FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Berkes Sándor

Tulajdonság alapú zárolás megvalósítása EMF modellek felett

Konzulens

Debreceni Csaba

doktorandusz

BUDAPEST, 2015

Tartalomjegyzék

[Összefoglaló 6](#_Toc419705988)

[Abstract 7](#_Toc419705989)

[1 Bevezetés 8](#_Toc419705990)

[1.1 Esettanulmány: Mondo 8](#_Toc419705991)

[1.2 Dolgozat célja 8](#_Toc419705992)

[1.3 Dolgozat struktúrája 8](#_Toc419705993)

[2 Háttér technológiák 9](#_Toc419705994)

[2.1 Verziókezelő Rendszerek 9](#_Toc419705995)

[2.1.1 SVN 9](#_Toc419705996)

[2.1.2 GIT 9](#_Toc419705997)

[2.2 Eclipse 12](#_Toc419705998)

[2.3 Eclipse Platform 12](#_Toc419705999)

[2.3.1 Eclipse PDE 13](#_Toc419706000)

[2.3.2 OSGI 14](#_Toc419706001)

[2.3.3 EMF 14](#_Toc419706002)

[2.3.4 IncQuery 14](#_Toc419706003)

[2.3.5 Egyébb Eclipse által biztosított eszközök 15](#_Toc419706004)

[2.4 Jersey 18](#_Toc419706005)

[2.4.1 Rest 18](#_Toc419706006)

[2.4.2 JSON 19](#_Toc419706007)

[3 Áttekintés 21](#_Toc419706008)

[3.1 Kliens – Szerver felépítése 21](#_Toc419706009)

[3.2 Kliens részletesebben 22](#_Toc419706010)

[4 Megvalósítás 24](#_Toc419706011)

[4.1 Server 24](#_Toc419706012)

[4.1.1 projecthez fájl feltöltés 24](#_Toc419706013)

[4.1.2 adott projekt lock fájlainak lekérdezése 24](#_Toc419706014)

[4.1.3 adott projekt fájlának a letöltése 25](#_Toc419706015)

[4.2 Kliens 25](#_Toc419706016)

[4.2.1 projekt leíró 25](#_Toc419706017)

[4.2.2 zárak definiálás 26](#_Toc419706018)

[4.2.3 zárak megvalósítása 27](#_Toc419706019)

[4.2.4 beállítások 30](#_Toc419706020)

[4.2.5 Jelölök használata 31](#_Toc419706021)

[4.2.6 Jersey használata 31](#_Toc419706022)

[5 Kiértékelés 33](#_Toc419706023)

[6 Kapcsolódó munkák 34](#_Toc419706024)

[6.1 Zárak megvalósítása általában 34](#_Toc419706025)

[6.2 CDO 34](#_Toc419706026)

[6.3 EMFstore 35](#_Toc419706027)

[6.4 EMFstore és CDO összehasonlítása 35](#_Toc419706028)

[6.4.1 Adattárolás 35](#_Toc419706029)

[6.4.2 Kollaboráció 36](#_Toc419706030)

[6.4.3 Jogok kezelése, zárolás 36](#_Toc419706031)

[6.4.4 Konfliktusok megoldása 37](#_Toc419706032)

[6.4.5 Skálázhatóság 37](#_Toc419706033)

[6.4.6 Összefoglalás 37](#_Toc419706034)

[6.5 MetaEdit 38](#_Toc419706035)

[6.5.1 Modell szerkesztés 38](#_Toc419706036)

[6.5.2 Kollaboratív munka támogatása 39](#_Toc419706037)

[6.5.3 integrálhatóság, más környezetekkel való kapcsolat 39](#_Toc419706038)

[7 Továbbfejlesztési irányok 40](#_Toc419706039)

[8 Összegzés 42](#_Toc419706040)

[Irodalomjegyzék 43](#_Toc419706041)

[Függelék 44](#_Toc419706042)

Hallgatói nyilatkozat

Alulírott **Rezeda Kázmér**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/ diplomatervet (nem kívánt törlendő) meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2015. 05. 18.

...…………………………………………….

