Szolgáltatásbiztonságra tervezés laboratórium (VIMIM236)

Robusztusság vizsgálata hibainjektálással

Mérési segédlet

Készítette: Oláh János

dr. Majzik István és dr. Pintér Gergely korábbi útmutatóinak felhasználásával

Utolsó módosítás: 2010. október 16.

Verzió: 1.0

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

1. Bevezető

A labor mérései során egy bonyolultabb többrétegű alkalmazást kiszolgáló összetett infrastruktúra szolgáltatásbiztonságát vizsgáljuk különböző technikák segítségével. A rendszer lehetséges felépítési alternatíváit modellezéssel és mérésekkel elemezzük, az így azonosított hibamódok és szűk keresztmetszetek kiküszöbölésére pedig különböző hibatűrést és skálázhatóságot segítő technológiákat próbálunk ki a gyakorlatban. A mérések felépítése a következő:

1. Architektúra modellezés és konfiguráció generálás
2. Terheléselosztó fürtök és teljesítménytesztelés
3. Feladatátvételi fürtök
4. Megbízhatóság modellezés
5. Hibadiagnosztika
6. **Robosztusság vizsgálata hibainjektálással**
7. Dependability benchmarking

Jelen mérés során megismerünk egy módszert arra, hogy hogyan tudjuk a rendszerünket felépítő komponensek robosztusságát vizsgálni és kiértékelni, hibainjektáló kísérletek segítségével.

2. Hibainjektálás

Modern biztonságkritikus illetve nagy rendelkezésre állású informatikai rendszerek tervezése, fejlesztése során a rendszert irányító, kiszolgáló szoftverekkel szemben is igen magasak az elvárások. A különböző rendszerhibák és leállások döntő hányadát a szoftverből eredő hibák teszik ki, ezért is különösen fontos ilyen alkalmazások fejlesztése esetén az alkalmazás hibakezelési mechanizmusainak validálása, mivel az egyes rendszerkomponensek nem kielégítően működő hibakezelése az egész rendszer megbízhatóságát befolyásolja. A vizsgálat egyik, legelterjedtebb és leghatásosabb módszere a hibainjektálás, vagyis annak vizsgálata, hogyan viselkedik a rendszer mesterségesen generált hibák jelenlétében.

A hibainjektálás kivitelezésére többféle megközelítést és technikát dolgoztak ki. A különböző technikák összehasonlításához azonban fontos bevezetni néhány hibainjektálási alapfogalmat. A *cél rendszer* az a rendszer, vagy programkomponens, melyet hibainjektálásnak vetünk alá. A kísérlet során az alkalmazással valamilyen előre meghatározott *terhelést* hajtatunk végre, mely során az injektált hibák összességét *hibaterhelésnek (faultload)* nevezzük. Egyetlen hiba injektálását, és az injektálás hatásának megfigyelését *kísérletnek* nevezzük. Ezt meg kell különböztetni a *kísérletsorozattól (campaign)*, mely több kísérlet végrehajtását foglalja magába.[13]

A hibainjektálási technikák legalapvetőbb megkülönböztetése aszerint történik, hogy a valós rendszerbe, vagy annak valamilyen módon szimulált változatába (modelljébe) történik a hibák injektálása. A szimulációs esetet tovább lehet bontani két ágra, vagyis a hibainjektálás történhet a célrendszer szoftveresen szimulált, vagy hardveres emulált modelljébe. Mindhárom mód számos hibainjektálási lehetőséget takar. A különböző technikákat a hibainjektálás fő követelményei, illetve ezek teljesülése alapján lehet a legkönnyebben összehasonlítani, melyek a következőek:

- **Irányíthatóság** – A hibainjektálás irányításának képessége.

- **Megfigyelhetőség** – A hibák hatásának megfigyelhetősége és az eredmények rögzíthetőségének jellemzője.
- **Megismételhetőség** – Egyetlen kísérlet pontos megismételhetőségének képessége.
- **Reprodukálhatóság** – Egy teljes kísérletsorozat során kapott eredmények megismételhetősége.
- **Elérhetőség** – Annak jellemzője, hogy a célrendszerben található, potenciálisan hibás részeket milyen eséllyel érhetjük el az injektált hibákkal.
- **Terhelés reprezentativitás** – Annak a jellemzője, hogy a terhelés milyen valószerűen szimulál egy, az alkalmazás számára valós terhelést.
- **Hiba reprezentativitás** – Annak jellemzője, hogy az injektálható hibák mennyire tekinthetők tipikusnak az adott esetben.

Szoftverek vizsgálata esetén a hibákat nyilvánvalóan az elkészült valós rendszerbe injektálhatjuk legkönnyebben, nem pedig annak bármilyen szintű modelljébe. Szoftver hibák injektálására két alapvető módszert dolgoztak ki: *meghibásodás-injektálás (fault injection)*, illetve *hibainjektálás (error injection)*.

Meghibásodás-injektálás. A meghibásodás (fault) nem más, mint a hiba *feltételezett oka*, így az első módszer esetén az injektált szoftver hibák a fejlesztő által elkövetett hibákat imitálják a program forráskódjának megváltoztatásával. Az ilyen jellegű manipulációt mutációnak is nevezik. Hátránya nyilvánvalóan az, hogy kivitelezéséhez szükséges hozzáférni a célrendszer forráskódjához, tehát általában nem használható úgynevezett *black box* jellegű vizsgálatok elvégzéséhez.

Hibainjektálás. A hiba (error) definíció szerint a hibajelenséghez vezető rendszerállapot. A második módszer esetében tehát a program működését befolyásoló hibákat injektálunk a rendszerbe. Az ilyen jellegű hibainjektálásnak két fő lehetősége a program állapotának manipulációja, illetve a hívási paraméterek korrupciója.

A hibainjektálás (error-injection) első lehetősége a változók értékei, pointerek, program által tárolt adatok manipulációját jelenti. A második lehetőség *API paraméter korrupció* néven is ismeretes, és a *robosztusság-tesztelés* kategóriájába tartozik.

A robosztusság annak az extrém bemenetnek és szélsőséges működési feltételeknek a mértéke, amelyek mellett a rendszer még megfelelően működik (IEEE 610). A robosztusságot szokás a *CRASH* hibahatás modell segítségével meghatározni. Az akronim a hibák hatására létrejövő lehetséges események kezdőbetűiből áll össze, úgymint:

- **Catastrophic** – A teljes rendszer összeomlik.
- **Restart** – A működés helyreállításához újraindítás szükséges.
- **Abort** – Az alkalmazás abnormálisan fejezi be a működést.
- **Silent** – Hibás működés következik be jelzés nélkül.
- **Hindering** – Valamilyen helytelen hibakóddal jelez az alkalmazás.

A robosztusságot az egyes tesztesetek által előidézett események egymáshoz viszonyított előfordulási arányával lehetséges kifejezni.

Hibainjektálási kísérletek során a hiba bevitelének négy fő szempontja van.

Hiba helye Az injektált hiba helye a szoftverben (pl melyik metódusban, konstruktorban, illetve ezeken belül pontosan hol, stb.)

