JUnit

Java technológia

Sipos Róbert
doktorjelölt
BME Híradástechnikai Tanszék
siposr@hit.bme.hu

2014. 04. 29.

Tartalom

- Teszt vezérelt fejlesztés, egységteszt
- JUnit telepítése, használata
- Fontosabb osztályok leírása
  - Tesztesetek készítése
  - Tesztelési technikák bemutatása

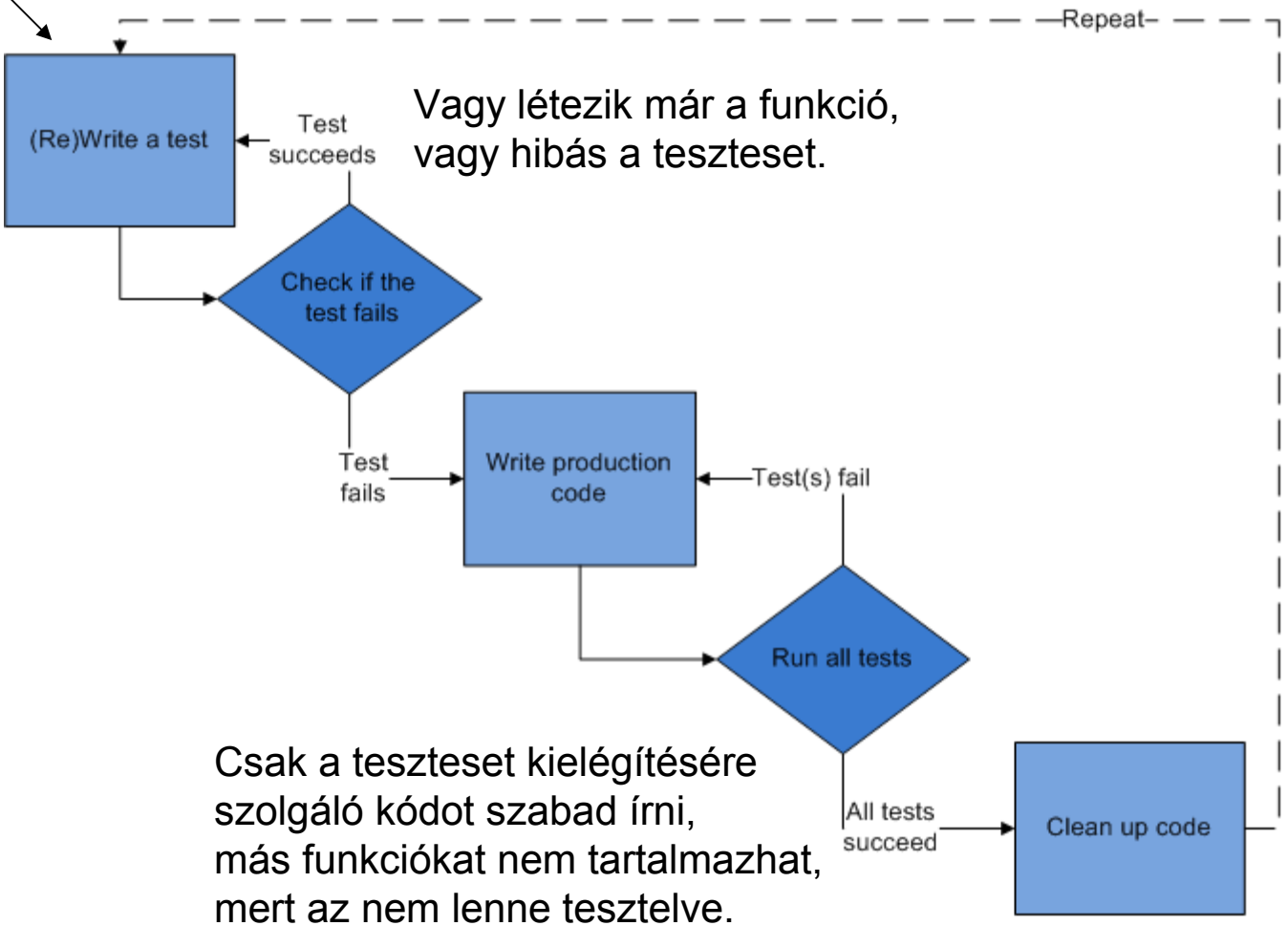
Teszt vezérelt fejlesztés (TDD)