Berkes Sándor

Összefoglaló

Ide jön a ½-1 oldalas magyar nyelvű összefoglaló, melynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

# Bevezetés

## Esettanulmány: Mondo

## Dolgozat célja

A szakdolgozatomban kitűzött célok tehát a következők:

* Megismerkedni több EMF modellek kollaborációs szerkesztését támogató eszközzel, majd összehasonlítani őket funkcionalitásuk terén
* Felkutatni az EMF modellek zárolási lehetőségének a mikéntjét
* Megvalósítani egy példa alkalmazást és bemutatni képességeit

## Dolgozat struktúrája

A következező az az a 2. fejezetben az általam elkészített alkalmazáshoz kapcsolódó és felhasznált technológiákat, eszközöket mutatnám be. Ezek után a 3. fejezetben megoldásom kulcs fontosságú komponenseit, és működésének fő logikáját mutatnám be. A 4. fejezet részletesen bemutatnám az eszköz működését, majd az 5.-ben értékelném azt és mérések segítségével elemzem hatékonyságát. A 6. fejezetben s témával kapcsolatos megoldásokat hasonlítanék össze, majd ezek tükrében a 7. fejezetben esetleges továbbfejlesztési irányokat határoznék meg. Végül a 8. fejezetben összegezném és értékelném az elvégzett munkámat.

# Háttér technológiák

Ebben a fejezetben a munkám során felhasznált technológiákat, eszközöket szeretném bemutatni. Mivel az általam elkészített megoldás lényegében egy lehetséges kiegészítése egy verziókezelő rendszernek, így elsőre is ezek közül szeretnék bemutatni kettőt, az GIT és SVN verziókezelőket. Ezek után az Eclipse Platformot fogom bemutatni és általam használt szolgáltatásait.

## Verziókezelő Rendszerek

http://lf.estontorise.hu/archives/164

A verziókezelő rendszerek fő lényege, hogy a szoftverfejlesztő az által írt program forráskódját megfelelően tudja menedzselni. Értjük ez alatt a forrás rendszeres mentését, nyomon követést, igény esetén egy pontra visszaállítása, illetve egyéb forrás manipulációs lehetőségeket. Mindezeket manapság kollaboratív környezetben valósítják meg a rendszerek, tehát több személy is dolgozhat ugyanazon a forráson egyszerre. Jelenleg a piacon két fő rendszer a domináló, ezeket fogom most bemutatni részletesen.

### SVN

Az SVN eredeti nevén Apache Subversion egy központi, centralizált verziókezelő rendszer. Ebben az esetben a felhasználók egy közös központi szervert használnak. A változtatásaikat ide töltik fel, illetve a mások által végrehajtott változtatásokat is innen töltik le. Fontos megjegyezni, hogy minden egyes feltöltést egy letöltést előz meg, hisz a lokális módosítást muszáj összefésülni a szerveren lévő jelenlegi kóddal, majd az összefésült verzió kerül továbbításra a szerve felé, és ez válik majd az aktuális változattá a szerveren. Az összefésülést vagy a rendszernek sikerül önmagától megoldania, vagy ha a rendszer feloldhatatlan problémába ütközik, akkor a felhasználónak kell feloldania a konfliktust.

### GIT

A GIT egy elosztott, decentralizált rendszer. Ez azt jelenti, hogy ebben az esetben minden egyes felhasználó rendelkezik egy saját tárolóval, ami képes a központi tárolónak használt szervertől teljesen függetlenül működni. Ennek számos előnye van, például átláthatóbbá válhat a fejlesztés az által, hogy a kisebb változtatások csak a helyi tárolóba kerülnek, az ezekből álló komplexebb egységet alkotó változtatások pedig már felkerülhet a központinak jelölt tárolóba. Ezáltal a központi változat mindig konzisztens állapotban lehet. Másik nagy előnye a biztonság, mivel itt minden felhasználó rendelkezik egy tárolóval, és azok mind rendelkeznek a fejlesztés történetével, így a központi elem kiesése esetén sem fordulhat elő adatvesztés, hisz a többi tároló ugyan úgy őrzi a fejlesztés állapotait. Ezek után a GIT néhány fontosabb műveletét szeretném bemutatni.

