4. gyakorlat: Teszttervezés és integráció

Kombinatorikus teszttervezés

Az első feladatban a kombinatorikus teszttervezés módszerét nézzük meg a gyakorlatban *n* paraméter *t*-szeres lefedettségét (t-wise coverage) garantáló tesztkészlet előállításával.

Adott a következő tesztelendő metódus:

```
int calculateEngineInput( GearMode g, bool economyMode, int acceleration )
```

ahol a `GearMode` a `{Backward, None, Parking, Drive}` halmazból veheti az értékét, az `acceleration` pedig egy 0-3 közötti szám.

A paraméterek kombinációját akarjuk vizsgálni, de nem akarjuk az összes lehetséges 32 kombinációt végigpróbálni. Először megelégszünk azzal, ha tetszőleges két paraméter esetén az összes lehetséges kombináció legalább egyszer szerepel (2-wise vagy pair-wise coverage).

T-szeres lefedettség számolásához az ACTS[1] eszközt fogjuk használni.

1. Hány tesztet kéne generálni a pair-wise lefedettség teljesítéséhez? Hogyan állnánk neki egy ilyen minimális elemszámú tesztkészlet kézi előállításának?
2. Vegyük fel a paraméterek fenti rendszerét az eszközben, és generáltassunk hozzá 2-szeres fedést biztosító tesztkészletet! Hány tesztesetünk lett?
3. A fenti metódus finomítása során kiderül, hogy nem minden bemeneti kombináció lehetséges, Backward esetén nem lehet az economyMode értéke igaz. Vegyük fel ezt a kényszert az ACTS-be[2], és generáljunk új tesztkészletet. Hogyan módosultak a tesztek?
4. A rendszer fejlesztése során bekerült egy új `BreakStatus` paraméter `{Pressing, Releasing, None}` értékkészlettel. Hogyan változik ekkor a tesztkészletünk?
5. Állítsuk át, hogy 3-szoros paraméter lefedettséget generáljon az eszköz, és készítsünk így is egy tesztkészletet.

(Az eszköz előnye igazából nagyobb méretű paramétertérnél jön ki még inkább, töltsük be a tcas.xml fájlban elmentett rendszert is, és generáljunk ehhez 3-szoros lefedettséget garantáló készletet.)

[1] http://csrc.nist.gov/groups/SNS/acts/index.html
[2] Az ACTS-ben használandó kényszerek szintaktikáját a User_Guide.pdf-ben megtaláljuk.

Kód lefedettség mérése

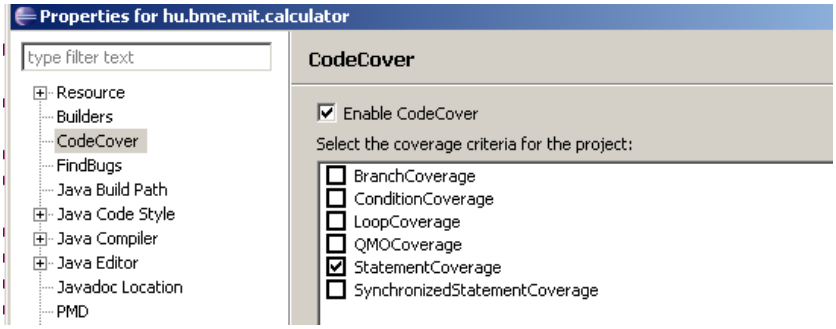
Az első feladat során egy meglévő tesztkészlet által elért lefedettséget (coverage) fogjuk megvizsgálni különböző kritériumok alapján, majd az azonosított hiányosságok alapján új teszteseteket implementálunk (`C:\code\GYAK4\Calculator`).

A kód lefedettség méréséhez a ***CodeCover***[3] eszközt használjuk, mely a következő URL-ről telepíthető az Eclipse-en belül: http://update.codecover.org/ (a kiadott virtuális gépen ezt már elvégeztük).

Figyelem: a CodeCover nem működik megfelelően az Eclipse újabb verzióival, ezért most a gyakorlaton a 4.3-as Eclipse példányt kell elindítani!

