Budapesti Műszaki és Gazdaságtudományi Egyetem
Témalaboratórium (BMEVIMIAL00)

Automatikus tesztelés

Témalaboratórium beszámoló

Kövér Márton (W6HYOZ)
Konzulensek:
Honfi Dávid
Micskei Zoltán
2017/2018 1. félév

1 Tartalomjegyzék

2 Bevezető

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

3 Unit tesztelés

3.1 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 (https://github.com/pedrovgs/Algorithms), ott fork-oltam, majd megnyitottam IntelliJ IDEA segítségével.

A projekt nyolcvankilenc osztályt tartalmazott, ezek egyszerű problémák megoldására készültek (pl. Adatszerkezetek, Matematikai műveletek, Rendezések, String műveletek stb.), így általában csak néhány rövidebb függvényt tartalmaznak.

3.2 Meglévő tesztek a projekthez

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.

3.3 Saját tesztek készítése

Először is részletesebben megismerkedtem a tesztek felépítésével. Szükség lehet olyan kódrészletre, pl osztályok példányosítása, amit minden teszt előtt vagy után le kéne, 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. Létezik ezeknek másik változata is (BeforeClass, AfterClass), ezek csak egyszer futnak le, olyankor lehet rá 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. Utána olvastam, hogyan kell lefuttatni a teszteket kódlefedettség méréssel. A lefedettség elég magas volt, de sikerült még így is javítani rajta. Hozzáadtam egy új tesztosztályt, ami a Hello World! kiírását teszteli. Számomra érdekes volt, hogy tudom megnézni mi került a kimenetre. A MoveZeroesInArrayTest-hez és a GoThroughMatrixInSpiralTest-hez adtam még hozzá egy új teszteseteket.

	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)

1. táblázat

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

4 Függőségek kezelése

A függőség alatt modulok közötti kapcsolatot értünk. A külső függőségek problémát okazhatnak 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.

4.1 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 elérhető (https://github.com/FTSRG/swsv-labs/tree/master/development-testing/hu.bme.mit.spaceship). Ez a projekt egy űrhajót valósít meg, ami képes torpedók kilövésére, két különböző tárolóból. Kénytelen voltam átalakítani az űrhajó osztályt, ugyanis az általam modellezendő osztály privát attribútuma volt annak. Megoldás lehet, ha írunk Getter és Setter függvényeket, vagy ha protected- re állítjuk az attribútumot privát helyett. Én a másodikat választottam és egy TestShip osztályt hoztam létre, amely az eredeti űrhajóból származott, azonmban egy Setter segítségével megadhattuk, milyen torpedó tárolókat használjon.

4.2 Mockito

A Mockito (http://site.mockito.org) egy nyíltforráskódú projekt, a segítségével izolációs teszteket lehet készíteni. Forráskódja megtalálható GitHub-on, de azzal én nem foglalkoztam, csak használtam.

A Mockito-t több módon is próbáltam hozzáadni a projektemhez, de nem sikerült működére bírnom. Utána olvasva rájöttem, hogy nekem nem a core, hanem az all verzióra van szükségem, ugyanis az tartalmas minden szükséges függőséget, így azt hozzáadtam a projekthez maven könyvtárként. Innentől kezdve tudtam használni a funkcióit.

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. Azt a funkciót, ami megadja egy adott eljárás milyen argumentummal lett meghívva nem próbáltam ki, mert nem volt rá szükség a feladatom megoldásához.

5 Unit tesztek generálása

Unit tesztek automatikus generálásának az a lényege, hogy valamilyen eszköz segítségével teszteket generálunk a projektünkhöz. Automatikus generálás előnye, hogy rávilágíthat olyan hibákra, amit mi nem feltétlen vettünk volna észre, ugyanakkor nem biztos, hogy minden általunk kívánt funkció tesztelésre kerül.