- Szoftver fejlesztési ciklus, ahol először az egységek tesztjét kell elkészíteni, és csak ezt követően magát a kódot.
  - Csak akkor írhatunk új kódot, amíg nem működik minden teszteset, különben előbb a tesztje kell.
  - Tesztelés az érintett kód minden változását követően → a regressziós tesztelés biztonságot nyújt.
  - Rövid ciklusokból áll, nem a teljes termék tesztjét kell előre elkészíteni, csak az egyes rész-funkciókét.

Teszt vezérelt fejlesztés (TDD)

Új *feature* → specifikáció
(követelmények, elvárt viselkedés)

Fontos, hogy így mindenképpen a követelmények teljes ismeretében készül csak kód.



Vagy létezik már a funkció,
vagy hibás a teszteset.

Csak a teszteset kielégítésére
szolgáló kódot szabad írni,
más funkciókat nem tartalmazhat,
mert az nem lenne tesztelve.

Forrás: Wikipedia

Teszt vezérelt fejlesztés (TDD)

- Előnyök
  - Növeli a produktivitást
  - Segít a valós kód tervezése során
    - A nehezen tesztelhető kód rossz tervezést sejtet
    - Kisebb, lazábban csatolt modulokat és tisztább interfészeket eredményez
  - Mivel csak olyan kódot írunk, ami egy teszteset sikerességéhez szükséges, így a kód lefedettsége (*code coverage*) elvileg teljes lesz
  - Lehetőséget ad arra, hogy a kódot és a teszteket más programozó készítse el
    - Nem követi el ugyanazt a hibát
    - Könnyebb ütemezés
  - Kisebb lépésekben lehet haladni
  - Növeli a biztonságérzetet

Teszt vezérelt fejlesztés (TDD)

- Hátrányok
  - Projekt menedzsment szintű támogatás szükséges
  - Bizonyos esetekben a teszt bonyolultsága megegyezhet a tesztelt funkcióéval (pl. összetett numerikus számítások)
  - A követelmények nem mindig ismertek előre

Egységtesztelés

- Az egyes egységeket teszteljük a specifikációval szemben.
- Külön teszteset minden nem-triviális metódushoz.
- Fontos, hogy mindegyik önállóan, a többitől különválasztva készüljön el.
- Kérdés: egység = osztály vagy metódus(ok)?
- Az egységtesztelést követi az integrációs teszt, majd a rendszerteszt.

„Hagyományos” módszerek

- `System.out.println(a);`
  - lábbal-hajtós
  - „teleszemeteli” a forráskódot
  - manuális ellenőrzést igényel minden futtatás során
  - esetenként túl hosszú kimenet
- Debugger
  - Szintén manuális összehasonlítását igényli az elvárt és a kapott értékeknek
  - A kód minden változtatása után lépésről lépésre végig kell haladni
- Ezeknek a célja eltérő: egy ismert hiba megoldásában nyújthatnak segítséget, nem egy ismeretlen hiba felfedezésében.

JUnit

- Tesztelő keretrendszer
- Kifejezetten egységtesztelés céljára
- Lehetővé teszi a TDD alkalmazását
- Külső, fordítási időben csatolt JAR (`org.junit`)
- Jelenlegi legfrissebb stabil verzió: 4.11
- © Kent Beck, Erich Gamma és David Suff
- Eredetileg Smalltalk nyelvhez (SUnit)
  - Később portolták Java-ra
  - C/CUnit, C++/CPPUnit, C#/NUnit, PHP/PHPUnit, …
  - Összefoglaló néven: xUnit

