Budapesti Műszaki és Gazdaságtudományi Egyetem
Témalaboratórium (BMEVIMIAL00)

Automatikus tesztelés

Témalaboratórium beszámoló

Kövér Márton (W6HYOZ)

Konzulensek:
Honfi Dávid
Micskei Zoltán
2017/2018 1. félév

Tartalomjegyzék

1 Bevezető

A mai világban körülvesznek bennünket a szoftveres rendszerek, találkozunk velük hivatalos ügyek intézésekor (banki ügyintézés, Ügyfélkapu stb.), de hétköznapjainkban, akár szórakoztató jelleggel (okostelefon, televízió stb.) is. Előfordulhat, hogy találkoztunk már olyannal, hogy nem megfelelően működik valami ilyen alapokon nyugvó rendszer. Ez azon felül, hogy bosszantó tud lenni, veszélyes is lehet, ellophatják a pénzünket a bankszámláról, vagy akár személyi sérülés is előfordulhat, pl. egy önvezető autó esetén.

1.1 Tesztelés célja

Célunk, hogy minél kevesebb hiba legyen a szoftverünkben. Ezt úgy érhetjük el, hogy a fejlesztés során különböző teszteket készítünk és az általuk felfedett problémákat még kibocsátás előtt kijavítjuk. Ha egy adott kódban a jól megírt tesztek kevés hibát találnak, vagy nem is találnak, akkor az növelheti a vélt megbízhatóságát, ugyanakkor még tesztelés segítségével sem könnyű minden hibát felfedezni. A tesztelés mennyisége a szoftver céljától nagymértékben függhet.

1.2 Elvégzendő feladatok

A félévre kitűzött feladatom volt megismerkedni tesztelési módszertanokkal. Ilyenek voltak a tesztek manuális implementációja, tesztek izolációja és tesztgenerálás eszköz segítségével. Ezek megvalósításához az alábbi eszközökkel dolgoztam: JUnit keretrendszer, Mockito, Randoop.

2 Unit tesztelés

2.1 Algorithms projekt

A munka elkezdéséhez szükségem volt egy kész projektre, amivel tudok dolgozni. Az én választásom az Algorithms nevű, nyíltforráskód projektre esett. Ez megtalálható GitHub-on[1], ott fork-oltam, fejlesztéshez pedig az IntelliJ IDEA-t használtam.

A projekt 89 osztályt tartalmazott, ezek egyszerű problémák megoldására készültek (pl. adatszerkezetek, matematikai műveletek, rendezések, string műveletek stb.), így általában csak néhány rövidebb függvényt tartalmaznak.

2.2 Meglévő tesztek vizsgálata

Két osztály kivételével az összeshez volt írva teszt. A tesztosztályok jó felépítésűek voltak, minden osztályhoz csak egy tesztosztály tartozott és viszont. Ugyan nem voltak kommentezve a tesztek, de a függvények nevéből könnyen kitalálható volt, mi az adott teszt célja.

2.3 Saját tesztek készítése

Először is részletesebben megismerkedtem a tesztek felépítésével. Szükség lehet olyan kódrészletre, pl. osztályok példányosítása, ami minden teszt előtt vagy után le kell, hogy fusson. A `Before` és az `After` rész arra szolgál, hogy az ilyen részt elég legyen egyszer megírni és minden teszt előtt, illetve után meghívni automatikusan. Léteznek ezeknek másik változataik is (`BeforeClass`, `AfterClass`), ezek csak egyszer, az osztályban tartalmazott összes teszt előtt vagy után futnak le, olyankor lehet rájuk szükség, mikor valami számításigényes feladatot akarunk használni és sok tesztünk van.

Magamtól írtam teszteket a Factorial, a DivideUsingSubstraction és a SquerRoot osztályokhoz. A korábbi tesztosztályokat példaként felhasználva ez viszonylag könnyedén ment. Miután

[1] https://github.com/pedrovgs/Algorithms

megismerkedtem a kódfedettség mérésének lehetőségeivel, lefuttatam egy eszközt a már meglévő, majd később a saját tesztjeimmel kiegészített tesztkészletre is. A lefedettség magas volt, de sikerült még így is javítani rajta. Számomra érdekesnek tűnt, hogy tudom ellenőrizni, mi került a konzolkimenetre, így kipróbáltam azt is. A MoveZeroesInArrayTest-hez és a GoThroughMatrixInSpiralTest-hez adtam még hozzá egy új tesztesetet.

