4. gyakorlat: Specifikáció alapú tesztelés

Kombinatorikus teszttervezés

Az első feladatban a kombinatorikus teszttervezés módszerét nézzük meg a gyakorlatban *n* paraméter t-szeres lefedettségét (t-wise coverage) garantáló tesztkészlet előállításával.

Adott a következő tesztelendő metódus:

```
int calculateEngineInput( GearMode g, bool economyMode, int acceleration )
```

ahol a GearMode a {Backward, None, Parking, Drive} halmazból veheti az értékét, az acceleration pedig egy 0-3 közötti szám.

A paraméterek kombinációját akarjuk vizsgálni, de nem akarjuk az összes lehetséges 32 kombinációt végigpróbálni. Először megelégszünk azzal, ha tetszőleges két paraméter esetén az összes lehetséges kombináció legalább egyszer szerepel (2-wise vagy pair-wise coverage).

T-szeres lefedettség számolásához az ACTS[1] eszközt fogjuk használni.

1. Hány tesztet kéne generálni a pair-wise lefedettség teljesítéséhez? Hogyan állnánk neki egy ilyen minimális elemszámú tesztkészlet kézi előállításának?
2. Vegyük fel a paraméterek fenti rendszerét az eszközben, és generáltassunk hozzá 2-szeres fedést biztosító tesztkészletet! Hány tesztesetünk lett?
3. A fenti metódus finomítása során kiderül, hogy nem minden bemeneti kombináció lehetséges, Backward esetén nem lehet az economyMode értéke igaz. Vegyük fel ezt a kényszert az ACTS-be, és generáljunk új tesztkészletet. Hogyan módosultak a tesztek?
4. A rendszer fejlesztése során bekerült egy új BreakStatus paraméter {Pressing, Releasing, None} értékkészlettel. Hogyan változik ekkor a tesztkészletünk?
5. Állítsuk át, hogy 3-szoros paraméter lefedettséget generáljon az eszköz, és készítsünk így is egy tesztkészletet.

(Az eszköz előnye igazából nagyobb méretű paramétertérnél jön ki még inkább, érdemes megnézni a hivatalos oldalon lévő példát, ott azért van olyan eszköz, ahol a generálás órákig tart.)

[1] http://csrc.nist.gov/groups/SNS/acts/index.html

Specifikáció alapú teszttervezés mintapélda

A gyakorlat további 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:

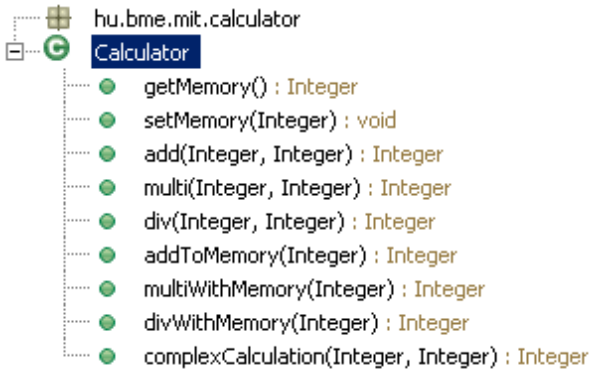
```
public Integer complexCalculation(Integer a, Integer b)
```

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.

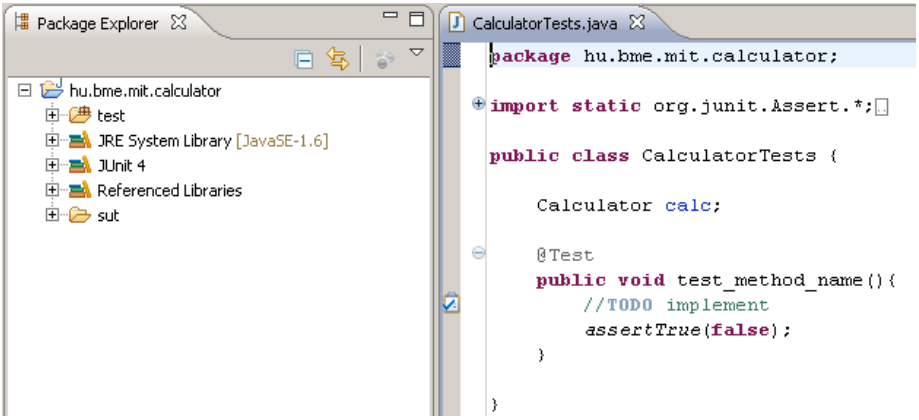
1. Határozzuk meg a paraméterek ekvivalencia partícióit, és tervezzünk ezek alapján teszteseteket a művelethez.
2. Határérték analízis segítségével egészítsük ki az előbb elkészült tesztkészletet.

Tesztek implementálása és végrehajtása

Az előző feladat programjához elkészült a következő Java nyelvű implementáció.



Implementáljuk az előző feladatban specifikált teszteseteket. A tesztek lefuttatásához a JUnit[2] keretrendszert fogjuk használni. A következő projekt struktúra áll a rendelkezésünkre.



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

Elvégzendő feladatok:

1. Implementáljunk kettőt a definiált teszteseteinkből.
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,

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

használjunk helyette inkább parametrizált tesztet[3]. Adjuk meg a parametrizált tesztnek a maradék definiált tesztesetet.

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.

[3] Lásd például: http://www.ibm.com/developerworks/java/tutorials/j-junit4/section6.html