Telepítés, használat

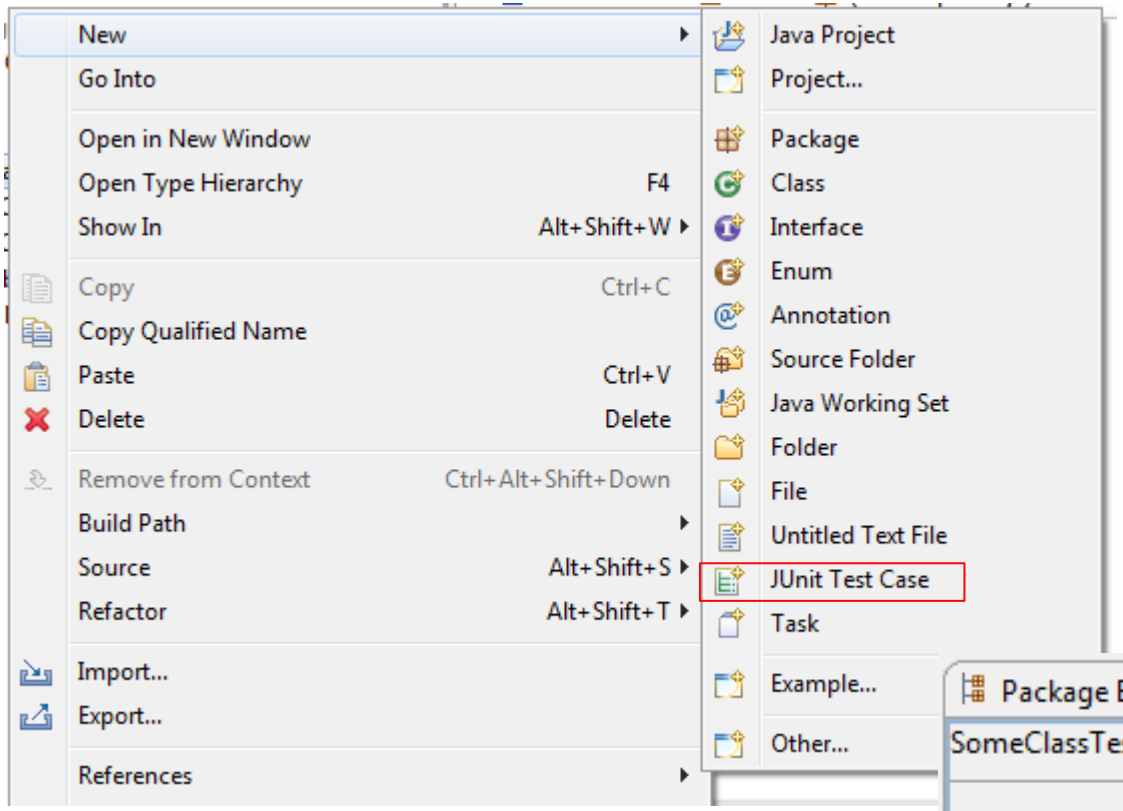
1. Le kell tölteni a JAR fájlt a JUnit oldaláról.
2. Hozzá kell adni a CLASSPATH-hoz.
   - Eclipse
     Project / Properties / Java Build Path / Libraries / Add JARs
   - Parancssor

```
set CLASSPATH=%CLASSPATH%;
%JUNIT_HOME%\junit.jar
```

   vagy

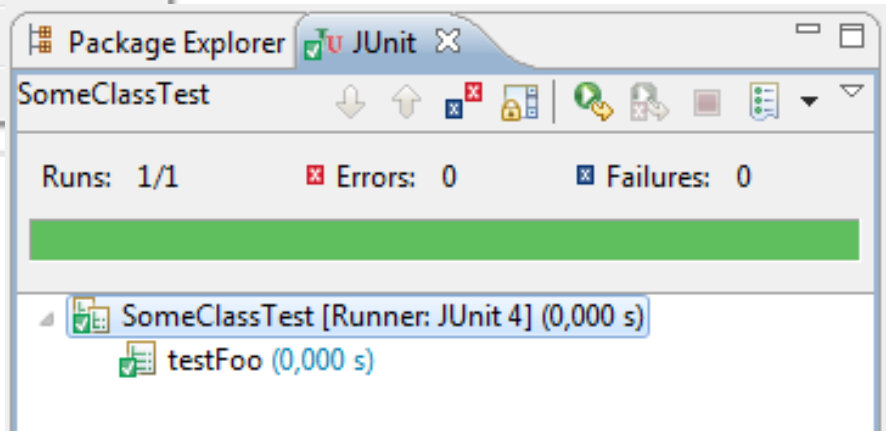
   a `javac` és `java` parancs után `-cp` vagy `-classpath` kapcsolóval

