JEGYZŐKÖNYV

Szoftvertesztelés

Féléves feladat

Készítette: **Vincze Zoltán László**

Neptunkód: **CM4LG9**

A feladat leírása

A választott feladatom szoftvertesztelés tárgyból egy egységteszt bemutatása Eclipses java környezetben.

Írok egy „AutoLista” nevű osztály, amely az előzőleg implementált „Lista” nevű osztály metódusait teszteli. Minden teszt lefutása előtt feltöltöm a tárolót eszközökkel, amit az (init) - @Before -al valósítok meg. A lista méretének ellenőrzésére vonatkozó tesztesetet (size). Tesztelem a lista ürességének ellenőrzését vonatkozó tesztesetet (isEmpty) - assertTrue/assertFalse. Ellenőrzők egy új elem hozzáadására vonatkozó tesztesetet. Írok egy a lista tartalmát törlő metódust vonatkozó tesztesetet (removeAll) – törlés, majd méret ellenőrzés. Készítek egy olyan tesztesetet, amely nem létező autó törlésére vonatkozik - exception! A legvégén pedig egy olyan metódust írok, amely minden teszt után lefut és kiüríti a listát (destroy) - @After [1]

„Egységtesztelés a JUnit segítségével

Egy egységtesztelés tipikusan fejlesztői tesztelés: a tesztelendő programot fejlesztő programozó gyakorlatilag a programozási munkája szerves részeként készíti és futtatja az egységteszteket. Erre azért van szükség, hogy ő maga is meggyőződhessen arról, hogy amit leprogramozott, az valóban az elvárások szerint működik. Az egység tesztelésére létrehozott tesztesetek darabszáma önmagában nem minőségi kritérium: nem állíthatjuk bizonyossággal, hogy attól, mert több tesztesetünk van, nagyobb eséllyel találjuk meg az esetleges hibákat. Ennek oka, hogy a teszteseteinket gondosan meg kell tervezni. Pusztán véletlenszerű tesztadatok alapján nem biztos, hogy jobb eséllyel fedezzük fel a rejtett hibákat, azonban egy olyan tesztesettervezési módszerrel, amely például a bemenetek jellegzetességeit figyelembe véve alakítja ki a teszteseteket, nagyobb eséllyel vezet jobb teszteredményekhez. Egy fontos mérőszám a tesztlefedettség, amely azon kód százalékos aránya, amelyet az egységteszt tesztel. Ez minél magasabb, annál jobb, bár nem éri meg minden határon túl növelni.

A JUnit egységtesztelő keretrendszert Kent Beck és Erich Gamma fejlesztette ki, e jegyzet írásakor a legfrissebb verziója a 4.11-es. A 3.x változatról a 4.0-ra váltás egy igen jelentős lépés volt a JUnit életében, mert ekkor jelentős változások történtek, köszönhetően elsősorban a Java 5 által bevezetett olyan újdonságoknak, mint az annotációk megjelenése. Még ma is sokan vannak, akik a 3.x verziójú JUnit programkönyvtár használatával készítik tesztjeiket, de jelen jegyzetben csak a 4.x változatok használatát ismertetjük.

A JUnit 4.x egy automatizált tesztelési keretrendszer, ami annyit tesz, hogy a tesztjeinket is programokként megfogalmazva írjuk meg. Ha már egyszer elkészítettük őket, utána viszonylag kis költséggel tudjuk őket újra és újra, automatizált módon végrehajtani. A JUnit keretrendszer a tesztként lefuttatandó metódusokat annotációk segítségével ismeri fel, tehát tulajdonképpen egy beépített annotációfeldolgozót is tartalmaz. Jellemző helyzet, hogy ezek a metódusok egy olyan osztályban helyezkednek el, amelyet csak a tesztelés céljaira hoztunk létre. Ezt az osztályt tesztosztálynak nevezzük.

Az Eclipse-ben egy tesztosztály létrehozását a File → New → JUnit → JUnit Test Case menüpontban végezhetjük el.

