3. gyakorlat: Fejlesztői tesztelés

Egyszerű tesztelési mintapélda

A gyakorlat első részében egy egyszerű számológépet megvalósító programot fogunk vizsgálni. A számológép egész számokkal képes műveleteket végezni, és memória funkcióval is rendelkezik.

A számológépnek van egy komplex művelete is:

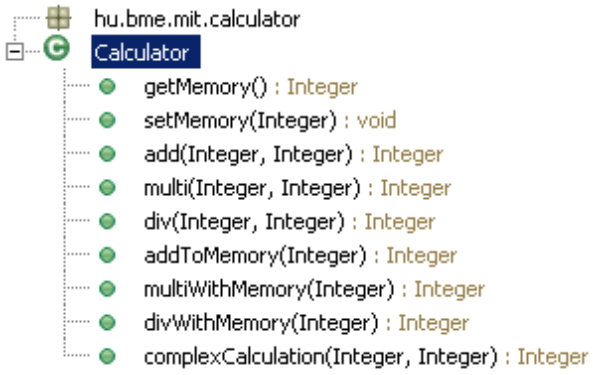
```
public Integer complexCalculation(Integer a, Integer b)
```

A művelet működése a következő. Az **a** paraméternek 10 feletti számnak kell lennie. Ha 100-nál kisebb, akkor szorozni kell **b** abszolút értékével, egyébként meg hozzáadni azt. Ha **b** negatív szám, akkor az egész eredményt még negálni kell, és azt kell visszaadni.

1. Határozzuk meg a paraméterek ekvivalencia partícióit, és tervezzünk ezek alapján teszteseteket a művelethez.
2. Határérték analízis segítségével egészítsük ki az előbb elkészült tesztkészletet.

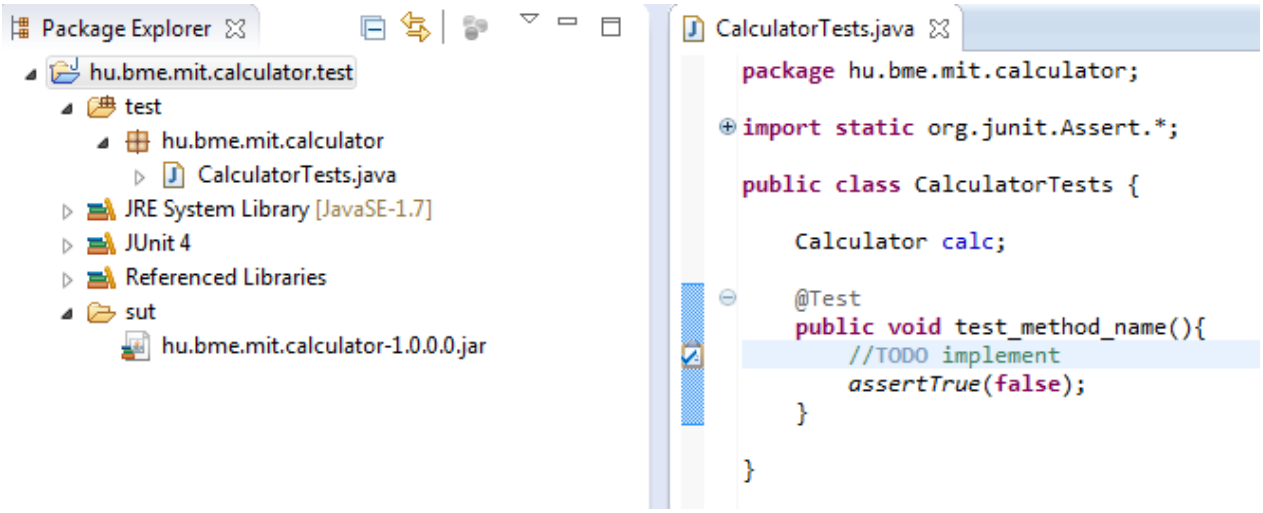
Tesztek implementálása és végrehajtása

Az előző feladat programjához elkészült a következő Java nyelvű implementáció.



1. ábra: A Calculator osztály

Implementáljuk az előző feladatban specifikált teszteseteket. A tesztek lefuttatásához a JUnit[1] keretrendszert fogjuk használni. A következő projekt struktúra áll a rendelkezésünkre.



2. ábra: A Calculator teszt projekt váza

- A *test* könyvtár alá kerülnek az elkészítendő tesztek. Ezekhez egy alap váz már elkészült a CalculatorTests.java fájlban.
- A *sut* könyvtárban van a *System Under Test,* jelen esetben egy jar fájl formájában.

Elvégzendő feladatok:

1. Implementáljunk kettőt a definiált teszteseteinkből.

[1] Bevezető leírás: http://www.vogella.de/articles/JUnit/article.html

2. Mindegyik teszteset inicializálása hasonló, ezért a közös részt mozgassuk át egy közös `setUp` részbe (`@Before` annotáció használata).
3. A tesztek még így is csak a bemeneti paraméterekben és az elvárt eredményben térnek el. Ehhez felesleges mindegyik tesztesethez külön-külön metódust készíteni, használjunk helyette inkább parametrizált tesztet[2]. Adjuk meg a parametrizált tesztnek a maradék definiált tesztesetet.
    a. A 4.11-esnél újabb JUnit esetén el lehet nevezni az egyes eseteket is, használjuk ki ezt a funkciót is[3].
4. Annál a tesztesetnél, ahol az elvárt működés az, hogy hibát ad vissza a program, használjunk `@Test(expected = Exception.class)` típusú annotációt.
5. A felhasználók panaszkodnak arra, hogy az összeadás művelet nagyon lassú. Megfelelő időkorlát kiválasztásával és a `@Test(timeout = <ertek>)` annotáció segítségével készítsünk egy olyan tesztet, ami ezt vizsgálja.
6. Készítsünk egy tesztkészletet (test suite), ami magában foglalja az összes, eddig elkészített tesztesetet.