Eclipse IDE integráció



Teszteset hozzáadása:

Futtatás eredménye:



@Test annotáció

- A `@Test` annotáció jelzi, hogy egy tesztesetet implementáló metódus következik:

```
@Test
public void myTest() {
    BigInteger a = new BigInteger(„11”);
    BigInteger b = new BigInteger(„42”);
    BigInteger c = a.add(b);
    assertTrue(c.toString().equals(„53”));
}
```

- A teszteset végrehajtása akkor sikeres, ha az `assertTrue(boolean)` logikai igaz értéket kap paraméterként.

Assert osztály

- Az egyes tesztek sikerességét ellenőrző statikus metódusokat foglalja magában az Assert osztály.
  - Tipikus az `assertTrue(boolean)` használata, de emellett ~40 összehasonlító függvény (`assert*(...)`) áll rendelkezésre.
  - Az egyszerű használathoz célszerű az Assert osztályt statikusan importálni:
    ```
    import static org.junit.Assert.*;
    ```
  - Az esetleges eltéréshez szöveges üzenet is megadható, például az `assertTrue(String, boolean)` esetén.
  - A `fail()` metódus hívásával jelezhető a teszt sikertelensége.

Assert osztály

- `assertEquals([String message,] ? expected, ? actual)`
- `assertArrayEquals([String message,] ?[] expected, ?[] actual)`
  - Opcionális hibaüzenet
  - ? helyén állhat tetszőleges primitív típus és Object
  - float/double esetén érdemes megadni egy ε paramétert is
- `assertTrue/assertFalse([String message,] boolean condition)`
- `assertNull/assertNotNull([String message,] Object reference)`
- `assertSame/assertNotSame([String message,] Object expected, Object actual)`
- … (lásd JavaDoc)

AssertionError

- Amennyiben valamelyik *assertion* metódus az elvárt értékektől való eltérést észlel egy `AssertionError` kivétel keletkezik (az opcionálisan megadott üzenettel).
- Az `Error` osztály leszármazottja, így nem ellenőrzött kivételnek számít.
- Az egyes ellenőrzőpontok, illetve a hozzájuk tartozó üzenetek csak hiba esetén kerülnek rögzítésre.
- Altípusa a `ComparisonFailure`, amely két hosszú string összehasonlítása során keletkezik, amennyiben nem egyeznek meg
  - `assertEquals(String, String)` függvény
  - Az üzenetben kiemeli a két string különbségét

Assume osztály

- Az `Assume` osztály az `Assert` osztály párja azzal a különbséggel, hogy a feltétel nem teljesülése nem a tesztelt kód hibájára utal, hanem a tesztelhetőség egy előfeltétele nem teljesül ilyenkor, például:

```
@Test
public void calculateTotalSalary() {
    DBConnection dbc = Database.connect();
    assumeNotNull(dbc);
    // …
}
```

- Hiba esetén a teszteset *ignored* eredményt kap, és nem lesz *failed*.

Elvárt kivételek kezelése

- Mi van abban az esetben, ha egy kivétel keletkezése az elvárt működés?
- Opcionális paraméter a @Test-hez:

```
@Test
(expected=IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

- Ha egy kivétel nem az elvárt viselkedés része, akkor egyszerűen a throws részben jelezni kell, és tovább dobni (vagy catch blokkban fail() hívás).

Fixture

- Sok esetben szükség van bizonyos objektumok egy csoportján több művelet sikerességét is ellenőrizni, esetleg bizonyos erőforráshoz csatlakozni, majd elengedni.
  - A különböző funkciók tesztelésére külön, egymástól független teszteseteket célszerű készíteni.
  - Nem szeretnénk kódot duplikálni.
- Megoldás: az úgynevezett *fixture* használata.
- Minden teszteset számára ugyanaz a kiindulási „környezet” érhető el, és nem osztják meg egymás között a példányokat.

@Before és @After

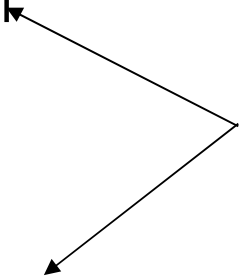
```
private o;
private dbc;

