

GEIAL31H-BL Szoftvertesztelés

Beadandó Feladat



JUnit

Készítette: Kolláth Zoltán

Neptun kód: PELL8A

Kelt: 2020.11.30.

Feladat leírása

A feladat kiírása alapján egységteszt kerül bemutatása java környezetben.

Szoftvertesztelés

A szoftvertesztelés segítségével megbizonyosodhatunk arról, hogy a szoftver termékünk az elvárásoknak megfelelően működik. Megtaláljuk a hibákat, ezeket által növeljük a minőségét és a megbízhatóságát. Nincs tökéletes szoftver, emberek fejlesztenek és az emberek hibázhatnak.

Tesztelési technikák

„A tesztelési technikákat csoportosíthatjuk a szerint, hogy a teszteseteket milyen információ alapján állítjuk elő. E szerint létezik:

- Ø Feketedobozos (black-box) vagy specifikáció alapú, amikor a specifikáció alapján készülnek a tesztesetek.
- Ø Fehérdobozos (white-box) vagy strukturális teszt, amikor a forráskód alapján készülnek a tesztesetek.

Tehát beszélünk feketedobozos tesztelésről, amikor a tesztelő nem látja a forráskódot, de a specifikációkat igen, fehérdobozos tesztelésről, amikor a forráskód rendelkezésre áll.

A feketedobozos tesztelést specifikáció alapúnak is nevezzük, mert a specifikáció alapján készül. Ugyanakkor a teszt futtatásához szükség van a lefordított szoftverre. Leggyakoribb formája, hogy egy adott bemenetre tudjuk, milyen kimenetet kellene adni a programnak. Lefuttatjuk a programot a bemenetre és összehasonlítjuk a kapott kimenetet az elvárttal. Ezt alkalmazzák pl. az ACM versenyeken is.

A fehérdobozos tesztelést strukturális tesztelésnek is nevezzük, mert mindig egy már kész struktúrát, pl. program kódot, tesztelünk. A strukturális teszt esetén értelmezhető a (struktúra) lefedettség. A lefedettség azt mutatja meg, hogy a struktúra hány százalékát tudjuk tesztelni a meglévő tesztesetekkel. Általában ezeket a struktúrákat teszteljük:

kódsorok,

elágazások,

metódusok,

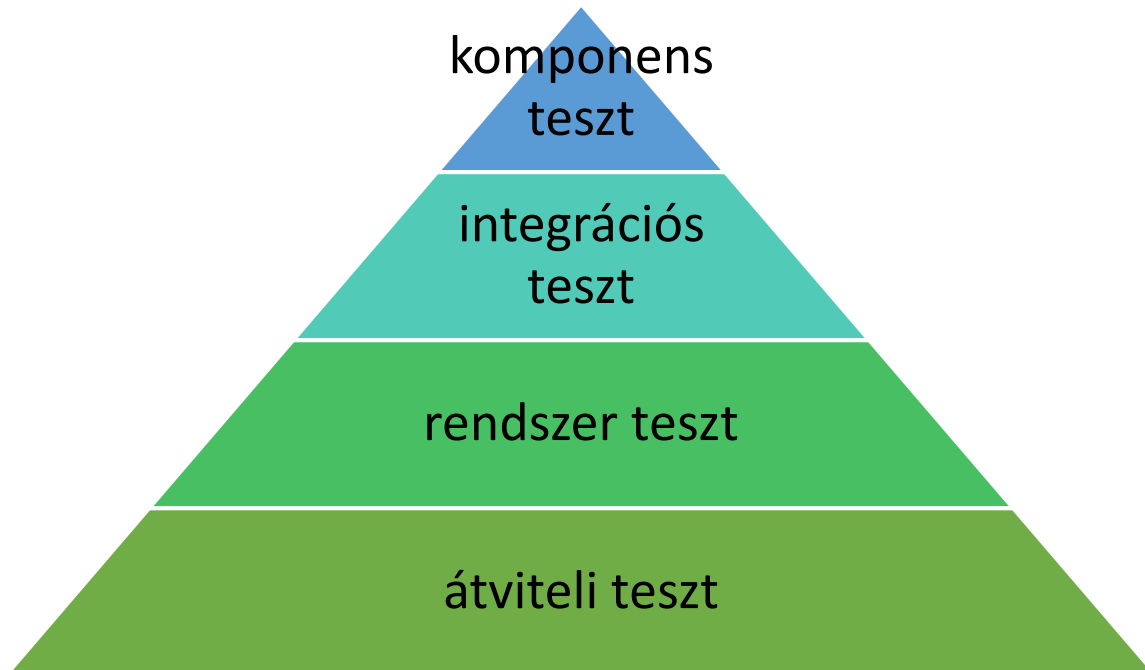
osztályok,

funkciók,

modulok.

