MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR

Szoftvertesztelés
JUnit keretrendszer

GEIAL31H-B

Tompa Tamás
egyetemi tanársegéd
Általános Informatikai Intézeti Tanszék

Miskolc, 2019

xUnit

- Az **xUnit** a tesztttechnikák eszközeinek és az azokhoz kapcsolódó módszertani elgondolásoknak az elnevezései

- Alacsonyszintű tesztek írásának hatékony eszköze
  - számos programozási nyelven megvalósították

xUnit

- Célja, hogy a programozó számára keretet adjon hatékony teszteket írására fókuszálva egy adott programegységre
- Az xUnit eszközök a Driver szerepét töltik be a Driver & Stub technikák esetében
- Őse a Smalltalk világában jelent meg Kent Beck tollából SUnit névvel (1998), majd később követte a JUnit

xUnit

- Az alábbi eszköz architektúrát javasolja:
  - **Tesztfuttató keretrendszer (Test Runner)**
  - **A teszt megvalósítását támogató eszköz (Test / Test Case)**
  - **Teszt előfeltételeket támogató eszköz (Test Fixtures)**
  - **Tesztfutás és életciklus (Test Execution & Lifecycle**)
  - **Tesztgyűjtemények (Test Suites)**
  - **Teszt elvárások (Assertions)**
  - **Tesztjelentés**

xUnit

- Az alábbi eszköz architektúrát javasolja:
  - **Tesztfuttató keretrendszer (Test Runner)**
    - a tesztek futtatása egy adott xUnit keretrendszerben
    - a teszteredmények rögzítése
    - gondoskodik arról, hogy egy teszt példány futtatásra alkalmas állapotba kerüljön
    - meghatározhatja a futtatandó tesztek körét

xUnit

- Az alábbi eszköz architektúrát javasolja:
  - **A teszt megvalósítását támogató eszköz (Test / Test Case)**
    - eszközrendszer, amellyel az xUnit keretrendszer támogatja a tesztek fejlesztését
    - ősosztályt, annotáció, függvénygyűjtemény

xUnit

- Az alábbi eszköz architektúrát javasolja:
  - **Teszt előfeltételeket támogató eszköz (Test Fixtures)**
    - a teszt futása előtt a teszteset számára megfelelő, előírt állapotba hozza az alkalmazást
    - a teszt futása után eltüntessen minden a teszt előfeltételeivel és futásával kapcsolatos változtatást

xUnit

- Az alábbi eszköz architektúrát javasolja:
  - **Tesztfutás és életciklus (Test Execution & Lifecycle**)
    - a teszt előfeltételeit, a futás által hátrahagyott módosításokat, lefoglalt erőforrásokat a teszt keretrendszer által kezelt életciklus metódusokkal lehet kezelni
    - a tesztek futásának függetlenségét biztosíthatja ezzel az eszközzel

xUnit

- Az alábbi eszköz architektúrát javasolja:
  - **Tesztgyűjtemények (Test Suites)**
    - tesztek csoportosítása
    - a csoportosítás alapját képezhetik logikai, infrastrukturális vagy környezeti tényezők
    - legtöbbször a tesztek előfeltételeinek egyezése a csoportosítás alapja

xUnit

- Az alábbi eszköz architektúrát javasolja:
  - **Teszt elvárások (Assertions)**
    - általában valamilyen logikai feltételt megvalósító függvények, melyek vagy "sikeres" vagy "elbukott" eredményt adnak
    - jelezik a programhibákat és befolyásolják a teljes teszt futásának sikerességét
    - egy teszt több elvárást is előírhat
    - ha az elvárás nem teljesült és a teszt elbukott általában valamilyen kivétellel jelzik a tesztfuttatók a keretrendszer számára

xUnit

- Az alábbi eszköz architektúrát javasolja:
  - **Tesztjelentés**
    - a teszteredményeket valamilyen formában történő elérhetővé tétele a teszt-keretrendszert futtató környezet számára
    - a jelentés formátumának testre szabására ad támogatást az xUnit megvalósítás

JUnit

- A **JUnit** egy nyílt forráskódú modulteszt rendszer, Java programok automatikus teszteléséhez
- A rendszer letölthető
  - http://junit.org
  - további hasznos információk
  - több verzió
- A JUnit rendszer használatával a programok minősége javítható, amivel hibakeresési időt takarítható meg