Hiba típusa Milyen jellegű az injektált hiba. Természetesen a hibainjektálással a célunk reprezentatív hibák injektálása, ami azt jelenti, hogy az adott hiba valóban előfordulhat a rendszer üzemszerű működése során is.

Hiba aktiválódási ideje Mikor aktiválódjon az injektált hiba. Ez lehet például abszolút (időbélyeg), vagy adott lefutásra aktiválódó.

Terhelés biztosítása Milyen forgatókönyvet hajt végre a rendszer a hibainjektálás során. Lényeges, hogy a terhelés reprezentatív legyen, illetve – amennyiben ez függ a terheléstől – aktiválva az injektált hibákat (pl. adott kódrészlet fusson le). Amennyiben a terhelés rögzíthető, akkor lehetségessé válik különböző hibaterhelések mellett ugyan azt a forgatókönyvet lefuttatni, így igen jól összehasonlíthatóak lesznek a különböző hibainjektálások hatásai.

További szempontként szokás még említeni a kísérlet elvégzésének *költségét* (idő és eszközök). Részletesebb vizsgálatokhoz, illetve extrém kis hibagyakoriság esetén hosszú megfigyelési illetve szimulációs idő várható, aminek csökkentésére különféle technikákat dolgoztak ki. Azonban ezzel mi jelenleg nem foglalkozunk.

A kísérletek során gyűjtött adatokat *statisztikailag értékelni kell*. Általában várható érték (pl. hiba lappangási ideje) illetve arány (pl. az egyes hibadetektáló eljárások hibafedése) becslésére van szükség. Ezen számítások alapjait tekintjük át.

2.1. Konfidenciaintervallumok meghatározása

Adott az X változóra vonatkozó n számú $x_1, x_2, \ldots, x_n$ mérési eredmény (mintaértékek), amelyek egyenként mint az $X_1, X_2, \ldots, X_n$ valószínűségi változók értékei tekinthetők, ahol az X_i-k függetlenek és azonos eloszlásúak X-szel. X valamely Θ paraméterét szeretnénk becsülni (legjobb közelítését meghatározni), $X_1, X_2, \ldots, X_n$ valamely függvényével. A Θ becsléséhez használt $\hat{\Theta} = \hat{\Theta}(X_1, X_2, \ldots, X_n)$ függvényt *becslőnek*, a $\hat{\Theta} = \hat{\Theta}(x_1, x_2, \ldots, x_n)$ értéket pedig Θ egy *pontbecslésének* nevezzük. A $\hat{\Theta}$ becslő torzítatlan, ha $E\{\hat{\Theta}\} = \Theta$; minimális varianciájú, ha $Var\{\hat{\Theta}\} = E\{(\hat{\Theta} - \Theta)^2\}$ minimális. Közismertek a várható érték és a szórás torzítatlan, minimális varianciájú becslői.

A pontszerű becslés kevés információt hordoz a felhasználó számára, ezért rendszerint **konfidenciaintervallumok illetve konfidenciaszintek megadása is szükséges** (intervallumbecslés). A konfidenciaintervallummal kapcsolatos $P\{\Theta \in (k_1, k_2)\} = 1-\varepsilon$ valószínűségi állítás, ahol (k_1, k_2) a konfidenciaintervallum, $1-\varepsilon$ pedig a konfidenciaszint, azt fejezi ki, hogy a mérési eredmények alapján származtatott konfidenciaintervallum az adott valószínűséggel tartalmazza (lefedi) a becsült paramétert. Általában célszerű a becslőre szimmetrikus konfidenciaintervallumot választani, $P\{\hat{\Theta} - e < \Theta < \hat{\Theta} + e\} = 1 - \varepsilon$ alakban, így a mérési eredmények *eredmény* $\pm$ *hiba* szokásos megadásához hasonló alakot kaphatunk.

A konfidenciaintervallum mérete függ a konfidenciaszinttől illetve a becslő tulajdonságaitól. Kisebb konfidenciaszint mellett a konfidenciaintervallum is kisebb; rögzített konfidenciaszint mellett a konfidenciaintervallum nagysága a megfigyelésszám növelésével csökkenthető.

2.1.1. A várható érték konfidenciaintervalluma

A várható érték a mintaértékek számtani közepével becsülhető, ami $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. Elég sok ($n > 50$) mintavett érték esetén a konfidenciaintervallum is egyszerűen meghatározható, mivel

ekkor $Z = \frac{\bar{X}-\mu}{S/\sqrt{n}}$ közelítőleg standard normális eloszlásnak tekinthető; itt μ a tényleges várható érték, S^2 pedig a mintaértékek korrigált tapasztalati szórásnégyzete: $S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2$. Így Z-re felírható: $P\left\{u_{1-\varepsilon/2} < Z < u_{\varepsilon/2}\right\} = 1-\varepsilon$, azaz

$$P\left\{u_{1-\varepsilon/2} < \frac{\bar{X}-\mu}{S/\sqrt{n}} < u_{\varepsilon/2}\right\} = 1-\varepsilon$$

.
Mivel $u_{1-\varepsilon/2} = -u_{\varepsilon/2}$, a várható érték egy $(1-\varepsilon)$ konfidenciaszintű konfidenciaintervalluma

$$\left(\bar{X} - u_{1-\varepsilon/2}\frac{S}{\sqrt{n}},\ \bar{X} + u_{1-\varepsilon/2}\frac{S}{\sqrt{n}}\right)$$

ahol u_α a standard normális eloszlás táblázatából határozható meg: $P\{Z \le u_\alpha\} = \alpha$. A leggyakrabban használt konfidenciaszintekre ez az érték a 1 táblázat szerinti.

Konfidenciaszint: $1-\varepsilon$	$1-\varepsilon/2$	$u_{1-\varepsilon/2}$
0.9	0.95	1.645
0.95	0.975	1.96
0.98	0.99	2.33

1. táblázat. Gyakori konfidenciaszintek

(Kisszámú mért érték esetén a fenti közelítés már nem alkalmazható. Ekkor, ha X normális eloszlású, hasonló gondolatmenettel az $n-1$-edfokú $student-t$ eloszlás használata szükséges. Ha X exponenciális eloszlású, akkor a χ^2 eloszlással való közelítés használható fel.)

Példa:
Adott egy 52 elemű mintasorozat, amelyre $\bar{X} = 3.9$, $S = 0.95$. *A várható érték konfidenciaintervalluma 90%-os szinthez* $3.9 \pm 1.645\frac{0.95}{\sqrt{52}} = (3.68, 4.12)$, *tehát a várható érték 90%-os konfidenciaszinttel ebbe az intervallumba esik.*

2.1.2. Arányok konfidenciaintervalluma

Egy q arány konfidenciaintervallumának meghatározása hasonló a várható érték konfidenciaintervallumának meghatározásához. A mintaértékekből számított aránybecslő itt $\hat{q} = \frac{Y}{n}$, ahol Y a kitüntetett esetek száma. Ha a mérési eredmények száma elég nagy, azaz pontosabban $nq \ge 5$ és $n(1-q) \ge 5$, akkor $\hat{q} = \frac{Y}{n}$ közelítőleg normális eloszlású, $\mu = q$ és $\sigma^2 = \frac{q(1-q)}{n}$ paraméterekkel.
Így q-nak az $(1-\varepsilon)$ konfidenciaszinthez tartozó konfidenciaintervallumára a

$$\left(\hat{q} - u_{1-\varepsilon/2}\sqrt{\frac{\hat{q}(1-\hat{q})}{n}},\ \hat{q} + u_{1-\varepsilon/2}\sqrt{\frac{\hat{q}(1-\hat{q})}{n}}\right)$$

közelítés adható.