Például a gyakran használt unit-teszt a metódusok struktúra tesztje.”[3]

A fejlesztés folyamatában betöltött szerepe alapján az alábbi csoportosítás végezhető:



Forrás: saját készítésű

„Az első, komponens teszt csak a rendszer egy komponensét teszteli önmagában. Ehhez képest az integrációs teszt kettő vagy több komponens együttműködését vizsgálja. A rendszerteszt triviális módon az egész rendszert, tehát minden komponens együtt teszteli. Ez első három teszt szintet együttesen fejlesztői tesztnek hívjuk, mert ezeket a fejlesztő cég alkalmazottai vagy megbízottjai végzik. Az átviteli teszt során a felhasználók kezébe kerül az adott szoftver és a már kész rendszert tesztelik. Ezek általában időrendben is így követik egymást.

- > **Egységtesztelés** (modultesztelés vagy komponens tesztelés): az egyes program egységeket (például modulokat, osztályokat) egymástól függetlenül teszteljük. Az adott egység fejlesztését végző programozó végzi. Ennél a tesztelésnél egy másik modul szolgáltatását használja, akkor valamilyen módon emulálni kell majd a másik modulban lévő szolgáltatás viselkedését.
- > **Integrációs tesztelés:** az egyes komponensek közötti interfészek, valamint az operációs rendszerrel, állományrendszerrel, hardverrel vagy más rendszerrel való interakciók tesztelését végezzük. Az integrációt végző fejlesztő, vagy (szerencsésebb esetben) egy direkt e célra létrehozott integrációs tesztelő csapat végzi. Az integrációs tesztelésnek több szintje is lehet, különféle célokkal. Tesztelhetjük a komponens integrációt vagy a rendszer integrációt.
- > **Rendszertesztelés:** A teljes rendszer (vagyis a késztermék) viselkedésének vizsgálatával foglalkozik. Általában ez az utolsó, a fejlesztést végző szervezet részéről elvégzett tesztelésfajta.

- > *Elfogadási tesztelés:* olyan felhasználó vagy ügyféloldali tesztelés, amelynek célja, hogy megbizonyosodjanak arról, hogy az elkészült rendszer megfelel-e a célnak, elfogadható-e a felhasználók/ügyfelek számára." [2]



