tą definicją),
  - całą zawartość tego pliku,
  - numerację linii,
  - komentarze do etapów testów,
  - komentarze do mało oczywistych rzeczy,
  - wcięcia i inne formatowania ułatwiające czytelność.

Błędy i braki

Jeśli opisane niżej rzeczy mają miejsce, ocena za ten etap laboratoriów może być niższa:

– Testowana operacja nie jest zależna od innych i jest tak prosta, że bez testu można łatwo stwierdzić prawidłowość jej działania.

– Nazwa klasy testu nie zawiera nazwy testowanej klasy lub słowa Test.

– Brak jednoznacznie określonej kolejności wykonywania testów w klasie testu.

– Nie przygotowano wszystkich danych potrzebnych do wykonania testu.

– Nie przygotowano ponownie wszystkich danych potrzebnych do kolejnego wykonania testu.

– Wykorzystano mniej niż 3 różne asercje.

– Wykorzystano mniej niż 2 różne sposoby parametryzacji testu.

– Nie utworzono symulacji obiektu lub operacji – fragmentu kodu, od którego testowana operacja jest zależna.
– Symulacja obiektu zależnego od innego obiektu nie ma wstrzykniętej symulacji tego drugiego obiektu.
– Nie określono zachowania symulacji – skutku wykonania operacji symulowanego obiektu.
– Operacja testowana w zadaniu 1 jest zależna od jeszcze nieprzetestowanego fragmentu kodu.
– Operacja testowana w zadaniu 2 jest zależna od jeszcze nieprzetestowanego fragmentu kodu, który nie został zasymulowany, lub którego symulacja jest niepełna.
– Brak zestawu testów klas warstwy encji lub zestaw ten nie bazuje na pakiecie tych klas.
– Brak zestawu testów klas warstwy kontroli lub zestaw ten nie bazuje na pakiecie tych klas.
– Brak zestawu testów oznaczonych wybranymi tagami i nieoznaczonych innymi tagami.
– Zestaw testów bazujący na ich tagach nie ma praktycznego zastosowania.
– Kod nie jest sformatowany lub jest niepraktycznie sformatowany, np. brak komentarzy, wcięć czy numeracji linii.
– Brak komentarzy określających etapy testu (*jeśli*, *gdy*, *wtedy*).

Ta lista zawiera typowo pojawiające się błędy i braki i nie wyczerpuje powodów, dla których ocena za laboratoria może być niższa.