JUnit

- Egy egységtesztelő keretrendszer Javához
  - Szerzők: Erich Gamma, Kent Beck
- Célkitűzés:
  - "Ha a teszteket könnyű létrehozni és lefuttatni, akkor a programozók jobban hajlanak majd arra, hogy teszteket hozzanak létre és futtassanak le."

Bevezetés

- Mire van szükségünk az automatizált teszteléshez?
  - Teszt szkript
    - A műveletek, melyeket el kell küldeni a tesztelt rendszernek (TR).
    - A TR-től várt válaszok.
    - Hogyan döntsük el, hogy egy teszt sikeres volt-e vagy nem?
  - Teszt végrehajtó rendszer
    - Egy mechanizmus, ami beolvassa a teszt szkripteket, és a TR-hez köti a teszteseteket.
    - Nyomon követi a tesztek eredményeit.

Tesztesetek döntései (verdiktek)

- Egy verdikt egyetlen teszt végrehajtásának meghatározott eredménye.
- Átment (pass): a teszteset elérte kijelölt célját, a tesztelt szoftver pedig a vártnak megfelelően viselkedett.
- Megbukott (fail): a teszteset elérte kijelölt célját, de a szoftver nem a vártnak megfelelően viselkedett.
- Hiba (error): a teszteset nem érte el kijelölt célját.
  - Ennek lehetséges okai:
    - A teszteset során egy nem várt esemény következett be.
    - A tesztesetet nem lehetett megfelelően beállítani.

Egy megjegyzés a JUnit verziókról...

- A jelenlegi verzió a 4.3.1, ami 2007 márciusától elérhető
  - A JUnit 4.x verziók használatához Java 5 vagy 6 verziót kell használni
- A 2006 áprilisában bemutatott Junit 4 komoly (azaz nem kompatibilis) változást hozott a korábbi verziókhoz képest.
- Ebben a prezentációban JUnit 4-et használunk.
- A jelenleg elérhető JUnit dokumentáció nagy része még a JUnit 3-hoz készült, ami egy kicsit más.
  - A JUnit 3 használható a Java korábbi verzióihoz (pl. 1.4.2.)
  - A junit.org weboldal a JUnit 4-es verziójáról ad információt, ha csak nem kéred a régebbi verziót.
  - Az Eclipse 3.2-ben választhatsz a JUnit 3.8 és a JUnit 4.1 használata között, mindkettő megtalálható benne.

Mi az a JUnit Teszt?

- Egy teszt „szkript” csupán Java metódusok egy csoportja.
  - Az alapötlet az, hogy hozzunk létre néhány Java objektumot, csináljunk velük valami érdekeset, és döntsük el, hogy a megfelelő tulajdonságokkal rendelkeznek-e.
- Mit adunk meg? Assertion-öket.
  - Olyan metódusok egy csoportját, melyek különböző tulajdonságokat ellenőriznek:
    - Objektumok „egyenlőségét”
    - Egyező objektumhivatkozásokat
    - Null / nem null objektumhivatkozásokat
  - Az assertion-öket használjuk a tesztesetek verdiktjeinek meghatározásához.

Mikor helyes alkalmazni a Junit-ot?

- Mint ahogy arra a név is utal…
  - Kis mennyiségű kód egységtesztéléséhez (unit test)
- Önmagában nem elegendő komplex teszteléshez, rendszertesztéléshez, stb.

- A tesztalapú fejlesztési módszertanban a JUnit tesztet kell először megírni (még a kód előtt), és lefuttatni.
  - Ezután kell megírni az implementációs kódot, ami nem lesz más, mint a minimális kód, ami átmegy a teszten – semmi egyéb funkcionalitás.
  - Ha megírtuk a kódot, futtassuk le a tesztet újra, és át kell mennie.
  - Valahányszor új kódot adunk hozzá, futtassuk le újra az összes tesztet, így megbizonyosodva róla, hogy semmi nem romlott el.