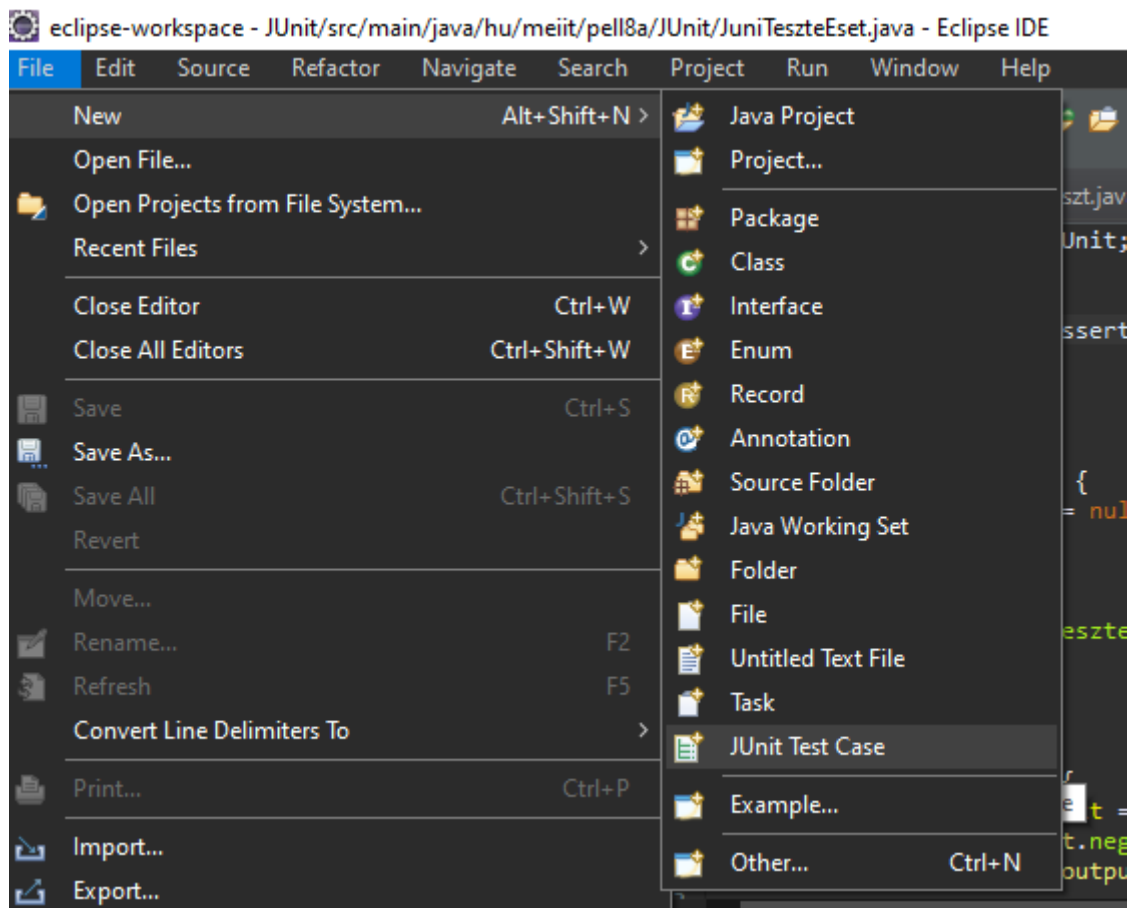
```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.1</version>
  </dependency>
</dependencies>
```

JUnit egy egységtesztelő keretrendszer a Java nyelvhez.

„A JUnit keretrendszer a tesztként lefuttatandó metódusokat annotációk segítségével ismeri fel, tehát tulajdonképpen egy beépített annotációfeldolgozót is tartalmaz. Jellemző helyzet, hogy ezek a metódusok egy olyan osztályban helyezkednek el, amelyet csak a tesztelés céljaira hoztunk létre. Ezt az osztályt *tesztosztálynak* nevezzük.

A JUnit tesztfuttatója az összes, @Test annotációval ellátott metódust lefuttatja, azonban, ha több ilyen is van, közöttük a sorrendet nem definiálja. Épp ezért tesztjeinket úgy célszerű kialakítani, hogy függetlenek legyenek egymástól, vagyis egyetlen tesztmetódusunkban se támaszkodjunk például olyan állapotra, amelyet egy másik teszt eset állít be. Egy teszt eset általában úgy épül fel, hogy a tesztmetódust az @org.junit.Test annotációval ellátjuk, a törzsében pedig meghívjuk a tesztelendő metódust, és a végrehajtás eredményeként kapott tényleges eredményt az elvárt eredménnyel össze kell vetni. A JUnit keretrendszer alapvetően csak egy parancssoros tesztfuttatót biztosít, de ezen felül nyújt egy API-t az integrált fejlesztőeszközök számára, amelynek segítségével azok grafikus tesztfuttatókat is implementálhatnak.” [1]

Az Eclipse-ben egy tesztosztály létrehozását a File → New → JUnit → JUnit Test Case menüpontban végezhetjük el.



A JUnit a @Test annotáció mellett további annotációtípusokat is definiál, amelyekkel a tesztjeink futtatását tudjuk szabályozni. Az alábbi táblázat röviden összefoglalja ezen annotációkat.

Állítás	Leírás
<code>fail([String])</code>	Feltétel nélkül elbuktatja a metódust. Annak ellenőrzésére használhatjuk, hogy a kód egy adott pontjára nem jut el a vezérlés, de arra is jó, hogy legyen egy elbukott tesztünk, mielőtt a tesztkódot megírnánk.
<code>assertTrue([String], boolean)</code>	Ellenőrzi, hogy a logikai feltétel igaz-e.
<code>assertFalse([String], boolean)</code>	Ellenőrzi, hogy a logikai feltétel hamis-e.
<code>assertEquals([String], expected, actual)</code>	Az equals metódus alapján megvizsgálja, hogy az elvárt és a tényleges eredmény megegyezik-e.
<code>assertEquals([String], expected, actual, tolerance)</code>	Valós típusú elvárt és aktuális értékek egyezőségét vizsgálja, hogy belül van-e tűréshatáron.
<code>assertArrayEquals([String], expected[], actual[])</code>	Ellenőrzi, hogy a két tömb megegyezik-e

