class: inverse, center, middle

# Unit tesztelés

## Unit tesztelés

* Forráskód legkisebb egységének tesztelésére
* OO nyelv (pl. Java) esetén osztály
* Az adott egység helyesen oldja meg a rá bízott feladatot
* Előírt bemenetre az elvárt kimenetet adja-e
* Tipikusan automatizált
* Tipikusan a szoftverfejlesztők írják

## Unit tesztelés ígéretei

* Hibák mielőbbi megtalálása, ezáltal költségcsökkentés
* Refactoring
* Hibajavításkor regresszió, funkció nem romolhat el, hiba nem jelenhet meg újra
* Komplex rendszer esetén hamarabb találjuk meg a hibás komponenst
* High cohesion, osztály használható interfésszel rendelkezik
* Komponensek közötti interakció és függőségek mielőbbi átgondolása
* Tiszta, lazán kapcsolt kód, low coupling
* Fejlesztés egyszerűsítése
* Dokumentációként viselkedik

## Mi nem unit teszt

* Több osztály kapcsolata
* Keretrendszer bevonása, pl. Spring, Java EE
* Adatbázis műveletek bevonása

## Unit tesztek tulajdonságai

* Izolált
* Unit teszt keretrendszerrel, biztosítja
* A teszt esetek futtatását, csoportosítását
* Az előkészítést (fixture)
* Az ellenőrzést (assertion)
* Az izoláltságot

## JUnit

* Beck, Gamma
* Unit Testing Framework
* Ant, Maven, Gradle, IDE támogatás
* Legelterjedtebb: JUnit 4
* JUnit 5

## Tesztelendő kód - metódus, állapot nélkül

public class Calculator {  
  
 public int add(int a, int b) {  
 return a + b;  
 }  
}

## Tesztelendő kód - osztály, metódus, állapottal

public class Person {  
  
 private String name;  
  
 private String address;  
  
 // Konstruktorok  
  
 private void moveTo(String newAddress) {  
 address = newAddress;  
 }  
  
 // Getter metódusok  
  
}

## Given - when - then

* Java osztály - *test class*
* @Test annotációval ellátott metódusok (public void) - *test case*
* Assert (import static org.junit.Assert.\*)

## Metódus tesztelés, állapot nélkül

import org.junit.Test;  
import static org.junit.Assert.assertEquals;  
  
public class CalculatorTest {  
  
 @Test  
 public void testAdd() {  
 // Given  
 Calculator c = new Calculator();  
  
 // When  
 int result = c.add(10, 20);  
  
 // Then  
 assertEquals(30, result);  
 }  
}

## Állapottal rendelkező osztály esetén

@Test  
public void testCreate() {  
 assertEquals("John Doe", new Person("John Doe", "1115 Budapest, Hengermalom u 1.").getName());  
}

@Test  
public void testMove() {  
 Person person = new Person("John Doe", "1115 Budapest, Hengermalom u 1.");  
 person.moveTo("1115 Budapest, Hengermalom u 1.");  
 assertEquals("1115 Budapest, Hengermalom u 1.", person.getAddress());  
}

## Maven integráció

* Maven életciklus - test fázis
* Surefire plugin - test cél

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

## Test fixture

* @Before, @After annotációk
* @BeforeClass, @AfterClass annotációk

## Test fixture példa

import org.junit.Test;  
import static org.junit.Assert.assertEquals;  
  
public class CalculatorTest {  
  
 private Calculator calculator;  
  
 @Before  
 public void init() {  
 calculator = new Calculator();  
 calculator.setScale(5);  
 }  
  
 @Test  
 public void testAdd() {  
 assertEquals(30, c.add(10, 20));  
 }  
}

## Kivételkezelés

* Dobhat kivételt
* expected
* try-catch használata
* ExpectedException

@Test(expected = IllegalArgumentException.class)  
public void testDiv() {  
 new Calculator().div(10, 0);  
}

## Kivételkezelés try-catch használatával

@Test  
public void testDiv() {  
 try {  
 new Calculator().div(10, 0);  
 fail("IllegalArgumentException should be thrown");  
 }  
 catch (IllegalArgumentException iea) {  
 assertEquals("Division by zero", iea.getMessage());  
 }  
}

