Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék
Önálló laboratórium (BMEVIMIAL01)

Mutációs tesztelés

SZABÓ ÁRON (AE0F10)

KONZULENS:
DR. MICSKEI ZOLTÁN

2018. MÁJUS 20.

Tartalomjegyzék

1. Bevezetés **2**
1.1. Motiváció és célkitűzés . . . 2
1.2. A mutációs tesztelésről . . . 2

2. Elterjedt mutációs tesztelési eszközök **2**
2.1. MAJOR Mutation Framework . . . 2
2.2. PITest . . . 2

3. Saját eszköz megtervezése **3**
3.1. Forráskód vagy bytekód manipuláció? . . . 3
3.1.1. Forráskód manipuláció . . . 3
3.1.2. Bytekód manipuláció . . . 4
3.1.3. Fordítóba integrált . . . 4
3.1.4. Döntés . . . 4
3.2. Lefedett kódok azonosítása . . . 4
3.2.1. Statikus analízis alapján . . . 4
3.2.2. Elnevezési konvenciók alapján . . . 5
3.2.3. Döntés . . . 5
3.3. Mutációs operátorok . . . 5
3.4. Mutációk beillesztése kódba . . . 5
3.4.1. Elágazás-struktúra . . . 6
3.4.2. Külön fileba írás . . . 6
3.4.3. Döntés . . . 6
3.5. Tesztek futtatása, kiértékelése . . . 6
3.5.1. Felügyelet a futtatás felett, párhuzamos futtatás . . . 6
3.6. Kimenet formátuma . . . 6

4. Saját eszköz fejlesztése **6**
4.1. Programstruktúra . . . 7
4.2. ASM Framework . . . 7
4.3. A mutánsokat előállító Java program . . . 7
4.3.1. Statikus analízis . . . 8
4.3.2. Mutator . . . 9
4.4. Shell program . . . 9
4.5. Kipróbálás . . . 11
4.5.1. Hiányosságok és nehézségek . . . 12
4.5.2. Tesztelés egy nagy projekten . . . 12

5. Végszó **12**

1. Bevezetés

1.1. Motiváció és célkitűzés

Az önálló laborom témájának a mutációs tesztelést választottam. Témalaborom alatt találkoztam ezzel a módszerrel, és úgy gondoltam, érdemes vele tovább foglalkozni. Célom volt a félévben elmélyülni ebben a témában, és egy saját mutációs tesztelő eszközt létrehozni.

1.2. A mutációs tesztelésről

Unit tesztelésnél a tesztkészlet minősége többféleképpen mérhető. Általában a programkód tesztek általi lefedettsége a meghatározó szempont, ezt egyszerűe és gyorsan lehet statikus analízis eszközökkel vizsgálni. A lefedettség-mérés viszont csak a lefutott kód hányadát nézi, a program és a tesztek összefüggését nem. Honnan tudjuk, hogy egy, a programban elkövetett hibát biztosan kiszűrnek-e a tesztesetek?

Vizsgáljuk a tesztkészlet és a tesztelendő szoftver kölcsönhatását! Tegyünk apró hibákat mesterségesen az eredeti programkódba, és nézzük meg, hogy a tesztek megtalálják-e ezeket. Az megtalált és az összes hibák arányából készítsünk egy pontszámot. A mutációs tesztelés alapvetően ebből a két lépésből áll.

Mutánsok létrehozásakor mindig csak egy-egy utasítást változtatunk azokban a kódokban, amik a tesztek során végrehajtódnak. Ezeken az utasításokon mutációs operátorokat futtatunk, ezek határozzák meg, hogy egy utasítás milyen másik utasítássá változhat át. Ezekre néhány példa: matematikai (összeadásból kivonás), feltétel negáló (egyenlőből nem egyenlő), visszatérési érték (változóból null). A mutáns programokat ezután lefuttatjuk a tesztkészleten: ha a teszt hibát jelez, akkor jó a teszteset, mivel "megölte" a mutánst. Ha viszont egy mutánsra az összes teszt rendben lefutott, a mutáns "életben maradt", a tesztkészlet nem tudta lebuktatni. Fontos, hogy egy mutánsban egyszerre több operátort nem hajtunk végre: mivel a mutációknak egyenként is felderíthetőknek kell lenniük, több mutáció együttes végrehajtásának nincs információtartalma.

A mutációs tesztelés egy nagy hátránya, hogy sokkal erőforrásigényesebb, mint egy szimpla lefedettségmérés. Egy nagyobb program vagy programkönyvtár több ezer mutánssal rendelkezhet, és ezekre a teszteket egyenként végrehajtani igencsak hosszadalmas procedúra.

2. Elterjedt mutációs tesztelési eszközök

A mutációs tesztelés ötlete több, mint 40 éves [1], ez idő alatt különféle tudású mutációs teszteszközt fejlesztettek ki különböző programnyelvekhez. Elsősorban a Java nyelvhez fejlesztett eszközökre fókuszáltam, mert a JUnit környezettel már sok tapasztalatom volt, és így otthonosabban ki tudtam próbálni ezeket az eszközöket. Két eszközt vizsgáltam meg alaposabban, ezek a Major és a PIT.