Példa:
Egy hibadetektáló eljárás $n = 1000$ *injektált hibából* $Y = 10$*-et detektál. A mintaértékekből számított aránybecslő tehát* $\hat{q} = 0.01$, *az* $nq \ge 5$ *és* $n(1-q) \ge 5$ *feltétel várhatóan teljesül. A 90%-os szinthez tartozó konfidenciaintervallum így* $0.01 \pm 1.645\sqrt{\frac{0.01-0.0001}{1000}} = (0.005, 0.015)$, *tehát 0.5% és 1.5% közé esik a hibadetektáló eljárás hibafedése. Ugyanez az intervallum 95% konfidenciaszinthez* (0.004, 0.016).

2.2. A mintavett értékek számának meghatározása

A konfidenciaintervallum nagysága függ a mintavett értékek számától. Minél több mintaértékünk van, annál nagyobb lehet a konfidenciaszint, illetve csökkenthető a konfidenciaintervallum. Költségkímélés céljából az adott konfidenciaintervallumhoz illetve -szinthez tartozó legkevesebb szükséges mérendő érték meghatározása a cél.

2.2.1. Várható érték számítása

A várható értéket $\pm 100\beta$ százalékos konfidenciaintervallummal, $1-\varepsilon$ konfidenciaszinttel szeretnénk meghatározni. Egy n mért értékből álló sorozatra az előzőek alapján a konfidenciaintervallum $\bar{X} \pm u_{1-\varepsilon/2}\frac{S}{\sqrt{n}}$.

A kívánt (százalékos alakban megadott) intervallum szerint ez $\bar{X} \pm \bar{X}\beta$ kell legyen, azaz $u_{1-\varepsilon/2}\frac{S}{\sqrt{n}} = \bar{X}\beta$. Ebből

$$n = \left(\frac{u_{1-\varepsilon/2}S}{\bar{X}\beta}\right)^2$$

Példa:
Előzetes mérések alapján $\bar{X} = 20$, $S = 5$; $\pm 5\%$ *konfidenciaintervallumot szeretnénk, 95% konfidenciaszinttel. A szükséges mért értékek száma* $n = \left(\frac{1.96 \cdot 5}{0.05 \cdot 20}\right)^2$, *tehát kb.* 100 *mért értékre van szükségünk.*

2.2.2. Arányok mérése

Az előző elgondolás aránymérésre is alkalmazható. Az arányt $\pm\beta$ konfidenciaintervallummal, $1-\varepsilon$ konfidenciaszinttel szeretnénk meghatározni. Az arány konfidenciaintervalluma és a kívánt pontosság alapján $\bar{q} \pm \beta = \bar{q} \pm u_{1-\varepsilon/2}\sqrt{\frac{\bar{q}(1-\bar{q})}{n}}$, innen

$$n = \frac{u_{1-\varepsilon/2}^2 \bar{q}(1-\bar{q})}{\beta^2}$$

Példa:
Előzetes mérések szerint 100 *hibából* 54*-et detektál egy hibadetektáló eljárás. Hány hibainjektálási kísérletet kell végeznünk, ha* $\pm 5\%$ *konfidenciaintervallummal, 95% konfidenciaszinttel szeretnénk a detektált hibák arányát megállapítani?*
Itt $\bar{q} = 0.54$, $\beta = 0.05$, $u_{1-\varepsilon/2} = 1.96$, *így* $n = \frac{1.96^2 \cdot 0.54 \cdot 0.46}{0.05^2} = 382$

3. Reflektív programozás

A számítógépes programok tervezésének és megvalósításának módszereit programozási paradigmáknak nevezzük. Napjaink legelterjedtebb paradigmája az objektum orientált programozás. Ebben a programot tulajdonképpen egymással együttműködő, de saját felelősséggel és szerepkörrel rendelkező objektumok hálózataként írjuk le. Ismert paradigmák továbbá a deklaratív programozás, az aspektus-orientált programozás és a reflektív programozás is. [1]

[1] Nézzen utána milyen egyéb paradigmák léteznek még!

A reflekció a programozásban egy program szerkezetének futásidőbeli felfedezését és/vagy módosítását jelenti. Egy reflektív paradigma szerint elkészített komponens képes akár a saját kódját is futásidőben módosítani. Ilyen módon képessé válhat arra, hogy dinamikusan reagáljon a környezet változásaira, például paraméterek hangolásán keresztül. Ennek egy másik variációja, mikor a fordításkor a program leendő környezetének bizonyos paraméterei nem ismertek, így bár dinamizmusra nincs szükség, ezekhez futásidőben kell hangolódni.

Végül nagy hasznát vehetjük a reflektív programok képességeinek hibák injektálása esetén is. Beágyazottól rendszerektől egészen a nagyvállalati rendszerekig az alkalmazandó komponenseket úgy akarjuk összeválogatni, hogy egy előre meghatározott megbízhatóságot érjünk el. Azonban ezek jellemzően COTS (Commercial Off The Shelf) komponensek, így nem áll rendelkezésre a forráskódjuk, csak a binárisok. Ilyenkor tehát nincs lehetőség a forráskód módosításával kísérletezni, vagyis kénytelenek vagyunk a már lefordított binárisokba hibát injektálni.

3.1. Javassist

A Javassist [5] Java nyelvű szoftverekhez egy elterjedt bájtkód manipulátor. Nagy előnye, hogy rendelkezik Java forráskódszintű API-val is. Ennek hiányában ugyanis a programozónak ismernie kellene a Java bájtkód szintaxist, ami igen nehezen olvasható és értelmezhető.

Működése meglehetősen egyszerű. Első lépésként a megadott útvonalon be kell olvasni a Java class fájlokat, CtClass objetumokba. A Javassist számára ezek az objektumok jelképezik az absztrakciós szintet - ezeken keresztül lehet felfedezni és manipulálni a class fájlokat. Ha ez megtörtént, akkor vissza kell menteni a fájlokat (akár felülírva a régi fájlokat, akár egy külön helyre).

A Javassist a forráskódszintű API ellenére rendelkezik egy speciális karakterrel (*$*). Ennek felhasználása a teljesség igénye nélkül:

- **$_** Az adott metódus visszatérési értéke.
- **$n** Az adott metódus n. paramétere (tehát $1, $2, ...).
- **$$** Az összes bemenő paraméter tömbje.
- **$r** A visszatérési érték típusa.