JUnit - verziók

Verzió	Megjelenés
JUnit 4.6	2009-04-14
JUnit 4.7	2009-07-28
JUnit 4.8	2009-12-01
JUnit 4.9	2011-08-22
JUnit 4.10	2011-09-28
JUnit 4.11	2012-11-14
JUnit 4.12	2016-04-18
JUnit 5 Milestone 1	2016-07-07
JUnit 5 Milestone 2	2016-07-23

JUnit 4.x

- **JUnit 4.x** (legelterjedtebb)
  - Java 1.5
- A Junit alap eszközei a **tesztosztály** és a **tesztmetódus**
- A tesztosztály egy POJO, melynek metódusai annotációkkal vannak ellátva
- A tesztek a junit.framework.TestCase leszármazottjai
- A tesztelő kódot metódusokban lehet megadni
  - megfelelő annotációval ellátva
  - a `@Test` annotációval ellátott tesztmetódusokat számítanak teszt metódusnak, ezek (is) lesznek végrehatjva

JUnit 5

- **JUnit 5**
  - = JUnit Platform + JUnit Jupiter + JUnit Vintage
    - JUnit Platform: teszt keretrendszer tesztek futtatásához, TestEngine API definiálása, CMD, IDE
    - JUnit Jupiter: kombinációja a „programming model” és az „extension model”-nek tesztek írásához (Runner, TestRule, MethodRule)
    - JUnit Vintage: TestEngine futtatását biztosítja
  - Java 8

Tesztek

- Egy teszt futtatása során az összes ilyen metódus végrehajtásra kerül, amelynek háromféle eredménye lehet
  - sikeres végrehajtás (pass)
  - sikertelen végrehajtás (failure)
  - hiba (error)

Tesztek

- Sikeres végrehajtás (pass)
  - a teszt rendben lefutott és az összes ellenőrzési feltétel teljesült
- Sikertelen végrehajtás (failure)
  - a teszt lefutott de valamelyik ellenőrzési feltétel nem teljesült
- Hiba (error)
  - a teszt futása során komolyabb hiba keletkezett
    - például Exception dobódott valamelyik tesztmetódus futása során

JUnit dependecia megadása

- **Maven dependency megadása példa (4.12):**

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.12</version>
   <scope>test</scope>
</dependency>
```

- **Gradle dependency megadása példa (4.12):**

```
apply plugin: 'java'

dependencies {
   testCompile 'junit:junit:4.12'
}
```

Teszt példa

```
1.  public class MessageUtil {
2.      private String message;
3.
4.      public MessageUtil(String message) {
5.          super();
6.          this.message = message;
7.      }
8.
9.      public String printMessage() {
10.         System.out.println(this.message);
11.         return this.message;
12.     }
13. }
```

Teszt példa

```
1.  import org.junit.Test;
2.  import static org.junit.Assert.assertEquals;
3.
4.  public class MessageUtilTest {
5.     String message = "Hello World";
6.     MessageUtil messageUtil = new MessageUtil(message);
7.
8.     @Test
9.     public void testPrintMessage() {
10.       assertEquals(message,messageUtil.printMessage());
11.    }
12. }
```

Teszt további példa

```
1.  import static org.junit.Assert.assertEquals;
2.  import org.junit.Test;
3.
4.  public class MyTest {