[2] Lásd például: http://www.ibm.com/developerworks/java/tutorials/j-junit4/section6.html
[3] Lásd például: http://stackoverflow.com/questions/650894/changing-names-of-parameterized-tests

Modul izolációs tesztelés

Modul/unit tesztelés esetén az egyik legfontosabb feladat annak megoldása, hogy a modul tényleg izoláltan fusson, és az összes függőségét valahogy helyettesítsük. Az irányítható és megfigyelhető csonkok kézi elkészítése helyett most a *Mockito*[4] nevű keretrendszert fogjuk használni ezek automatikus generálására.

Mockito bemutatása

A *Mockito* mind stubok és mock objektumok használatát lehetővé teszi.

- Állapot ellenőrzése:

```
// create a test double (there is no separate stub() call)
List mockedList = mock(List.class);

// specify how the test double should respond to calls from the SUT
when( mockedList.get(0) ).thenReturn( "first" );

// call the SUT as in any test
String r = sut.queryList(mockedList, 0);

// assert the state or the return value of the SUT to decide the outcome of the test
assertEquals("first", r);
```

- Interakciók ellenőrzése:

```
// create a test double, it automatically records all the calls received
List mockedList = mock(List.class);
sut.setList(mockedList);

// call the SUT as in any test
sut.add(0);

// verify the mock and not the SUT
// check that the SUT sent the required interactions
verify(mockedList).add(0);
```

A fenti kód lefordulásához még importálni kell a Mockito statikus metódusait:

```
import static org.mockito.Mockito.*;
```

A Mockito használata során a kulcslépés egy általános *test double* létrehozása a `mock` hívással (pl. `List mockedList = mock(List.class)`). Lehet interfészt vagy egy konkrét osztályt is „mockolni”. Ennek eredményeképp:

- a `mockedList`-en hívható a `List` összes metódusa,
- minden külön nem definiált hívásra valami alapértéket ad vissza, pl. null, 0, üres lista,
- a `mockedList`-nek küldött hívásokat eltárolja, később azok ellenőrizhetőek.

[4] Mockito weboldal: http://mockito.org/

A fenti alapokon kívül még a következő funkciók lehetnek hasznosak:

- *Argument matcher*: ha nem equals egyezést akarunk a paramétereknél (lásd a `Matchers` osztály metódusait)

```
when(mockedList.get(anyInt())).thenReturn("element");
```

- *ArgumentCaptor*: ellenőrzés során lehetséges a mock-nak átadott paraméter elkapása
- Adott számú meghívás ellenőrzése (`atLeast(2)`, `atMost(5)`, `never()` is használható)

```
verify(mockedList, times(2)).add("twice");
```

- Kivétel dobása

```
when(mockedList.get(-1)).thenThrow(new Exception());
```

- when szabályok felüldefiniálhatják egymást (legutolsó számít)

További leírást lásd a Mockito dokumentációjában:

http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html

Feladatok

Adott egy űrhajók működését modellező kódváz (`C:\code\GYAK3\Spaceship` workspace, `hu.bme.mit.spaceship` projekt). A feladat egy konkrét hajó torpedókilövő rutinjának a modul szintű tesztelése. A metódus definíciója a következő:

```
/**
* Tries to fire the torpedo stores of the ship.
*
* @param firingMode how many torpedo bays to fire
* SINGLE: fires only one of the bays.
*  - For the first time the primary store is fired.
*  - To give some cooling time to the torpedo stores,
     torpedo stores are fired alternating.
*  - But if the store next in line is empty the ship
     tries to fire the other store.
*  - If the fired store reports a failure, the ship
     does not try to fire the other one.
* ALL: tries to fire both of the torpedo stores.
*
* @return whether at least one torpedo was fired successfully
*/
public boolean fireTorpedos(FiringMode firingMode)
```

A unit teszteléshez az adott modult (`GT4500` osztály) izoláltan kell tesztelni:

- Azonosítsuk, hogy milyen függőségei vannak a modulnak.
- Tudjuk-e minden függőségét befolyásolni a tesztek során, könnyen vezérelhető-e a tesztelendő modul? Ha nem, milyen megoldást javasolunk?

- Készítsünk a modul leírása alapján unit teszteket a `fireTorpedos` metódushoz, a függőségek helyettesítéséhez használjuk a Mockito keretrendszert.
- A tesztekben próbáljuk végiggondolni, hogy az állapot vagy interakciók ellenőrzése a célszerűbb, és használjuk a megfelelő módszert.

Olvasnivaló

A Mockito hatékonyabb használatához érdemes átolvasni a következő blog bejegyzést és a folytatásait.

[1] Holger Staudacher. „Effective Mockito Part 1”, EclipseSource blog, 2011. URL: http://eclipsesource.com/blogs/2011/09/19/effective-mockito-part-1/