Már szóbakerült a CtClass osztály, mely egy class fájl reprezentációja. Ezen felül még érdemes ismerni a fontosabb Javassist osztályokat: CtField, CtConstructor, CtMethod - ezek jelentése nevük alapán egyértelmű, részletes dokumentációjuk megtalálható a Javassist oldalán. (Ez természetesen használható a mérés alatt, ennek ellenére mind a tutorial [7], mint az API [6] előzetes megismerése ajánlott mérés előtt.)[2]

Végezetül egy rövid példakód:

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
CtMethod m = cc.getDeclaredMethod("move");
m.insertBefore("{ System.out.println(\$1); System.out.println(\$2); }");
cc.writeFile();
```

E fenti példa tehát egy *Point* nevű class fájlból készít egy CtClass objektumot, melyben a *move* metódus elejére beszúrja az első két paraméter kiírását.

[2] A hibainjektálásról és a reflektív programozásról szerzett ismereteit felhasználva gondolja végig, milyen jellegű hibákat tudna injektálni a Javassist könyvtár segítségével!

Meg kell azonban jegyezni, hogy a Javassist használatának vannak bizonyos korlátai is. Például egy visszatérési érték felülírása problémás lehet, ha más típusú objektumok szeretnénk visszaadni, mint ami a függvény eredeti deklarációjában szerepel. Ilyenkor a Javassist hibát fog dobni, és sok esetben a konverzió sem megoldás.

4. A hibainjektáló keretrendszer bemutatása

A keretrendszer használatának célja, hogy a tesztelendő alkalmazások hibakezelési mechanizmusait lehessen vizsgálni a segítségével, mégpedig alacsony szintű konfigurálás nélkül. A keretrendszer a korábban említettek szerint *hibainjektálásra* képes, azon belül is a második módszert támogatja, vagyis a robosztusság-tesztelést. (Az első hibainjektálási módszer használatára, vagyis a tesztelendő alkalmazás állapotának megváltoztatására közvetlenül nincsen mód.)

Az alacsony szintű konfigurálás elkerülése érdekében a keretrendszer felhasználója egy modellen jelölheti ki a hibák helyeit, melyek fajtáit a keretrendszer hibakönyvtárából válogathatja össze. A modell a a Javassist reflektív könyvtár felhasználásával jön létre a célalkalmazás implementációjának bejárásával. A modell alapjául egy EMF metamodell szolgál. A modell elemei az osztályok, ezek változói, konstruktorai és metódusai, valamint az úgynevezett **kapcsolatok**. Egy kapcsolat forrása egy konstruktor vagy egy metódus, a cél pedig egy változó (adatelérés), egy konstruktor (objektum létrehozása) vagy egy másik metódus (függvényhívás). A kapcsolatok a modellben tehát felfoghatóak egyfajta *hívási gráfként* is. A kapcsolatok a modellben *x2y* néven jelennek meg, melynek jelentése, hogy x metódusból meghívjuk y metódust (vagy elérjük y változót, attól függően, hogy micsoda y).

A létrejött modellhez lehetséges hozzáadni hibákat és monitorozási pontokat, és az így létrejött konfigurációt elmenteni. Egy ilyen konfiguráció tehát egy EMF példánymodell, melyet injektáláskor az eszköz bejár, és szintén a Javassist segítségével injektálja a hibákat és a monitorozási pontokat. Hibákat csak kapcsolatokra, monitorozási pontokat csak konstruktorba vagy metódusba lehet injektálni.

Maga a keretrendszer egy Eclipse plug-in, mely rendelkezik saját varázslóval. Ezek segítségével lehet létrehozni egy új hibainjektáló projektet, egy másik, a workspace-ben található Eclipse projektből, vagy akár külső class fájlokból is. Ha átváltunk az eszköz saját perspektívájára (*Yellow Fault Injection Perspective*), akkor láthatóvá válik a *Yellow* menü. Itt két fontos akció található.

Create model Bejárja az implementációt, és létrehozza a modellt (*ymodel*). Ezt jelenleg egy tree editorban lehet megnyitni és böngészni. Az editorban lehet hozzáadni a modellhez a hibákat és monitorozási pontokat, és a properties nézetben konfigurálni.

Instrument code A létrehozott konfigurációknak megfelelően instrumentálja a kódot, vagyis injektálja a hibákat és monitorozási pontokat.

A monitorozási pontok hatására a háttérben egy olyan függvény kerül bele az instrumentált binárisba, mely a 85-ös portra küldi ki a monitorozott információt. (A keretrendszer induláskor nyit egy socketet, és a ide kapott üzeneteket egyszerűen kiírja a konzolra. Természetesen a begyűjtött információkkal bármi egyéb művelet is elképzelhető, pl. log fájlba mentés.)

Az injektálható hibák és monitorozási pontok egy-egy könyvtárban találhatóak. Mindkét könyvárnak van néhány alapvető eleme, melyeket fel lehet használni.

4.1. A hibakönyvtár

A hívott metódus lefagy (CallHang) Ebben az esetben a hívott metódusba egy olyan hiba kerül beinjektálásra, aminek a hatására a metódus lefagy, és sohasem tér vissza. Ezzel

vizsgálható hogy van-e a hívó oldalon bármilyen timeout mechanizmus. Csak hívásra injektálható, változó elérésére nem!

Hívás elmarad (CallCancelled) Ilyenkor a forrás metódusból indított hívás nem következik be. Általában e hiba hatására valamilyen, a metódus vagy más metódusok által későbbiekben használandó változó értéke nem állítódik be. Tesztelhető vele a hibakezelés általában, vagyis hogy a különböző metódusok ellenőrzik-e az értékeket mielőtt felhasználnák azokat.