5.   @Test
6.   public void onePlusOneShouldBeTwo() {
7.      int sum = 1 + 1;
8.      assertEquals(2, sum);
9.   }
10.
11. }
```

Assert osztály metódusai

Alapvető építőelemek, amelyekkel logikai állítások fogalmazhatóak meg:

Metódus	Funkció
`assertTrue(boolean)`	Ellenőrzi, hogy a paraméter igaz-e.
`assertFalse(boolean)`	Ellenőrzi, hogy a paraméter hamis-e.
`assertNull(Object)`	Ellenőrzi, hogy a paraméter null-e
`assertNotNull(Object)`	Ellenőrzi, hogy a paraméter nem null
`assertNotSame(Object, Object)`	Ellenőrzi, hogy a két paraméter különböző objektumra mutat-e.
`assertEquals(int, int)`	Ellenőrzi, hogy a két paraméter egyenlő-e. (Az összes primitív típushoz létezik változata.)

Assert osztály metódusai

Metódus	Funkció
`assertEquals(double, double, double)`	Ellenőrzi, hogy a két double paraméter egyenlő-e a megadott tolerancián belül. (A tolerancia a harmadik double paraméter.)
`assertEquals(Object, Object)`	Ellenőrzi, hogy a két paraméter egyenlő-e. (Az equals() metódust használva.)
`fail()`	A tesztet azonnal sikertelenné teszi(megbuktatja).

JUnit annotációk

Annotáció	Funkció
`@Test public void method()`	egy metódust tesztmetódust jelöl meg
`@Test(expected = Exception.class) public void method()`	elvárt kivétel megadása, a teszt elbukik ha nem dobódik az elvárt kivétel
`@Test(timeout=100) public void method()`	teszt elbukik, ha a végrehajtási idő 100 ms-nál hosszabb
`@Before public void method()`	teszteseteket inicializáló metódus, minden teszteset előtt lefut, feladata a tesztkörnyezet előkészítése
`@After public void method()`	minden egyes teszteset végrehajtása után lefut, feladata az ideiglenes adatok törlése

JUnit annotációk

Annotáció	Funkció
`@BeforeClass public static void method()`	egyszer fut le, még az összes teszteset és a hozzájuk kapcsolódó @Before-ok végrehajtása előtt. Itt lehet egyszeri inicializálós lépéseket megadni, pl. adatbázis-kapcsolat kiépítése. Ilyen annotációval ellátott metódusnak statikusnak kell lennie!
`@AfterClass public static void method()`	egyszer fut le, miután az összes tesztmetódus, és a hozzájuk tartozó @After metódusok végrehajtása befejeződött. Egyszeri tevékenységet helyezünk ide, amely a @BeforeClass metódusban lefoglalt erőforrások felszabadítását végzi el. Az ilyen annotációval ellátott metódusnak statikusnak kell lennie!

JUnit annotációk

Annotáció	Funkció
`@Ignore`	Figyelmen kívül hagyja a tesztmetódust, vagy a tesztosztályt. Olyankor használható, ha megváltozott a tesztelendő kód, de a tesztesetet még nem frissítettük vagy abbaz az esetben ha a teszt végrehajtása túl hosszú ideig. Ha osztály szinten van definiálva, akkor az osztály összes tesztmetódusát figyelmen kívül hagyja

JUnit annotációk példa

```
1.  public class Annotations {
2.      private Collection<String> collection;
3.
4.
5.    @BeforeClass
6.    public static void oneTimeSetUp() {
7.        System.out.println("@BeforeClass - oneTimeSetUp");
8.    }
9.
10.    @AfterClass
11.    public static void oneTimeTearDown() {
12.        System.out.println("@AfterClass - oneTimeTearDown");
13.    }
14.
15.    @Before
16.    public void setUp() {
17.        collection = new ArrayList();
18.        System.out.println("@Before - setUp");
19.    }
```

JUnit annotációk példa kód

```
20. @After
21.     public void tearDown() {
22.         collection.clear();
23.         System.out.println("@After - tearDown");
24.     }
25.
26.     @Test
27.     public void testEmptyCollection() {
28.         assertTrue(collection.isEmpty());
29.         System.out.println("@Test - testEmptyCollection");
30.     }
31.
32.     @Test
33.     public void testOneItemCollection() {
34.         collection.add("itemA");
35.         assertEquals(1, collection.size());
36.         System.out.println("@Test - testOneItemCollection");
37.     }
38.
39.}
```

JUnit annotációk példa kód

Futás eredménye:

```
@BeforeClass - oneTimeSetUp
@Before - setUp
@Test - testOneItemCollection
@After - tearDown
@Before - setUp
@Test - testEmptyCollection
@After - tearDown
@AfterClass - oneTimeTearDown
```

Parametrizált tesztek

- **JUnit 4-ben jelentek meg**
- Célja
  - ugyanazon tesztesetek többszöri lefuttatási de más-más paraméterekkel
  - teszt kód és a teszt adatok szétválasztása
- @RunWith(Parameterized.class) annotációval
- Példa:
  - metódus mai összead 2 számot