2.1. MAJOR Mutation Framework

A Major egy fejlett mutációs tesztelőkörnyezet Java platformhoz. Sok szempontból optimalizált, hatékonyan és gyorsan állítja elő és futtatja a mutánsokat. Saját konfigurációs nyelvvel (Major Mutation Language) rendelkezik, amivel a mutációs operátorok részletekbe menően beállíthatóak. Saját beépített Java fordítójának köszönhetően fordítás közben hozza létre a mutációkat, az így előállított mutációk pedig jobban igazodnak a nyelv sajátosságaihoz, és a programozó által potenciálisan elkövethető hibákat fogják szimulálni. A beépített fordító sajnos a hátránya is, mert jelen pillanatban a Major utolsó verziója Java 1.7-es fordítót használ, és Java 1.8-as projektekhez nem használható. Apache Anttel és önmagában is futtatható (az indító shell script némi módosítása után), futása gyors. Az eredmények CSV fileokban állnak elő, amikből részletes statisztika készíthető. [2]

2.2. PITest

A PIT egy modernebb eszköz, a kettő közül ennek a használata jobban kézre állt. A Majortól eltérően itt a már lefordított bytekódon történik a mutációs tesztelés, a mutánsok a memóriában jönnek létre és több szálon

hajtódnak végre. Mivel a bytekódban jönnek létre a muatációk, előfordulhatnak úgynevezett "junk" mutációk is, amik nem követhetők vissza a forráskódban elkövetett programozási hibákra, de a PIT algoritmusai ezeket képesek kiszűrni. Kényelmesen használható, Maven és Ant pluginokon kívül Eclipse és IntelliJ beépülők is készültek hozzá (habár ez utóbbinál a beállítási lehetőségek igen szegényesek). Mavenes indításnál az eszköz egyszerűbb paraméterei parancssorban adhatók meg (lásd 1. shell script), a részletes beállítások, mint például a mutációs operátorok, a pom.xml-ben. Ha a program muatációja jellemzően végtelen ciklusokat eredményez, érdemes egy timeout paramétert megadni, ami idő letelte után a PIT leállítja a teszt processzt. [3]

```
mvn test
mvn org.pitest:pitest-maven:mutationCoverage -DmaxDependencyDistance=1 -DtimeoutConstant=250
```

Listing 1. Példa a PITest paraméterezésére

Lefuttattam a PIT-et egy olyan projektre, aminek fejlesztésében részt vettem (vasutas játék): a sorlefedettség 18% (276/1495), a mutációs pontszám pedig 9% (93/1007) volt. A lefedettség azért lett ilyen alacsony, mert Unit tesztek csak a kontroller és modell osztályokra léteztek, a program jelentős hányadát pedig UI kódok, segédeszközök tették ki. Az eredményt (2. részlet) megvizsgálva látható, hogy mely utasítások hatása nincs vizsgálva a teszt során (hiszen ha lenne, a mutáció során sikertelen lett volna a teszt). A kimutatásból visszakereshető a kódrészlet sorra pontosan, így a hiányosság nagyon gyorsan javítható.

```
102     1. negated conditional - KILLED
...
116     1. removed call to serpenyoe/model/rails/Tunnel::setActive - KILLED
117     1. removed call to serpenyoe/model/rails/Tunnel::setB - SURVIVED
119     1. removed call to serpenyoe/model/Rail::setA - KILLED
120     1. removed call to serpenyoe/model/Rail::setB - KILLED
```

Listing 2. PIT kimenete

3. Saját eszköz megtervezése

Az önálló laborom egyik célja egy saját eszköz megtervezése és implementációja volt. Természetesen a piacon lévő eszközökhöz képest, amiket évek óta fejlesztenek és működésük nagymértékben optimalizált, jelentős egyszerűsítésekkel éltem. A fő szempont a mutációs tesztelés lépéseinek szemléltetése és egy működő prototípus elkészítése volt.

