Programozási technológia 2.

Egységtesztelés (JUnit)

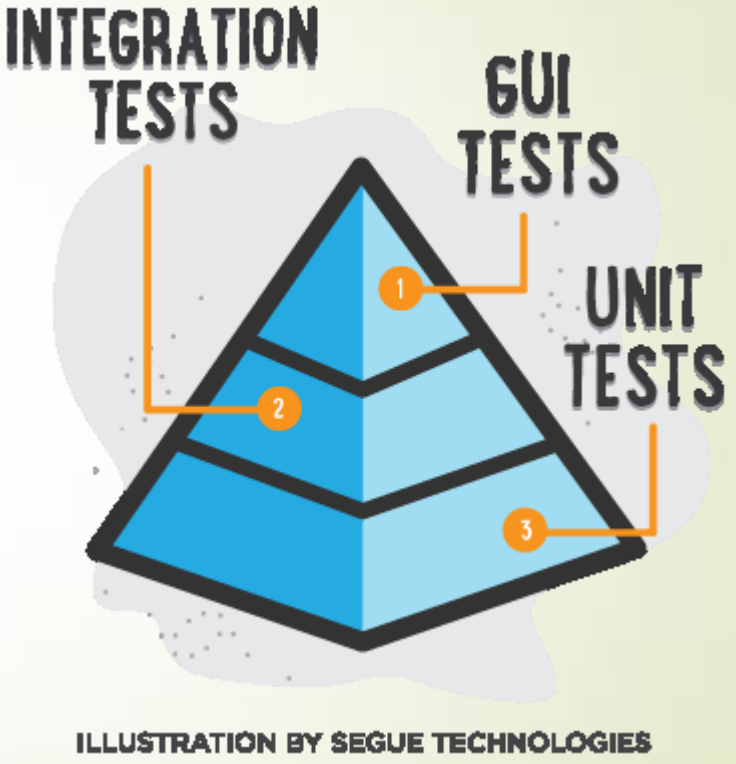
Dr. Szendrei Rudolf

ELTE Informatikai Kar

2018.

Egységtesztelés

- A legalacsonyabb szintű, a programot felépítő egységek tesztelése
- Egység: egy rendszer legkisebb önálló egységként tesztelhető része.
- Egység tesztekkel ellenőrizhető, hogy egy egység az elvárásoknak megfelelően működik.
- Egy egység függvényeiről ellenőrizzük, hogy különböző bemenetek esetén megfelelő eredményt, vagy hibát produkálnak.
- Az egységeket egymástól függetlenítve kell tesztelni.



Egységtesztelés

Előnyök

- A hibák sokkal korábban észlelhetőek
- Minden komponens legalább egyszer tesztelt
- Az egységek elkülönítése miatt a hibák helyének meghatározása könnyű
- A funkciók könnyen módosíthatóak, átalakíthatók
- Dokumentációs szerep: példákat biztosít egyes funkciók használatára

Egységtesztelés

Elvek

- **Gyors:** A teszteknek gyorsan kell futnia, lassú teszteket senki nem futtatja gyakran, így a hibák nem derülnek ki idejében.
- **Független:** A teszteknek egymástól függetlennek és bármilyen sorrendben végrehajthatónak kell lennie.
- **Megismételhető:** A teszteknek bármilyen környezetben, hálózat nélkül is végrehajthatónak kell lennie. A különböző futtatások eredményének meg kell egyeznie.
- **Önellenőrző:** A tesztek eredménye egy logikai érték (futásuk vagy sikeres, vagy sikertelen)
- **Automatikus**: Automatikusan, interakció nélkül futó tesztek

Egységtesztelés

Mit kell tesztelni?

- Egy osztály minden publikus metódusát tesztelni kell
- „Triviális” eseteket
- Speciális eseteket
  - Pl.: számok esetén: negatív, 0, pozitív,
    a megengedettnél kisebb, nagyobb,…
- Pozitív / Negatív teszteseteket
  - A negatív teszteset szándékosan hibás paraméterekkel hívja a tesztelt metódust, célja a hibakezelés ellenőrzése
- Végrehajtási lefedettség: a tesztekből indított hívásoknak a tesztelt metódus lehető legtöbb során végig kell haladnia.
- Egy teszt egyetlen „dolog”

Egységtesztelés

Mit NEM kell tesztelni?