#### commit

A helyi tárolóban megvizsgálja az állományok régi és új állapotait, majd ennek differenciájából születik meg a commit. A commithoz még egy szöveges leírási is tartozik, ez rendszerint egy pár szavas leírás ami commit által érintett változásokat összegezi. A művelet végrehajtása után a tárolóban a legújabb verzió lesz az aktív verzió. Fontos hogy a művelet elvégzésekor, csak azok a fájlok vesznek részt a procedúrában, amelyek ki lettek rá jelölve, más szóval indexelve lettek.

#### push

Ez a művelet egy másik távoli repository állományait tudja felfrissíteni a mi saját lokális példányunkból. A távoli tárolót tipikusan a központinak használt tároló személyesíti meg. Ha a rendszer konfliktusba ütközik a művelet során, és nem tudja a két forrást egymásba olvasztani, akkor nekünk kézzel kell feloldani a konfliktusokat, majd commit és újra push műveletet végrehajtani.

#### fetch

A távolinak megjelölt tárolóban található olyan változásokat töltene le a mi tárhelyünkre, amikkel mi nem rendelkezünk. Fontos, hogy ez esetben a lokális munkapéldányunkat ez nem változtatja meg, ahhoz hogy változtatások azon is megjelenjenek egy merge utasítás is szükséges még. Ez a parancs főleg akkor lehet hasznos, ha mások által végrehajtott változásokat felül szeretnénk vizsgálni és eldönteni, hogy mely változások jelenjenek meg a saját munka példányunkban.

#### pull

Lényegében ez a parancs a fetch majd a merge végrehajtását jelenti.

#### merge

https://www.atlassian.com/git/tutorials/merging-vs-rebasing

A fejlesztés során gyakran előfordul, hogy egyszerre akár több problémát is meg kell oldanunk párhuzamosan vagy több verziót is fenn kell tartani a fejlesztendő alkalmazásból. Ekkor a verziókezelő úgynevezett branch az az ág eszközéhez nyúlnak. Ez lehetőséget biztosít arra, hogy egy adott szakaszban több felé szétválasszanak egy kódbázist, majd lehetőség szerint akár össze is fésüljék őket. Jó példa erre lehet, hogy ha a munka folyamán az alkalmazásból létezik egy teszt és egy stabil ág. Ez esetben a fejlesztések a teszt ágban történnek, majd ha ott jónak bizonyulnak, akkor ennek az ágnak a változásai bekerülhetnek a stabil ágba. Ezt a két át közötti összefésült a merge utasítás végzi el.

#### rebase

https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Az előző utasításhoz hasonlóan ez is összefésülést végez, azonban egészen más megközelítésben. A különbség abból adódik, hogy míg a merge esetében a másik branch új változásai időrendben előrehaladva egy új commitként jelenik meg, addig a rebase esetén a mi águnk alapjának veszi a kiválasztott ágat, majd a jelenlegi branch commitjai ennek a „tetején” jelennek meg, tehát a rabasere jelölt ág változásai időrendben hátrébb lesznek, a mi jelenlegi águnk commitjai. Ennek számos előnye és hátrányai is lehet, egyik fő aspektusa a nyomon követhetőség. Rebase esetén lényegében feltétel nélkül elfogadjuk a másik ág változásait, míg ha merge utasítást választjuk, nagyobb szabadsággal élhetünk. Pont emiatt is a merge átláthatatlanabb hisz túl sok fölösleges járulékos napló bejegyzést jelenthet, ellenben a rebase ami nem zavarja meg ilyen szempontból a fejlesztés nyomon követhetőségét.

#### reset

Ezzel az utasítással visszaállíthatatlanul vehetünk fel egy másik commitolt állapotot a tárolónkban. Ennek egy felhasználó barátiabb verziója a revert utasítás, ez esetben már visszatérhetünk az épp elhagyott pontra is később.

