

Testowanie – Testy jednostkowe

Wstęp

W trakcie zajęć zapoznasz się z koncepcją testów jednostkowych, narzędziem JUnit przeznaczonym do ich tworzenia oraz zastosujesz kilka podstawowych podejść doboru odpowiednich przypadków testowych.

JUnit to narzędzie umożliwiające automatyczne testowanie jednostkowe kodu napisanego w Javie (istnieją również odpowiedniki dla innych języków, np. Microsoft Tests i NUnit dla C#, czy CppUnit dla C++). Sposób pracy z JUnitem będzie pokazany na przykładzie środowiska programistycznego Eclipse, z którym JUnit bardzo dobrze się integruje.

Testowanie jednostkowe

Pierwsza grupa zadań ma na celu przedstawienie podstawowych mechanizmów i koncepcji dotyczących tworzenia i uruchamiania testów jednostkowych.

Zadanie 1. Mój pierwszy test jednostkowy

Klasa `Calculator` zawiera m.in. metody `add` i `multiply`, które służą do obliczania sumy i iloczynu dwóch liczb całkowitych. Twoim pierwszym zadaniem będzie napisanie testów dla tych metod.

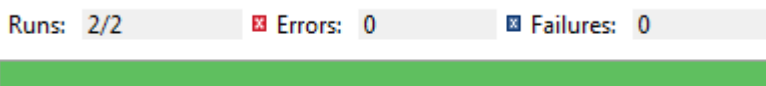
1. Zaimportuj do Eclipse'a projekty, które znajdują się w archiwum *Projekty.zip*

W pierwszej części zajęć będziesz operował w obrębie projektu *01-TestowanieJednostkowe*.

2. Dodaj do projektu folder źródłowy *test* (**File > New > Source folder**)
3. Dodaj klasę testującą: zaznacz klasę `put.io.testing.Calculator` i wybierz komendę **File > New > JUnit Test case**
 - a. W pierwszym oknie kreatora wybierz opcję **New JUnit 4 test**, zmień **Source folder** na *test* i upewnij się, że **Class under test** to `Calculator`. Kliknij **Next**.
 - b. Zaznacz wszystkie metody, które dostarcza klasa `Calculator`
 - c. Eclipse zapyta Cię czy dodać JUnit 4 do ścieżki budowania. Zgódź się.
4. Napisz metody testujące dla metody `add` i `multiply`. W każdej z nich zweryfikuj przy pomocy asercji wyniki przynajmniej dwóch wywołań danej metody, podając różne zestawy argumentów.
5. Uruchom przed chwilą napisane testy: zaznacz klasę testującą, wybierz komendę **Run > Run as > JUnit Test**.

Czy Twoje testy wykryły jakieś błędy w kodzie klasy `Calculator`? Jeśli nie, postaraj się bardziej i dodaj kolejne asercje – te błędy naprawdę tam są.

6. Napraw kod błędnej metody
7. Uruchom testy ponownie



Wszystko świeci się na zielono? Brawo! Możesz przejść do kolejnego zadania.

Zadanie 2. Oczekiwanie na wyjątek

Trzecia metoda klasy `Calculator`, `addPositiveNumbers`, przyjmuje jako argumenty dwie liczby i zwraca ich sumę, ale tylko wtedy, kiedy ich wartości są większe lub równe 0. W przeciwnym wypadku metoda rzuca wyjątek `IllegalArgumentException`. W trakcie tego zadania nauczysz się jak wyrazić

w teście, że oczekujesz od testowanego kodu by w konkretnej sytuacji rzucił właściwy wyjątek.

1. Dodaj w odpowiedniej metodzie testującej wywołanie metody `addPositiveNumbers` dla argumentów, takich że $x < 0$ i $y > 0$.
2. Uruchom testy. Jaki jest wynik dla przed chwilą dodanego testu?
3. Dodaj parametr `expected=IllegalArgumentException.class` do adnotacji `@Test` przypisanej do rozpatrywanej metody testującej.
4. Uruchom ponownie wszystkie testy

Jeżeli wszystkie testy przechodzą – udało Ci się!