- Ami nyilvánvalóan működik
  - külső lib-ek, JDK, JRE, …
- Adatbázis
  - feltételezhető, hogy ha elérhető, akkor helyesen működik
- Triviális metódusok
  - getter / setter
- GUI
  - a felhasználó felület nem tartalmazhat üzleti logikát

Egységtesztelés

JUnit 4.x

- A JUnit egy egységtesztelő keretrendszer a Java nyelvhez
- Annotációkkal konfigurálható.
- Elvárt eredmények ellenőrzése Assert-ekkel.

```
public class MyTests {
  @Test
  public void multiplicationOfZeroIntegersShouldReturnZero() {
    // MyClass is tested
    MyClass tester = new MyClass();
    // assert statements
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10,  0));
    assertEquals("0 x 10 must be 0", 0, tester.multiply( 0, 10));
    assertEquals("0 x 0 must be 0",  0, tester.multiply( 0,  0));
  }
}
```

Egységtesztelés

Teszt osztály felépítése

Annotáció	Eredmény
`@Test` `public void method()`	Teszt metódus jelölése
`@Test(expected = Exception.class)`	A teszt sikertelen a megadott típusú kivétel hiányában
`@Test(timeout = 100)`	A teszt sikertelen, ha a metódus nem fejeződik be adott idő alatt
`@Before / @After` `public void method()`	Metódus amely minden teszt előtt/után lefut
`@BeforeClass / @AfterClass` `public static void method()`	Egyszer fut le az osztályban lévő első teszt metódus indulása előtt/után
`@Ignore / @Ignore("Why disabled")`	A teszt metódus kihagyása

Egységtesztelés

Ellenőrzések, Assert-ek

Utasítás	Leírás
fail(message)	A hívó teszt sikertelen futását eredményezi
assertTrue([message,] boolean condition**)** vagy **assertFalse**(…)	Ellenőrzi, hogy a megadott feltétel igaz (hamis)–e.
assertEquals([message,] expected, actual**)**	Ellenőrzi, hogy két objektum egyenlő-e.
assertNull([message,] obj) vagy **assertNotNull(**…**)**	Az objektum null (nem null).
assertSame([message,] expected, actual) vagy **assertNotSame**(…)	Ellenőrzi, hogy két változó ugyanarra az objektumra mutat (== / !=)

Egységtesztelés

Kivételkezelés ellenőrzése

- A `@Test(expected = Exception.class)` annotáció limitált, csak egyetlen kivételt tud tesztelni.

- Alternatíva

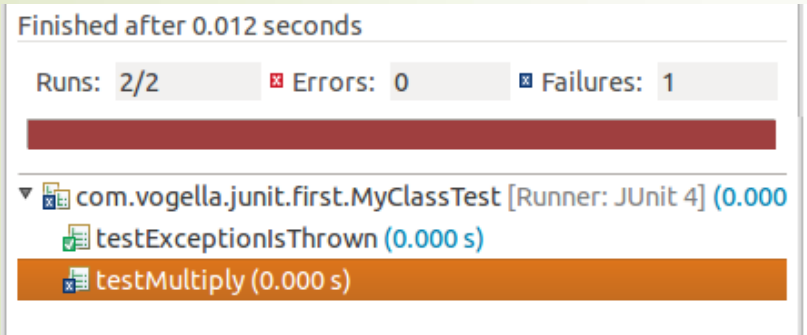
```
try {
    mustThrowException();
    fail();
} catch (Exception e) {
    // expected
}
```

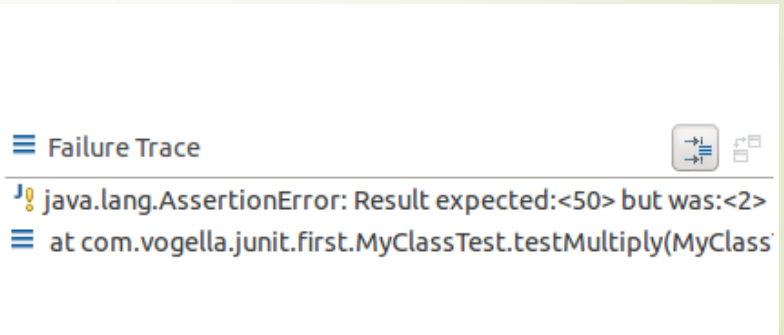
Egységtesztelés

Példa

```
public class MyClass{
  public int multiply(int x, int y) {
    if (x > 999) {
      throw new IllegalArgumentException();
    }
    return x / y;
  }
}
```

```
public class MyClassTest {
  @Test(expected =
     IllegalArgumentException.class)
  public void testExceptionIsThrown(){
    MyClass m = new MyClass();
    m.multiply(1000, 5);
  }
  @Test
  public void testMultiply() {
    MyClass m = new MyClass();
    assertEquals(50,m.multiply(10,5));
  }
}
```





