

Miskolci Egyetem
Gépészmérnöki És Informatikai Kar
Informatikai Intézet
Alkalmazott Informatikai Intézeti Tanszék

Jegyzőkönyv

Szoftvertesztelés - egységteszt
féléves feladat



Harnócz Fanni
JTZX6o
mérnök-informatikus BSc hallgató
rendszer-mérnök szakirány

2020 Miskolc

Tartalo

Feladat	3
1. A szoftver tesztek célja	3
2. Tesztelés terminológiája	3
2.1. Kód (vagy alkalmazás) tesztelés alatt	3
2.2. Test fixture – Teszt hozzávalók.....	3
2.3. Egységteszt	3
2.4. Integrációs tesztek	4
2.5. Teljesítmény teszt	4
2.6. Viselkedés és állapot tesztelés	4
2.7. Java keretrendszerek tesztelése.....	4
2.8. Hol kell elhelyezni a tesztet?	4
3. A JUnit használata általánosságban	5
3.1. Teszt meghatározása JUnitban.....	5
3.2. JUnit nevezési konvenciók	5
3.3. JUnit nevezési konvenciók Maven számára	5
3.4. Teszt végrehajtási rend.....	6
4. A JUnit 4 használata	6
UML diagram.....	9
Gyakorlati példa bemutatása.....	10
1. NoSuchElementException	11
2. Méret ellenőrzés	12
3. Hibátlan tesztelés	13
Felhasznált irodalom	14



Feladat

A fél éves feladatomban Eclipse környezetben fogok részletesen bemutatni egy egységtesztet JUnit segítségével.

1. A szoftver tesztek célja

A szoftver teszt egy szoftver, amely egy másik szoftvert hajt végre. Validálja, ha ez a kód a várt állapotot eredményezi (állapot teszt), vagy végrehajtja a várható eseménysorozatot (viselkedés teszt).

A szoftveregység-tesztek, röviden egységtesztek, mellyel a dolgozatomban is foglalkozok, segítenek a fejlesztőnek ellenőrizni, hogy a program egy részének logikája helyes-e.

A tesztek automatikus futtatása segít beazonosítani a szoftver regressziókat, melyeket a forráskódban történő változások okoznak. A kód magas tesztlefedettsége lehetővé teszi, hogy a funkciókat tovább fejlesszük manuális tesztek sokasága nélkül.

2. Tesztelés terminológiája

Kezdő tesztelőként fontosnak találom, hogy a felmerülő fogalmakat tisztán lássuk, hogy a későbbi gyakorlati életben ezek ne okozzanak félreértést, így ezeket legnagyobb egységtől a legkisebb felé haladva sorra is veszem

2.1. Kód (vagy alkalmazás) tesztelés alatt

A tesztelt kódot általában *tesztelés alatt álló kódnak* hívják. Ha egy alkalmazást tesztelünk, akkor ezt *tesztelés alatt álló alkalmazásnak* nevezzük.

2.2. Test fixture – Teszt hozzávalók

A teszt hozzávalók az objektumok halmazának rögzített állapota, amelyeket kiindulási alapként használnak a tesztek futtatásához. Ennek másik leírási módja a teszt előfeltétele.

2.3. Egységteszt

Az egységteszt egy fejlesztő által írt kóddarab, amely a tesztelendő kódban egy bizonyos funkciót hajt végre, és bizonyos viselkedést vagy állapotot érvényesít.

Az egységtesztekkel tesztelt kód százalékát általában teszt lefedettségnek nevezzük.

Az egységteszt egy kis kódegységet céloz meg, például egy módszert vagy osztályt. A külső függőségeket el kell távolítani az egységtesztekből, például úgy, hogy a függőséget egy tesztmegvalósítással vagy egy tesztkeret által létrehozott (ál) objektummal helyettesítjük.