Zadanie 3. Oczekiwanie na wiele wyjątków

W poprzednim zadaniu utworzyłeś metodę testującą, która oczekiwała na wyjątek spowodowany wywołaniem metody `addPositiveNumbers` z argumentami, z których pierwszy był liczbą ujemną. By w pełni przetestować tę metodę należałoby rozważyć również i inne przypadki, tj. gdy tylko drugi argument jest liczbą ujemną oraz gdy oba argumenty są mniejsze od 0. Jednym ze sposobów na realizację tego zadania jest dodanie dwóch kolejnych metod testujących, zaimplementowanych w analogiczny sposób, jak w poprzednim zadaniu – z wykorzystaniem parametru `expected`. Drugi ze sposobów pozwala na to, by całość zawrzeć w jednej metodzie testującej – wewnątrz znajdują się wszystkie 3 wywołania metody `addPositiveNumbers`. Tylko jak to zrobić?

Pytanie 3.1. W jaki sposób oczekiwać na wiele różnych wyjątków w jednej metodzie testującej?

W razie problemów ze znalezieniem rozwiązania, prowadzący dysponuje wskazówką.

1. Zastosuj wymyślony sposób tak by sprawdzić metodę `addPositiveNumbers` dla 3 wyżej opisanych kombinacji
2. Uruchom testy i sprawdź czy przebiegły zgodnie z planem



Zadanie 4. Przygotowanie do testów i sprzątanie po

Poprzednie testy były dość proste – w każdym teście tworzyłeś obiekt testowany, uruchamiałeś testowaną metodę i przy pomocy asercji sprawdzałeś czy zwraca odpowiedni wynik. Może się jednak okazać, że proces tworzenia i konfiguracji obiektu jest bardziej skomplikowany, ale za to identyczny dla każdego przypadku testowego. Zamiast powielać kod w każdej metodzie lepiej „wypchnąć” go do osobnej metody, która będzie uruchamiana przed każdym wywołaniem metody testującej.

1. Dodaj do klasy testującej pole prywatne `Calculator calculator`
2. Dodaj metodę, którą możesz nazwać np. `setUp`. Oznacz ją taką adnotacją, by JUnit uruchamiał ją przed każdym testem. Dodaj kod, który przypisze polu `calculator` nową instancję klasy `Calculator`.
3. Zmodyfikuj wszystkie metody testujące, by zamiast samodzielnie tworzyć obiekt klasy `Calculator`, korzystały z pola `calculator`

Pytanie 4.1. Czy testy przestałyby działać, gdyby zmieniono adnotację przy metodzie `setUp` z `Before` na `BeforeClass`? Uzasadnij swoją odpowiedź.

Zadanie 5. Failure kontra Error

JUnit zlicza niepowodzenia testów i przedstawia je jako dwie liczby: **Errors**  i **Failures** . Czy potrafisz odpowiedzieć „z pamięci”, kiedy niepowodzenie testu jest traktowane jako Error, a kiedy jako Failure? Zamiast zaglądać do wykładu sprawdź to w praktyce!

1. Dodaj nową klasę testującą: **File > New > JUnit Test case**, nazwij ją `FailureOrErrorTest`, zmień **Source folder** na test i kliknij **Finish**.
2. Dodaj dwie metody testujące: `test1` i `test2`.
3. W metodzie `test1` wpisz asercję, dla której zdefiniowany warunek zawsze będzie fałszywy
4. W metodzie `test2` wpisz instrukcję, która rzuca dowolny wyjątek
5. Uruchom testy by znaleźć odpowiedź na niżej postawione pytanie

Pytanie 5.1. Która metoda zostanie oznaczona jako **Failure**, a która jako **Error**?

Czy zastanawiałeś się w jaki sposób JUnit dowiaduje się o tym, że test nie przeszedł? Zobacz co by się stało jakbyś przechwycił w teście „wszystko co da się rzucać” (w tym stwierdzeniu zawarta jest odpowiedź).

1. Dodaj metodę `test3` do klasy testującej
2. Dodaj block try-catch. W sekcji `try` umieść asercję, która zawsze jest fałszywa. W sekcji `catch` spróbuj wyświetlić `stackTrace` dla przechwyconego obiektu.
3. Uruchom testy i sprawdź co to za obiekt

Pytanie 5.2. Na jaki typ obiektu rzucanego oczekuje JUnit by stwierdzić, że test się nie powiódł w sensie kategorii Failure (będzie oznaczony na granatowo).

Jak nie masz pomysłu na rozwiązanie zapytaj prowadzącego, na pewno podrzuci Ci jakąś wskazówkę.

Zadanie 6. Tworzenie zestawu testów

W tej chwili w folderze test masz dwie klasy testujące. Uruchamianie każdej z nich z osobna jest kłopotliwe, a co by było gdyby tych klas było znacznie więcej? W tej sytuacji z pomocą przychodzą zestawy testów, tzw. Suite case.

