JEGYZŐKÖNYV

Szotvertesztelés

Féléves feladat

Készítette: **Pintér Judit**
Neptun-kód: **E4UHUX**

A feladat leírása: JUnit segítségével egységteszt bemutatása java környezetben.

Készítek egy „list_colors” elnevezésű osztályt, amely az előzőleg implementált „List” nevű osztály metódusait teszteli. Minden teszt lefutása előtt feltöltöm a tárolót eszközökkel, amit az (init) @Before teszteseteket inicializáló metódussal valósítok meg, mely minden teszteset előtt lefut. Feladata a tesztkörnyezet előkészítése. Tesztelem a lista méretét és az ürességének ellenőrzésére vonatkozó tesztesetet, hogy hamis e, hogy üres a listám, valamint ellenőrzök egy új elem hozzáadására vonatkozó tesztesetet. Mindezek után írok egy nem létező elem törlésére vonatkozó tesztesetet és a lista tartalmát törlő metódusra utaló tesztesetet. A végére írok egy metódust, amely minden teszt után lefut és kiüríti a listát - @After-el.

a, <u>Témakör kidolgozása:</u> Egységtesztelés 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 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 alábbi kódrészlet egy JUnit tesztmetódust tartalmaz. Az Eclipse-ben egy tesztosztály létrehozását a File → New → JUnit → JUnit Test Case menüpontban végezhetjük el.

```
@Test
public void testMultiply() {
  // MyClass a tesztelendő osztály
  MyClass tester = new MyClass();
  // leellenőrizzük, hogy a multiply(10,5) 50-nel tér-e vissza
  assertEquals("10 x 5 must be 50",  50, tester.multiply( 10,  5));
}
```

A JUnit tesztfuttatója az összes, `@Test` annotációval ellátott metódust lefuttatja, azonban, ha töb 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.

3.2. táblázat - JUnit annotációk

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.

Annotáció	Leírás
`@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.

Megjegyzés

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 „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 részsztringgel kezdődnek, és lehetővé teszik, hogy megadjunk egy hibaüzenetet, valamint az elvárt és téényleges 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.

3.3. táblázat - Az `Assert` osztály metódusai

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.

Állítás	Leírás
`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 ojektum `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

Parametrizált tesztek

A JUnit 4-es verziójában jelent meg a parametrizált tesztek készítésének lehetősége. Ezek célja, hogy lehetővé tegyék ugyanazon tesztesetek többszöri lefuttatását, persze rendre különböző értékekkel.

Leegyszerűsített példaként tekintsünk egy olyan osztályt, amelynek az `add` metódusa összeadja a paramétereként kapott két számot:

```
public class Addition {
  public int add(int x, int y) {
    return x + y;
  }
}
```

Parametrizált tesztek készítéséhez az alábbi öt tevékenységet kell elvégeznünk:

- El kell látni a tesztosztályt a `@RunWith(Parameterized.class)` annotációval.

```
@RunWith(Parameterized.class)
public class AdditionTest {
  private int expected;
```

```
  private int first;

  private int second;

  ...

}
```

- Készíteni kell egy olyan konstruktort, amely egy sornyi tesztadat információit képes befogadni.

```
  public AdditionTest(int expected, int first, int second) {

    this.expected = expected;

    this.first = first;

    this.second = second;

  }
```

- Létre kell hozni egy, a `@Parameters` annotációval ellátott olyan publikus statikus metódust, amely egydimenziós tömbök kollekcióját adja vissza. Egy-egy tömb ebben a kollekcióban az egyes tesztvégrehajtások során használt adatokat tartalmazza. A kollekció mérete azt mondja meg a tesztfuttatónak, hogy hányszor kell majd az egyes teszteseteket lefuttatni. Az egyes tömböknek azonos elemszámúaknak kell lennie, ráadásul ez pont annyi, mint a konstruktor paramétereinek a száma, hiszen a tesztfuttató majd a tömb elemei alapján fogja létrehozni a paraméterezett teszt végrehajtásához szükséges objektumot.

```
  @Parameters

  public static Collection<Integer[]> addedNumbers() {

    return Arrays.asList(new Integer[][] {{3, 1, 2}, {5, 2, 3}, {7, 3, 4}, {9, 4, 5}});

  }
```

  A példában látható tömbök (például a 3, 1, 2 elemekt tartalmazó) felhasználásával hozza majd a futtató rendszer létre a tesztosztály egy-egy objektumát (az `AdditionTest` konstruktornak átadva a tömb elemeit).