Egy JUnit 4 Teszteset

```
/** A setName() metódus tesztje a Value osztályban */

    @Test
    public void createAndSetName()
    {
        Value v1 = new Value( );

        v1.setName( "Y" );

        String expected = "Y";
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
    }
```

Egy JUnit 4 Teszteset

```
/** A setName() metódus tesztje a Value osztályban */
    @Test
    public void
    {
        Value v1 = new Value( );

        v1.setName( "Y" );

        String expected = "Y";
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
    }
```

Jelzi a tesztfuttató számára, hogy ez a Java metódus egy teszteset

Egy JUnit 4 Teszteset

```
/** A setName() metódus tesztje a Value osztályban */

    @Test
    public void createAndSetName()
    {
        Value v1 = new Value(

        v1.setName( "Y" );

        String expected = "Y"
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
    }
```

Célkitűzés:
Erősítsük meg, hogy a **setName** elmenti a megadott nevet a **Value** objektumba

Egy JUnit 4 Teszteset

```
/** A setName() metódus tesztje a Value osztályban */

    @Test
    public void createAndSetName()
    {
        Value v1 = new Value( );

        v1.setName( "Y" );

        String expected = "Y"
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
    }
```

Ellenőrizzük,hogy a **Value** objektum tényleg elmentette-e a nevet

Egy JUnit 4 Teszteset

```
/** A setName() metódus tesztje a Value osztályban */

    @Test
    public void createAndSetNam
    {
        Value v1 = new Value( )

        v1.setName( "Y" );

        String expected = "Y";
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
    }
```

Azt akarjuk, hogy az `expected` és az `actual` egyenlő legyen.

Ha ez nincs így, a tesztesetnek meg kell buknia.

Assertion-ök

- Az állítások (assertion) az `Assert` JUnit osztályban vannak definiálva
  - Ha egy állítás igaz, folytatódik a metódus végrehajtása.
  - Ha bármely állítás hamis, a metódus végrehajtása megszakad azon a ponton, és a teszteset eredménye megbukott (fail) lesz.
  - Ha bármilyen más kivétel dobódik a metódus végrehajtása során, a teszteset eredménye hiba (error) lesz.
  - Ha egyetlen állítás sem sérül az egész metódusban, akkor a teszteset átmegy (pass).
- Minden assertion metódus statikus.

Assertion metódusok (1)

- A Boolean típusú feltételek értéke igaz vagy hamis

  `assertTrue(condition)`

  `assertFalse(condition)`

- Az objektumok értéke null vagy nem null

  `assertNull(object)`

  `assertNotNull(object)`

- Az objektumok egyformák (azaz két hivatkozás ugyanarra az objektumra), vagy nem egyformák.

  `assertSame(expected, actual)`

  - Igaz ha: `expected == actual`

  `assertNotSame(expected, actual)`

Assertion metódusok (2)

- Objektumok „egyenlősége”:

  `assertEquals(expected, actual)`

  - Igaz ha: `expected.equals( actual )`

- Tömbök „egyenlősége”:

  `assertArrayEquals(expected, actual)`

  - A tömbök hossza meg kell, hogy egyezzen
  - Minden létező `i` értékre megfelelően ellenőrizzük a következőt:

    `assertEquals(expected[i],actual[i])`

    vagy

    `assertArrayEquals(expected[i],actual[i])`

- Létezik egy feltétel nélküli bukás assertion is, a `fail()`, ami mindig bukás verdiktet eredményez.

Assertion metódusok paraméterei

- Bármely kétparaméterű assertion metódus esetén az első paraméter a várt érték, a második pedig a valós érték.
  - Ez nincs hatással a hasonlításra, de ezt a sorrendet feltételezzük a felhasználónak szóló hibaüzenet létrehozásakor.
- Minden assertion metódusnak lehet egy további `String` paramétere, első paraméterként. Ez a sztring bekerül a hibaüzenetbe, ha az állítás megbukik.
  - Példák:

```
fail( message )
assertEquals( message, expected, actual)
```