Az egységtesztek nem alkalmasak összetett felhasználói felület vagy komponens kölcsönhatás tesztelésére. Ehhez ki kell dolgoznia az integrációs tesztek.



2.4. Integrációs tesztek

Az integrációs teszt célja egy komponens viselkedésének vagy az összetevők halmaza közötti integráció tesztelése. A funkcionális teszt kifejezést néha az integrációs teszt szinonimájaként használják.

Az integrációs tesztek ellenőrzik, hogy a teljes rendszer megfelelően működik-e, ezért csökkentik az intenzív kézi tesztek szükségességét.

Az ilyen típusú tesztek lehetővé teszik a user storyk tesztorozatba történő lefordítását. A teszt hasonlít az alkalmazás várható felhasználói interakciójára.

2.5. Teljesítmény teszt

Teljesítményteszteket használnak a szoftverkomponensek ismételt összehasonlítására. Céljuk annak biztosítása, hogy a tesztelt kód elég gyors legyen, még nagy terhelés alatt is.

2.6. Viselkedés és állapot tesztelés

A teszt egy viselkedésteszt (más néven interakciós teszt), ha ellenőrzi, hogy bizonyos módszereket a megfelelő bemeneti paraméterekkel hívtak-e meg. A viselkedési teszt nem érvényesíti a metódushívás eredményét.

Az állapot tesztelés az eredmény validálásáról szól. A viselkedés tesztelése a tesztelt alkalmazás viselkedésének teszteléséről szól.

Ha algoritmusokat vagy rendszerfunkciókat tesztelünk, akkor a legtöbb esetben érdemes állapotot és nem interakciókat tesztelni.

2.7. Java keretrendszerek tesztelése

Számos tesztelési keretrendszer áll rendelkezésre a Java számára. A legnépszerűbbek a JUnit és a TestNG. Ez a leírás a JUnit-re összpontosít. A JUnit 4.x és az 5. JUnit egyaránt lefedi.

2.8. Hol kell elhelyezni a tesztet?

Tipikusan egységteszteket külön projektben vagy külön forrásmappában hozták létre, hogy a teszt kód elkülönüljön a valós kódtól.

A Maven és a Gradle buildeszközök szokásos konvenciója a következő:

- src/main/java - for Java classes
- src/test/java - for test classes

Hogy mit kell tesztelni, az nagyon ellentmondásos téma. Egyes fejlesztők úgy vélik, hogy a kódban szereplő minden állítást tesztelni kell.



Mindenestre szoftveres teszteket kell írni az alkalmazás kritikus és összetett részeihez. Ha új funkciókat vezet be, akkor egy szilárd tesztcsomag is megvéd a regressziótól a meglévő kódban.

Általában biztos, hogy figyelmen kívül hagyható a triviális kód. Például általában haszontalan olyan teszteket írni a *getter* és *setter* módszerekhez, amelyek egyszerűen értékeket rendelnek a mezőkhöz. Tesztek írása ezekhez az utasításokhoz időigényes és értelmetlen, mivel a Java virtuális gépet tesztelnénk. Magának a JVM-nek már vannak tesztesei erre.

3. A JUnit használata általánosságban

A JUnit egy tesztkeret, amely kommentárokat használ a tesztet meghatározó módszerek azonosítására. A JUnit egy nyílt forráskódú projekt, amelynek a Github ad otthont.

3.1. Teszt meghatározása JUnitban

A JUnit teszt egy osztályba tartozó metódus, amelyet csak tesztelésre használnak. Ezt nevezzük *Teszt osztálynak*.

Annak meghatározásához, hogy egy metódus teszt metódus, el kell látni **@Test** annotációval. Ez a metódus végrehajtja a kódot teszt alatt. A JUnit vagy más állítási keretrendszer által biztosított assert metódust használva a várt eredmény és a tényleges eredmény összehasonlítható.