Az 1. táblázat mutatja, mekkora volt a kódlefedettség, mielőtt írtam (Előtte sor) és miután írtam új teszteket (Utána sor).

1. táblázat. Kódlefedettség a projekten

	Lefedett osztályok	**Lefedett metódusok**	**Lefedett sorok**
Előtte	97% (87/89)	97% (270/277)	97% (1676/1713)
Utána	98% (88/89)	98% (272/277)	98% (1693/1713)

3 Függőségek kezelése

A függőség alatt modulok közötti kapcsolatot értünk. A külső függőségek problémát okozhatnak teszteléskor, ugyanis a tesztelt osztályunk viselkedése függ másik osztályétól is, ilyenkor nem várt viselkedést tapasztalhatunk.

Erre megoldás, ha modellezzük a tesztelendő modul környezetét, vagyis biztosan az fog történni, amit mi szeretnénk. Ezt mock objektumok segítségével tehetjük meg, így megadhatóvá válik a függvények visszatérési értéke és ellenőrizhetjük az átadott argumentumainkat.

3.1 Spaceship projekt

A projekt, amivel korábban dolgoztam, nem felet meg az elvégzendő feladathoz egyszerűsége miatt. Az új projektem egy MSc-s labor anyagaként szolgál, GitHub-on[2] elérhető. Ez a projekt egy űrhajót modellez, ami képes torpedók kilövésére, két különböző tárolóból. Át kellett alakítanom az űrhajó osztályt, ugyanis az általam modellezendő osztály privát attribútuma volt annak. Megoldás lehet, ha írunk Getter és Setter függvényeket, vagy ha protectedre állítjuk az attribútumot privát helyett. Én a másodikat választottam és egy TestShip osztályt hoztam létre, amely az eredeti űrhajóból származott, azonban egy Setter segítségével megadhattuk, milyen torpedó tárolókat használjon.

3.2 Mockito

A Mockito[3] egy nyíltforráskódú projekt, a segítségével izolációs teszteket lehet készíteni. Forráskódja megtalálható GitHub-on.

A Mockito-t több módon is próbáltam hozzáadni a projektemhez, elsőre nem sikerült működésre bírnom. Ennek oka, hogy először olyan verziót próbáltam használni, amely nem tartalmaz minden függőségeket. Ennek megfelelően letöltöttem a program azon verzióját, amely az összes függést tartalmazza. Ezzel már sikerült működtetni a keretrendszert.

[2] https://github.com/FTSRG/swsv-labs/tree/master/development-testing/hu.bme.mit.spaceship
[3] http://site.mockito.org

Mockito segítségével be tudtam állítani, mi legyen egy-egy függvény kívánt visszatérési értéke, ha meghívásra kerülnek. Ezt a `when(mockolt osztály.függvény).thenReturn(kívánt visszatérési érték);` alakban tudtam megtenni.

4 Unit tesztek generálása

Unit tesztek automatikus generálásának az a lényege, hogy módszer segítségével a tesztelt program alapján teszteket generálunk. Az automatikus generálás előnye, hogy rávilágíthat olyan hibákra, amiket mi nem feltétlen vettünk volna észre, ugyanakkor nem biztos, hogy minden általunk kívánt funkció tesztelésre kerül.

4.1 Randoop

A Randoop[4] egy nyíltforráskódú projekt, forráskódja megtalálható GitHub-on. A Unit tesztek visszajelzéses tesztgenerálás segítségével készülnek, azaz véletlenszerűen készít metódus, illetve konstruktor hívás szekvenciákat a tesztelés alatt lévő osztályhoz. Ezeket végrehajtva ellenőrzéseket készít, amik a program aktuális kimenetét rögzítik. Az ellenőrzésekből és a kódszekvenciákból készülnek a tesztek. A végeredmény véleményem szerint nehezen átlátható, olvasható.