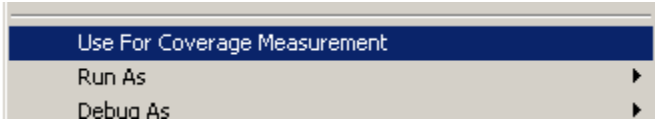
1. Váltsunk át a CodeCover perspektívára, és állítsuk be a CodeCovert!

A CodeCover használatához először be kell kapcsolni a projekten, hogy milyen típusú lefedettséget akarunk mérni (*Project / Properties*). Válasszuk az utasítás lefedettséget (*StatementCoverage*).



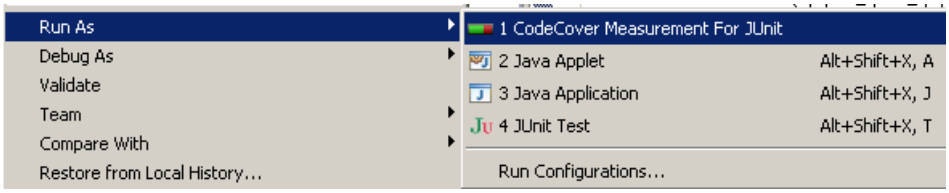
1. ábra: CodeCover engedélyezése a projekt beállításainál

Ezután ki kell választani, hogy melyik forrásfájlokat akarjuk instrumentáltatni (jobb gomb a *Calculator.java* forrásfájlon, majd *Use For Coverage Measurement*):



2. ábra: Forrásfájlon az instrumentálás engedélyezése

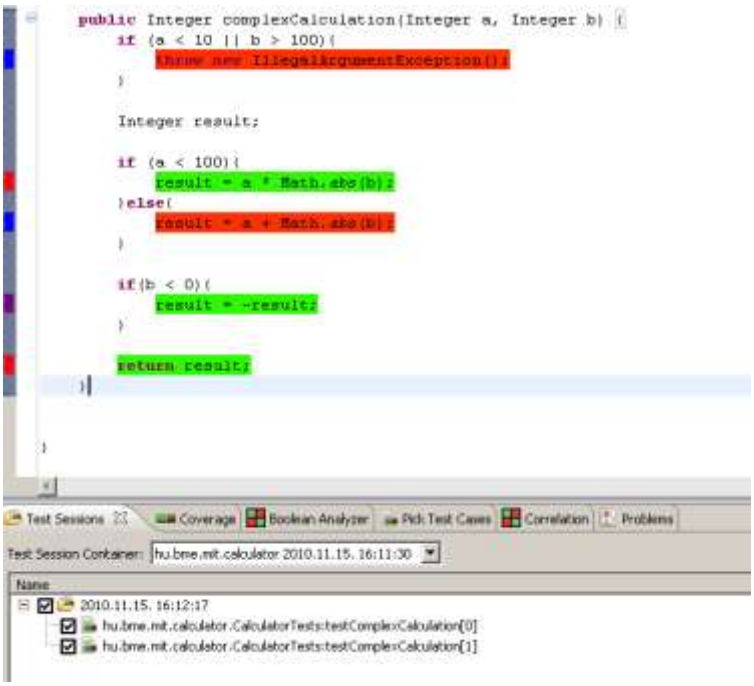
Majd a *CodeCover Measurement For JUnit* opcióval kell a teszteket indítani.