Ezeket a metódushívásokat általában asserteknek vagy assert utasításoknak nevezik. Jelentőségteljes üzeneteket kell megadni egy assert állításban. Ez megkönnyíti a felhasználó számára a probléma azonosítását és kijavítását. Ez különösen igaz, ha valaki megnézi a problémát, aki nem a tesztelt kódot vagy a tesztkódot írta.

3.2. JUnit nevezési konvenciók

A JUnit teszteknel számos lehetséges elnevezési megállapodás létezik. Az osztályok számára széles körben alkalmazott megoldás a "Test" utótag használata a tesztosztályok nevének végén.

Általános szabály, hogy a tesztnévnek meg kell magyaráznia a teszt működését. Az egyik lehetséges szokás a "should" használata a vizsgálati módszer nevében. Például: "orderShouldBeCreated" vagy "menuShouldGetActive". Ez tippet ad arra, hogy mi történjen a teszt módszer végrehajtása esetén.

Egy másik megközelítés az "Adott [ExplainYourInput] Amikor [WhatIsDone], majd [ExpectedResult]" használata a vizsgálati módszer megjelenítési nevéhez.

3.3. JUnit nevezési konvenciók Maven számára



Ha a Maven használata esetén a "Test" utótagot hozzáadása szükséges a tesztosztályokhoz. A Maven build rendszer (a surfire plug-in keresztül) automatikusan felveszi az ilyen osztályokat a tesztkörébe.

3.4. Teszt végrehajtási rend

A JUnit feltételezi, hogy az összes tesztelési módszer tetszőleges sorrendben végrehajtható. A jól megírt teszt kód nem vehet fel sorrendet, vagyis a tesztek nem függhetnek más tesztektől.

A JUnit 4.11-től kezdve az alapértelmezett egy determinisztikus, de nem kiszámítható sorrend használata a tesztek végrehajtásához.

Annotációvaé meghatározható, hogy a teszt metódusok a metódus neve szerint, szótártani sorrendben legyenek rendezve. Ennek a funkciónak az aktiválásához a teszt osztályt a `@FixMethodOrder(MethodSorters.NAME_ASCENDING)` annotációval szükséges annotálni. Alapértelmezettként is beállítható a `MethodSorters.DEFAULT` paraméter megadásával az annotációban. Használható a `MethodSorters.JVM` is ami a JVM alapértelmezetteket használja, ami futásról futásra változhat.

4. A JUnit 4 használata

A JUnit annotációkat használ a metódusok tesztelési módszerként történő megjelölésére és konfigurálására. Az alábbi táblázat áttekintést nyújt a JUnit legfontosabb annotációiról a 4.x és az 5.x verzióhoz. Mindezek az annotációk felhasználhatók a metódusokban.

JUnit 4	Leírás
<code>import org.junit.*</code>	Állítások importálása a következő annotációk használatához.
<code>@Test</code>	A metódust teszt metódusként azonosítja.
<code>@Before</code>	Minden teszt előtt végrehajtva. A tesztkörnyezet előkészítésére szolgál (pl. Bemeneti adatok kiolvasása, az osztály inicializálása).
<code>@After</code>	Minden teszt után végrehajtva. A tesztkörnyezet tisztítására szolgál (például ideiglenes adatok törlésére, az alapértelmezettek visszaállítására). Emellett memóriát takaríthat meg a drága memóriaszerkezetek megtisztításával.
<code>@BeforeClass</code>	Egyszer végrehajtják, az összes teszt megkezdése előtt. Időigényes tevékenységek végrehajtására használják, például csatlakozáshoz egy adatbázishoz. Az ezzel az annotációval megjelölt