```
1.public class Addition {
2.    public int addNumbers(int x, int y) {
3.        return x + y;
4.    }
5.}
```

Parametrizált tesztek

- **A tesztosztály** @RunWith(Parameterized.class) annotációval való ellátása

```
1.@RunWith(Parameterized.class)
2.
3.public class AdditionTest {
4.    private int expected;
5.    private int first;
6.    private int second;
7.
8.       …
9.}
```

Parametrizált tesztek

- Konstruktor, amely egy tesztadatok információit képes befogadni

```
1.public AdditionTest(int expected, int first, int second) {
2.        this.expected = expected;
3.        this.first = first;
4.        this.second = second;
5.    }
```

Parametrizált tesztek

- @Parameters annotációval ellátott publikus statikus metódus, ami egydimenziós tömbök kollekcióját adja vissza
- egy-egy tömb ebben a kollekcióban az egyes tesztvégrehajtások során használt adatokat tartalmazza
- mérete azt mondja meg, hogy hányszor kell majd az egyes teszteseteket futtatni
- tömbök azonos elemszámúak, és pont annyi amennyi a konstruktor paramétereinek a száma

```
1.@Parameters
2.  public static Collection<Integer[]> addedNumbers() {
3.      return Arrays.asList(new Integer[][] {{3, 1, 2}, {5, 2, 3},
{7, 3, 4}, {9, 4, 5}});
4.  }
5.
```

Parametrizált tesztek

- Tesztmetódus létrehozása

```
1.@Test
2.    public void sum() {
3.        Addition add = new Addition();
4.        assertEquals(expected, add.addNumbers(first, second));
5.    }
```

Parametrizált tesztek

- Esetleg futtatható osztály létrehozása

```
1.import org.junit.runner.JUnitCore;
2.import org.junit.runner.Result;
3.import org.junit.runner.notification.Failure;
4.
5.public class AdditionTestRunner {
6.    public static void main(String[] args) {
7.        Result result = JUnitCore.runClasses(AdditionTest.class)
;
8.        for (Failure fail : result.getFailures())
9.            System.err.println(fail.toString());
10.         if (result.wasSuccessful())
11.             System.out.println("Az összes teszt sikeresen lefut
tatva!");
12.    }
13.}
```

Kivételek tesztelése

- Előfordulhatnak olyan esetek mikor a normál működéshez tartozik, hogy kivétel keletkezzen
  - Push művelet: ha megtelt a verem akkor FullStackException dobása a metódsuban
  - Vagy akár saját kivétel dobása ha adott összeg nem emelhető le a bankkártyáról
  - Hozzátartozik a normál működéshez, adott kivétel keletkezését várjuk
  - Kivétel keletkezése esetén a teszt elbukhat

Kivételek tesztelése

- De ha az a cél, hogy kivétel keletkezzen akkor ezt jelezni kell
  - @Test(expected=xxxException.class)
  - pl.: @Test(expected=FullStackException.class)

- Ebben az esetben akkor bukik el a teszt ha nem keletkezik az adott elvárt kivétel

Kivételek tesztelése

- Ha a kivétel üzenetének tartalmát akarjuk tesztelni, vagy a kivétel várt kiváltódásának helyét akarjuk szűkíteni akkor a következő a forgatókönyv:
  - 1. kapjuk el a kivételt
  - 2. Fail metódus használata, ha egy adott pontra nem volna szabad eljutni
  - 3. a kivételkezelőben nyerjük ki a kivétel szövegét majd hasonlítsuk az elvárt szöveghez

Kivételek tesztelése

```
1.public void testException() {
2.    try {
3.       exceptionCausingMethod();
4.       // Ha eljutunk erre a pontra, a várt kivétel nem váltó
dott ki, ezért megbuktatjuk a tesztesetet.
5.       fail("Kivételnek kellett volna kiváltódnia");
6.    }
7.    catch(ExceptedTypeOfException exc) {
8.       String expected = "Megfelelő hibaüzenet";
9.       String actual = exc.getMessage();
10.       Assert.assertEquals(expected, actual);
11.    }
12.}
```

Tesztkészlet

- **Tesztkészlet (test suite)**
- Összetartozó, együttesen végrehajtandó teszteseteket összesége
- Hasznos mikor az áttekinthetőség miatt a tesztesek több különálló osztályban vannak kiszervezve
- A különböző tesztosztályokban lévő teszteket mégis egyszerre szeretnénk futtatni
- Test suit definiálása
  - @RunWith(Suite.class)
  - @Suite.SuiteClasses

Tesztkészlet példa