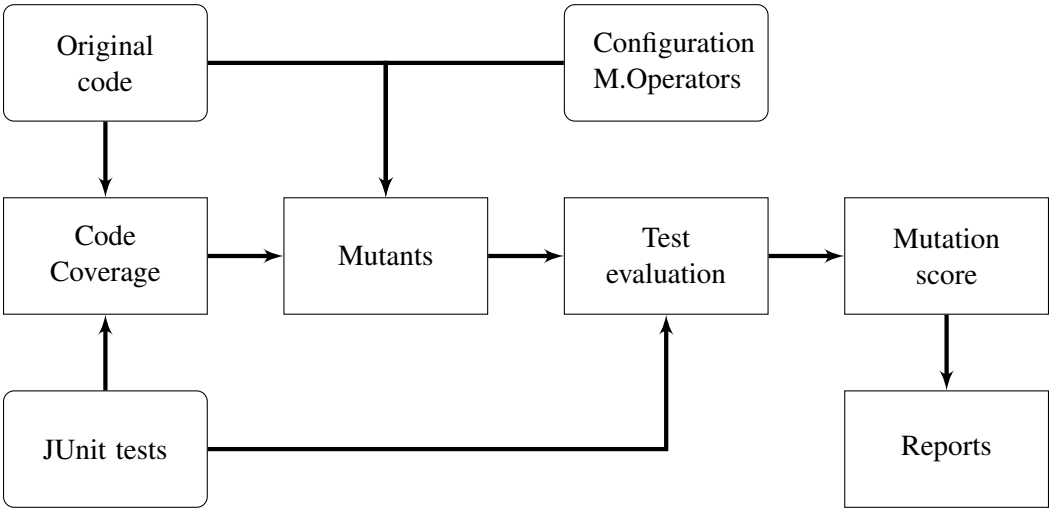
A tevezés során is leginkább az egyszerűségre törekedtem, de ha kettő közül a jobb megoldás nem volt nagyságrendekkel nehezebb, inkább azt választottam. Megfigyeltem a fent említett eszközök működését, és felvázoltam a mutációs tesztelés egyes lépéseit (1. ábra), valamint, hogy milyen lehetőségek vannak azok megvalósítására.

3.1. Forráskód vagy bytekód manipuláció?

A legfontosabb kérdés, hogy a manipulációt a szoftverfolyamat melyik fázisában hajtjuk végre: fordítás előtt, közben, vagy utána? Mindegyik módszernek megvan a maga előnye és hátránya, és mind sajátos megközelítést kívánnak, ezért ezt a kérdést a tervezés legelején el kellett dönteni.

3.1.1. Forráskód manipuláció

Elsőre a forráskód manipuláció egyszerűbbnek és látványosabbnak tűnhet. Itt közvetlenül meg lehet figyelni a mutációs eszköz működését, és a bonyolultabb nyelvi elemek mutációjára is lehetőség van. Viszont itt szükség van java szintaxis értelmező modulra, hogy a módosított forrráskódot előállíthassuk, illetve a fordítás jelentősen időigényesebb lehet, mivel minden egyes mutációnál meg kell ismételni.



1. ábra. A mutációs tesztelés lépései

3.1.2. Bytekód manipuláció

A már lefordított program bytekódját egyszerű módosítani (akár framework segítségével: ASM). Előállhatnak így "junk" mutációk, amik nem vezethetők vissza forráskódban elkövethető hibákra. Előnye, hogy a mutáció egyszerűen elvégezhető a memóriában is, és (egyébként is) kevésbé erőforrásigényes. Hátránya, ha a forrásban látni szeretnénk a mutációkat, azokat vissza kell követni.

3.1.3. Fordítóba integrált

Ha a mutációs eszköz a fordítóba van építve, az ötvözni tudja az előző két módszer jó tulajdonságait. Hátránya, hogy viszonylag bonyolult elkészíteni, és meglehetősen rugalmatlan (nem követi a Java verziószámokat, körülményes futtatni), és ilyenkor a mutánsokat fileba kell írni.

3.1.4. Döntés

A bytekód manipulációt választottam, mert alapvetően megfelel az igényeimnek, és elég egyszerű ahhoz, hogy az önálló labor alatt megvalósítható legyen.

3.2. Lefedett kódok azonosítása

Ahhoz, hogy tudjuk, a mutációkat hova kell beilleszteni a kódba, tudni kell, hogy a vizsgált tesztre mely részek fognak lefutni a programból. A célom az volt, hogy a mutációk csak a tesztosztállyal azonos packageban lévő osztályokon történjenek csak meg, elkerülve azt, hogy külső könyvtárakon és irreleváns kódokon is lefut az eszköz.

3.2.1. Statikus analízis alapján

Megvizsgáljuk, hogy a teszt milyen függvényeket hív meg az eredeti programból. Ehhez vagy egy forráskód értelmezőre van szükség, vagy a bytekódból gyűjtjük ki a más osztályokra mutató függvényhívásokat. Ki kell szűrni azokat a programrészeket (libraryket), amik nem tartoznak szorosan a vizsgált programhoz, például a tesztek és az osztály package egyezésének vizsgálatával. Hibája, hogy az osztályok statikus inicializáló függvényét figyelmen kívül hagyja, mivel azokat nem hívja semmi [4], de ez legtöbb esetben nem okoz problémát.

3.2.2. Elnevezési konvenciók alapján

A módszer lényege az, hogy a teszt/tesztkészlet nevéből konvenciók alapján következtetünk, melyik részek/osztályok fognak futni a programból. Legtöbb esetben a tesztosztály neve visszautal arra az osztályra vagy unitra, amihez tartozik, így ez a módszer a külső könyvtárakat, és a mutálni nem kívánt más programkódokat működéséből fakadóan kiszűri. Ez a módszer viszont igen pontatlan lehet, ha nem követjük pontosan az elnevezés szabályait. Előnye viszont, hogy igen egyszerű megvalósítani, csupán String-átalakítások szükségesek hozzá..