3. ábra: Futtatás fedési információk gyűjtésével

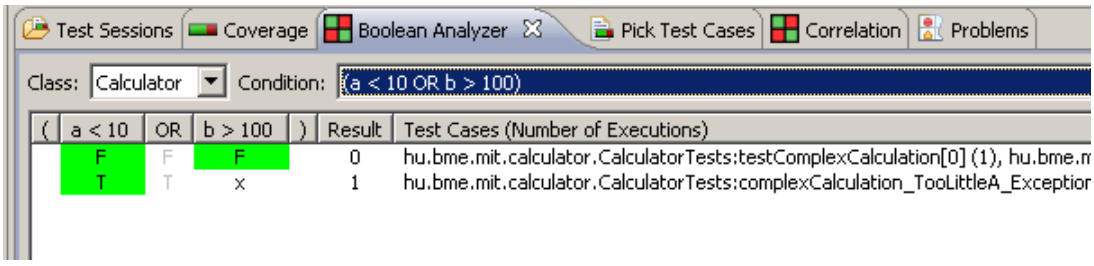
[3] A házi feladatban majd az EclEmma eszközt használjuk, mert az stabilabb. De a CodeCover tud feltétel lefedettséget is megjeleníteni, ezért a gyakorlaton most ezt nézzük meg.

A tesztek lefutása után alul a *Test Sessions* nézetben ki kell választani, hogy melyik futtatások eredményét akarjuk megnézni. Ha kiválasztjuk valamelyiket, akkor a forráskódot megfelelően színezni fogja az eszköz.



4. ábra: Teszt futtatás kiválasztása és a kód színezése

2. Bővítsük a tesztkészletünket, hogy a complexCalculation metódusnál 100%-os utasítást lefedettséget érjünk el.
3. Állítsuk át, hogy csak döntés lefedettséget vizsgáljon az eszköz (*BranchCoverage*), majd így is egészítsük ki a tesztkészletet.
4. Végül állítsuk át, hogy csak feltétel lefedettséget vizsgáljon (*ConditionCoverage*), és így is egészítsük ki a tesztkészletet. A lefedetlen kombinációk azonosításához segítséget nyújt a *Boolean Analyzer* nézet.



5. ábra: Feltétel lefedettség vizsgálata

(Figyelem! A CodeCover a következőt érti condition coverage alatt: *„Term coverage takes a closer look to the decision expression in if-statements. Each basic boolean term of the decision must at least once effect the overall result to true and to false. CodeCover implements the Ludewig term coverage, which subsumes MC/DC for boolean short circuit semantics."*)

Eclipse plug-in tesztelése

A házi feladatban használt alkalmazás Eclipse alapokon van megvalósítva, így röviden áttekintjük az Eclipse plug-inek tesztelésével kapcsolatos alapokat. A gyakorlaton egy nagyon egyszerű plug-int fogunk használni, amely egy nézetben megjeleníti, hogy a workspace-ben található fájloknál melyik fájlkiterjesztésből hány darab található.

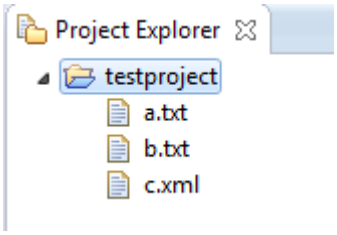
A példa plug-in kipróbálása

A plug-in egy *jar* formájában áll rendelkezésünkre. Indítsuk el a 4.5-ös Eclipse példányt, majd nyissuk meg a `c:\code\GYAK4\FileCounter` workspace-t. Ez egy üres `hu.mit.bme.filecounter.sut` nevű projektet tartalmaz, amiben van egy `dist` nevű könyvtár, és benne a tesztelendő plug-in jar fájl formában.

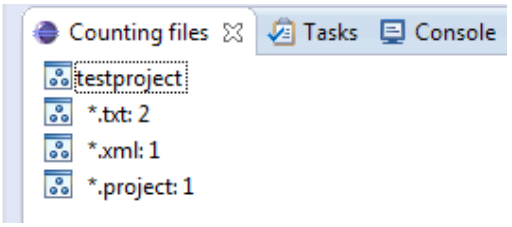
Az Eclipse azonban ettől még nem tud az új plug-inról, ehhez létre kell hozni egy új *target platform*ot, ami tartalmazza ezt a plug-int. Ehhez tegyük a következőket:

1. *Window / Preferences / Plug-in Development / Target Platform / Add...*
2. *Target definition:* válasszuk a *Current Target: Copy settings from the current target platform* opciót.
3. *Target content:*
    a. A target platform neve legyen *FileCounter Target.*
    b. *Locations / Add... / Directory,* a könyvtár elérési útja pedig legyen `${workspace_loc}/hu.mit.bme.filecounter.sut/dist` (így relatív lesz az útvonal, és nem kell az adott gép abszolút útvonalát beleégetni).
    c. A megadott könyvtárban meg is kell találnia egy új plug-int.
4. A létrehozás után válasszuk ki az új target platformot aktívnak.