NULL érték tér vissza (ReturnNull) Ilyen hiba esetén megtörténik ugyan a hívás, de a hívott függvény *NULL-al* tér vissza. A nem várt visszatérési értékek kezelést lehet tesztelni ezzel a módszerrel.

Beállított érték tér vissza (ReturnValue) Tulajdonképpen az előző esethez hasonlít, de itt egy tesztelő által adott érték tér vissza, nem pedig NULL érték. Segítségével vizsgálható nem csak a viselkedés egy adott, valamilyen szemantikai megfontolásból érdekes értékre, hanem például *MAXINT+1* jellegű értékekre való válasz is. A visszaadandó értéket a felhasználó a hiba konfigurálásánál az *Argument* mezőben állíthatja be.

Beállított paraméterekkel indul a hívás (SetParameter) Ez a fajta hiba meglehetősen nagy jelentőséggel bír, hiszen segítségével a felhasználó a program aktuális működésétől függetlenül rögzítheti egy-egy metódus hívási paramétereit. Így vizsgálható a hívott metódus hibakezelése is, ahogyan a hívási paramétereit ellenőrzi felhasználásuk előtt, ugyanakkor itt is előkerülhetnek a különböző szemantikai megfontolások. (Ilyen lehet például a program vezérlésének egy nem megfelelő ágba kényszerítése a hívási paraméterek beállításával, stb.) Értelemszerűen ez sem konfigurálható adatelérésre.

Kivétel dobása (AddException) E hiba injektálásának hatására a kívánt helyen a lefutáskor egy konfigurálásnál megadott típusú kivétel dobása (*throw* mechanizmus segítségével) következik be. A kivétel típusát szintén az Argument mezőben lehet beállítani (ha szükséges, akkor nyilván teljes elérési úttal, pl. *java.lang.Exception*). Jelenleg ez a hibatípus nem konfigurálható adatelérésre.

Késleltetés (AddTimeout) Ilyen jellegű hiba injektálásának hatására az adott kódrészlet a lefutás során a tesztelő által megadott ideig blokkolódik, így téve vizsgálhatóvá például egy missziókritikus, valós idejű rendszerben implementált *timeout* mechanizmust. Jelenleg ez a hibatípus sem konfigurálható adatelérésre.

4.2. A monitorozási könyvtár

Hívási paraméterek monitorozása (MonitorCallParameters) Amikor a felhasználó ilyen monitorozást állít be egy metódusra vagy konstruktorra, akkor ezzel lehet rögzíteni azt, hogy az adott metódust milyen értékekkel hívták meg. A socketre az összes paramétert egy sztringbe összefűzve küldi ki.

Visszatérési érték monitorozás (MonitorReturnValue) A visszatérési érték monitorozása a kérdéses metódus visszatérési értéket figyeli meg, és küldi ki a socketre.

Hívási paraméterek és visszatérési érték monitorozás (Monitor...) Értelemszerűen az előző kettő kombinációja. Ez egy sztringbe fűz minden adatot és küldi a socketre.

Try-catch monitorozás (InsertTryCatch) Ez a fajta monitorozó mechanizmus egy metódus vagy konstruktor köré injektál egy try-catch blokkot. Ebben elkap bármilye, a törzsben nem kezelt kivételt, és kiírja annak típusát a socketre. Természetesen az elkapott kivételt tovább is dobja, hiszen ennek hiányában meghamisítaná a tesztalkalmazás viselkedését.

Változó értékének monitorozása (MonitorVariable) Ebben az esetben egy szál futása jelenti a monitorozást, mely a célrendszer futása alatt időnként felébred, és a socketre küldi a változó aktuális értékét. (Sajnos ez a módszer jelenleg nem működik.)

Hívás monitorozása (MonitorCall) A hívási paraméterek monitorozásához hasonlóan egy metódus hívásáról szolgáltat információt, azzal a különbséggel, hogy ez csak a meghívás tényét monitorozza, semmilyen plusz információt nem.

4.3. A hibainjektálási szempontok a keretrendszerben

Korábban már szóba kerültek a hibainjektálás szempontjai (2). Ezeket a kérdéseket a keretrendszer segítségével a következő módon lehet megválaszolni:

A hiba típusa Ezt a hiba létrehozásakor lehet kiválasztani. A lehetséges típusok az előbb bemutatott hibakönyvtár elemei.

Hibainjektálás helye A helyet az adott kapcsolat határozza meg, melyre a hibát konfigurálják.

Hiba aktiválódásának ideje Ezt a paramétert a keretrendszerben egy hiba konfigurálásánál, a *frequency* paraméter beállításával lehet megadni. Ez azt adja meg, hogy milyen gyakran forduljon elő az adott hiba. (Pl. az adott kódrészlet minden második lefutásánál.)

Terhelés A terhelés biztosítása grafikus programok tesztelése esetén egy olyan szoftver használatát jelenti, mely képes a felületen előre rögzített felhasználói akciókat végrehajtani. Ezekről bővebben a következő alfejezetben (4.4) lesz szó.

Kötségek Ezek meghatározására a keretrendszerben közvetlenül nincs támogatás.

4.4. A terhelés biztosítása

Hibainjektálási kísérletek végrehajtásánál fontos elvárás, hogy a tesztelés alatt álló szoftverrel végrehajtatott terhelés egységes legyen, hiszen csak így lehet közvetlenül összehasonlítani a futások eredményeit. Mivel a mérés során grafikus felhasználói felülettel rendelkező szoftverekbe szeretnénk hibát injektálni, ezért olyan terhelésbiztosító alkalmazásra lesz szükségünk, mely képes a GUI-n rögzített akciókat végrehajtani.