3.2.3. Döntés

Az ASM képes modellt alkotni a vizsgált bytekódról, tehát arra is megfelelő, hogy a tesztekben a programkód függvényeinek hívásait azonosítsuk vele. Mivel így nem járt többletmunkával, az előnyök miatt a statikus analízist választottam, azzal az egyszerűsítéssel, hogy csak a tesztből közvetlenül hívott függvényeken hajtódik végre mutáció.

3.3. Mutációs operátorok

Egy meglévő operátor készletet könnyen ki lehet egészíteni továbbiakkal, erre a projektre a feltétel negáló operátorokat, néhány matematikai, valamint függvényhívás operátort valósítottam meg (a bytekód manipuláció miatt ezek JVM instrukciók, a típusnevek stacken lévő értékeket jelölnek):

- `if(int==int)` → `if(int!=int)`
- `if(int!=int)` → `if(int==int)`
- `if(int>=int)` → `if(int<int)`
- `if(int<=int)` → `if(int>int)`
- `if(int>int)` → `if(int<=int)`
- `if(int<int)` → `if(int>=int)`
- `if(int==0)` → `if(int!=0)`
- `if(int!=0)` → `if(int==0)`
- `if(int>=0)` → `if(int<0)`
- `if(int<=0)` → `if(int>0)`
- `if(int>0)` → `if(int<=0)`
- `if(int<0)` → `if(int>=0)`
- `if(Object==null)` → `if(Object!=null)`
- `if(Object!=null)` → `if(Object==null)`
- `float+float` → `float-float`
- `double+double` → `double-double`
- `long+long` → `long-long`
- `int+int` → `int-int`
- `float-float` → `float+float`
- `double-double` → `double+double`
- `long-long` → `long+long`
- `int-int` → `int+int`
- virtuális függvényhívás → nincs művelet
- interface függvényhívás → nincs művelet

Ezek azok az operátorok, amiket viszonylag könnyű volt implementálni, és a legtöbb programra hatékonyan működnek.

3.4. Mutációk beillesztése kódba

Fontos szempont, hogy a mutációkor létrehozott kódok később, a futáskor milyen módon lesznek azonosíthatóak. Többféle módszer létezik a mutációk generálás utáni rendszerezésére, hogy a tesztek futtatásakor kiválaszthatók legyenek.

3.4.1. Elágazás-struktúra

Az első esetben egy program jön létre, ami az összes mutációt tartalmazza elágazásos szerkezetben. Futáskor a megfelelő flag beállításával kiválasztható az az egy mutáció, ami le fog futni. A mutáció azonosítóját paraméterként kell átadni a programnak, és gondoskodni róla, hogy az flag formájában eljusson az elágazásig. A legtöbb teszteszköz ezt alkalmazza, mert hatékony, takarékos a tárhellyel. Emiatt, ha a mutációk a memóriában történnek, szinte csak ez jöhet szóba.

3.4.2. Külön fileba írás

Minden mutáció egy külön class filet kap. Az egyes mutációkat a filenév azonosítja. Ez a módszer több tárhelyet foglal, de egyszerűbben végrehajtható.

3.4.3. Döntés

Végül a fileba írás mellett döntöttem, mivel a JUniton keresztül körülményes a paraméterek átadása. A mutációk átmásolódnak az eredeti class helyére a JUnit futása előtt, a futás után pedig törlődnek.

3.5. Tesztek futtatása, kiértékelése

Az eszköz a mutánsokat kiértékelteti a JUnit segítségével. Paraméterként a JUnitnak át kell adni valamilyen módon, hogy mit hajtson végre, a kimenenetét pedig értelmezni kell. Arra jutottam, hogy egy shell program segítségével ez egyszerűen megoldható, hiszen a mutáns a JUnit futása előtt másolással lett kiválasztva, a kimenetet (OK/Failed) pedig a grep segédprogram segítségével egyszerűen lehet értelmezni. A programom keretét ezért egy egyszerű shell script adja, és csak a mutációk előállítását fogja egy Java program végezni.

3.5.1. Felügyelet a futtatás felett, párhuzamos futtatás

Egy ciklus mutációjakor (vagy egyébként is) létrejöhet olyan helyzet, hogy a tesztelt program végtelen ciklusba kerül, vagy egyszerűen rendellenesen sokáig tart egy teszteset futása. Szükség lehet egy maximum futási idő meghatározására, ami után a főprogram kilövi a teszt processt. A tervezett operátorkészlet feltétel negáló operátorai sajnos potenciálisan előállíthatnak ilyen helyzetet. A fejlettebb teszteszközök képesek arra is, hogy több szálon, akár több processzormagot használva több mutációt futtatnak egyszerre. A processek párhuzamos kezelése viszont sajnos egy olyan extra funkció, ami nem férne bele az önálló labor kereteibe.