A plug-in kipróbálásához indítani kell egy olyan új Eclipse példányt, amiben már benne van ez a plug-in is. Ehhez hozzunk létre egy új *Run configuration*t, aminek a típusa *Eclipse Application*. A Run configuration beállításainál láthatjuk, hogy ez majd egy új workspace-t fog megnyitni (*Main* fül / *Workspace Data*).



6. ábra: Teszt projekt fájlokkal



7. ábra: A plug-in nézete

Az új Run configuration elindítása után elindul egy új Eclipse. Hozzunk létre ebben egy új, üres projektet, majd hozzunk létre benne pár fájlt. Jelenítsük meg a *Counting files* nézetet (*Windows / Show view*), és itt látszik a plug-in működés közben.

A tesztelendő metódus

A plug-innek van egy nagyon egyszerű „üzleti logikája” (a fájlok összeszámolása) és egy GUI része (a nézet megjelenítése). Most az „üzleti logikát” fogjuk vizsgálni, annak is a következő metódusát:

```
public static Hashtable<String, Integer> getFileNumbersByExtension(IProject project)
```

A metódus bemenetként megkap egy projektet, és visszaad egy hash táblát, aminek egy eleme egy <fájlkiterjesztés, ilyen kiterjesztésű fájlok száma> páros.

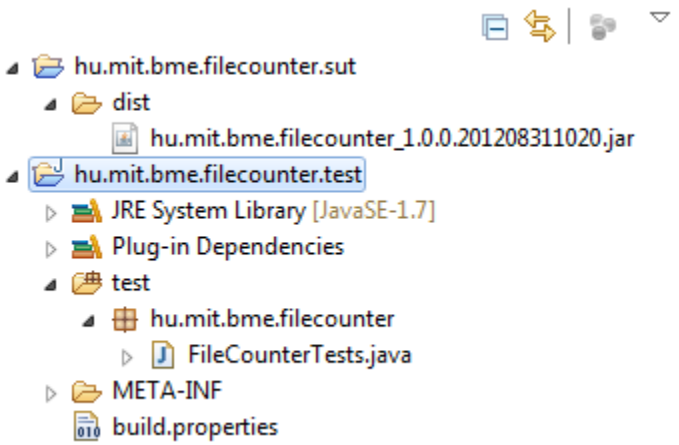
Egy tesztelési cél lehet például, hogy ellenőrizzük, hogy ha egy adott fájlkiterjesztéshez csak egy fájl van a projektben, akkor az ehhez a fájlkiterjesztéshez tartozó elem értéke 1 a hash táblában.

Tesztesetek megvalósítása JUnit tesztként

Ha ellenőrizni akarjuk a metódust automatikusan futó tesztekkel, akkor rögtön az első kérdés, hogy hova rakjuk ezeket a teszteket. Több lehetőségünk is van[4]:

- A tesztelendő projektbe rakjuk egy külön `test` könyvtárat. Ez most nem járható út, hisz nincs meg a forrása a projektnek. Ráadásul ebben az esetben a tesztelendő éles plug-inhez függőségként hozzá kéne adni a JUnit komponenseit is, ami nem szerencsés.
- Külön plug-inbe kerül a teszt kód, így nem kell az éles kódhoz teszt függőségeket hozzáadni. Ennek viszont az az ára, a tesztelendő plug-innak csak az exportált felületét éri el a teszt kód.
- Egy úgynevezett *plug-in fragment*be kerül a teszt kód, ami kapcsolódik a tesztelendő plug-inhez, és hozzáfér futásidőben a tesztelendő plug-in nem exportált részéhez is.