A JUnit tesztfuttatója az összes, @Test annotációval ellátott metódust lefuttatja, azonban, ha több ilyen is van, közöttük a sorrendet nem definiálja. Épp ezért tesztjeinket úgy célszerű kialakítani, hogy függetlenek legyenek egymástól, vagyis egyetlen tesztesetmetódusunkban se támaszkodjunk például olyan állapotra, amelyet egy másik teszteset állít be. Egy teszteset általában úgy épül fel, hogy a tesztmetódust az @org.junit.Test annotációval ellátjuk, a törzsében pedig meghívjuk a tesztelendő metódust, és a végrehajtás eredményeként kapott tényleges eredményt az elvárt eredménnyel össze kell vetni. A JUnit keretrendszer alapvetően csak egy parancssoros tesztfuttatót biztosít, de ezen felül nyújt egy API-t az integrált fejlesztőeszközök számára, amelynek segítségével azok grafikus tesztfuttatókat is implementálhatnak. Az Eclipse grafikus tesztfuttatóját a Run → Run as → JUnit test menüpontból érhetjük el. Egy tesztmetódust kijelölve lehetőség nyílik csupán ennek a tesztesetnek a lefuttatására is.

A JUnit a @Test annotáció mellett további annotációtípusokat is definiál, amelyekkel a tesztjeink futtatását tudjuk szabályozni. Az alábbi táblázat röviden összefoglalja ezen annotációkat.

Annotáció	**Leírás**
@Test public void method()	A @Test annotáció egy metódust tesztmetódusként jelöl meg.
@Test(expected = Exception.class) public void method()	A teszteset elbukik, ha a metódus nem dobja el az adott kivételt
@Test(timeout=100) public void method()	A teszt elbukik, ha a végrehajtási idő 100 ms-nál hosszabb
@Before public void method()	A teszteseteket inicializáló metódus, amely minden teszteset előtt le fog futni. Feladata a tesztkörnyezet előkészítése (bemenetei adatok beolvasása, tesztelendő osztály objektumának inicializálása, stb.)
@After public void method()	Ez a metódus minden egyes teszteset végrehajtása után lefut, fő feladata az ideiglenes adatok törlése, alapértelmezések visszaállítása.
@BeforeClass public static void method()	Ez a metódus pontosan egyszer fut le, még az összes teszteset és a hozzájuk kapcsolódó @Before-ok végrehajtása

	előtt. Itt tudunk olyan egyszeri inicializálós lépéseket elvégezni, mint amilyen akár egy adatbázis-kapcsolat kiépítése. Az ezen annotációval ellátott metódusnak mindenképpen statikusnak kell lennie!
@AfterClass public static void method()	Pontosan egyszer fut le, miután az összes tesztmetódus, és a hozzájuk tartozó @After metódusok végrehajtása befejeződött. Általában olyan egyszeri tevékenységet helyezünk ide, amely a @BeforeClass metódusban lefoglalt erőforrások felszabadítását végzi el. Az ezzel az annotációval ellátott metódusnak statikusnak kell lennie!
@Ignore	Figyelmen kívül hagyja a tesztmetódust, illetve tesztosztályt. Ezt egyrészt olyankor használjuk, ha megváltozott a tesztelendő kód, de a tesztesetet még nem frissítettük, másrészt akkor, ha a teszt végrehajtása túl hosszú ideig tartana ahhoz, hogy lefuttassuk. Ha nem metódus szinten, hanem osztály szinten adjuk meg, akkor az osztály összes tesztmetódusát figyelmen kívül hagyja.

Ez csak egy lehetséges végrehajtási sorrend, ugyanis a JUnit tesztfuttatója nem definiál sorrendet az egyes tesztmetódusok végrehajtása között, ezért a testEmptyCollection és testOneItemCollection végrehajtása a fordított sorrendben is végbemehetett volna. Ami viszont biztos: a @Before mindig a teszteset futtatása előtt, az @After utána fut le, minden egyes tesztesetre. A @BeforeClass egyszer, az első teszt, illetve hozzá tartozó @Before előtt, az @AfterClass ennek tükörképeként, a legvégén, pontosan egyszer.