	metódusokat statikusként kell definiálni, hogy a JUnit programmal működjenek.
@AfterClass	Egyszer hajtják végre, miután az összes teszt befejeződött. Tisztítási tevékenységek elvégzésére használják, például az adatbázisból való leválasztáshoz. Az ezzel az annotációval jegyzetelt metódusokat statikusként kell meghatározni, hogy a JUnittel működjenek.
@Ignore or @Ignore("Why disabled")	Jelzi, hogy a tesztet le kell tiltani. Ez akkor hasznos, ha az alapul szolgáló kód megváltozott, és a tesztet még nem adaptálták. Vagy ha ennek a tesztnek a végrehajtási ideje túl hosszú ahhoz, hogy belefoglalják. A legjobb gyakorlat, ha megadja az opcionális leírást, miért tiltják le a tesztet.
@Test (expected = Exception.class)	Nem sikerül, ha a metódus nem dobja meg a megnevezett kivételt.
@Test(timeout=100)	Nem sikerül, ha a módszer 100 milliszekundumnál tovább tart.

A JUnit statikus módszereket kínál bizonyos feltételek tesztelésére az `Assert` osztályon keresztül. Ezek az assert statementek tipikusan asserttel kezdődnek. Lehetővé teszik a hibaüzenet, a várható és a tényleges eredmény megadását. Egy assert metódus összehasonlítja a teszt által adott tényleges értéket a várható értékkel. Ha az összehasonlítás nem sikerül, akkor `AssertionException`ot dob.

Az alábbi táblázat áttekintést nyújt ezekről a módszerekről. A [] zárójelben lévő paraméterek opcionálisak és String típusúak.

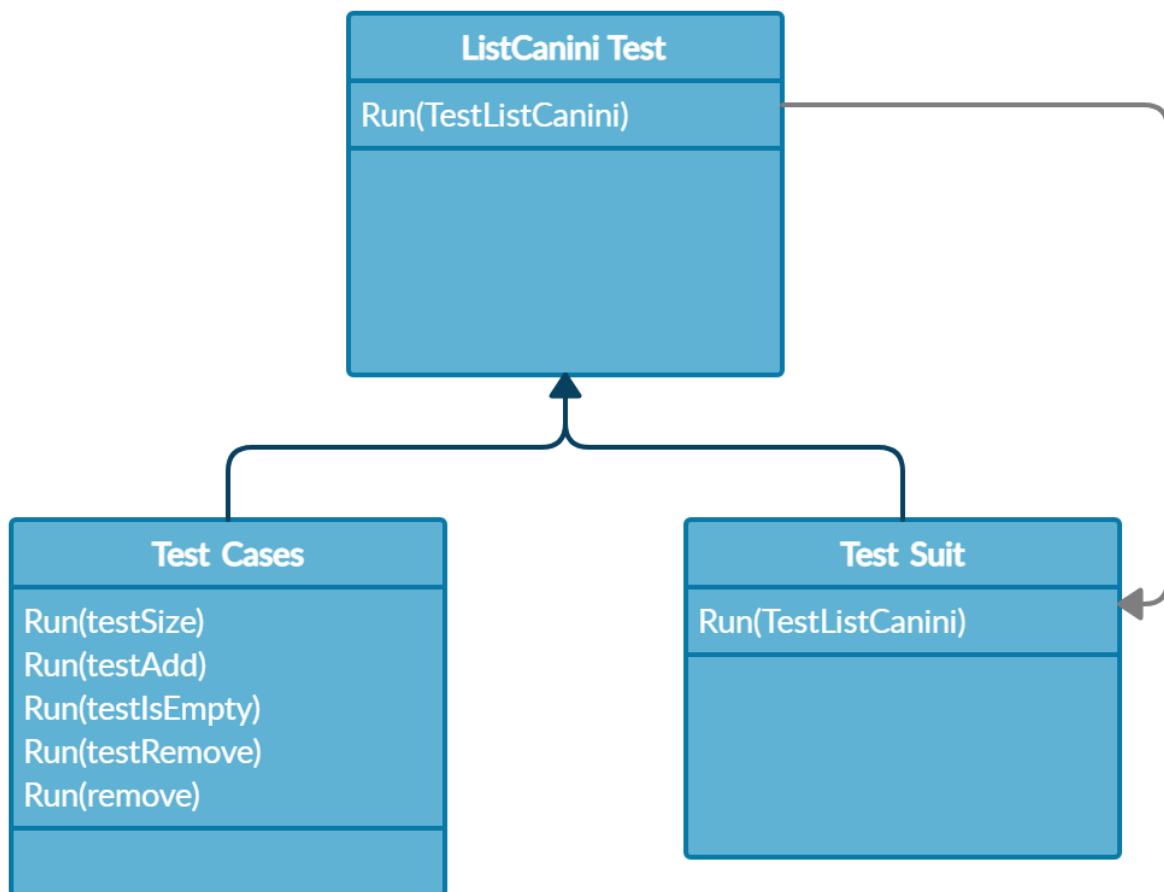
Asserts	Leírás
fail([message])	Állítások importálása a következő annotációk használatához.
assertTrue([message,] boolean condition)	A metódust teszt metódusként azonosítja.
assertFalse([message,] boolean condition)	Minden teszt előtt végrehajtva. A tesztkörnyezet előkészítésére szolgál (pl. Bemeneti adatok kiolvasása, az osztály inicializálása).
assertEquals([message,] expected, actual)	Minden teszt után végrehajtva. A tesztkörnyezet tisztítására szolgál (például ideiglenes adatok törlésére, az alapértelmezettek visszaállítására). Emellett memóriát takaríthat meg a drága memóriaszerkezetek megtisztításával.