Állítás	Leírás
<code>assertNull([message], object)</code>	Ellenőrzi, hogy az objektum null-e
<code>assertNotNull([message], object)</code>	Ellenőrzi, hogy az objektum nem null-e
<code>assertSame([String], expected, actual)</code>	Ellenőrzi, hogy az elvárt és a tényleges objektumok referencia szerint megegyeznek-e
<code>assertNotSame([String], expected, actual)</code>	Ellenőrzi, hogy az elvárt és a tényleges objektumok referencia szerint nem egyeznek-e meg

[1]

„A végrehajtás tényleges eredménye és az elvárt eredmény közötti összehasonlítás során *állításokat* fogalmazzunk meg. Az állítások nagyon hasonlóak az „Állítások (assertions)” alfejezetben már megismertekhez, azonban itt nem az assert utasítást, hanem az org.junit.Assert osztály statikus metódusait használjuk ennek megfogalmazására. Ezen metódusok nevei az assert rész sztringgel kezdődnek, és lehetővé teszik, hogy megadjunk egy hibaüzenetet, valamint az elvárt és tényleges eredményt. Egy ilyen metódus elvégzi az értékek összevetését, és egy AssertionError kivételt dob, ha az összehasonlítás elbukik. (Ez a hiba ugyanaz, amelyet az assert utasítás is kivált, ha a feltétele hamis.) A következő táblázat összefoglalja a legfontosabb ilyen metódusokat. a szögletes zárójelek ([]) közötti paraméterek opcionálisok.” [1]

Az Assert osztály metódusai

Állítás	Leírás
<code>fail([String])</code>	Feltétel nélkül elbuktatja a metódust. Annak ellenőrzésére használhatjuk, hogy a kód egy adott pontjára nem jut el a vezérlés, de arra is jó, hogy legyen egy elbukott tesztünk, mielőtt a tesztkódot megírnánk.
<code>assertTrue([String], boolean)</code>	Ellenőrzi, hogy a logikai feltétel igaz-e.
<code>assertFalse([String], boolean)</code>	Ellenőrzi, hogy a logikai feltétel hamis-e.
<code>assertEquals([String], expected, actual)</code>	Az equals metódus alapján megvizsgálja, hogy az elvárt és a tényleges eredmény megegyezik-e.
<code>assertEquals([String], expected, actual, tolerance)</code>	Valós típusú elvárt és aktuális értékek egyezőségét vizsgálja, hogy belül van-e tűréshatáron.
<code>assertArrayEquals([String], expected[], actual[])</code>	Ellenőrzi, hogy a két tömb megegyezik-e
<code>assertNull([message], object)</code>	Ellenőrzi, hogy az objektum null-e
<code>assertNotNull([message], object)</code>	Ellenőrzi, hogy az objektum nem null-e
<code>assertSame([String], expected, actual)</code>	Ellenőrzi, hogy az elvárt és a tényleges objektumok referencia szerint megegyeznek-e

Állítás	Leírás
<code>assertNotSame([String], expected, actual)</code>	Ellenőrzi, hogy az elvárt és a tényleges objektumok referencia szerint nem egyeznek-e meg

[1]

Gyakorlati feladat - a teszt osztályok négyzet számítást és a megadott a_A karakter/karakterek számát ellenőrzi