4.1.1 Telepítés és futtatás

A kiadott leírás[5] segítségével próbáltam telepíteni és futtatni a Randoop-ot. A telepítés nem volt nehéz, lényegében csak ki kellett csomagolni a fájlt, amit letöltöttem. A futtatással több gondom is volt. Először a Linuxos leírást kellett átalakítanom Windows környezethez, vagyis az elérési utakat ";" kell, hogy elválassza és nem ":". A ";" miatt a parancssor két utasításnak értelmezte, amit beírtam, ezt javítottam idézőjelek használatával. A másik problémát az jelentette, hogy csak úgy lehet tesztgenerálást futtatni, ha az eredeti csomag struktúrában vannak a tesztelendő osztályok. A futtatáshoz hozzá kellett adnom a Path-hoz a jdk-t. A tesztek készítéshez a tesztelendő osztályok lefordított változatára van szükség, így egy python kód segítségével kilistáztam a mappaszerkezetet, amiben ezek megtalálhatóak voltak.

Az alábbi paranccsal futtattam a Randoop-ot:

```
java -ea -classpath "path1;path2" randoop.main.Main gentests
```

A `path1` és `path2` a helye a randoop-all-3.1.5.jar fájlnak és annak a mappának, ahol a lefordított tesztelendő osztályok találhatóak. Fontos paraméter a `--classlist=`, ez után adható meg a tesztelendő osztályok nevét tartalmazó fájl neve és kiterjesztése, pl. classes.txt. A `--testclass=` paraméter után megadhatunk egy osztály nevet, amihez a teszteket generálja a Randoop, pl. java.util.TreeSet. Általam használt paraméter még a `--timelimit=`, itt adhatjuk meg, mennyi ideig generálja a teszteket másodpercben mérve, pl. 60.

4.1.2 A tesztelt projekt

Tesztek készítésekor a korábban már leírt Algoritms projektet használtam. A projekt összes osztályához generáltattam teszteket. A generált teszteket hozzáadtam a projekthez.

4.1.3 Felfedezett hibák

A generált tesztek, amik kivételt dobtak, külön osztályba lettek elmentve. Sikerült három darab NullPointerExceptionre is fényt deríteni a tesztek segítségével.

[4] https://randoop.github.io/randoop
[5] https://randoop.github.io/randoop/manual/index.html

1. ábra. Kódrészlet a projektből. A legalsó sorban keletkezett hiba

```
public BinaryNode getIterative(BinaryNode root, BinaryNode n1, BinaryNode n2) {
  validateInput(root, n1, n2);

  List<BinaryNode> pathToA = getPathTo(root, n1);
  List<BinaryNode> pathToB = getPathTo(root, n2);
  BinaryNode result = null;
  int size = Math.min(pathToA.size(), pathToB.size());
```

Az 1. ábrán látszik az egyik létrejöttének a helye. A getPathTo visszatérési értéke nincs megfelelően leellenőrizve, ezért keletkezik hiba a Math.min hívása során.

Érdekesség, hogy a generált tesztek között volt, ami a Randoop szerint sikeresen lefutott, azonban, ha én próbáltam futtatni, StackOverflow Exception keletkezett.

4.1.4 Tesztgenerálás alapértelmezett futási idővel

Kezdetben azt gondoltam időparaméter nélkül, addig készít teszteket a Randoop, amíg el nem ér egy adott lefedettségi szintet. Többszöri futtatás után rájöttem, hogy az időparaméter értéke alapértelmezetten száz másodperc. Arra voltam kíváncsi, mennyire különböznek egymástól a tesztek a véletlen tesztgenerálás miatt.

2. táblázat. Tesztgenerálás során kapott eredmény tulajdonságai

	Generált tesztek	Hibát adó tesztek	Osztály lefedettség	Metódus lefedettség	Sor lefedettség
Generálás 1	112	1	67	145	667
Generálás 2	112	1	69	152	692
Generálás 3	114	1	69	154	727
Generálás 4	113	1	69	152	693
Generálás 5	112	1	67	145	667
Átlag	112,6	1	68,2	149,6	689,2
Szórás			**1,095445**	**4,27785**	**24,68198**