- Létre kell hozni a tesztmetódus(oka)t. Ezeket szokás szerint a `@Test` annotáció jelöli, és a tesztosztály példányváltozóin operálnak.

```
  @Test

  public void sum() {

    Addition add = new Addition();

    assertEquals(expected, add.addNumbers(first, second));

  }
```

- Ha parancssorból szeretnénk a tesztfuttatót végrehajtani, akkor szükségünk lesz még az `org.junit.runner.JUnitCore` osztály `runClasses` nevű statikus metódusára is (grafikus tesztfuttató esetén erre nincs szükség).

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. Példa:

```
@Test(expected=FullStackException.class)

public void testPush() {

  FixedLengthStack<Integer> s = new FixedLengthStack<Integer>(3);

  for (int i = 0; i < 4; i++)

    s.push(i);

}
```

A példában egy háromelemű verembe 4 beszúrást kísérlünk meg. Arra számítunk, hogy ekkor a megjelölt kivétel kerül kiváltásra.

Ha nem váltódik ki kivétel, vagy nem a 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 (egy hosszabb tesztmetóduson belül), arra ez a módszer nem jó. Ilyenkor tegyük a következőt:

- kapjuk el a kivételt mi magunk,
- használjuk a fail-t, ha egy adott pontra nem volna szabad eljutni,
- a kivételkezelőben pedig nyerjük ki a kivétel szövegét, és hasonlítsuk az elvárt szöveghez.

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.

A példában a `Test1`, a `Test2` és a `JunitTestFirstExample` tesztosztályok által tartalmazott tesztesetek végrehajtását végző tesztkészlet létrehozását láthatjuk.

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, minde 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 csak a túl kevés, de a túl sok állítás is problémás lehet: ha egy tesztmetódusban több állítás is van, az általában azt jelenti, hogy a tesztmetódus túl sokat próbál tesztelni. Ezt orvosolhatjuk, ha szétvágjuk a tesztmetódust több tesztmetódusra.

Megjegyzés

Ez nem feltétlenül jelenti azt, hogy tesztenként pontosan egy állítás kerüljön megfogalmazásra! Tapasztalt tesztelők is készítenek néha olyat, hogy egy tesztmetódus több (de csak néhány) állítást tartalmaz. Általában azzal van a probléma, hogy összekeveredik a funkcionalitást tesztelő kód és az elvárt eredmények ellenőrzését végző kód, mert ilyenkor a hiba okát elég nehéz meglelni.

A redundáns feltételek szintén kerülendőek. Egy redundáns állítás egy olyan `assert` metódus, amelyben a feltétel beleégetett módon `true`. Általában ezzel a helyes működési mód demonstrálását szeretnék elvégezni, azonban a szükségtelen bőbeszédűség csak zsúfolttá teszi a metódust. Amennyiben egyéb állítások nincsenek is, akkor ez tulajdonképpen a kézzel ellenőrzés antimintával egyenértékű. Ha ilyennel találkozunk, egyszerűen csak szüntessük meg azon állításokat, amelyek a beégetett feltételt tartalmazzák.

A rossz állítás alkalmazása szintén problémát okozhat. Az `Assert` osztálynak elég sok metódusa kezdődik `assert`-tel, ráadásul sokszor csak kicsit eltérő közöttük a paraméterek száma és a szementikájuk. Sokan talán épp emiatt csupán egyetlen `assert` metódust használnak, mégpedig az `assertTrue`-t, és annak a logikai kifejezés részébe szuszakolják bele, hogy mit is szeretnének levizsgálni. Példák a rossz használatra:

```
assertTrue("Objects must be the same", expected == actual);

assertTrue("Objects must be equal", expected.equals(actual));

assertTrue("Object must be null", actual == null);
```

assertTrue("Object must not be null", actual != null);

Ezek helyett használjuk rendre az alábbiakat:

assertSame("Objects must be the same", expected, actual);

assertEquals("Objects must be equal", expected, actual);

assertNull("Object must be null", actual);

assertNotNull("Object must not be null", actual);

Felszínes tesztlefedettség

A kezdő egységtesztelők gyakorta csak valamilyen alapvető tesztkódot írnak, és nem vizsgálják meg teljesen a tesztelendő kódot. Ennek többféle megjelenési formája is van:

- Csak az alapvető lefutás tesztelése: csak a rendszer ***elvárt*** viselkedése kerül tesztelésre. Érvényes adatokat megadva az elképzelt helyes eredmény ellenében történik az ellenőrzés, hiányoznak azonban a kivételes esetek vizsgálatai. Ilyen például, hogy mi történik hibás bemeneti adatok esetén, az elvárt kivételek eldobásra kerültek-e, melyek az érvényes és érvénytelen adatok ekvivalenciaosztályainak határai, stb.
- Csak a könnyű tesztek: az előzőhez némiképpen hasonlóan, csak arra koncentrálunk, amit egyszerű ellenőrizni, és így a tesztelendő rendszer igazi logikája figyelmen kívül marad. Ez tipikusan a tapasztalatlan fejlesztő komplex kódot tesztelni célzó próbálkozásainak a tünete.