#### cherry pick

<http://think-like-a-git.net/sections/rebase-from-the-ground-up/cherry-picking-explained.html>

Ebben az esetben lehetőségünk nyílik arra, hogy egy adott ágba egy másik ágból átmozgassunk commitokat. ÁBRA ÉS MAGYARÁZAT, mert ábra nélkül ☹

## Eclipse

<https://eclipse.org/#sec_ide>

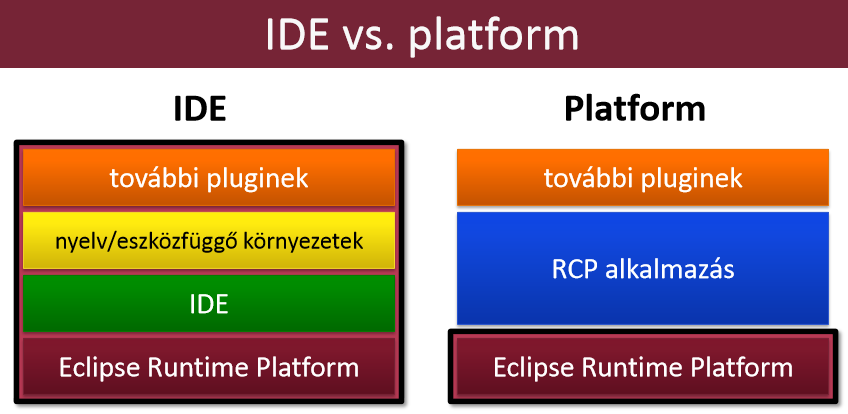
<https://inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/rendszertervez%C3%A9s-szakk%C3%B6r/12/szakkor_eclipse_eloadas.pdf>

target platfrom, run config KELL E ?! product ?

Az Eclipse magát egy nyílt forráskódú közösségnek nevezi, ami magában foglal eszközöket, projekteket és az ezeket létrehozó fejlesztők csoportosulásait. Mindezt pedig a szintén Eclipse nevű alapítvány irányítja, hangolja össze. Fő projektjük az Eclipse Platfromnak nevezett környezetük. Lényegében majdnem az összes Eclipse technológia ennek segítségével érhető el, illetve futatható.

## Eclipse Platform

A Platform számos keretrendszert és szolgáltatás definiál, amik segítségével már könnyen alkalmazásokat lehet készíteni. Ilyen eszközök például a fontosabbakat említve a nagy testreszabhatósággal rendelkező grafikus felület (workbench,SWT), projekt alapú erőforrás kezelés, erőforrások figyelése, univerzális nyelv független fordító és hibakereső eszköz, több felhasználós verziókezelő rendszerek integrálhatósága (Team api) illetve dinamikus modul kezelő rendszer (OSGI) . Ezen szolgáltatások egy részéből áll össze a RCP az az Rich Client Platform, lényegében ez adja meg a lehetőségét arra, hogy vastag kliens alkalmazásokat készítsünk. Fontos megjegyezni, mivel a platfrom erős modularitással rendelkezik így az RCP használata nem zárja ki más Eclipse alapú eszközök használatát sem. Többek közt az Eclipse IDE az a fejlesztői környezete is erre a platformra épül.



### Eclipse PDE

A Plug-in Development Environment röviden PDE az Eclipse egy olyan eszköze, ami lehetőséget biztosít Eclipse IDE beépülő modulok, szolgáltatások, modul frissítő oldalak és RCP alkalmazások elkészítéséhez. A PDE szerteágazó OSGi eszközöket is tartalmaz, így környezetet biztosíthat komolyabb komponens alapú programozásra is. Én mivel csak Eclipse beépülő modult másnéven plug-int készítettem így csak az ezzel kapcsolatos funkciókat mutatnám be.

Ahhoz hogy egy Eclipse plug-int hozhassunk létre, ennek megfelelő nevű típusú projektet kell létrehozni. A projektben automatikusa létrejön az plugins.xml nevű fájl. Ez a plug-in legfontosabb fájlja. Itt történik meg a plug-in tulajdonságainak meghatározása. Itt adhatjuk meg a nevét, azonosítóját és egyéb alap leíróját. Lehetőségünk van, a modulunkhoz aktivátor osztályt is hozzárendelni, ekkor a kiválasztott osztálynak az AbstractUIPlugin nevű osztályból kell leszármaznia. Az itt felüldefiniált start és stop függvény pedig akkor hívódik meg, ha modulunkat elindította a rendszer illetve leállította.