@Before
public void setUp() {
    o = new Object();
    DBConnection dbc = Database.connect();
    // …

}
@After
public void tearDown() {
    dbc.close();
    // ...
}
```

Végrehajtási sorrend

- A @Before és @After annotációval jelölt metódusok lefutnak minden teszteset előtt és után.
- Előfordulhat olyan eset, amikor valamelyik rész elég lenne csak egyszer: @BeforeClass és @AfterClass.
  - Bár ez tipikusan annak a jele, hogy szétcsatolásra lenne szükség.
- Tehát a végrehajtás sorrendje:
  - @BeforeClass
  - @Before
  - @Test #1
  - @After
  - ...
  - @Before
  - @Test #N
  - @After
  - @AfterClass



Az egyes tesztesetek a forráskódban betöltött helyük alapján.

@RunWith annotáció

- A @RunWith annotáció segítségével megadható egy végrehajtó osztály az alapértelmezett futtató helyett:

```
@RunWith(Suite.class)
public class JunitTest {
    // ...
}
```

  - Ilyenkor a teszt végrehajtása során a megadott osztály fog meghívódni és végrehajtani a tesztet
  - Mély bővítési lehetőségeket rejt

Suite

- Lehetőség van több tesztosztály egységbefoglalására:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    JunitTest1.class,
    JunitTest2.class
})
public class JunitSuiteTest {
    // ...
}
```

  - A hivatkozott osztályok tesztjei kerülnek sorban végrehajtásra

Paraméteres tesztesetek

- Sok esetben szükség lehet egy teszteset elvégzésére több különböző paraméterrel
  - Tipikus példája a határérték tesztelés
- Megoldás: parametrizált tesztesetek
  - A megadott tesztesetek és paraméterek minden kombinációban (Déscartes-szorzat) lefutnak
  - Minden esethez külön példány konstruktor híváson keresztül
  - `@RunWith(Parameterized.class)`

Paraméteres tesztesetek

```
@RunWith(Parameterized.class)
public class ParamTest {

    private int number;

    public ParamTest(int number) {
        this.number = number;
    }

    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] { { 1 }, { 2 }, { 3 }, { 4 } };
        return Arrays.asList(data);
    }

    @Test
    public void test() {
        System.out.println("My number is: " + number);
    }
}
```

További lehetőségek

- @Ignore annotáció
  - Teszt metódus vagy egész osztály figyelmen kívül hagyása a tesztelés során
  - Opcionálisan „indokolható”:

```
@Ignore(„Not implemented yet.”)
@Test
public void test() {
    // …
}
```

- @Rule annotáció
  - Speciális szabályok definiálhatóak
  - Olyan public, nem-statikus attribútumra lehet kitenni, amely implementálja a `TestRule` interfészt:

```
@Rule
public TestRule rule = …;
```

Rule - ExpectedException

```
@Rule
public TestRule thrown =
      new ExpectedException();

@Test
public void throwsNullPointerException() {

  thrown.expect(NullPointerException.class);
    thrown.expectMessage("Null");
    throw new NullPointerException("Null pointer.");
}
```

Rule - Timeout

- Timeout limitet határoz meg:

```
@Rule
public TestRule globalTimeout =
    new Timeout(500);
```

  - Minden tesztesetre érvényes
  - Ezredmásodpercben kell megadni

Rule - Verifier

- Verifikációs szabály:

```
@Rule
   public TestRule verifier =
       new Verifier() {
       @Override public void verify() {
           assertTrue(errorLog.isEmpty());
       }
   }
```

  - Elvégzi az ellenőrzést minden tesztesetet követően
  - Az @After-el ellentétben sikertelenné teheti a tesztet

Kimeneti fájlok ellenőrzése

- Érdemes könnyen tesztelhető architektúrát alkalmazni:
  - `writeSomething(String filename)` helyett
    - `writeSomething(Writer writer)`
    - `writeSomethingToFile(String filename)`
      - magában foglalja a fájlnév alapján a szükséges `FileWriter` létrehozását és a másik metódus meghívását
  - tesztelés során a `StringWriter`-el használhatjuk
- (Persze maradhatunk a hagyományos módszernél is.)