3.6. Kimenet formátuma

A kimenet formátumára rengetegféle lehetőség van, az elterjedt eszközök is sokféleképpen oldják meg. Az eredményt legjobban szemléltető módszer HTML formátumban, diagramokat és hiperhivatkozásokat használni. CSV-ben is eltárolhatók az adatok, a későbbi könnyű feldolgozás érdekében. Mivel egyelőre az eredménnyel semmi más tervem nincs, úgy döntöttem, hogy az egyes pontszámokat a konzol kimenetére írom ki.

4. Saját eszköz fejlesztése

A félévben fejlesztett szoftverem egy parancssoros eszköz: paraméterként egy Java projekt elérési útját kapja, kimenetként pedig előállítja az osztályok mutációs pontszámát. Az eszköz Maven könyvtárstruktúrát feltételez, a megfelelő mappákban keresi a lefordított program bytekódot és teszt bytekódot. Futtatás előtt ajánlott kiadni a `mvn test` parancsot, hogy a megfelelő állományok előálljanak, valamint hogy meggyőződjünk arról, hogy a tesztkészlet sikerrel lefut, hiszen a mutációs tesztelésnek csak ekkor van értelme.

4.1. Programstruktúra

A mutációs tesztelőeszközöm két fő részből áll: egy, a mutációkat előállító Java program, és egy UNIX shellben futtatható keret script. A shell script feladata a tesztek megkeresése, rájuk a mutációkat előállító Java program meghívása, majd a létrejött mutáns osztályokra egyenként a tesztelést elindítani. Ezután a script egybegyűjti az eredményeket, és a mutációs pontszámot osztályokra lebontva mutatja.

4.2. ASM Framework

Ahhoz, hogy a Java bytekódhoz hatékonyan hozzáférhessek, az ASM Java Bytecode Manipulation Frameworköt használtam. Ez a könyvtár lehetővé teszi, hogy a kód manuális beolvasása helyett, a kódstruktúrát már feldolgozva kapjuk. A bytekód Visitor elvvel járható be, az egyes függvényeket, utasításokat könnyedén lehet egyenként kezelni. Így egyaránt alkalmas a tesztosztályok statikus analízisére és a mutációk végrehajtására is. A Visitor metódusokban eszközölt változások a végén visszaírhatók fileba.

Az osztályok beolvasása és kiírása ClassReader és ClassWriter osztályokkal lehetséges (3. kódrészlet). A bejárás során először a ClassVisitor osztály lép működésbe, ez lépked végig az osztály elemein. Szempontunkból a visitMethod függvény érdekes, ez adja tovább a vezérlést a MethodVisitor osztálynak. Ebben az egyes utasítások bejárására különböző paraméterezésű függvények vannak (4. kódrészlet), amelyek OpCode paramétere azonosítja az utasításokat (Opcodes.IF_ICMPEQ, Opcodes.FADD...), és itt le is cserélhetők bármelyik másikra.

```
ClassReader reader = new ClassReader(new FileInputStream(...));
ClassWriter writer = new ClassWriter(reader, 0);
ClassVisitor visitor = new ClassVisitor(writer);
reader.accept(visitor, 0);
byte[] newBytecode = writer.toByteArray();
FileOutputStream output = new FileOutputStream(...);
output.write(newBytecode);
output.flush();
output.close();
```

Listing 3. Java bytekód beolvasása, feldolgozása, kiíratása ASM segítségével

```
public void visitJumpInsn(int opcode, Label label);
public void visitIntInsn(int opcode, int operand);
public void visitInsn(int opcode);
```

Listing 4. A MethodVisitor általam használt bejáró függvényeinek fejlécei

4.3. A mutánsokat előállító Java program

A Java programom az egyszerűbb adatátadás érdekében két egymásra épülő feladatot hajt végre: a lefedettség analízist, és a mutációk előállítását. A kódbázist ennek megfelelően két packagere osztottam fel: mutator és testanalyzer, ezek működése független egymástól, csak az adatmodell osztályon osztoznak. Ezen kívül egy harmadik package tartalmazza a kontroller Main függvényt (5. kódrészlet), aminek feladata a két feladat elindítása, köztük az adatáramlás biztosítása, valamint a mutator felkonfigurálása mutációs operátorokkal.

A Main függvény parancssori paramétere egy tesztosztály elérési útja, az eszköz ezt fogja átvizsgálni, innen hivatkozott függvényekben a mutációkat végrehajtani, majd a mutáns classokat az eredetik könyvtárába kimenteni.