A plugins.xml fájljában kötelezően meg kell adni, hogy az adott plug-in melyik másik modulokat is használja még fel. Illetve lehetőségünk van csak osztály package megnevezése által megnevezni a kívánt komponenseket, ekkor a rendszer maga deríti fel, hogy melyik másik modulban található a hivatkozott package. A leíró fájl egy másik általam használt funkciója az Extensions az az a kiterjesztések rész. Itt van lehetőségünk regisztrálni, hogy modulunk mivel szeretné kiegészíteni a rendszert. A kiegészítés tipikusan abban valósul meg, hogy az adott osztály megvalósítja a kiválasztott interfészt majd az Extensions résznél regisztrálásra kerül.

### OSGI

https://inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/msc-t%C3%A1rgyak/rendszerintegr%C3%A1ci%C3%B3-%C3%A9s-fel%C3%BCgyelet-laborat%C3%B3rium/12/osgi.pdf

Az OSGI (Open Services Gateway Initiative) egy olyan felsőbb szintű szolgáltatási réteget valósít meg a java virtuális gépe fölött, ami lényegében operációs rendszer szerű folyamat kezelést tesz lehetővé. Két fontos aspektusa a rendszernek a szeparáció illetve az erőforrások dinamikus megosztása. A rendszer egységként az úgynevezett batyut (bundle) kezeli. Akár csak egy folyamatnak, ennek is vannak állapotai és életciklusa. Fontos tulajdonsága még egy ilyen modulnak hogy a futtatásához követelményeket más komponenseket határozhat meg, illetve ő is kiajánlhat szolgáltatások más modulok felé. Maga az Eclipse platform is OSGI szolgáltatást nyújt, épp ezért a könnyű átjárhatóság végett az Eclipse-en belül egy plug-in az OSGi rendszerén belül egy bundle-nek feleltethető meg.

### EMF

Az EMF hosszúnevén Eclipse Modeling Framework egy modellezésre használt eszköz. Lehetőséget nyújt modellek magas fokú szerkesztésére és modellekből többféle célra felhasználható kód generálására. Ilyen generált eszköz lehet például a modell alkalmazásban egyből felhasználható kódja, különböző adat módosítást segítő eszközök, illetve grafikus szerkesztői felületek.

### IncQuery

Az EMF modellekkel való munka során nagy problémát jelenthet a rajtuk való hatékony keresés megvalósítása. Ehhez nyújt segítséget az IncQuery nevű eszköz. Egyik fő sajátossága hogy a kéréseinket deklaratív nyelven kell megfogalmazni majd rendszer a kérésekhez megfelelő imperatív kódot generál és futtat. A keresés végrehajtáshoz szükséges kód generálódhat már fordítási időben, de lehetőség van futás időben is beolvasni a deklaratív mintát majd egyből felhasználni azt.

Egy IncQuery minta a következőképp néz ki:

**import** <http://WTSpec/2.0>

**pattern** subsystem2(SysID) {

**Subsystem**.**sysId**(\_Sys,SysID);

}

Elsőre is minden fájl elején szükségünk van importálni azon meta modellek nevét amiket használni szeretnénk. Később e modellek elemeire tudunk majd hivatkozni.

A minta deklarálása a pattern kulcsszóval kezdődik, majd a neve és a paraméter lista következik. A függvény törzsében egy mintát alakíthatunk ki az importált modell típusai és tulajdonságai alapján. Itt több illeszkedést is meghatározhatunk, ha ugyan az a változó több helyen is megjelenik, akkor nyílván szűkül annak a változónak az eredményhalmaza. A pattern eredményként csak a paraméterlistán megjelölt változók eredményhalmazával tér vissza. Ahol szükségünk van egy változóra azonban értéke számunkra lényegtelent, azt az ’\_’ karakterrel kell kezdeni. A mintánál a megfogalmazása során lehetőség van még már meglévő mintát újra felhasználni, logikai műveleteket alkalmazni, illetve tranzitív pattern is deklarálható.