Egyenlőségi állítások

- Az `assertEquals(a,b)` a tesztelendő osztály `equals()` metódusára hagyatkozik.
  - A hatása a `a.equals( b )` kifejezés kiértékelése.
  - A tesztelendő osztályra van bízva, hogy megfelelő egyenlőségi relációt definiáljon. A JUnit azt használja, ami rendelkezésre áll.
  - Bármely olyan osztály esetén, amely nem definiálja felül az `Object` osztály `equals()` metódusát, az alapértelmezett `equals()` viselkedés érvényesül – az objektumok azonossága.
- Ha `a` és `b` típusa egy olyan primitív típus, mint pl. az `int` vagy a `boolean`, akkor az `assertEquals(a,b)`-nél a következő történik:
  - `a` és `b` az ekvivalens objektumtípusra konvertálódik (`Integer`, `Boolean`, stb.), majd kiértékelődik az `a.equals( b )` kifejezés.

Lebegőpontos állítások

- Lebegőpontos típusok (`double` vagy `float`) hasonlításánál egy további, `delta` paraméterre is szükség van.
- Az állítás a

  ```
  Math.abs( expected - actual ) <= delta
  ```

  kifejezést értékeli ki, hogy kiküszöbölje a lebegőpontos hasonlításoknál a kerekítési hibák okozta problémákat.
- Példa:

  ```
  assertEquals( aDouble, anotherDouble, 0.0001 )
  ```

A JUnit tesztek szerveződése

- Minden metódus egyetlen tesztesetet reprezentál, melynek önálló verdiktje van (átment, hiba, megbukott).
- Normális esetben minden, egy Java osztályhoz tartozó teszt egy, különálló osztályba van csoportosítva.
  - Elnevezési konvenció:
    - A tesztelendő osztály: `Value`
    - A teszteket tartalmazó osztály: `ValueTest`

JUnit Tesztek futtatása (1)

- A JUnit keretrendszer nem biztosít grafikus tesztfuttatót. Ehelyett biztosít egy API-t, amit az IDE-k használhatnak tesztesetek futtatásához, és egy szöveges futtatót, amit használhatunk a parancssorból.
- Az Eclipse és a Netbeans is biztosít a környezetükbe integrált grafikus tesztfuttatót.

JUnit tesztek futtatása (2)

- A JUnit által biztosított futtatóval:
  - Amikor kiválasztunk egy osztályt végrehajtásra, az osztály összes teszteset-metódusa futtatásra kerül.
  - A sorrend, amelyben a metódusok meghívásra kerülnek (azaz a tesztesetek végrehajtásának sorrendje), nem megjósolható.
- Az IDE-k tesztfuttatói lehetőséget biztosíthatnak a felhasználónak arra, hogy kiválasszon bizonyos metódusokat, vagy hogy meghatározza a végrehajtás sorrendjét.
- Jó módszer, ha a futtatási sorrendtől, és bármely korábbi teszt(ek) állapotától független teszteket írunk.

Tesztek környezete (fixture)

- Egy teszt környezete az a kontextus, amelyben a teszteset lefut.
- Jellemzően a következőek tartoznak hozzá egy teszt környezetéhez:
  - Objektumok és erőforrások, amelyek használatra elérhetőek bármely teszteset számára.
  - Az ezen objektumok elérhetővé tételéhez és/vagy erőforrások allokálásához és felszabadításához szükséges tevékenységek: „setup" és „teardown"

Setup és Teardown

- Egy konkrét osztályhoz tartozó tesztek egy csoportjához gyakran szükség van ismétlődő tevékenységekre, amelyeket minden teszteset előtt el kell végezni.
  - Példák: hozzunk létre néhány „érdekes” objektumot, melyekkel dolgozhatunk, nyissunk meg egy hálózati kapcsolatot, stb.
- Ehhez hasonlóan minden teszteset végén is szükség lehet ismétlődő feladatokra, amelyek „eltakarítanak” a teszt futása után.
  - Biztosítja az erőforrások felszabadítását, a következő tesztesethez a tesztrendszer ismert állapotban lesz, stb.
  - Mivel egy teszteset bukása véget vet az adott ponton a tesztelő metódus futásának, a takarítókód nem lehet a metódus végén.