Több eszköz is létezik, melyek közül kiemelkedik a Froglogic cég Squish [2] nevű megoldása, illetve további ismert szoftver pl. az IBM-től a Rational Robot[9]. Ezek jellemzően bármilyen ablakozóval készül GUI tesztelését végre tudják hajtani (Qt, SWT, MFC, .Net, ...), mivel alacsony szintű paraméterekkel (pl. egérmutató pozíciója) dolgoznak. Sajnos ezeket az eszközöket a laborban nem tudjuk kipróbálni.

Más eszközök is léteznek, melyek nem ilyen alacsony szintű paramétereket figyelnek, hanem megjegyzik azokat az objektumokat, melyeket a felhasználó aktivál. Ez a megoldás természe-

tesen nem lehet univerzális az előzőhöz képest, ám így is létezik olyan eszköz [12] mely képes Eclipse pluginekhez, RCP, SWT és Swing alkalmazásokhoz GUI teszteket rögzíteni. [3]

A laboron mi is ezt a szoftvert fogjuk kipróbálni, és bár használata egyszerű, nem árt röviden belenézni a weboldalon található tutorialokba.

5. OLAP

Hibainjektálási kísérletek eredményeinek elemzésére is jól használható technika az OLAP (*Online Analytical Processing*), melyet arra fejlesztettek ki, hogy nagy mennyiségű adaton lehessen gyors analízis műveleteket végrehajtani. Az OLTP-vel (*Online Transaction Processingel*) ellentétben itt jellemzően csak olvasási műveletek vannak. Az OLAP-on belül ROLAP-nak nevezik, mikor az adatok egy relációs adatbázisban vannak eltárolva, és az adatokhoz való hozzáféséhez az OLAP szoftvernek SQL lekérdezéseket kell generálnia a háttérben.

OLAP során a komplex elemzések végrehajtása úgynevezett multi-dimenzionális adatok felett történik. Ezt nagyjából úgy kell elképzelni, hogy egy központi úgynevezett *ténytáblában* vannak eltárolva azok az adatok, melyeket a többi, *dimenziótáblában* levő adatok alapján kategorizálhatunk.

Az OLAP koncepció alapeleme az *OLAP kocka* (valójában nincsen megkötés a dimenzióra, így helyesebb hiperkockának nevezni), mely kiválóan előállítható például egy csillag-séma [10] szerint rendezett adattárházból (a labor során használt csillag séma a 1 ábrán látható). Az elemzés lényege, hogy a hiperkocka egy dimenzióját lerögzítve vizsgáljuk az adatokat. A korábban írtak fényében ezt úgy lehet elképzelni, hogy egy dimenziótábla elemei alapján szűrjük a ténytáblában található elemeket.

Ez még szemléletesebben a hibainjektálási kísérletek elemzésénél azt jelentheti például, hogy egy adott hibaterhelés mellett (ez lesz a rögzített dimenzió) a különböző vizsgált rendszerek milyen válaszokat adtak a CRASH hibahatás modell szerint. (A rögzített hibaterhelés tartalmazhatja a hibák számát, típusát, gyakoriságát, stb.)

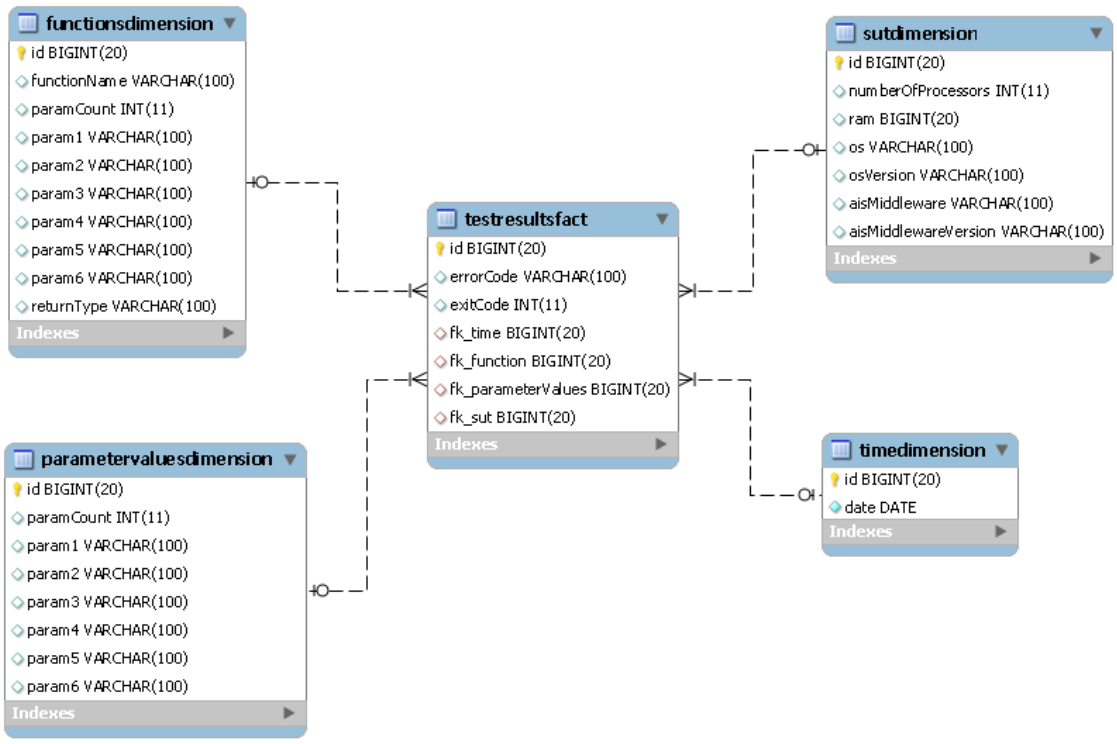
Előfordulhat olyan eset is, mikor új ténytábla hozzáadása válik szükségessé, az adatok nagymértékű eltérése miatt. Ha azonban az új táblához is alkalmazható egy már korábban is használt dimenzió, akkor a csillag-sémák a közös dimenziók mentén össze lesznek kötve, így a köztük lévő összehasonlítások is értelmesek lehetnek.

A legtöbb OLAP eszközzel az adatok analíziséhez nincs szükség magas szintű ismeretekre a lekérdezések készítésének területén. (Általában grafikus felhasználói felületet nyújtanak a lekérdezések összeállításához, az ezeket végrehajtó komplex parancsokat pedig a háttérben állítják össze. Az eredmények megjelenítéséhez pedig számos grafikus ábrázolás közül lehet választani.) Azonban természetes, hogy az OLAP szoftvert valahogy fel kell konfigurálni, hogy hozzáférhessen a relációs adatbázisban tárolt adatokhoz, és az adattárház struktúráját is kezelni tudja. Erre való a sémaleírás, mely természetesen alkalmazásfüggő (a mi esetünkben egy XML fájl lesz).

Ezen felül szükség van egy lekérdező nyelvre is, amit az OLAP szoftver értelmezni tud. Ez az iparban de facto szabványként elterjedt *MDX* [8] nyelv[4]. Szintaxisa nagyban hasonlít az SQL-hez, azonban használata éppen az OLAP lekérdezésekhez van kialakítva.

[3] Meg kell jegyezni, hogy a WindowTester Pro a *Instantiations* cég korábbi piacvezető terméke, komoly ipar felhasználótáborral. Ezt a céget a Google a közelmúltban vásárolta fel, és 2010 szeptember közepén tette ingyenes elérhetővé számos korábbi terméket, többek között a WindowTester Pro-t is. Az eszköz egészen ára egészen a közelmúltig $1295 US volt. Akit érdekel a hír: `http://dev.eclipse.org/blogs/mike/2010/09/16/a-good-day-for-java-development/`

[4] Sajnos nem elhanyagolható hátrány, hogy mivel az MDX de facto szabvány, így a legtöbb gyártó igyekszik úgy módosítani az MDX szintaxist, hogy az nekik kényelmes legyen.



1. ábra. Csillag séma

Kiemelten fontos egy speciális elem, a *Measure*. Ezek jelképezik azokat az adatokat, melyeket éppen *vizsgálunk*. Egyszerűen fogalmazva ezek többnyire a ténytábla elemei lesznek, melyeket szűrünk a dimenziók lerögzítésével.

Rövid példaként tekintsük a már említett csillag sémánkat (1). Vegyük úgy, hogy az OLAP szoftver számára létrehozott séma változtatás nélkül ezeket az elnevezéseket használja (tábla illetve oszlopnevek). Egy olyan MDX query, mely az összes olyan kísérlet hibakódjait a szerint válogatja szét, hogy mi volt a tesztelt rendszer operációs rendszere (a query a hibakódokat rendezi az oszlopokba, és az OS-ek kerülnek a sorokba):