Mindkét feladathoz az ASM keretrendszerre volt szükség: a statikus analízis és a mutációk előállítása is bytekód szinten valósul meg. A ClassVisitor és MethodVisitor osztályok leszármaztatásával ezekhez tudtam igazítani a kódok bejárását.

```
    public static void main(String[] args) throws Exception {
        List<MethodId> methods;
        methods = TestClassVisitor.analyzeTest(new File(args[0]), "");

        Mutator mutator = new Mutator();
        mutator.addMutationOperator(Opcodes.IF_ICMPEQ, Opcodes.IF_ICMPNE);
        mutator.addMutationOperator(Opcodes.IF_ICMPNE, Opcodes.IF_ICMPEQ);
        // ...
        mutator.addMutationOperator(Opcodes.IFNONNULL, Opcodes.IFNULL);
        mutator.addMutationOperator(Opcodes.INVOKEVIRTUAL, Opcodes.NOP);
        mutator.addMutationOperator(Opcodes.INVOKEINTERFACE, Opcodes.NOP);
        String classpath=getClassesPath(new File(args[0]));
        for(MethodId method : methods){
            MutationClassVisitor.mutateMethod(method, classpath, mutator);
        }
    }
```

Listing 5. Main függvény

4.3.1. Statikus analízis

Ez a modul egy listába gyűjti a kapott teszt osztályban hivatkozott függvényeket a tesztelt programból. A visszatérő adatstruktúra minden lényeges információt tartalmaz ahhoz, hogy a Mutator modul a hivatkozott részletet megtalálhassa (6. kódrészlet).

```
public class MethodId {
    public String owner;
    public String name;
    public String descriptor;
    public boolean isInterface;
    // ...
}
```

Listing 6. MethodId adatstruktúra

A modul működése egyszerű: a TestClassVisitor (7. kódrészlet) meghívja az egyes JUnit tesztfüggvényekre a TestMethodVisitort (8. kódrészlet), amiben a visitMethodInsn keresi a külső (eredeti programra hivatkozó) függvényhívásokat. A megtalált hívásokat visszajuttatja a TestClassVisitornak, mely megvizsgálja, hogy az megfelel-e a feltételeknek: a függvény osztálya azonos packageben legyen, mint a tesztosztály, illetve még ne szerepeljen a listában (mivel a mutációkat tesztosztályonként maximum egyszer szeretnénk csak végrehajtani).

```
public class TestClassVisitor extends ClassVisitor{
    List<MethodId> visitedMethods = new LinkedList<>();
    // ...
    public void pushMethod(MethodId method){
        if(!checkPackage(method.owner))return;
        for(MethodId mid : visitedMethods){
            if(mid.equals(method))return;
        }

        visitedMethods.add(method);
    }
    @Override
    public MethodVisitor visitMethod(int access, String name, String descriptor,
                                     String signature, String[] exceptions) {
        // ...
        return new TestMethodVisitor(DEFAULT_API, super.visitMethod(access, name,
                                    descriptor, signature, exceptions),this);
    }
    // ...
}
```

Listing 7. TestClassVisitor

```
public class TestMethodVisitor extends MethodVisitor {
    // ...
    @Override
    public void visitMethodInsn(int opcode, String owner, String name,
                                String descriptor, boolean isInterface) {
        if(opcode== Opcodes.INVOKEVIRTUAL){
            mtv.pushMethod(new MethodId(owner, name, descriptor, isInterface));
        }
        super.visitMethodInsn(opcode, owner, name, descriptor, isInterface);
    }
}
```

Listing 8. TestMethodVisitor

4.3.2. Mutator

A Main függvény a statikus analízis során kapott függvényekre sorban meghívja a mutateMethod függvényt (9. kódrészlet). Ennek feladata a kapott függvény többszöri bejárása, egészen addig, amíg az összes lehetséges mutáció nem lett végrehajtva rajta. A mutáció adatait egy Mutation nevű adatstruktúra (11. kódrészlet) tartalmazza, ami a bejárás során (10. kódrészlet) feltöltődik a megfelelő adatokkal. A Mutator osztály feladata a mutációs operátorok tárolása, valamint egy lista a már elvégzett mutációkról (hogy a bejárások során ugyanaz a mutáns ne jöjjön létre többször).

```
    public static void mutateMethod(MethodId mid, String cp, Mutator mut) throws IOException {
        File originalFile = new File(cp+"/"+mid.owner+".class");
        File renamedOriginalFile = new File(cp+"/"+mid.owner+".class.original");
        originalFile.renameTo(renamedOriginalFile);

        Mutation m;
        do{
            m = mut.createMutation();
            ClassReader cr = new ClassReader(new FileInputStream(renamedOriginalFile));
            ClassWriter cw = new ClassWriter(cr,0);
            MutationClassVisitor cv = new MutationClassVisitor(cw, mid, m);
            cr.accept(cv, 0);
            byte[] newBytecode = cw.toByteArray();
            if(m.label==null) break;
            File mutationFile = new File(cp+"/"+mid.owner+".class."+m.label+".mutation");
            FileOutputStream fos = new FileOutputStream(mutationFile);
            fos.write(newBytecode);
            fos.flush();
            fos.close();
        }while(true);
    }
```

Listing 9. Mutate Method

Az egyes mutációk egy sorszámmal lettek azonosítva, ami egyértelműen meghatározza a tartalmazó függvényt és a mutáció helyét. Ez az azonosító a Mutation osztályon keresztül jut el a mutateMethod függvényhez, ahol az a filenév részeként használja, és ez kerül bele a Mutator osztály listájába is. A canMutate függvény visszaadja, hogy a mutáció lehetséges-e vagy nem: az alapján, hogy volt-e már a mutáció, illetve az operátorok között szerepel-e. Ezután a mutate függvény visszaadja a megfelelő mutáns Opcodeot, és felveszi a mutációt a listába. Amennyiben az opcode változott, a visit metódusok visszatérte után az ASM keretrendszer végrehajtja a megfelelő változtatást a bytekódon.