A végrehajtás tényleges eredménye és az elvárt eredmény közötti összehasonlítás során állításokat fogalmazunk meg. Az állítások nagyon hasonlóak az 2. szakasz - Állítások (assertions) alfejezetben már megismertekhez, azonban itt nem az assert utasítást, hanem az org.junit.Assert osztály statikus metódusait használjuk ennek megfogalmazására. Ezen metódusok nevei az assert rész sztringgel kezdődnek, és lehetővé teszik, hogy megadjunk egy hibaüzenetet, valamint az elvárt és tényleges eredményt. Egy ilyen metódus elvégzi az értékek összevetését, és egy AssertionError kivételt dob, ha az összehasonlítás elbukik. (Ez a hiba ugyanaz, amelyet az assert

utasítás is kivált, ha a feltétele hamis.) A következő táblázat összefoglalja a legfontosabb ilyen metódusokat. a szögletes zárójelek ([]) közötti paraméterek opcionálisak.

Állítás	**Leírás**
fail([String])	Feltétel nélkül elbuktatja a metódust. Annak ellenőrzésére használhatjuk, hogy a kód egy adott pontjára nem jut el a vezérlés, de arra is jó, hogy legyen egy elbukott tesztünk, mielőtt a tesztkódot megírnánk.
assertTrue([String], boolean)	Ellenőrzi, hogy a logikai feltétel igaz-e.
assertFalse([String], boolean)	Ellenőrzi, hogy a logikai feltétel hamis-e.
assertEquals([String], expected, actual)	Az equals metódus alapján megvizsgálja, hogy az elvárt és a tényleges eredmény megegyezik-e.
assertEquals([String], expected, actual, tolerance)	Valós típusú elvárt és aktuális értékek egyezőségét vizsgálja, hogy belül van-e tűréshatáron.
assertArrayEquals([String], expected[], actual[])	Ellenőrzi, hogy a két tömb megegyezik-e
assertNull([message], object)	Ellenőrzi, hogy az objektum null-e
assertNotNull([message], object)	Ellenőrzi, hogy az objektum nem null-e
assertSame([String], expected, actual)	Ellenőrzi, hogy az elvárt és a tényleges objektumok referencia szerint megegyeznek-e
assertNotSame([String], expected, actual)	Ellenőrzi, hogy az elvárt és a tényleges objektumok referencia szerint nem egyeznek-e meg

Kivételek tesztelése

Néha előfordul, hogy az elvárt működéshez az tartozik, hogy a tesztelt program egy adott ponton kivételt dobjon. Például a kivételkezeléssel foglalkozó alfejezetben így működött a push művelet, ha már tele volt a fix méretű verem: FullStackException kivételt vált ki, ha már elfogytak a helyek a veremben. Elvárásunkat, mely szerint kivételnek kellene bekövetkeznie, a teszteset @Test annotációjának expected paraméterének megadásával fogalmazhatjuk meg. Ilyenkor a tesztfuttató majd akkor tartja sikeresnek a tesztet, ha valóban a megjelölt kivétel hajtódik végre, és sikertelennek számít minden más esetben.

Tesztkészletek létrehozása

Egy tesztkészlet (test suite) alatt összetartozó és együttesen végrehajtandó teszteseteket értünk. Ez akkor igazán hasznos, ha egy összetettebb funkció teszteléséhez számos, a részfunkciókat tesztelő teszteset tartozik, amelyeket ilyenkor sokszor különálló osztályokba szervezünk a könnyebb áttekinthetőség érdekében. A különböző tesztosztályokban elhelyezett teszteket azonban mégis szeretnénk együttesen (is) lefuttatni, amelyhez egy olyan tesztosztályra van szükségünk, amelyet a @RunWith(Suite.class) és a @Suite.SuiteClasses annotációkkal is el kell látnunk. Az előbbi a JUnit tesztfuttatónak mondja meg, hogy tesztkészlet végrehajtásáról lesz szó, míg a második paraméteréül a tesztkészletet alkotó tesztosztályok osztályliteráljait adjuk.