Ezek ellen valamilyen tesztlefedettség-mérő eszköz alkalmazásával védekezhetünk, amely segít meghatározni, hogy a kód melyik része nincs kielégítő módon tesztelve.

Túlbonyolított tesztek

Az egységtesztek kódjának az éles rendszer kódjához hasonlóan könnyen érthetőnek kell lennie. Általánosságban azt mondhatjuk, hogy egy programozónak a lehető leggyorsabban meg kell értenie egy teszt célját. Ha egy teszt olyan bonyolult, hogy nem tudjuk azonnal megmondani róla, jó-e vagy sem, akkor nehéz megállapítani, hogy egy sikertelen tesztvégrehajtás a tesztelendő vagy a tesztelő kód rossz mivolta miatt következett-e be. Vagy ami még ennél is rosszabb, fennáll a lehetősége annak, hogy egy kód úgy megy át egy teszten, hogy nem volna neki szabad.

A túlbonyolított tesztek egyszerűsítését ugyanúgy végezzük, mint bármilyen más túlbonyolított kód egyszerűsítését: kódújraszervezést (refaktorálást) hajtunk végre a minél könnyebben érthető kód érdekében. Általában ezt a lépéssorozatot mindaddig végezzük, mígnem könnyen felismerhető módon a következő szerkezettel fog rendelkezni:

1. Inicializálás (set up)
2. Az elvárt eredmények deklarálása
3. A tesztelendő egység meghívása
4. A tevékenység eredményeinek beszerzése
5. Állítás megfogalmazása az elvárt és a tényleges eredményről.

Külső függőségek

Annak érdekében, hogy a kód helyesen működjön, számos külső függőségre kell támaszkodnia, például függhet:

- egy bizonyos dátumtól vagy időtől,
- egy harmadik fél által készített (úgynevezett third-party) jar formátumú programkönyvtártól,
- egy állománytól,
- egy adatbázistól,
- a hálózati kapcsolattól,
- egy webszervertől,
- egy alkalmazásszervertől,
- a véletlentől,
- stb.

Az egységtesztek a tesztelési hierarchia legalsó szintjén helyezkednek el, a céljuk az, hogy kis mennyiségű kóddal izoláltan próbára tegyék az éles kód egy kis részét, vagyis az egységet. A magasabb szintű teszteléssel szemben az egységtesztelés célja tehát csakis önálló egységek ellenőrzése. Minél több függőségre van egy egységnek szüksége a futtatásához, annál nehezebb igazolni a megfelelő működést. Ha adatbázis-kapcsolatot kell konfigurálni, el kell indítani egy távoli szervert, stb., akkor az egységteszt futtatásáért nagy erőfeszítéseket kell tenni.