A Java program futása során előálltak a mutáns classok, a megfelelő package könyvtárban, futásra készen.

4.4. Shell program

A shell script feladata a shellből egyszerűbben elvégezhető feladatok megoldása: a tesztosztályok, majd a mutánsok megtalálása (find), JUnit futtatása és az eredmények értelmezése (java+grep), mutációs pontszám összesítése (wc+grep). Az egyes string műveleteket reguláris kifejezésekkel végeztem.

```
public class MutationMethodVisitor extends MethodVisitor {
    private Mutation mutation;
    private int opCounter=0;
    // ...
    private int mutate(int opcode){
        opCounter++;
        String opLabel = "OP" + opCounter;
        if(mutation.canMutate(opLabel,opcode)){
            int mutated=mutation.mutate(opLabel, opcode);
            return mutated;
        }
        return opcode;
    }
    // ...
    @Override
    public void visitIntInsn(int opcode, int operand) {
        opcode=mutate(opcode);
        super.visitIntInsn(opcode, operand);
    }
}
```

Listing 10. MutationMethodVisitor

```
public class Mutation {
    Mutator mut;
    String label=null;

    public Mutation(Mutator mut) {
        this.mut = mut;
    }
    public boolean canMutate(String label, int opcode){
        return (!mut.mutatedLabels.contains(label)) && mut.mutationOperators.containsKey(opcode)
                                && (this.label==null);
    }
    public int mutate(String label, int opcode){
        this.label=label;
        mut.mutatedLabels.add(label);
        return mut.mutationOperators.get(opcode);

    }
}
```

Listing 11. A Mutation osztály

A JUnit futtatása minden egyes mutációra lezajlik, a megelelő mutáció bemásolódik az eredeti class helyére. A program futásának végén helyreállnak az előzetesen kimentett eredeti class fileok.

Az eredmények összesítése a projekten belül létrehozott mappában történt: megölt mutáció esetén a megfelelő classhoz tartozó fileba egy 1-es került, életben maradt mutánshoz 0. A végén az 1-esek aránya az összeshez jelentette a mutációs pontszámot.

```
  echo "RUNNING $(basename $mut) [$clsname]"
  result=$(java -cp "$BASEDIR/lib/*:$1/target/test-classes:$1/target/classes" \
                                    org.junit.runner.JUnitCore $2 | grep "Tests run\|OK")
  if echo $result | grep "OK"; then
    echo "0" >> "$1/target/arontest-reports/$clsname"
  else
    echo "1" >> "$1/target/arontest-reports/$clsname"
  fi
```

Listing 12. Részlet a shell scriptből

4.5. Kipróbálás

Az elkészült eszközt először egy nagyon egyszerű projekttel próbáltam, ami egyetlen függvényt tartalmazott. A függvényen belüli összehasonlító elágazásokkal jól lehetett szemléltetni a mutációs operátorok működését.

```
public class SimpleClass {
    public int something(int param){
        if(param<1) hey(1);
        if(param>2){
            return 2;
        }else{
            if(param==3)return 4;
        }
        return param;
    }
    void hey(int i){
    }
}
```

Listing 13. Simple Class

```
public class SimpleClassTest {
    @Test
    public void SomethingGreater(){
        SimpleClass h = new SimpleClass();
        Assert.assertEquals(2,h.something(3));
    }
    @Test
    public void SomethingSmaller(){
        SimpleClass h = new SimpleClass();
        Assert.assertEquals(1,h.something(1));
    }
}
```

Listing 14. Simple Class Test

```
WITHOUT MUTATION
OK (2 tests)
ANALYZING TEST [...] testproj/SimpleClassTest.class
METHOD testproj/SimpleClass/something
CREATED MUTATION OP7131091982 (IF_ICMPGE->IF_ICMPLT)
WRITTEN SimpleClass.class.OP7131091982.mutation
CREATED MUTATION OP7131091984 (INVOKEVIRTUAL->NOP)
WRITTEN SimpleClass.class.OP7131091984.mutation
CREATED MUTATION OP7131091986 (IF_ICMPLE->IF_ICMPGT)
WRITTEN SimpleClass.class.OP7131091986.mutation
CREATED MUTATION OP71310919810 (IF_ICMPNE->IF_ICMPEQ)
WRITTEN SimpleClass.class.OP71310919810.mutation
METHOD testproj/SimpleClass/something
RUNNING SimpleClass.class.OP71310919810.mutation [testproj.SimpleClass]
Killed (Tests run: 2,  Failures: 1)
RUNNING SimpleClass.class.OP7131091982.mutation [testproj.SimpleClass]
OK (2 tests)
Survived (OK (2 tests))
RUNNING SimpleClass.class.OP7131091984.mutation [testproj.SimpleClass]
OK (2 tests)
Survived (OK (2 tests))
RUNNING SimpleClass.class.OP7131091986.mutation [testproj.SimpleClass]
Killed (Tests run: 2,  Failures: 2)
testproj.SimpleClass: 2/4
```