1. Dodaj folderu test zestaw testów: **File > New > Other > Java > JUnit > JUnit Test Suite**
 - a. Upewnij się, że pole **Package** wskazuje na `put.io.testing`
 - b. Zaznacz obie klasy testujące
 - c. Kliknij **Finish**
2. Uruchom nowopowstały zestaw testów

Jak wszystko działa i nie masz żadnych wątpliwości co do zadań do tej pory wykonanych, możesz spokojnie przejść do drugiej części dzisiejszych laboratoriów.

Strategie testowania

Teraz już wiesz jak tworzyć testy jednostkowe. Zdrowy rozsądek na pewno podpowiada Ci, że w większości przypadków nie ma możliwości całkowitego przetestowania systemu – możliwych kombinacji wartości argumentów i oczekiwanych wyników jest zbyt wiele, by je wszystkie sprawdzić. Tworząc testy musisz więc wybrać te przypadki, które najskuteczniej testują daną metodę. W trakcie kolejnych zadań poznasz 3 proste strategie doboru przypadków testowych.

Zadanie 7. Analiza ścieżek

Projekt 2-Strategie Testowania zawiera klasę `AnotherCalculator`. Twoim zadaniem jest napisanie testów dla metody `calculate` wykorzystując podejście analizy ścieżek działania programu. Zanim przejdziemy do pisania testów, proste pytanie:

Pytanie 7.1. Jaki to typ testowania: blackbox czy whitebox?

Testowanie przy pomocy analizy ścieżek zostało krótko opisane w [Wikipedii](#). W skrócie, powinieneś stworzyć tyle testów ile jest możliwych ścieżek w metodzie `calculate`. Dla uproszczenia załóż, że dla pętli istnieją dwie możliwe ścieżki: 1) warunek pętli jest niespełniony, pętla nie jest wykonana 2) warunek pętli jest spełniony i pętla zostanie wykonana dokładnie raz.

Pytanie 7.2. Ile możliwych ścieżek działania znajduje się w metodzie `calculate`, zakładając że punkt startowy to początek metody, a punkt końcowy to jej koniec? (Zanim przejdziesz do kolejnego kroku zadania, skonsultuj odpowiedź z prowadzącym.)

1. Dodaj ***JUnit Test Case*** dla klasy `AnotherCalculator`, nazywając go `AnotherCaclulatorTest` i umieszczając w folderze `test`
2. Dodaj po jednym przypadku testowym dla każdej z możliwych ścieżek metody `calculate`
3. Uruchom testy, powinny przechodzić

Jeżeli wszystkie Twoje testy kończą się sukcesem przejdź do kolejnego zadania.

Zadanie 8. Naprawa błędów regresyjnych

Testy jednostkowe doskonale nadają się do wykrywania błędów regresyjnych, czyli takich, które powstały omyłkowo w skutek zmian w kodzie, np. przy dodawaniu nowej funkcji do systemu.

Przerwa na ważny komunikat! Właśnie dostałeś maila od Jenkinsa, który powiadamia Cię, że podczas ostatniego nocnego builda testy zakończyły się porażką. Jak to możliwe, myślisz, wczoraj działało. Zamiast się zastanawiać lepiej sprawdź czy ktoś nie „grzebał” w kodzie.

1. Otwórz linię poleceń w folderze projektu *2-StrategieTestowania*
2. Wpisz polecenie `hg update Zadanie-8`. Czy pojawił się jakiś nowy kod?
3. Odśwież projekt w Eclipse (menu kontekstowe na projekcie)
4. Uruchom testy i zobacz ile z nich kończy się porażką. Na podstawie informacji o tym, które testy się nie powiodły spróbuj zlokalizować błędy w kodzie.
5. Napraw znalezione błędy

Dzięki testom regresyjnym niemal od razu dowiedziałeś się, że ktoś popsuł dobrze działający kod i dzięki Twojemu refleksowi udało Ci się w porę je poprawić. Gratulacje!

Zadanie 9. Analiza klas równoważności (blackbox)

Projekt *2-StrategieTestowania* ma załączoną bibliotekę `TaxCalculator`. W środku znajduje się klasa `put.io.testing.blackbox.TaxCalculator`, która ma jedną publiczną metodę `calculate`. Twoim zadaniem jest przetestować tę metodę, nie mając dostępu do kodu źródłowego (testowanie typu blackbox).