Setup és Teardown

- Setup:
  - Használjuk a **`@Before`** annotációt egy metóduson, ami a minden teszteset előtt lefuttatandó kódot tartalmazza.
- Teardown (verdikttől függetlenül):
  - Használjuk az **`@After`** annotációt egy metóduson, ami a minden teszteset után lefuttatandó kódot tartalmazza.
  - Ezek a metódusok akkor is lefutnak, ha kivétel dobódik vagy ha egy állítás elbukik a tesztesetben.
- Ezen annotációkból bármennyit használhatunk.
  - Minden **`@Before`** annotációval ellátott metódus lefut minden teszteset előtt, de futásuk bármely sorrendben történhet.

Példa: Használjunk egy fájlt szöveges környezetként

```
public class OutputTest
{
    private File output;

    @Before public void createOutputFile()
    {
       output = new File(...);
    }

    @After public void deleteOutputFile()
    {
        output.delete();
    }

    @Test public void test1WithFile()
    {
       // code for test case objective
    }

    @Test public void test2WithFile()
    {
       // code for test case objective
    }
}
```

A metódusok futtatási sorrendje

1. `createOutputFile()`
2. `test1WithFile()`
3. `deleteOutputFile()`
4. `createOutputFile()`
5. `test2WithFile()`
6. `deleteOutputFile()`

- Feltételezzük, hogy a `test1WithFile` a `test2WithFile` előtt fut le, ami nem biztosított.

Egyszeri setup

- Az is lehetséges, hogy egy metódust egyetlen egyszer fusson le egy egész osztályhoz, bármely teszt lefutása előtt, és bármely `@Before` metódus előtt.
- Ez hasznos lehet szerverek elindításához, kommunikáció megnyitásához, stb., melyek lezárása és újranyitása minden tesztnél nagyon időigényes lenne.
- Ezt a `@BeforeClass` annotációval jelöljük (csak egy metódushoz használható, mely static kell, hogy legyen):

```
@BeforeClass public static void anyNameHere()
{
    // class setup code here
}
```

Egyszeri tear down

- Ennek megfelelően egy egyszeri tisztító metódus is alkalmazható. Ez azután kerül futtatásra, hogy az osztály minden teszteset metódusa, és bármely `@After` metódus végrehajtódott.
- Használható szerverek leállításához, kommunikációs linkek lezárásához, stb.
- Ezt az `@AfterClass` annotációval jelöljük (csak egy metódushoz használható, mely static kell, hogy legyen):

```
@AfterClass public static void anyNameHere()
{
    // class cleanup code here
}
```

Kivételtesztelés (1)

- Adjunk hozzá egy paramétert a `@Test` annotációhoz, ami arra utal, hogy egy meghatározott osztályú kivétel váltódhat ki a teszt során.

```
@Test(expected=ExceptedTypeOfException.class)
public void testException()
{
    exceptionCausingMethod();
}
```

- Ha nem váltódik ki kivétel, vagy nem várt kivétel váltódik ki, a teszt elbukik.
  - Azaz ha kivétel nélkül jutunk el a metódus végére, a teszteset megbukik.
- Ha a kivétel üzenetének tartalmát akarjuk tesztelni, vagy a kivétel várt kiváltódásának helyét akarjuk szűkíteni, ahhoz a következő dián látható megközelítés szükséges.

Kivételtesztelés (2)

- Kapjuk el a kivételt, és használjuk a `fail( )` -t ha nem váltódik ki.

```
public void testException()
{
    try
    {
        exceptionCausingMethod();

        // Ha eljutunk erre a pontra, a várt
        // kivétel nem váltódott ki.

        fail(„Kivételnek kellett volna kiváltódnia");
    }
    catch ( ExceptedTypeOfException exc )
    {
        String expected = „Megfelelő hibaüzenet";
        String actual = exc.getMessage();
        Assert.assertEquals( expected, actual );
    }
}
```

JUnit 3

- Korábban említetteknek megfelelően JUnit 3 és 4 nem kompatibilis.
- A JUnit archívumon belül a következő két csomagot alkalmazzák, hogy a két verzió párhuzamosan létezhessen:
  - JUnit 3: `junit.framework.*`
  - JUnit 4: `org.junit.*`