<code>assertEquals([message,] expected, actual, tolerance)</code>	Egyszer végrehajtják, az összes teszt megkezdése előtt. Időigényes tevékenységek végrehajtására használják, például csatlakozáshoz egy adatbázishoz. Az ezzel az annotációval megjelölt metódusokat statikusként kell definiálni, hogy a JUnit programmal működjenek.
<code>assertNull([message,] object)</code>	Egyszer hajtják végre, miután az összes teszt befejeződött. Tisztítási tevékenységek elvégzésére használják, például az adatbázisból való leválasztáshoz. Az ezzel az annotációval jegyzetelt metódusokat statikusként kell meghatározni, hogy a JUnittel működjenek.
<code>assertNotNull([message,] object)</code>	Jelzi, hogy a tesztet le kell tiltani. Ez akkor hasznos, ha az alapul szolgáló kód megváltozott, és a tesztesetet még nem adaptálták. Vagy ha ennek a tesztnek a végrehajtási ideje túl hosszú ahhoz, hogy belefoglalják. A legjobb gyakorlat, ha megadja az opcionális leírást, miért tiltják le a tesztet.
<code>assertSame([message,] expected, actual)</code>	Nem sikerül, ha a metódus nem dobja meg a megnevezett kivételt.
<code>assertNotSame([message,] expected, actual)</code>	Nem sikerül, ha a módszer 100 milliszekundumnál tovább tart.