A gyakorlaton ezt a harmadik lehetőséget fogjuk használni, ehhez elérhető egy teszt projekt váz (8. ábra). Importáljuk be a workspace-be a `hu.mit.bme.hu.filecounter.test` projektet.



8. ábra: FileCounter teszt projekt vázlata

Mi legyen a teszt kód a kiválasztott működés ellenőrzéséhez? Ha manuális tesztet definiálnánk, amit a GUI-t használva hajtunk végre, akkor valami olyasmit készítenénk, hogy hozzuk létre egy projektet példa

[4] Bővebben lásd itt: RCP Quickstart blog, Testing Plug-ins with Fragments. URL: http://rcpquickstart.wordpress.com/2007/06/20/unit-testing-plug-ins-with-fragments/

adatokkal (6. ábra), indítsuk el a runtime Eclipse példányt, nyissuk meg a nézetet, ellenőrizzük el, hogy a *.xml sorhoz tartozó értéknél 1 jelenik-e meg.

Figyelem*:* Itt rögtön látszik, hogy ennél a tesztnél már nem csak a tesztelendő metódus bemeneti paraméterei lesznek a fontosak, hanem azok a teszt adatok, teszt környezet, amiben elindítjuk (ilyen a példa projekt, amin a tesztelendő plug-in dolgozik). Tesztelés esetén gyakran ennek a teszt környezetnek az automatikus előállítása jelenti az egyik nagy kihívást.

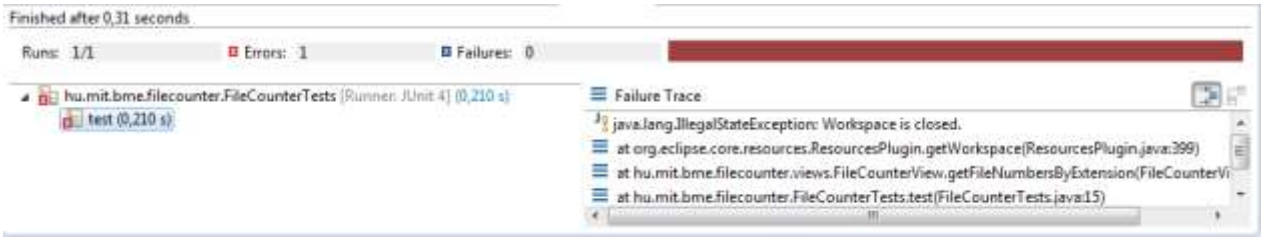
A példa projektünket már elkészítettük, így a teszt kódunk első próbálkozásban elég egyszerű:

```
@Test
public void test() {
	Hashtable<String, Integer> ht =
	 FileContentAnalyzer.getFileNumbersByExtension("testproject");

	assertEquals(Integer.valueOf(1), ht.get("xml"));
}
```

(A teszt adatok szövegként való elhelyezése a kódban nem túl karbantartható megoldás, gondoljuk el, hogy mi történik akkor, ha átnevezzük valamiért a `testproject` projektet más névre. Ezen később még érdemes finomítani.)

Futassuk le JUnit tesztként a korábban megtanult módon az új tesztünket! Ekkor `IllegalStateException` kivételt kapunk „Workspace is closed” üzenettel. Mi lehet a gond?



9. ábra: Teszt futtatása nem plug-in tesztként

Sima Java programként indítottuk el a tesztünket, így a runtime Eclipse példány nem indult el a háttérben, a plug-in kódját az Eclipse környezet nélkül próbáltuk meg végrehajtani.