```
SELECT
{[Measures].[errorCode]} ON COLUMNS,
{[sutdimension].[os]} ON ROWS
FROM fault_injection
```

5.1. A mérés során használt OLAP eszköz bemutatása

A mérés során a JasperForge [3] szoftvereit (a Business Intelligence területen szoftvereket kínáló Jaspersoft [4] nyílt forráskódú termékei) fogjuk használni, nevezetesen JasperServert[5] és a JasperAnalysist.

A JasperServer a dotCMS-hez hasonlóan egy Tomcaten futó webalkalmazás. Maga a JasperServer a felhasználókezelést, a megjelenítést, exportálási lehetőségeket és az adattárházzal való kapcsolatot biztosítja, tehát összességében egyfajta keretet biztosít az elemzéseket végrehajtó szoftver számára. A JasperAnalysis tulajdonképpen egy előre telepített alkalmazás a

[5]Maga a JasperServer azonban a Mondrian nevű OLAP szervert használja és erre ad egy saját felületet illetve néhány plusz funkciót, mint ahogy sok más OLAP szoftver is.

JasperServeren, mely az adatmegjelenítés kialakítását (az adatkocka szintjén is), a különféle diagramok megjelenítését, ezek paraméterezését teszi lehetővé, a minél érthetőbb és áttekinthetőbb megjelenítés érdekében.

A mérés során elvégzendő feladatok

Megjegyzés a méréshez

A mérés virtuális gépeken zajlik (VMware [11]), Eclipse platformon (Helios) [1]. Bár remélhetőleg ezekkel mindenki találkozott már legalább az eddigi mérések során, de ha valaki mégsem ismeri az alapfogalmakat, és nem tudja ezeket a szoftvereket legalább felhasználói szinten kezelni, az nagyban megnehezíti a mérés végrehajtását a megadott idő alatt. Az említetteken felül a feladatok végrehajtásához szükség lesz még legalább alapvető Java programozói tudásra is.

Egyszerű reflektív program készítése

A mérési feladat célja egy olyan egyszerű alkalmazás elkészítése, mely a Javassist könyvtár felhasználásával képes felfedezni és manipulálni egy Java class fájlt.

Ehhez az *Excercise1* workspace-ben rendelkezésre áll egy olyan keret-alkalmazás, mely inputként kap egy class fájlt, és annak kiírja a teljes elérési útját.[6]

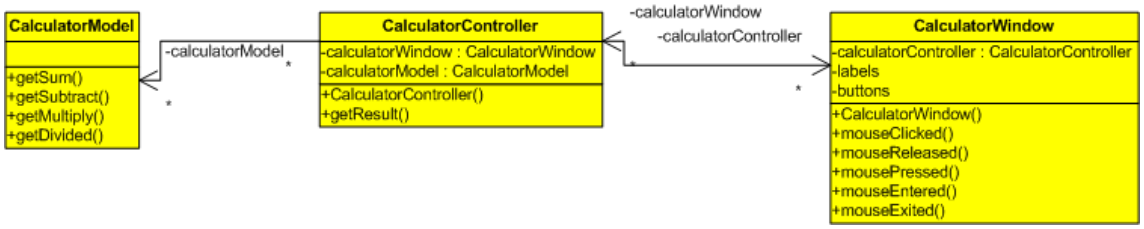
Az első feladat ennek a továbbfejlesztése olyan módon, hogy képes legyen a bemenetként kapott class fájl változóit, konstruktorait valamint metódusait megkeresni, és kiírni a konzolra. A workspace-ben található egy *Test* nevű alkalmazás. A feladat teljesítéséhez használhatja ennek bináris fájljait!

A feladat második részében módosítsa úgy a beolvasott class fájlt, hogy annak minden konstruktora és metódusa a meghíváskor írjon ki egy sort a konzolra. A kiírt információ tartalmazza a meghívott konstruktor vagy metódus nevét! Az injektálás sikerességének ellenőrzéséhez futtassa le a Test alkalmazást!

Hibainjektálási kísérletek konfigurálása - egyszerű célalkalmazás

A hibainjektáló keretrendszerrel való ismerkedés céljából először váltson az *Excercise2* workspacere, majd hozzon létre egy hibainjektáló projektet az itt található *TestCalculator*[7] alkalmazásból (Yellow Fault Injection Project from workspace). Az így létrehozott projektben hozzon létre egy modellt. (Ezt azonnal át is nevezheti, hogy minél könnyebben meg tudja különböztetni a konfigurációkat.)

Az tesztalkalmazás architektúrája az 2 ábrán látható, de természetesen ebben az esetben a forráskód is elérhető. Ezek alapján igyekezzen megismerni a generált modellt, és kitalálni olyan lehetséges helyeket, ahová jól megfigyelhető hibákat tud injektálni.



2. ábra. A tesztelendő számológép architektúrája

[6] Az alkalmazás valójában egy plug-int valósít meg, mely egy view-t rak ki az Eclipse GUI-ra, melyen a Browse gombbal tallózható egy tetszőleges class file, a Start gomb pedig a kiíratást indítja.

[7] A projekt egy nagyon egyszerű számológépet valósít meg, SWT felhasználói felülettel.

Készítsen a WindowTester Pro alkalmazás segítségével egy olyan workloadot, mely várhatóan az összes injektált hibát aktiválja! A későbbiekben az egyes hibainjektálások után használja ezt a workloadot.

Hozzon létre hibakonfigurációkat, és injektálja azokat. Amenyiben már injektált egy hibát, utána érdemes a célalkalmazás projektjén egy *Cleant* futtatni, mivel különben az Eclipse nem veszi észre, hogy megváltoztak a binárisok (*clean and build immediately*) [8].

Ezt követően próbálja ki a monitorozási lehetőségeket is! Állítson össze egy monitorozási konfigurációt, és azt is injektálja a célalkalmazásba. Jegyezze fel a tapasztalatait!

Tesztalkalmazás módosítása

Tegye hibatűrővé az eddig használt számológépet (elég két alapművelet esetén)! Ehhez használjon n-verziós programozást, megfelelő kivételkezelést, stb.

Ezt követően újra injektáljon hibákat az így felkészített célalkalmazásba (ehhez természetesen szükség lesz egy újabb hibainjektáló projekt létrehozására)! Végezzen több kísérletet (hibatűrővé tett és nem hibatűrő műveletre is, változatos hibatípusokkal), és jegyezze fel milyen arányban detektálta a hibákat a "hibatűrő számológép"!

A mérési eredmények felhasználásával, a 2.2.2 fejezetben leírtak alapján számítsa ki, hogy mennyi kísérletet kellene végezünk ahhoz, hogy 5%-os konfidenciaintervallummal, 99%-os konfidenciaszinttel meg tudjuk állapítani a detektált hibák arányát!

Opcionális: Amenyiben több mint 30 kísérletet elvégzett, kiszámolhatja az alkalmazott hibadetektáló eljárások hibafedését is (2.1.2)!

Hibainjektálási kísérletek konfigurálása - komplex célalkalmazás

Ebben a mérési feladatban a komplex célalkalmazás a dotCMS tartalomkezelő és annak komponensei lesznek[9]. Ehhez a dotCMS megfelelő könyvtárában (dotCMS/stable/common/lib) található különböző jar fájlokat fogjuk felhasználni. A jar fájlokat kibontva közvetlenül is hozzáférnénk a class fájlokhoz, azonban az Eclipse PDE lehetővé teszi, hogy a jar fájlokból egyszerűen plug-in projektet készítsünk, megkönnyítve ezzel a dolgunkat.

Váltson át az *Excercise3* workspace-be. Itt már előre létre van hozva 5 dotCMS plug-in projekt (dotCMSn), és az ezekhez tartozó hibainjektáló projektek (dotCMSn_fi). A plug-in projektek a dotCMS core komponensét megvalósító *dotCMS1.7a.jar* fájlból készültek.[10]