## Kivételkezelés ExpectedException használatával

@Rule  
public final ExpectedException exception = ExpectedException.none();  
  
@Test  
public void testDiv() {  
 exception.expect(IllegalArgumentException.class);  
 exception.expectMessage("Division by zero");  
 new Calculator().div(10, 0);  
}

## @Ignore

* Nem futtatja le a környezet
* Opcionálisan megadható üzenet
* Ne használjunk commentelést
* Ideiglenesen használjuk

## Timeout

* timeout attribútum használatával

@Test(timeout=1000)  
public void testWithTimeout() {  
 ...  
}

## Timeout rule-lal

@Rule  
public Timeout globalTimeout = Timeout.seconds(10); // 10 seconds max per method tested  
  
@Test  
public void testSleepForTooLong() throws Exception {  
 log += "ran1";  
 TimeUnit.SECONDS.sleep(100); // sleep for 100 seconds  
}

## TemporaryFolder

* Létrehozott könyvtárak és fájlok törlésre kerülnek a teszteset futtatása után
* Az operációs rendszer temp könyvtárában

@Rule  
public final TemporaryFolder folder = new TemporaryFolder();  
  
@Test  
public void testUsingTempFolder() throws IOException {  
 File createdFile = folder.newFile("myfile.txt");  
 File createdFolder = folder.newFolder("subfolder");  
 // ...  
}

## Paraméterezett tesztek - konstruktor injection

@RunWith(Parameterized.class)  
public class FibonacciTest {  
 @Parameters  
 public static Collection<Object[]> data() {  
 return Arrays.asList(new Object[][] {   
 { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }   
 });  
 }  
  
 private int fInput;  
  
 private int fExpected;  
  
 public FibonacciTest(int input, int expected) {  
 this.fInput = input;  
 this.fExpected = expected;  
 }  
  
 @Test  
 public void test() {  
 assertEquals(fExpected, Fibonacci.compute(fInput));  
 }  
}

## Paraméterezett tesztek - field injection

@Parameter // first data value (0) is default  
public /\* NOT private \*/ int fInput;  
  
@Parameter(1)  
public /\* NOT private \*/ int fExpected;

## Test suite

@RunWith(Suite.class)  
@Suite.SuiteClasses({  
 TestFeatureLogin.class,  
 TestFeatureLogout.class,  
 TestFeatureNavigate.class,  
 TestFeatureUpdate.class  
})  
  
public class FeatureTestSuite {  
 // Üres  
}

## Kategóriák

* @Category annotáció rátehető osztályra vagy metódusra
* Egy vagy több paraméterrel

@Category(IntegrationTest.class, PerformanceTest.class)  
public class A {  
 @Test  
 public void c() {  
  
 }  
}

## Kategória futtatása suite-tal

@RunWith(Categories.class)  
@IncludeCategory(IntegrationTest.class)  
// @ExcludeCategory(PerformanceTest.class)  
@SuiteClasses( { A.class, B.class })  
public class SlowTestSuite {  
 // Will run A.b and B.c, but not A.a  
}

## Kategória használata Maven esetén

<build>  
 <plugins>  
 <plugin>  
 <artifactId>maven-surefire-plugin</artifactId>  
 <configuration>  
 <groups>com.training360.IntegrationTest</groups>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

## JUnit tulajdonságok

* Minden metódushívás előtt példányosítás – izoláció

## JUnit tanácsok

* POJO tesztelése
* Egyszerű tesztek
* Összes elágazás tesztelve legyen
* Kivételes input adatok tesztelése
* Tesztek futtatási sorrendjére ne alapozzunk, metódusok között ne tároljunk állapotot
* Dokumentációs célzat
* Lefutásának az idejét minimalizáljuk

## Gyakori hibák JUnit használatakor

* A fejlesztő a teszt eredményét manuálisan ellenőrzi
* Egy teszt metódus egyszerre több mindent ellenőriz
* Szükségtelen assertion
* Assert metódus nem megfelelő formájának használata
* Nem teljes lefedettség
* Túl bonyolult teszt eset
* Külső függőségek
* Kivételeket nem hibának deklaráljuk
* Nincs unit teszt
* Test infected

## Problémák

* Statikus mezők és metódusok
* Függőségek