5.1 Randoop

A Randoop (https://randoop.github.io/randoop) egy nyíltforráskódú projekt, forráskódja megtalálható GitHub-on,de ezt az eszközt is csak használtam. Unit teszteket generál visszajelzés irányított tesztgeneráció segítségével, így véletlenszerűen készít metódus, illetve konstruktor hívás szekvenciákat a tesztelés alatt lévő osztályhoz. Ezeket végrehajtva ellenőrzéseket készít, ami a program működését vizsgálja. Az ellenőrzésekből és a kódszekvenciákból készülnek a tesztek.

5.1.1 Telepítés és futtatás

A kiadott manual (https://randoop.github.io/randoop/manual/index.html) 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 rá kellett jönnöm, hogy a leírás Linuxhoz van és Windows-on van kisebb eltérés, az elérési utakat; kell, hogy elválassza és nem :. A ; miatt a parancssor két utasításnak értelmezte, amit beírtam, ezt javítottam idézőjelek használatával. Ezután arra kellett rájönnöm, hogy csak úgy tudom futtatni a teszteket, ha az eredeti package struktúrában vannak. A futtatáshoz hozzá kellett adnom a Path-oz 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.

5.1.2 A tesztelt projekt

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

5.1.3 Felfedezett hibák

A generált tesztek, amik Exception-t dobtak külön osztályba lettek elmentve. Sikerült három darab NullPointerException-ra is fényt deríteni a tesztek segítségével.

```
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());
```

1. ábra

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.

5.1.4 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.

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

2. táblázat

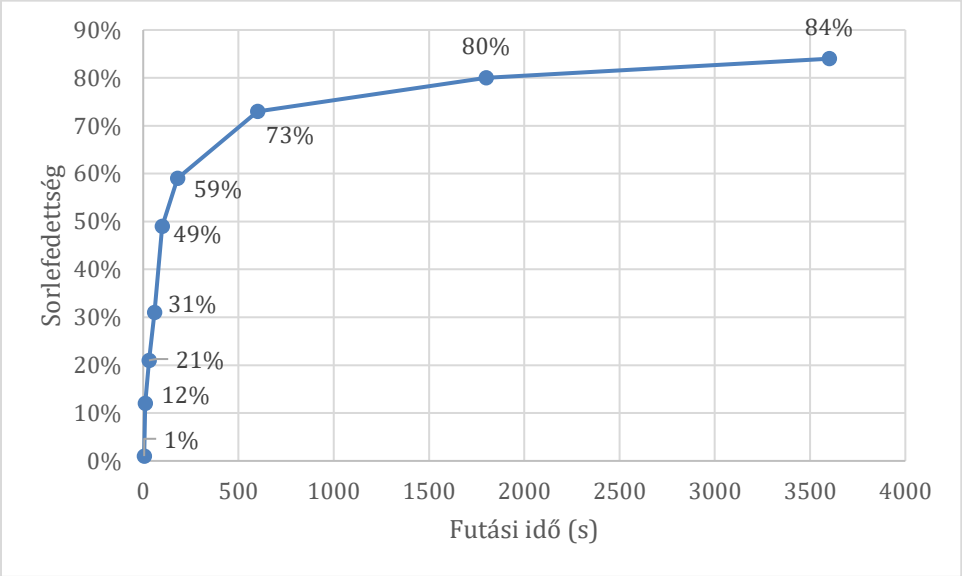
Osztályok száma	Metódusok száma	Sorok száma
89	277	1713

3. táblázat

A 2. táblázat adatinak az elvi maximuma szerepel a 3. táblázatban. A második táblázatból az olvasható ki, egy-egy futtatás után a generált tesztek, hogy néztek ki és mekkora lefedettséget értek el a kódon. Az utolsó két sor számított adatokat tartalmaz, az átalagos 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.

5.1.5 Tesztgenerálás változó futási idővel

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



1. diagram

Futási 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%

4. táblázat

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. A 1.diagram a 4. táblázat elő 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 háromszorosan a fele akkora idő alatt generáltak számának. Erre a projektre viszont, így is csupán kis mértékű növekedést sikerült elérni.

6 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 Randoop-ot más projekteken is és megvizsgálni a további paraméterei mire jók.