UML diagram

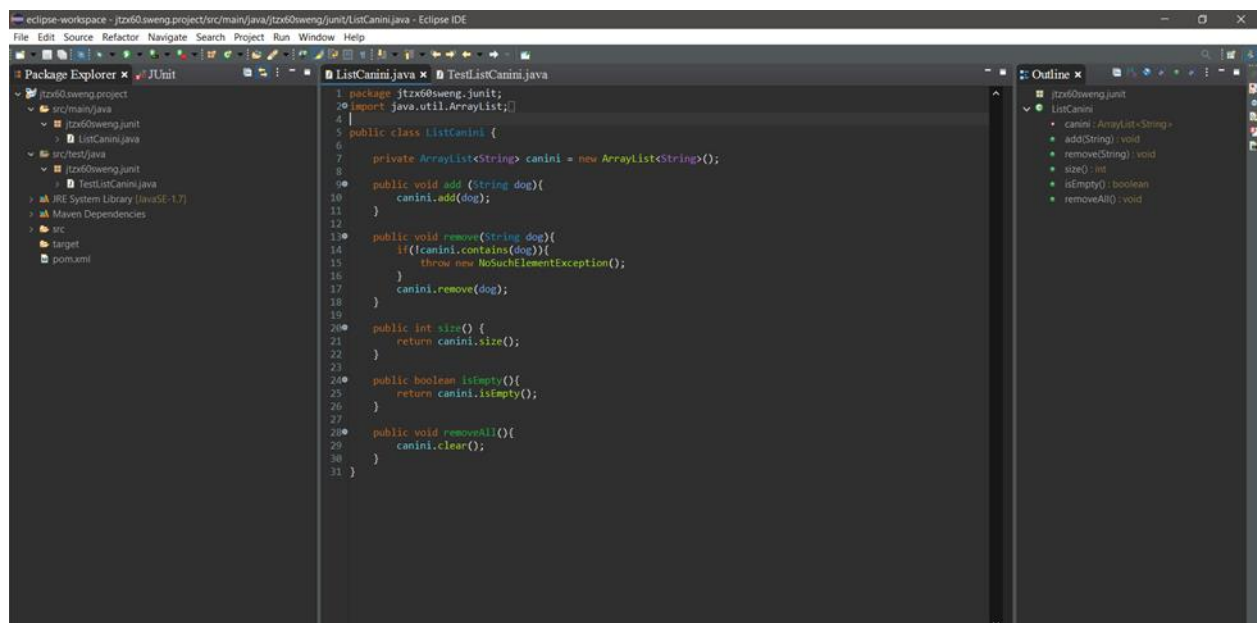


Gyakorlati példa bemutatása

A gyakorlati példában egy kutyákról szóló listán végrehajtható műveleteket fogjuk sorra ellenőrizni.

A műveletek a következők:

- › hozzáadás
- › eltávolítás
- › összes eltávolítása
- › üresség ellenőrzése
- › méret ellenőrzése



```
1 package jtzx60sweng.junit;
2 import java.util.ArrayList;
3
4 public class ListCanini {
5     private ArrayList<String> canini = new ArrayList<>();
6
7     public void add (String dog){
8         canini.add(dog);
9     }
10
11     public void remove(String dog){
12         if(!canini.contains(dog)){
13             throw new NoSuchElementException();
14         }
15         canini.remove(dog);
16     }
17
18     public int size() {
19         return canini.size();
20     }
21
22     public boolean isEmpty(){
23         return canini.isEmpty();
24     }
25
26     public void removeAll(){
27         canini.clear();
28     }
29
30 }
31 }
```

1. ábra: Forráskód



1. NoSuchElementException

A tesztelés során minden művelethez létrehoztam egy tesztesetet. Az első verzióban úgy, hogy csak egy hiba jelentkezzen. A második esetben már egy újabb hibával bővült a lista.

A forráskódban látható, hogy egy a `remove`, vagyis eltávolítás műveletnél hozzáadtam egy kivételt, `NoSuchElementException`, vagyis, ha olyan elemet szeretnének eltávolítani a listából, ami nem szerepel benne, akkor hibát jelez a tesztesetünknel.

Erre látható példa a következő ábrán.

The screenshot shows the Eclipse IDE with a Java project. The main editor displays the `TestListCanini.java` file. The code includes a `@Before` method to initialize a `ListCanini` object with three elements: "Border Collie", "Alaszakai Malamu", and "Corgi". There are several test methods: `testSize()` (asserts size is 3), `testAdd()` (adds "Beagle" and asserts size is 4), `testIsEmpty()` (asserts not empty), `testRemove()` (removes all), and `remove()` (removes "Pitbull"). The `remove()` method is highlighted with a red circle. The Package Explorer on the left shows the project structure, and the Outline view on the right shows the class hierarchy. The Run console at the bottom shows the execution results, including a `NoSuchElementException` error.