3. táblázat. A vizsgált projekt tulajdonságai

Osztályok száma	Metódusok száma	Sorok száma
89	277	1713

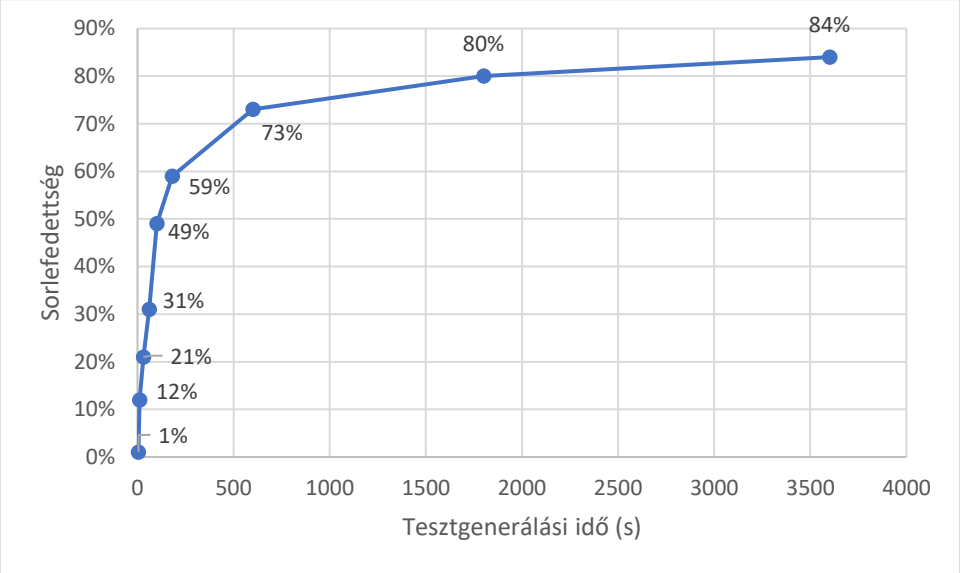
A 2. táblázat adatainak az elvi maximuma szerepel a 3. táblázatban. A második táblázatból az olvasható ki, egy-egy futtatás után a generált tesztek hogyan néztek ki, és mekkora lefedettséget értek el a kódon. Az utolsó két sor számított adatokat tartalmaz, az átlagos eredményeket és lefedettségnél pedig a szórást. A kapott eredmények miatt én úgy vélem, erre a projektre nincs nagy különbség az

adott idő alatt generált tesztekre, ugyanis csak 4% eltérés van sorlefedettségben a minimális és a maximális értékek között.

4.1.5 Tesztgenerálás változó futási idővel

Érdekes volt megvizsgálni, hogy a generálási idő mennyire befolyásolja a teszteket, illetve mennyig ideig van értelme egyáltalán futtatni a generálást. Ehhez végeztem méréseket.

2. ábra Sorlefedettség változása a tesztgenerálás futási idejének függvényében



4. táblázat. Változó idő alatt generált tesztek tulajdonságai

Tesztgenerálási idő (mp)	Generált tesztek	Hibát adó tesztek	Osztály lefedettség	Metódus lefedettség	Sor lefedettség
5	18	0	20%	1%	1%
10	54	0	51%	19%	12%
30	70	0	61%	31%	21%
60	96	1	71%	45%	31%
100	144	2	88%	64%	49%
180	182	4	91%	75%	59%
600	368	6	96%	88%	73%
1800	788	10	97%	92%	80%
3600	2161	17	96%	92%	84%

A 4. táblázatban különböző idő alatt generált tesztekről találhatóak meg adatok, mint a generálás futási ideje, generált tesztek száma, hibát adó tesztek száma, különböző lefedettségi értékek. Az 2. ábra a 4. táblázat első és utolsó oszlopának felhasználásával készült. Megfigyelhető, hogy kisebb generálási időknél sokat nyerhetünk kódlefedettségben az idő növelésével. Ez a 600 másodpercig generált tesztekig látszik, onnantól kevésbé lesz hatékony az idő növelése. A 3600 másodperces generálást egy erősebb számítógépen végeztem, látszik is abból, hogy a tesztek száma majdnem háromszorosa a fele

akkora idő alatt generáltak számának. Erre a projektre viszont így is csupán kis mértékű növekedést sikerült elérni.

5 Összegzés

A félév során sikeresen készítettem teszteket egy nyíltforráskódú projekthez, izolációt valósítottam meg egy másik projekten és teszteket generáltattam eszköz segítségével, amiket utána elemeztem is. Későbbiekben érdemes lehet kipróbálni a Randoopot más projekteken is és megvizsgálni, hogy a további paraméterei hogyan befolyásolják az eszköz viselkedését.