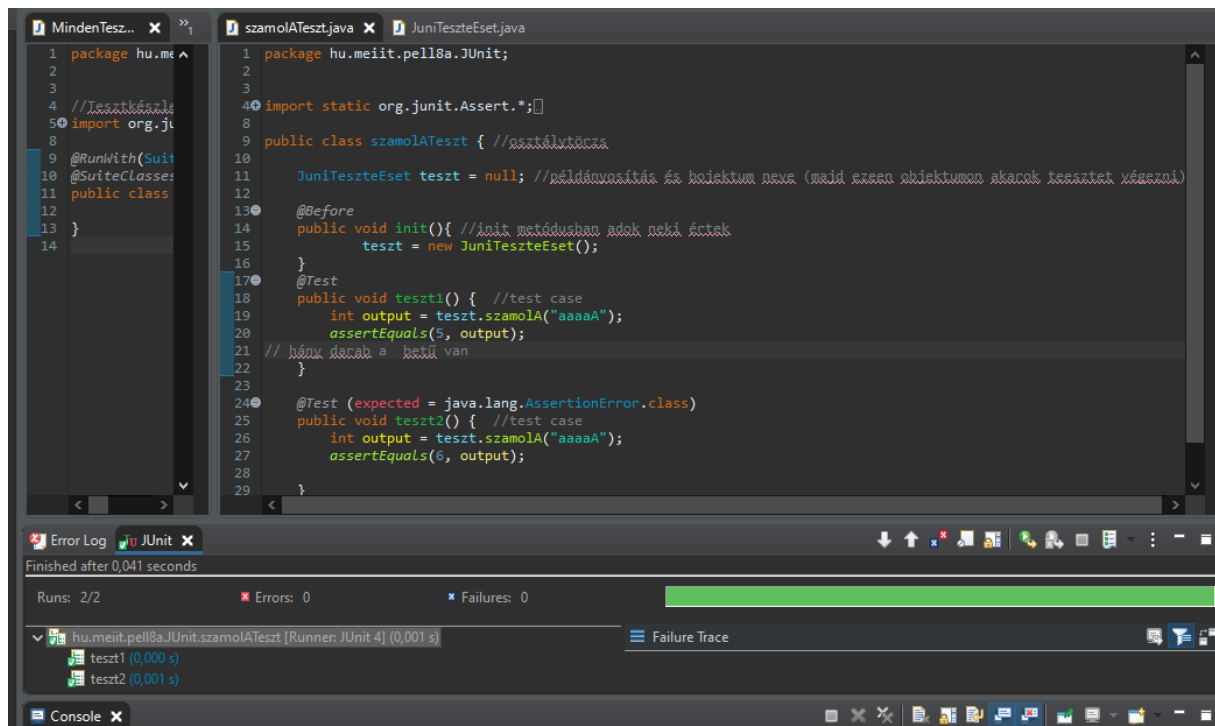
The screenshot displays an IDE with two Java files: `negyzetTeszt.java` and `MindenTeszt.java`. The `negyzetTeszt.java` file contains the following code:

```

1 package hu.meit.pell8a.JUnit;
2
3 import static org.junit.Assert.*;
4
5 public class negyzetTeszt {
6     JuniTeszteEset teszt = null;
7
8     @Before
9     public void init(){
10         teszt = new JuniTeszteEset();
11     }
12
13     @Test
14     public void teszt1() {
15         JuniTeszteEset teszt = new JuniTeszteEset();
16         int output = teszt.negyzet(5); //kimenet
17         assertEquals(25, output);
18     }
19
20     // Megadottan került a teszt argumentumában, az elvárt eredmény:
21     // hogy megszűnik a teszt, így a teszt lefutása sikeres lesz.
22
23     @Test (expected = java.lang.AssertionError.class)
24     public void teszt2() {
25         int output = teszt.negyzet(3); //kimenet
26         assertEquals(25, output);
27     }
28

```

The IDE also shows the execution results in the `Error Log` and `JUnit` tabs. The `JUnit` tab indicates that the tests were finished after 0.056 seconds, with 2 runs, 0 errors, and 0 failures. The `Console` tab shows the output of the tests, including the message `<terminated> negyzetTeszt [JUnit] C:\Users\kolla\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_14.0.2.v20200815-0932\jre\bin\javaw.exe (2020. dec. 2. 20:27:07 - 20:27:14)`.



```
1 package hu.meit.pell8a.JUnit;
2
3
4 //Tesztkészlet
5 import org.junit.*;
6
7
8
9 @RunWith(Suite.class)
10 @SuiteClasses({
11     szamolATeszt.class,
12 })
13 public class MindenTeszt {
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29 }
```

```
1 package hu.meit.pell8a.JUnit;
2
3
4 import static org.junit.Assert.*;
5
6 public class szamolATeszt { //osztálytörzs
7
8     JuniTeszteEset teszt = null; //példányosítás és objektum neve (máig ebben objektumon akarok tesztet végezni)
9
10     @Before
11     public void init() { //init metódusban adok neki értéket
12         teszt = new JuniTeszteEset();
13     }
14
15     @Test
16     public void teszt1() { //test case
17         int output = teszt.szamola("aaaaA");
18         assertEquals(5, output);
19     }
20
21     // ha az aaaaa aaaaa van
22
23
24     @Test (expected = java.lang.AssertionError.class)
25     public void teszt2() { //test case
26         int output = teszt.szamola("aaaaA");
27         assertEquals(6, output);
28     }
29 }
```