JUnit antiminták

A JUnit bemutatott eszközeinek segítségével viszonylag alacsony költséggel tudunk inkrementális módon olyan tesztkészletet fejleszteni, amellyel mérhetjük az előrehaladást, kiszúrhatjuk a nem várt mellékhatásokat, és jobban koncentrálhatjuk a fejlesztési erőfeszítéseinket. Az a többletkódolás, amit a tesztesetek kialakítása érdekében kell megtennünk, valójában általában gyorsan behozza az árát és hatalmas előnyöket biztosít fejlesztési projektjeink számára. Mindez persze csak akkor lesz, lehet így, amennyiben az egységteszteink jól vannak megírva – éppen ezért érdemes megvizsgálni, hogy melyek azok a tevékenységek, amelyek a leggyakoribb hibákat jelentik az egységtesztelés során. Ha ezekkel tisztában vagyunk, remélhetőleg már nem követjük el őket mi magunk is.

Rosszul kezelt állítások

A JUnit tesztek alapvető építőelemei az állítások (assertion-ök), amelyek olyan logikai kifejezések, amik ha hamisak, az valamilyen hibát jelez. Az egyik legnagyobb hiba velük kapcsolatban az, ha kézi ellenőrzést végzünk. Ez általában úgy jelenik meg (innen ismerhetünk rá), hogy a tesztmetódus viszonylag sok utasítást tartalmaz, azonban állítást egyet sem. Ilyenkor a fejlesztő a tesztet elsősorban arra használja, hogy ha valamilyen hiba (például egy kivétel) a teszt végrehajtása során bekövetkezik, akkor kézzel elkezdhesse debugolni. Ez a megközelítés azonban pontosan a tesztautomatizálás lényegét és egyben legnagyobb előnyét veszi el, nevezetesen,

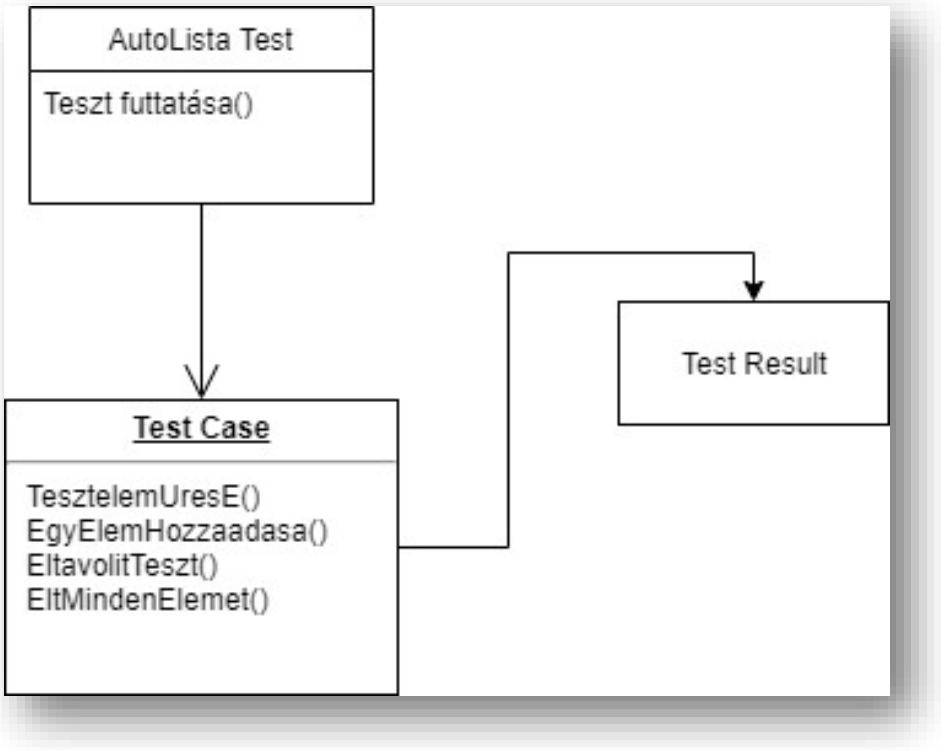
hogy tesztjeinket a háttérben, minden külső beavatkozás nélkül kvázi folyamatosan futtassuk. A kézi ellenőrzések másik tünete, ha a tesztek viszonylag nagy mennyiségű adatot írnak a szabványos kimenetre vagy egy naplóba, majd ezeket kézzel ellenőrzik, hogy minden rendben zajlott-e. Ehhez nagyon hasonló ellenminta a hiányzó állítások esete, amikor egy tesztmetódus nem tartalmaz egyetlen utasítást sem. Ezek a helyzetek kerülendőek.