Egységtesztelés

Mock

- A tesztelt metódusok általában más komponens nyújtotta szolgáltatásokat is használnak
- Az egyes funkciókat más komponensektől izoláltan kell tesztelnünk, a tesztek sikeressége nem függhet a függőségek helyes implementációjától
- A Mock egy olyan objektum, amely egy másik objektum működését szimulálja
- Egy osztály függőségeinek szimulálására használandó
- http://easymock.org/

Egységtesztelés

Mock példa

```
public interface ValidatorService {
    boolean allowedToRent(Member m, int numberOfBooksToRent);
}

public class RentalController {
    private ValidatorService validatorService;

    public RentalController(ValidatorService validatorService) {
        this.validatorService = validatorService;
    }

    public void rent(Member m, List<Book> books) {
        if(!validatorService.allowedToRent(m, books.size()) {
            throw new ValidationException();
        }
        ...
    }
}
```

Egységtesztelés

Mock példa – folytatás

```
public class RentalControllerTest {

  private ValidatorService validatorService;
  private RentalController controller;

  @Before public void setUp() throws Exception {

    // NiceMocks return default values for unimplemented methods
      validatorService = createNiceMock(ValidatorService.class);
      controller = new RentalController(validatorService);
  }


  @Test(expected = ValidationException.class)
  public void testRentMoreThanAllowed();

      Member m = new Member();
      List<Books> books = new ArrayList<>();
      expect(validatorService.allowedToRent(m, books))
                                        .andReturn(false).times(1);

      replay(validatorService);
      controller.rent(m, books);
  }
}
```

Test Driven Development

Teszt vezérelt szoftverfejlesztés

Definíció

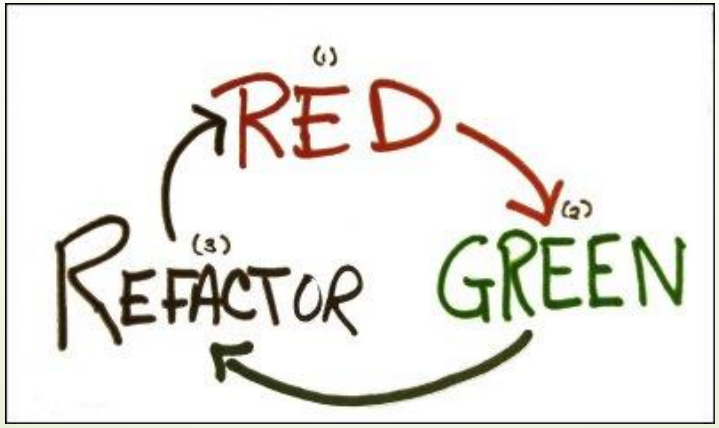
- A TDD a szoftverfejlesztés egy evolúció alapú megközelítése, amely ötvöződik az „előbb-teszteljünk” fejlesztési módszertannal.
  A teszteket már azelőtt megírjuk, mielőtt elkészítenénk azt az éppen elegendő kódot, ami átmegy a teszten, majd ezt refaktoráljuk.

- Ez is egy módja annak, hogy átgondoljuk a követelményeket vagy a tervet, mielőtt megírnánk a működő kódot.

Uncle Bob törvénye (Robert C. Martin)

A TDD három alapszabálya

- NE kódoljunk semmit, kivéve ami ahhoz kell, hogy a programunk átmenjen a sikertelen teszten.
- Tesztből csak éppen elegendő mértékűt írjunk a hiba demonstrálásához.
- Csak annyi kódot írjunk, amennyi éppen elegendő a sikeres teszthez.



+1 szabály

- Legyünk óvatósak az előbbi szabályokkal!
- Azt mondják általában, hogy csak annyit kódoljunk, ami pillanatnyilag épp szükséges. Ez nem elég!
- Végeredményként egy tökéletesen struktúrált és refaktorált kódot kell kapnunk, ami nem csak a teszteket elégíti ki.
- **A tesztek a lehetséges használati eseteknek csak egy részhalmazát fedik le.**
- A végső kódnak jól kell működnie az összes lehetséges tesztesetben. Ne felejtsük ezt el!

Bowling példa

- Készítsünk egy programot, ami kiszámítja a Bowling játék során szerzett pontunkat.
- Tíz mezőt kell teljesítenünk, és minden mezőben két gurítási lehetőségünk van.
- Ha tarolunk (strike), a következő két gurítás eredménye hozzáadódik a tarolás eredményéhez.
- Ha másodikra döntjük le a bábukat (spare), akkor csak a következő gurítás eredménye adódik pluszban a pontszámokhoz.
- A 10. mező hibátlan teljesítésénél bónusz dobást kapunk (strike-nál kettőt, spare-nél egyet).