Utánzatok (*mock objects*)

- Komplex objektumok működését, viselkedését utánozzák egy bizonyos szempontból (ugyanazzal az interfésszel rendelkeznek).
- Az egységtesztek során sok külső objektumot nem célszerű, vagy nem lehetséges felhasználni, tipikusan:
  - Nem determinisztikus működés (pl. aktuális idő, véletlen-szám)
  - Nehezen reprodukálható állapotot szeretnénk tesztelni (pl. hálózati hiba)
  - Lassú (pl. adatbázis kapcsolat)
  - Még nem készült el
- Ilyen módon az aktuális egység izoláltan tesztelhetővé válik, kiküszöbölhetőek a függőségek.

Absztrakt osztályok tesztelése

- Miért van erre szükség?
  - Az absztrakt ősosztályok gyakran feltételeket szabnak (*contract*), hogy a konkrét implementációk hogyan járjanak el, de előfordulhat, hogy ezt megsértik.
  - Például: `Object.equals()`.
- Megoldás: absztrakt tesztosztályok.

```
public abstract AbstractSuperclassTest {
    protected AbstractSuperclass impl;

    @Before
    public abstract void setUp();

    @Test
    public void test() {
        // …
    }
}
```

`impl` példányosítása

A konkrét implementációk egységtesztjei pedig ebből az egységtesztből származnak.

Metódusok láthatóságai

- public láthatóság
  - értelemszerűen tudjuk tesztelni
- protected / „package” láthatóság
  - ha a tesztosztályok is ugyanabban a package-ben vannak
- private láthatóság
  - Java Reflection API
    - `getDeclaredMethods()`
    - `setAccessible(true)`

Best practice

- Mit kell tesztelni? Mi az ami tönkremehet?
  - „Teszteljünk, amíg a félelem unalommá válik.”
- Ha mások mégis egy hibát fedeznek fel a kódunkban:
  - Készítsünk egy ezt ellenőrző tesztesetet, így többet biztosan nem fordulhat elő.
  - Tanuljunk belőle, és legközelebb az ilyen jellegű tesztek megírását ne felejtsük el.
- Tesztesetenként csak az első hibát jelzi.
  - Ha ez zavaró, az annak a jele, hogy túl összetett funkciót tesztelünk: érdemes lehet tesztesetek és kód szintjén is különválasztani őket.

Tesztek rendszerezése

- A tesztesetek osztályokba rendezése csak a rendszerezésükre szolgál.
- Konvenció: tesztelt osztályonként egy teszt osztály
  - SomeClass → SomeClassTest
- Teszt osztályok elhelyezése
  - A tesztelt osztály „mellé”
  - Jobb megoldás egy párhuzamos könyvtárstruktúra:

```
src/
    hu/
        bme/
            SomeClass.java
test/
    hu/
        bme/
            SomeClassTest.java
```

Tesztek futtatása

- IDE-n keresztül
- Parancssorból

```
java org.junit.runner.JUnitCore SimpleTest
```

- main() függvényen keresztül

```
public static void main(String args[]) {
    JUnitCore.main("SimpleTest");
}

java SimpleTest
```

Összefoglalás

- Tesztelés helye a szoftverfejlesztésben
  - Fejlesztési idő spórolható
  - A kódminőség javítható
- Tesztelés a JUnit segítségével
- Hatékony tesztesetek készítése
- IDE támogatás

Hasznos hivatkozások

- Hivatalos oldal: http://junit.org
- Sourceforge: http://junit.sourceforge.net
  - Getting started
  - FAQ
  - JavaDoc
  - Legfrissebb verzió letöltése

Kérdések?

KÖSZÖNÖM A FIGYELMET!

Híradástechnikai Tanszék

Sipos Róbert
doktorjelölt
BME Híradástechnikai Tanszék
siposr@hit.bme.hu