Metoda `calculate` oblicza wartość podatku, który trzeba oddać fiskusowi, naliczany od wartości pensji, podanej jako parametr `amount`. Zasady obliczania podatku są następujące:

- kwota wolna od podatku wynosi 3000. Oznacza to, że podatek jest obliczany tylko od części zarobków przekraczających tę wartość (`amount - 3000`). Jeżeli, podatnik ma mniejsze zarobki lub równe niż kwota wolna od podatku, nie płaci podatku wcale.
- do obliczenia podatku wykorzystana jest skala podatkowa składająca się z dwóch progów 20.000 oraz 100.000. Część pensji nieprzekraczająca pierwszego progu jest opodatkowana stawką 10%, część nieprzekraczająca drugiego – 20%, a pozostała część 15%.

Zadania do wykonania

1. Zaproponuj i zapisz na kartce klasy równoważności dla danych wejściowych.
2. Dodaj **JUnit Test Case** dla klasy `TaxCalculator`, nazywając go `EquivalenceTaxCalculatorTest` i umieszczając w folderze `test`. W kreatorze tworzenia, przy polu **Class under test** znajduje się przycisk **Browse ...**. Po jego kliknięciu pojawi się nowe okno, na początku puste. Zaczynij pisać `TaxCalculator`, pojawi się poszukiwana klasa. Wybierz ją. Zakończ działanie kreatora.
3. Dla każdej wyznaczonej klasy równoważności stwórz osobną metodę testującą. Zadbaj o poprawne, spójne i czytelne nazewnictwo tych metod.

Funkcja `calculate` operuje na kwotach trzymanyh w obiekcie `BigDecimal`. Asercje, które zaraz zaczniesz tworzyć, najlepiej by przyjęłyby postać:

```
BigDecimal amount = new BigDecimal("<amount>");
BigDecimal expected = new BigDecimal("<expected>");
BigDecimal result = calculator.calculate(amount);
assertTrue(result.compareTo(expected) == 0);
```

4. Dodaj po minimum jednej asercji do każdej metody testującej
5. Uruchom testy i zobacz czy są błędy

Hmm, znów ktoś psocił w kodzie. Pobierz najnowszą wersję z repozytorium i zobacz czy coś nie jest zepsute.

1. Wpisz polecenie `hg update Zadanie-10`
2. Odśwież projekt w Eclipse (menu kaskadowe na projekcie)
3. Uruchom testy i zobacz ile z nich kończy się porażką

Całkiem możliwe, że w 3 punkcie nie znajdziesz błędów w kodzie. Po zaktualizowaniu wersji roboczej projektu pojawił się w nim nowy plik `OriginalTaxCalculator.java`. Znajduje się w nim pierwotna implementacja testowanej przez Ciebie klasy (ta wersja, która nie miała posianych błędów). Spójrz na kod i zastanów się na ile skuteczna w tym przypadku była strategia analizy klas równoważności.

Błędy, które ktoś „umieścił” niechący w najnowszej wersji modyfikują stałe wartości progów. W kolejnym zadaniu przećwiczysz strategię, która koncentruje się na analizie wartości danych na brzegach klas równoważności.

Zadanie 10. Analiza wartości brzegowych (blackbox)

W tym zadaniu dodasz przypadki testowe dla tej samej klasy `TaxCalculator`, ale wykorzystując analizę wartości brzegowych.

1. Zaproponuj i zapisz na kartce możliwe wartości brzegowe (po 3 wartości dla każdej granicy przedziału: wartość graniczna, wartość o jeden grosz mniejsza i wartość o jeden grosz większa)
2. Dodaj **JUnit Test Case** dla klasy `TaxCalculator`, nazywając go `BoundaryTaxCalculatorTest`. Reszta ustawień kreatora będzie taka sama jak w poprzednim zadaniu.
3. Dla każdej granicy przedziału stwórz osobną metodę testującą. Dodaj po 3 asercje testujące wynik metody dla wartości granicznej. Ponownie, zadbaj o poprawne, spójne i czytelne nazewnictwo tych metod.

4. Uruchom testy i zobacz czy wykryły błędy

Zastanów się czy najprostszy sposób wyznaczenia wartości brzegowych jest wystarczający dla tego zadania, tj. $b - \Delta$, b , $b + \Delta$. Aby poprawnie przetestować klasę `TaxCalculator` należy wziąć pod uwagę precyzję operacji i zaokrąglenia do pełnych groszy. Wywołanie `calculate(new BigDecimal("5000.00"))` zwróci ten sam wynik co `calculate(new BigDecimal("5000.05"))`.