```
1.  import org.junit.runner.RunWith;
2.  import org.junit.runners.Suite;
3.
4.  @RunWith(Suite.class)
5.
6.  @Suite.SuiteClasses({
7.      TestJunit1.class,
8.      TestJunit2.class
9.  })
10.
11. public class JunitTestSuite {
12. }
```

JUnit antiminták

- A JUnit eszközeinek segítségével hatékonyan, inkrementális módon van lehetőség tesztkészlet fejlesztésére
  - előrehaladás mérése
  - nem várt mellékhatások kiszűrése

- A megírt tesztek által biztosítható a szoftver minősége, hibamentessége
  - de csak akkor ha az adott egységtesztek jól vannak megírva
  - meg kell vizsgálni, hogy melyek azok a tevékenységek amik a leggyakoribb hibákat okozzák az egységtesztek implementálása során
  - -> Antiminták

JUnit antiminták – rosszul kezelt állítások

- A Junit tesztek építőelemei az állítások (assertion)
  - logikai kifejezés
  - ha hamis akkor az hibát jelent
  - hiba ezzel kapcsolatban: kézi ellenőrzés
    - sok utasítás a kódban de állítás egy sem -> kézi debug
    - hiányzó állítások
    - ezek az estek kerülendők
  - Másik gond: ha túl sok állítás van
    - tesztmetódus túl sokat tesztel
      - összekeveredik a funkcionalitást tesztelő kód és az elvárt eredmények ellenőrzését végző kód
    - megoldás: tesztmetódus szétdarabolása

JUnit antiminták – rosszul kezelt állítások

- Kerülendő:

```
1.public class MyTestCase {
2.
3. @Test
4. public void testSomething() {
5.    // Set up for the test, manipulating local variables
6.    assertTrue(condition1);
7.    assertTrue(condition2);
8.    assertTrue(condition3);
9.  }
10.}
```

JUnit antiminták – rosszul kezelt állítások

- Javasolt:

```
1.public class MyTestCase {
2.  // Local variables become instance variables
3.
4.  @Before
5.  protected void setUp() {
6.    // Set up for the test, manipulating instance variables
7.  }
8.
9.   @Test
10.  public void testCondition1() {
11.    assertTrue(condition1);
12.  }
13.
14.  @Test
15.  public void testCondition2() {
16.    assertTrue(condition2);
17.  }
18.
19.  @Test
20.  public void testCondition3() {
21.    assertTrue(condition3);
22.  }
23.}
```

JUnit antiminták – rosszul kezelt állítások

- Redundáns feltételek kerülése
  - Assert metódus amiben a feltétel beégetett módon True

```
1.@Test
2.public void testSomething() {
3.     ...
4.     assertTrue("...", true);
5.}
```

JUnit antiminták – rosszul kezelt állítások

- Rossz állítás alkalmazása
  - Assert osztálynak sok metódusa van
  - csakis az AssertTrue alkalmazása: kerülendő

```
1.assertTrue("Objects must be the same", expected == actual);
2.assertTrue("Objects must be equal", expected.equals(actual));
3.assertTrue("Object must be null", actual == null);
4.assertTrue("Object must not be null", actual != null);
```

  - Helyette:

```
1.assertSame("Objects must be the same", expected, actual);
2.assertEquals("Objects must be equal", expected, actual);
3.assertNull("Object must be null", actual);
4.assertNotNull("Object must not be null", actual);
```

JUnit antiminták – felszínes lefedettség

- Alapvető tesztelő kód
  - csak a rendszer elvárt viselkedés kerül tesztelésre
  - érvényes adatokat megadva az elképzelt helyes eredmény ellenében történik az ellenőrzés
  - hiányoznak a kivételes esetek vizsgálatai
    - mi történik hibás bemeneti adatok esetén
    - az elvárt kivételek eldobásra kerültek-e
    - melyek az érvényes és érvénytelen adatok ekvivalenciaosztályainak határai, stb.
  - elkerülés: tesztlefedettség-mérő eszköz alkalmazásával
    - Pl. **EclEmma** (Java, Eclipse plugin)

JUnit antiminták – túlbonyolított tesztek

- Az egységteszt kódjának könnyen érthetőnek kell lennie
- Ha egy egységteszt bonyolult akkor nehéz megállapítani, hogy
  - a teszt bukása a tesztkód miatt vagy a tesztelendő kód miatt történt-e
- Másik gond: a kód úgy megy át a teszten, hogy nem lett volna szabad

- Következő lépések betartása:
  - Inicializálás (set up)
  - Az elvárt eredmények deklarálása
  - A tesztelendő egység meghívása
  - A tevékenység eredményeinek beszerzése
  - Állítás megfogalmazása az elvárt és a tényleges eredményről

JUnit antiminták – külső függőségek

- Számos külső függőség lehetséges, a kód helyes működése függhet:
  - dátum, idő
  - third-party jar formátumú programkönyvtártól
  - állománytól
  - adatbázistól
  - hálózattól, stb.
- Minél több függőség, annál nehezebb igazolni a helyes működést
- Többletmunka
  - Szerver konfigurálása
  - Adatbázis kapcsolat létrehozása
  - Stb.
- Ökölszabály, hogy a külső függőségeket el kell kerülni ha lehet

JUnit antiminták – külső függőségek

- Az alábbi módszerek alkalmazhatók
  - mock objektumok használata
  - tesztadatok a tesztkóddal együtt legyenek csomagolva
  - adatbázishívások elkerülése
    - Ha muszáj adatbázishívás akkor memóriában tárolt adatbázis használata (HSQLDB)

JUnit antiminták – nem várt kivételek elkapása

- Rossz, akkor is átmegy ha kivétel keletkezett
  - kerülendő

```
1.@Test
2.public void testCalculation () {
3.    try  {
4.        deepThought.calculate();
5.        assertEquals("Calculation wrong", 42, deepThought.getResult());

6.    } catch (CalculationException ex) {
7.        Log.error("Calculation caused exception", ex);
8.    }
9.}
```

JUnit antiminták – nem várt kivételek elkapása

- Elbukik ha kivétel keletkezett

```
1.@Test
2.public void testCalculation() {
3.    try {
4.        deepThought.calculate();
5.        assertEquals("Calculation wrong", 42, deepThought.getResult());

6.    }
7.    catch(CalculationException ex) {
8.        fail("Calculation caused exception");
9.    }
10.}
```

JUnit antiminták – nem várt kivételek elkapása

- Megoldás
  - kivétel továbbadása a JUnit-nak

```
1.@Test
2.public void testCalculation() throws CalculationException {
3.     deepThought.calculate();
4.     assertEquals("Calculation wrong", 42, deepThought.getResult());

5.}
```

JUnit antiminták – nem várt kivételek elkapása

- Ha azt kell igazolni, hogy adott kivétel keletkezik:

```
1.@Test
2.public void testIndexOutOfBoundsException() {
3.    try {
4.        ArrayList emptyList = new ArrayList();
5.        Object o = emptyList.get(0);
6.        fail("Exception was not thrown");
7.    }
8.    catch(IndexOutOfBoundsException ex) {
9.        // Success!
10.    }
11.}
```

- Vagy az expected paraméter használata:
  - `@Test(expected = IndexOutOfBoundsException.class)`

JUnit antiminták – éles és a teszt kód keveredése

- Nehéz megkülönböztetni a tesztelendő és a teszt kódot:

```
src/
 com/
   xyz/
     SomeClass.java
     SomeClassTest.java
```

```
src/
 com/
   xyz/
     SomeClass.java
     test/
        SomeClassTest.java
```

- Helyes:

```
src/
  com/
    xyz/
      SomeClass.java
test/
  com/
    xyz/
      SomeClassTest.java
```

JUnit antiminták – nem létező egységtesztek

- Tesztek hiánya
- Mindenki tudja, hogy teszteket szükséges lenne írni de kevesen teszik
- Minél nagyobb nyomás -> annál kevesebb teszt -> kód stabilitása csökken -> nagyobb nyomás…
- Teszteket írni muszáj
  - ne csak a végfelhasználó „tesztelje”
  - tesztelés növeli a szoftverminőséget
  - bizonyítja, hogy az API használható

Köszönöm a figyelmet!