2. ábra: NoSuchElementException

```

13 • @Before
14 public void init(){
15     testCanini.add("Border Collie");
16     testCanini.add("Alaszakai Malamu");
17     testCanini.add("Corgi");
18 }
19
20 • @Test
21 public void testSize(){
22     assertEquals("Méret ellenőrzés", 3, testCanini.size());
23 }
24
25 • @Test
26 public void testAdd(){
27     testCanini.add("Beagle");
28     assertEquals("Hozzáadás ellenőrzés", 4, testCanini.size());
29 }
30
31 • @Test
32 public void testIsEmpty(){
33     assertFalse(testCanini.isEmpty());
34 }
35
36 • @Test
37 public void testRemove(){
38     testCanini.removeAll();
39 }
40
41 • @Test
42 public void remove(){
43     testCanini.remove("Pitbull");
44 }
45

```



2. Méret ellenőrzés

A `@Before` annotációban 3 elemet adtunk meg a listának, ami azt jelenti, hogy a méret ellenőrzésénél ezt a számot várja vissza a tesztet. Szándékosan 2-t adtam meg paraméterként, így természetesen hibára futott a tesztelés.

```

1 package jtzx60sweng.junit;
2 import static org.junit.Assert.*;
3
4 public class TestListCanini{
5
6     private ListCanini testCanini = new ListCanini();
7
8     @Before
9     public void init(){
10         testCanini.add("Border Collie");
11         testCanini.add("Alaszakai Malamut");
12         testCanini.add("Corgi");
13     }
14
15     @Test
16     public void testSize(){
17         assertEquals("Méret ellenőrzés", 2, testCanini.size());
18     }
19
20     @Test
21     public void testAdd(){
22         testCanini.add("Beagle");
23         assertEquals("Hozzáadás ellenőrzés", 4, testCanini.size());
24     }
25
26     @Test
27     public void testIsEmpty(){
28         assertFalse(testCanini.isEmpty());
29     }
30
31     @Test
32     public void testRemove(){
33         testCanini.removeAll();
34     }
35
36     @Test
37     public void remove(){
38         testCanini.remove("Pitbull");
39     }
40 }

```

Failure Trace
 1 java.lang.AssertionError: Méret ellenőrzés expected:<2> but was <3>
 2 at jtzx60sweng.junit.TestListCanini.testSize(TestListCanini.java:22)