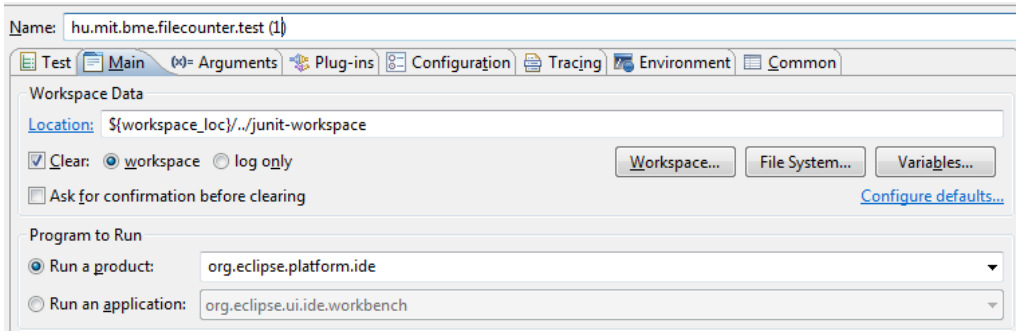
Figyelem: Talán már korábban is sejtettük, de ez most már biztos megerősített minket benne, hogy ez nem egy egységteszt, hiába a JUnit keretrendszert használjuk. Ez bizony egy integrációs teszt, hisz a teszt futtatásakor az Eclipse futtatókörnyezet rengeteg modulját is meghívjuk. Ha egységteszteket szeretnénk készíteni, akkor foglalkozni kell az izolációval.

Az `IllegalStateException` kivételt úgy tudjuk elkerülni, ha **JUnit Plug-in Test**ként indítjuk el a teszteket. Ilyenkor már látszik, hogy elindul egy Eclipse példány, dolgozik is, azonban megint hibát kapunk (de legalább most már egy másikat). Mit rontunk el?

```
<terminated> hu.mit.bme.filecounter.test (1) [JUnit Plug-in Test] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (2012.08.31. 12:08:08)
org.eclipse.core.internal.resources.ResourceException: Resource '/testproject' does not exist.
        at org.eclipse.core.internal.resources.Resource.checkExists(Resource.java:341)
        at org.eclipse.core.internal.resources.Resource.checkAccessible(Resource.java:215)
        at org.eclipse.core.internal.resources.Project.checkAccessible(Project.java:147)
        at org.eclipse.core.internal.resources.Container.members(Container.java:266)
        at org.eclipse.core.internal.resources.Container.members(Container.java:249)
        at hu.mit.bme.filecounter.views.FileCounterView.getFileNumbersByExtension(FileCounterView.java:59)
        at hu.mit.bme.filecounter.FileCounterTests.test(FileCounterTests.java:15)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

10. ábra: Nem található a testproject - melyik workspace-t használjuk is?

Nézzük meg a JUnit Plug-in Test elindításával létrejött run configuration beállításait!



11. ábra: Plug-in Test run configuration beállításai

A probléma tehát az volt, hogy ez a *junit-workspace* workspace-ben futott, és nem a korábban létrehozott *runtime-EclipseApplication* workspace-ben. Látszik tehát, hogy a teszt környezet kezelése tényleg nem triviális feladat.

Éljünk most a legegyszerűbb megoldással. Szedjük ki a pipát a *Clear* opciótól, hogy ne törölje a workspace tartalmát minden teszt futtatás előtt. Váltsunk át a *junit-workspace* workspace-re, hozzuk ott is létre a példa adatokat. Futassuk így a tesztet, így már elvileg sikeresen végrehajtódik.

Otthoni feladatok

1. Az a gond azzal, hogy ha a workspace tartalmát meghagyjuk a teszt futások között, hogy nem biztos, hogy ismert állapotból indulnak a tesztek. Most elvileg csak olvassa a teszt kód és a plug-in is a workspace tartalmát, de más esetben gond lehet később ebből. Milyen módszert alkalmazhatunk ennek megoldására?
2. Nézzük át a Run configuration további beállításait is, ezek is hasznosak. Például a tesztekhez használt runtime Eclipse elég lassan indul el, mert minden telepített plug-int betölt. Elég lenne csak a SUT-hoz szükségeseket használni. Persze ehhez az kell, hogy a fejlesztők definiálják a függőségeket. (Később persze érdemes olyan tesztet is futtatni, ahol minden plug-int betöltünk, hogy az esetleges konfliktusok kiderüljenek.)
3. A tesztelésünk elég hiányos még. Valamelyik tanult teszttervezési technikát használva bővítsük a teszteseteket, és vizsgáljuk meg a FileCounter plug-int!