Finished after 0,041 seconds

Runs: 2/2 Errors: 0 Failures: 0

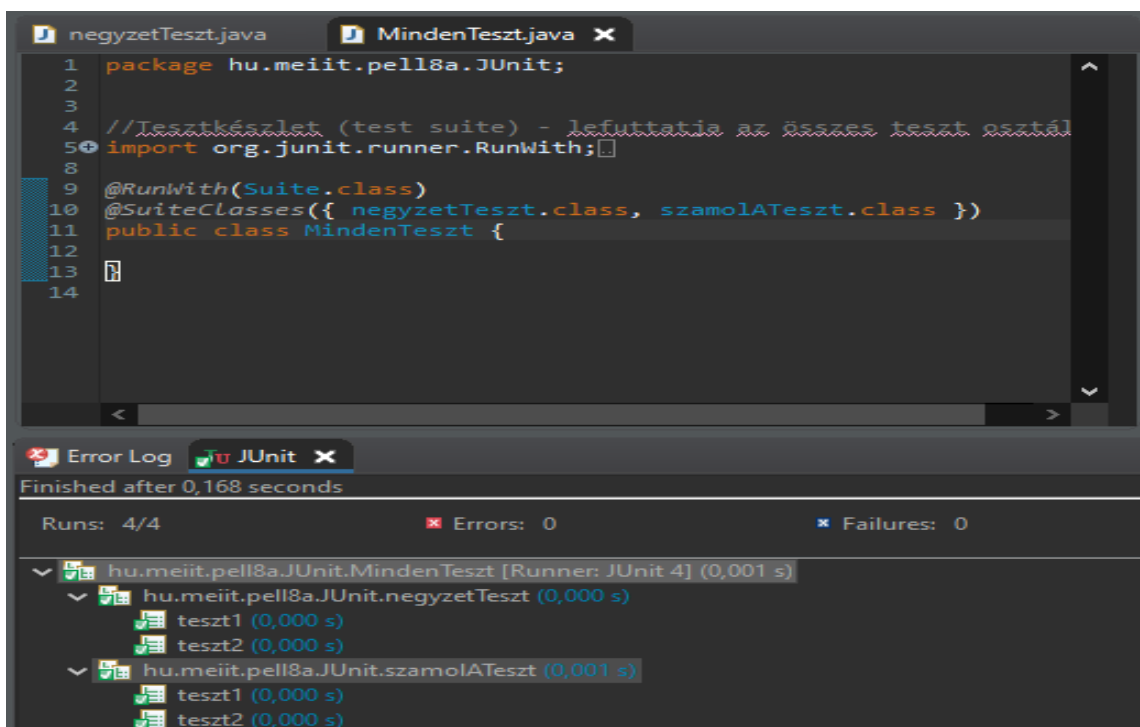
hu.meit.pell8a.JUnit.szamolATeszt [Runner: JUnit 4] (0,001 s)

- teszt1 (0,000 s)
- teszt2 (0,001 s)

Mivel minden paraméter helyesen lett megadva a teszt1 -ek sikeresen lefutottak. A teszt2-őknél megadásra került a teszt argumentumában, az elvárt eredmény, hogy megbukik a teszt, így a teszt lefutása szintén sikeres.

„A test suite (tesztkészlet), ebbe összegezzük a testcase-eket és ezáltal egyszerre lefuttathatjuk, amelyhez egy olyan tesztszámolóra van szükségünk, amelyet a @RunWith(Suite.class) és a @Suite.SuiteClasses annotációkkal is el kell látnunk.