3. ábra: Méret ellenőrzés: expected: <2> but was <3>.

```

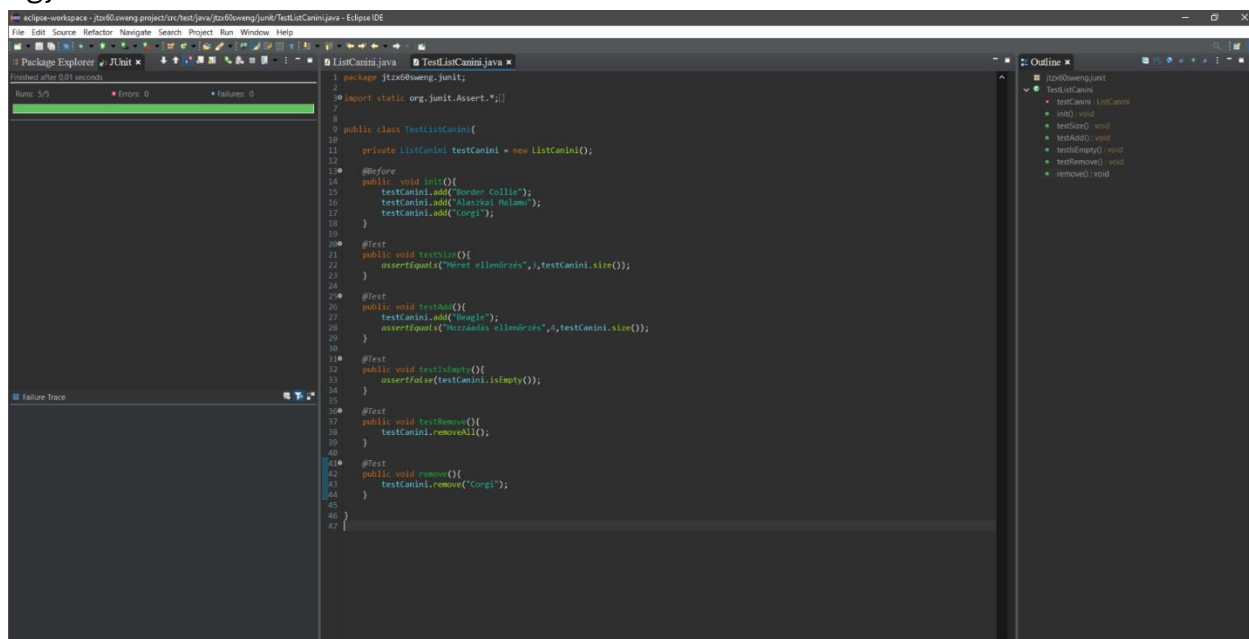
8
9 public class TestListCanini{
10
11     private ListCanini testCanini = new ListCanini();
12
13     @Before
14     public void init(){
15         testCanini.add("Border Collie");
16         testCanini.add("Alaszakai Malamut");
17         testCanini.add("Corgi");
18     }
19
20     @Test
21     public void testSize(){
22         assertEquals("Méret ellenőrzés", 2, testCanini.size());
23     }
24
25     @Test
26     public void testAdd(){
27         testCanini.add("Beagle");
28         assertEquals("Hozzáadás ellenőrzés", 4, testCanini.size());
29     }
30 }

```

3. Hibátlan tesztelés

Az utolsó bemutatott példában úgy módosítottam a paramétereket, hogy hibátlanul fussanak le a tesztesetek. Ehhez javítottam a korábban rosszul megadott listaméretet, a helyes 3-ra, illetve a listából egy olyan kutyaajtát távolítottam el, amely egyébként is szerepelt a listában.

Így o hibával futottak le a tesztesetek.



```
1 package jtzx6oeng.junit;
2
3 import static org.junit.Assert.*;
4
5 public class TestListCanini {
6     private ListCanini testCanini = new ListCanini();
7
8     @Before
9     public void init() {
10         testCanini.add("border collie");
11         testCanini.add("Alaszkai Malamé");
12         testCanini.add("corgi");
13     }
14
15     @Test
16     public void testSize() {
17         assertEquals("Hiret ellenörzés", 3, testCanini.size());
18     }
19
20     @Test
21     public void testAdd() {
22         testCanini.add("beagle");
23         assertEquals("Hirszándék ellenörzés", 4, testCanini.size());
24     }
25
26     @Test
27     public void testIsEmpty() {
28         assertFalse(testCanini.isEmpty());
29     }
30
31     @Test
32     public void testRemove() {
33         testCanini.removeAll();
34     }
35
36     @Test
37     public void remove() {
38         testCanini.remove("corgi");
39     }
40 }
41
42 }
```

4. ábra: Minden teszteset jól futott le



Felhasznált irodalom

- › [Unit Testing with JUnit – Tutorial](#)
- › [Unit tests with Mockito – Tutorial](#)
- › [JUnit 5 Tutorial: Running Unit Tests With Maven](#)
- › [Programozási technológiák – Jegyzet](#)
- › [Java Unit Testing Tutorial](#)
- › [Introduction to Unit Testing with Java](#)

SanFranciscobol Jottem videók:

- › [Tesztelés Java környezetben 02. TDD](#)
- › [Tesztelés Java környezetben 03. JUnit](#)
- › [Tesztelés Java környezetben 04. Mockito](#)