Listing 15. Az eszköz kimenete erre a programra

Működés közben az eszköz kiírja a standard outputra az éppen elvégzett feladatot, így könnyen nyomon lehet követni a működését. Az utolsó sorban az eszköz kiírja az egyes osztályokra adott mutációs pontot: ebben

	PITest	**Saját eszköz**
org.docx4j.toc.TocGenerator	12/41	9/46
org.docx4j.wml.SdtBlock	0/8	0/20
org.docx4j.wml.SdtContentBlock	1/3	4/4
org.docx4j.wml.SdtPr	4/19	24/24

1. táblázat. A docx4j mutációs tesztelésének eredménye

az esetben az egyetlen osztály négy mutációjából kettőt sikerült megölni. A kimenetből látható, hogy az eszköz a várt módon működik: a visszaadott értékre hatással lévő elágazások mutációját lebuktatja a teszten belüli assert. A függvény elején lévő elágazást és a függvényhívást viszont a tesztkészlet nem tudta vizsgálni, így a belőlük létrejött mutációk életben maradtak.

4.5.1. Hiányosságok és nehézségek

Függvényhívások törlésekor a programom nem tudta lekövetni a stack méretének változtatását. Mivel a program csak magát a hívást detektálta, a függvényparamétereket a stackbe helyező és a visszatérési értéket onnan kivevő utasítások a bytekódban maradtak. A programom arra az esetre jól működhet, amikor a paraméterek mérete és a visszatérési értékek mérete megegyezik (habár ilyenkor sem mindig), egyéb esetben viszont mindig hibát jelez (és ekkor hibásan a megölt mutánsok közé számolja ezt az esetet).

A mutációs teszteszközöm a tesztosztályokra egymástól függetlenül hajtja végre a mutációkat, valamint futtatja azokat, így több tesztosztály esetén elképzelhető, hogy ugyanaz a mutáció többször létrejön, és akár más-más eredményt adnak. Azonban mivel csak a tesztosztályhoz szorosan kapcsolódó (vele egy packageben lévő, egy hívás mélységben lévő) kódra hajt végre mutációkat, illetve általában egy tesztosztály egy unitot tesztel, ez a legtöbb esetben nem probléma.

4.5.2. Tesztelés egy nagy projekten

A szoftveremet egy nagy projektre is lefuttattam, erre a célra a docx4j (plutext/docx4j) nevű projekt megfelelt, mivel Java nyelvű, kevés külső függőséget alkalmaz, sok teszttel rendelkezik, és Mavenes projekt. Éltem azzal az egyszerűsítéssel, hogy a tesztosztályokból csak hármat hagytam meg, a gyors futás érdekében.

A projektet lefuttattam a PITesttel is, és törekedtem arra, hogy minél hasonlóbb paraméterezéssel fusson a két program. A mutációs operátorok közül a feltétel negáló és matematikai operátorokat alkalmaztam. A kimeneten (1. táblázat) látszik, hogy sok esetben hasonló eredményre jutott a két eszköz, habár az utolsó osztálynál a saját eszköz furcsa módon az összes mutációt lebuktatta (lehetséges, hogy a mutációk futtatásakor hiba történt).

5. Végszó

A félévben sikerült mélyebben megismerkednem a mutációs tesztelés lehetőségeivel, technikáival, számba vettem az ehhez szükséges lépéseket, majd kifejlesztettem egy saját eszközt, ami során kitapasztaltam a bytekód manipuláció működését. Az elkészült eszköz kicsi és demonstrációs célra alkalmas, de teljesítményben és lehetőségekben nem tud versenybe szállni a piacon lévő eszközökkel.

Úgy érzem, ezt a témakört a lehetőségekhez mérten kimerítettem, a következőkben a tesztelés egy más témakörével szeretnék majd foglalkozni.

Hivatkozások

[1] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. *MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler*.
`http://people.cs.umass.edu/~rjust/publ/major_compiler_ase_2011.pdf`

[2] *The Major Mutation Framework Manual*, version 1.3.2
`http://mutation-testing.org/doc/major.pdf`

[3] *PIT Quick Start*
`http://pitest.org/quickstart/`

[4] Henry Coles. *So you want to build a mutation testing system*.
`https://github.com/hcoles/pitest/`