A feladatban a negyzetTESZT, és a szamolATeszt tesztszámolók által tartalmazott tesztesetek végrehajtását végző tesztkészlet létrehozását láthatjuk”[1]



```
1 package hu.meit.pell8a.JUnit;
2
3
4 //Tesztkészlet (test suite) - lefuttatja az összes teszt osztály
5 import org.junit.runner.RunWith;
6
7
8
9 @RunWith(Suite.class)
10 @SuiteClasses({
11     negyzetTeszt.class,
12     szamolATeszt.class
13 })
14 public class MindenTeszt {
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29 }
```

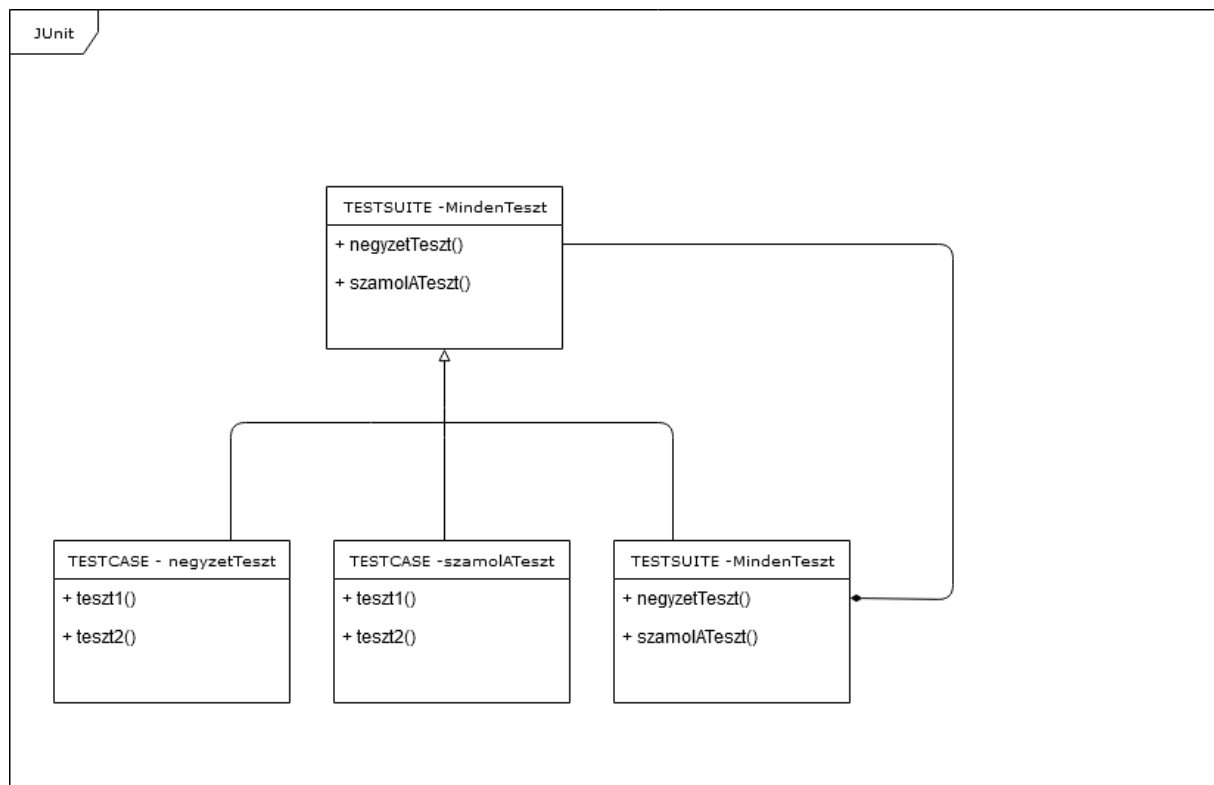
Finished after 0,168 seconds

Runs: 4/4 Errors: 0 Failures: 0

hu.meit.pell8a.JUnit.MindenTeszt [Runner: JUnit 4] (0,001 s)

- hu.meit.pell8a.JUnit.negyzetTeszt (0,000 s)
 - teszt1 (0,000 s)
 - teszt2 (0,000 s)
- hu.meit.pell8a.JUnit.szamolATeszt (0,001 s)
 - teszt1 (0,000 s)
 - teszt2 (0,000 s)

UML diagram



Az UML diagram saját készítésű

Felhasznált irodalom:

[1] <https://gyires.inf.unideb.hu/GyBITT/21/ch03s02.html>

[2] https://regi.tankonyvtar.hu/hu/tartalom/tamop425/0046_szoftverteszteles/index.html

[3] https://regi.tankonyvtar.hu/hu/tartalom/tamop425/0046_szoftverteszteles/ch01s02.html

JUnit logo:

<https://avatars0.githubusercontent.com/u/874086?s=280&v=4>