Nézze meg a létrehozott *ymodel* fájlokat. Használj a dotCMS dokumentációját is amennyiben szükséges, és próbáljon találni olyan helyeket, ahová érdemes lehet hibákat injektálni. Segítségként próbáljon beazonosítani a következő funkciókhoz tartozó függvényhívásokat, és azokba injektálni hibákat:

- Indulással kapcsolatos ellenőrzések, rendszerhívások.

[8] Ez pedig hosszútávon azt eredményezné, hogy az injektált hibák akkumulálódnak, meghamisítva ezzel a mérések eredményeit.

[9] A laboron használt image-en a dotCMS-t futtató Tomcat szerver szolgáltatásként van konfigurálva. Ezért az elindításához/leállításához be kell lépni a Windows szolgáltatásokat kezelő menübe, és végrehajtani a kívánt akciót.

[10] Mivel ebben az esetben a forrás nem áll rendelkezésre, sajnos a korábban használt *Clean-and-Build* módszer nem használható. Így ha egy projektbe már injektáltak egy létrehozott konfigurációt, akkor abba többet nem lehet/érdemes. Ezért ha további kísérleteke szeretnénk konfigurálni, akkor új projektet kell létrehozni a jar fájlból. Azonban ekkor az új projekthez létrehozott új hibainjektáló projekthez a modell létrehozása a hibainjektáló tool számára 4-5 perc. Ezt egyszer érdemes lehet kipróbálni, azonban sokszor nyilván nagyon lelassítaná a mérést, így érdemesebb egy másik, ugyanebből a jarból létrehozott projekthez készített hibainjektáló projekt *.ymodel* fájlját bemásolni a konfigurációk mappába.

- Felhasználó autentikáció.

Ha egy konfigurációt injektált, akkor a plug-in projektből újra elő kell állítani a jar fájlt. Ehhez **frissítsen egyet a plug-in projekten**, majd exportálja jar-ként. Az exportált jar fájlokat célszerű egy közös mappában gyűjteni (pl. . . . /Workspace/Exported_jars), és valamilyen névvel azonosítani, hogy tudni lehessen milyen hiba van injektálva. A futtatáshoz felül kell írni a dotCMS/stable/common/lib könyvtárban található jar fájlt a módosítottal (ez csak akkor lehetséges ha a Tomcat szerver le van állítva!)[11].

Végezzen el néhány kísérletet, és az eredményeket osztályozza a *CRASH* hibahatás modell alapján!

Az eredmények értékelése

Az eredmények értékelésénél egy korábbi, a tanszékünk közreműködésével zajló projekt mérési eredményeit fogjuk felhasználni. Ez az adatbázis több mint 100.000 kísérlet eredményeit tartalmazza. Az eredmények a már korábban is említett *csillag-séma* szerinti adattárházba vannak feltöltve. Első feladatként indítsa el a MySQL Workbench-et, keresse meg a *fault_injection* sémát, és vizsgálja meg a felépítését. Azonosítsa be az egyes táblákat, nézze meg tartalmukat.

A MySQL-ben található csillag sémához való hozzáféréshez a JasperServert konfigurálni kell. A *connection* köti össze a MySQL adatbázissal, a *data source* az aktuálisan használt sémával, végül egy XML séma írja le az OLAP kocka felépítését. Vizsgálja meg ezt az XML-fájlt[12], és az adatbázis séma, illetve az OLAP-ról szerzett ismeretei alapján próbálja meg kitalálni, hogy a Measure elemek ilyen kialakítása mi célt szolgál.

Miután elindította a JasperServert[13], lépjen be adminisztrátor felhasználóval[14]. A JasperAnalysis segítségével analízis nézeteket hozhatunk létre, melyek létrehozásához szükség van a korábban leírt három komponensre, és egy MDX query-re. az Analysis Components/ FaultInjectionViews pontban már van egy kész View, mely a kísérletek kilépési kódjainak adatait jeleníti meg az adatbázisból. Futtassa le, majd módosítsa az MDX query-t úgy, hogy az oszlopokban pedig a SUT dimenzió tagjai jelenjenek meg [15]. Ezt követően futtassa le a módosított nézetet, és próbálja ki a nézet egyéb lehetőségeit (szűrés - Change Data Cube opciónál, különböző megjelenítési lehetőségek).

A jelenlegi View csak az kilépési kódokkal foglalkozik. Azonban további hasznos információhoz jutunk, ha ugyanezt az elemzést a hibakódokra is elvégezzük. Ezért készítsen egy új nézetet (jobb klikk a FaultInjectionViews mappán, majd AddResource/ Other/ AnalysisView), melyben a Measure elemeket a korábban megvizsgált sémaleírásból válogatja össze!

[11] Felülíráskor figyeljen arra, hogy az eredeti jar fájlokat mentse el valahová vagy nevezze át, hogy később vissza tudja állítani az eredeti állapotot.

[12] C:/Program Files/jasperserver-ce-3.7.1/FaultInjection/FISchema.xml

[13] A JasperServer a dotCMS-hez hasonlóan szolgáltatásként van konfigurálva!

[14] A chrome-ban a könyvjelzők között be van állítva a JasperServer címe. Belépés után érdemes egyből bal alul az alapértelmezett "Visualization typesot" "All typesra" állítani!

[15] Jobb klikk a FaultInjectionErrorView-n, majd Edit, itt pedig Next amíg el nem ér az MDX query-ig.

Ellenőrző kérdések[16]

- Mit jelent a hibainjektálás irányíthatósága?
- Mit jelent a hibainjektálás megfigyelhetősége?
- Mit jelent a hibainjektálás reprodukálhatósága?
- Mit jelent a hibainjektálás terhelés reprezentativitása?
- Milyen módszereket ismer szoftverhibák injektálására?
- Mit jelent a meghibásodás-injektálás?
- Mit jelent a hibainjektálás?
- Mi a robosztusság definíciója?
- Miért hasznos a konfidenciaintervallumok meghatározása?
- Mi a reflektív programozás lényege?
- Mi az OLAP koncepció lényege röviden?

[16] Az itt felsorolt kérdések az önellenőrzést hivatottak szolgálni, a beugrón további kérdések is elképzelhetőek!

Hivatkozások

[1] Eclipse. http://www.eclipse.org/.

[2] FrogLogic Squish. http://www.froglogic.com/.

[3] JasperForge. http://jasperforge.org/.

[4] Jaspersoft. http://www.jaspersoft.com/.

[5] Javassist. http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[6] Javassist manual. http://www.csg.is.titech.ac.jp/~chiba/javassist/html/index.html.

[7] Javassist tutorial. http://www.csg.is.titech.ac.jp/~chiba/javassist/tutorial/tutorial.html.

[8] MultiDimensional eXpressions. http://en.wikipedia.org/wiki/MultiDimensional_eXpressions.

[9] Rational Robot. http://www-01.ibm.com/software/awdtools/tester/robot/.

[10] Star schema. http://en.wikipedia.org/wiki/Star_schema.

[11] VMware. http://www.vmware.com/.

[12] WindowTester Pro. http://code.google.com/webtoolkit/tools/wintester/html/index.html.

[13] AMBER consortium; Assessing, Measuring and Benchmarking Resilience. State of the Art. Technical report, 2008.