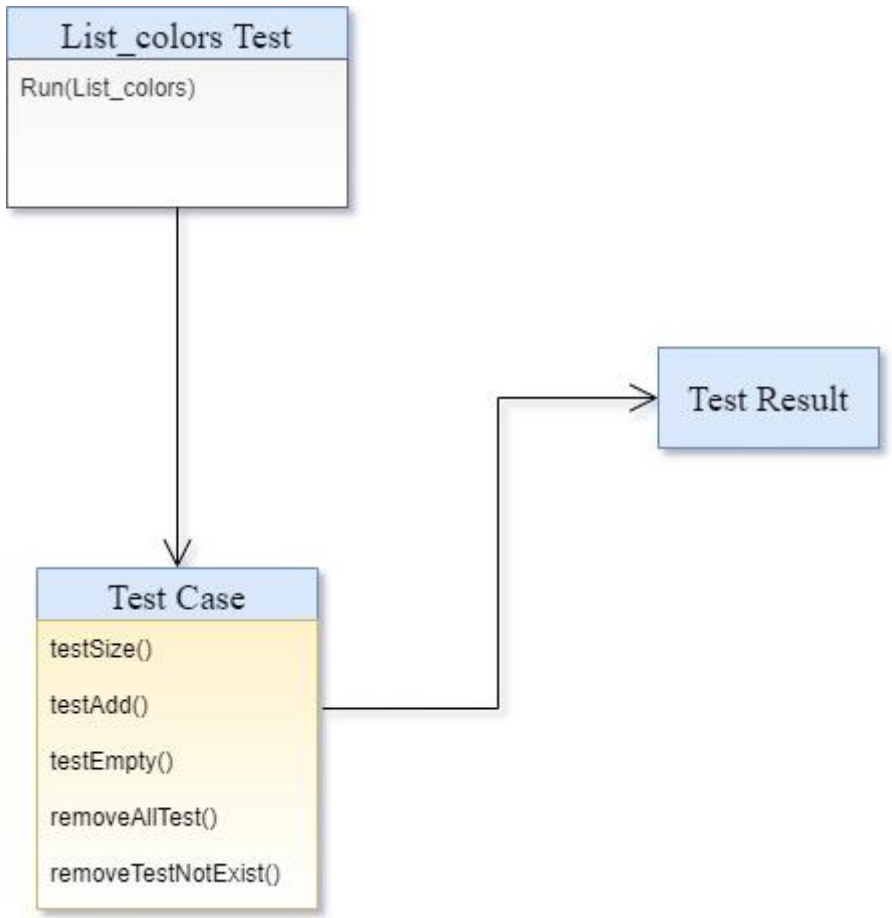
Az egységtesztek hatékonyságának jó mérőszáma, hogy egy kezdő fejlesztő mennyi idő alatt jut el a tesztek lefuttatásához onnantól kezdve, hogy a verziókezelő rendszerből beszerezte a kódokat. A legegyszerűbb megoldás a verziókezelőből (például cvs, svn vagy git) történő checkout után a build-elést végző eszköz (például ant vagy maven) futtatása, amely csak akkor megy ilyen egyszerűen, ha nincsenek külső függőségek. Ökölszabály, hogy a külső függőségeket el kell kerülni.

Ennek érdekében az alábbiakat tehetjük:

- a harmadik fél által készített könyvtáraktól való függés elkerüléséhez használjunk tesztduplázókat (test doubles), például mock objektumokat,
- biztosítsuk, hogy a tesztadatok a tesztkóddal együtt kerülnek csomagolásra,
- kerüljük el az adatbázishívásokat egységtesztjeinkben,
- ha ***mindenképpen*** adatbázisra van szükségünk, használjunk memóriában tárol adatbázist (például HSQLDB-t).”

Forrás: [1]Kollár Lajos, Sterbinszky Nóra: Programozási technológiák – Jegyzet 2014
Egységtesztelés JUnit segítségével (unideb.hu)

b, UML diagram:



Forrás: Saját készítés

c, Forráskód:

List.java:

```
package list_colors;

import java.util.ArrayList;

import java.util.NoSuchElementException;

public class List {

private ArrayList<String> colors;

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

public void add(String color) {this.colors.add(color);}

public void remove(String color)

{

int index = this.colors.indexOf(color);

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

this.colors.remove(index);

}

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

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

public void removeAll() {this.colors.removeAll(colors);}

}
```

ListTest.java:

```
package list_colors;

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 colors = new List();

	@Before

	public void init()

	{

		this.colors.add("Blue");
		this.colors.add("Red");
		this.colors.add("Brown");
		this.colors.add("Yellow");
		this.colors.add("Purle");
		this.colors.add("Pink");
		this.colors.add("White");
		this.colors.add("Black");
		this.colors.add("Silver");
		this.colors.add("Gold");
		this.colors.add("Green");
		this.colors.add("Turquoise");

	}

	@Test
	   public void testSize(){
	   assertEquals("Méret ellenőrzése",12,colors.size());
	   //Teszteli, hogy a listám mérete egyezik e 12-vel.
	    }

	@Test

	public void testEmpty() {
		assertFalse(this.colors.isEmpty());
		//Ellenőrzi, hogy hamis e az hogy a listám üres.
	}

	@Test
```

```
	public void testAdd()

	{

		this.colors.add("Burgundy");
		assertEquals("Új szin hozzáadása:", 13, this.colors.size());
		//Az adott új szín bekerülhet e a 13. színként a listába

	}

	@Test(expected = NoSuchElementException.class)

	public void removeTestNotExist() {
		this.colors.remove("Yellow");
		//Töröl egy nem létező színt
	}

	@Test

	public void removeAllTest()

	{

		this.colors.removeAll();
		assertTrue(this.colors.isEmpty());
		//Teszt a lista törlésének metódusára.

	}

	@After

	public void destroy() {
		this.colors.removeAll();
		//Minden teszt után lefut és kiüríti a listát.
	}

}
```

Forrás: [1]Kollár Lajos, Sterbinszky Nóra: Programozási technológiák – Jegyzet 2014
Egységtesztelés JUnit segítségével (unideb.hu)