Nem létező egységtesztek

Ez esetben nem magukkal a tesztekkel, hanem azok hiányával van a baj. Minden programozó tudja, hogy teszteket kellene írnia a kódjához, mégis kevesen teszik. Ha megkérdezik tőlük, miért nem írnak teszteket, ráfogják a sietségre. Ez azonban ördögi körhöz vezet: minél nagyobb nyomást érzünk, annál kevesebb tesztet írunk. Minél kevesebb tesztet írunk, annál kevésbé leszünk produktívak és a kódunk is annál kevésbé lesz stabil. Minél kevésbé vagyunk produktívak és precízek, annál nagyobb nyomást érzünk magunkon. Ezt a problémát elhárítani csak úgy tudjuk, ha teszteket írunk. Komolyan. Annak ellenőrzése, hogy valami jól működik, nem szabad, hogy a végfelhasználóra maradjon. Az egységtesztelés egy hosszú folyamat első lépéseként tekintendő.

Azon túlmenően, hogy az egységtesztek a tesztvezérelt fejlesztés alapkövei, gyakorlatilag minden fejlesztési módszertan profitálhat a tesztek meglétéből, hiszen segítenek megmutatni, hogy a kód újraszervezési lépések nem változtattak a funkcionalitáson, és bizonyíthatják, hogy az API használható. Tanulmányok igazolták, hogy az egységtesztek használata drasztikusan növelni tudja a szoftverminőséget.” [2]

UML diagram elkészítése



[Forrás: saját készítés]

Forráskód

```
package AutoLista;

import java.util.ArrayList;
import java.util.NoSuchElementException;

public class List {

        private ArrayList<String> cars;

        public List() {this.cars = new ArrayList<String>();}

        public void add(String car) {this.cars.add(car);}

        public void remove(String car)
```

```
        {
                int index = this.cars.indexOf(car);

                if(index == -1) throw new NoSuchElementException();

                this.cars.remove(index);
        }

        public int size()
        {
                return this.cars.size();
        }

        public boolean isEmpty() {return this.cars.isEmpty();}

        public void removeAll() {this.cars.removeAll(cars);}
}
```

Teszt osztály

```
package AutoLista;

import static org.junit.Assert.*;
import java.util.NoSuchElementException;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class ListTest {

        private List cars = new List();

        @Before
```

```
	public void init()
	{
		this.cars.add("Toyota");
		this.cars.add("Opel");
		this.cars.add("Dacia");
		this.cars.add("BMW");
		this.cars.add("Audi");
		this.cars.add("Trabant");
		this.cars.add("Renault");
	}

	@Test
	public void testEmpty()
	{
		assertFalse(this.cars.isEmpty());
	}

	@Test
	public void testAdd()
	{
		this.cars.add("Lada");
		assertEquals("Tesztelem, hogy bekerül-e az új elem a
listába?",6,this.cars.size());
	}

	@Test(expected = NoSuchElementException.class)
	public void removeTest()
	{
		this.cars.remove("Traktor");
		}

	@Test
	public void removeAllTest()
	{
```

```
            this.cars.removeAll();
            assertTrue(this.cars.isEmpty());
        }

        @After
        public void destroy() {this.cars.removeAll();}
}
```

Internetes forrás

[1] Tompa Tamás

[2] Kollár Lajos, Sterbinszky Nóra: Programozási technológiák - Jegyzet
https://regi.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0103_21_programozasi_technologiak/adatok.html (Utolsó letöltés: 2020. 11. 20.)