### Egyébb Eclipse által biztosított eszközök

#### Beépülő modul automatikus indítása

http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg\_eclipse\_ui\_startup.html

Az Eclipse lehetőséget biztosít arra, hogy a rendszer elindítása után, bizonyos modulok mindenképp elinduljanak. Ehhez a modulon belül az egyik osztálynak meg kell valósítani az IStartup nevű interfészt és implementálni kell az earlyStartup függvényt. Ez a metódus a rendszer betöltődése után hívódik meg, tehát a rendszer alap szolgáltatásai, komponensei már biztosan elérhetőek. Így a plug-in kikerülheti a rendszer alapértelmezett komponensi betöltődési szabályát a lazy loadingot, ami csak igény eseti betöltődést jelentene.

#### Erőforrás jelölés

A rendszer használata során tipikusan előforduló eset, hogy bizonyos erőforrásokhoz bizonyos információt kell csatolni, például szintaktikai hiba a fájlban, erőforrás nem elérhető stb. A rendszerben minden egyes erőforrás az IResource interfészt valósítja meg, ennek pedig a createMarker nevű függvényével társíthatunk jelzést az adott elemhez. A társított információk szinte bárhol megjelenhetnek, a projekt nézetben, az elem szerkesztőjében stb., de ezek tipikusan szintén megjelennek a rendszerszintű Problems az az a problémák nevezetű nézetben.

#### Feladatok ütemezett végrehajtása

Az Eclipse régebbi verzióinál problémásak voltak a rendszert kiegészítő egyéb műveletek végrehajtásai. Tipikus problémaként jelent meg hogy az egyes műveleteket nehézkesen lehetett ütemezni, kölcsönös kizárás megvalósítása problémás volt, mi több gyakran előfordult az is, hogy a folyamat lefutása közben a rendszer nem reagált egy ideig, hisz a folyamat blokkolta azt. Ezek kiküszöbölésére hozták létre a Job osztályt. Használata emlékeztethet a Thread osztályra, azonban jóval többet tud nála. Akár csak a Thread delegált Runnable interfészénél, itt is egy run metódust kell megvalósítani, azonban itt egy IProgressMonitor típusú objektumot is kap a metódus. A ProgressMonitor segítségével a felhasználó számára jelezhetünk, hogy a végrehajtás épp milyen fázisban van, illetve kezelhetjük segítségével, ha épp a felhasználó megszakította a feladatot. Egy Job fő tulajdonságai közé tartozik, hogy mennyire kritikus a lefutása (priority), rendszerszintű feladat-e, tehát szükség van-e arra hogy a felhasználót értesítsük a végrehajtásról, illetve hogy milyen más feladatokkal futtatható együtt, melyekkel zárják ki egymást (SchedulingRules). A Job objektumok végrehajtását komplexebben a statikus Platfrom osztálytól elkért IJobManager interfészt megvalósító objektummal lehet. Ilyen műveletek lehetnek például a feladatok újra ütemezése, futás időben történő feladatokhoz tartozó dinamikus erőforrás zárolás vagy épp feladatok végrehajtása közben megjelenő események figyelése.

#### Feladatok egyszeri végrehajtása

Akár csak a Job esetében az IHandler interfészt megvalósítok osztályok is tipikusan egy komplexebb műveletet valósítanak. Azonban míg az előző esetében a hangsúly tipikusan a többszörös, ütemezett lefutáson van, itt azonban inkább egyszeri lefutásról beszélhetünk. Ahhoz hogy egy osztály megvalósíthassa ezt a tulajdonságot az IHandler interfészt kell megvalósítania. Ebben több függvény közül a legfontosabb az execute nevű függvény, ez hívódik meg végrehajtáskor. Az interfészt megvalósító osztályt a rendszertől elkért, IHanderService típusú objektum segítségével lehet futtatni. Ez mellett azonban lehetőség van implicit módon is meghívni az adott osztályt. A platformon belül jó példa erre a menüpontok vagy billentyűzet események kötése ilyen utasításokhoz. Ekkor a rendszer az adott trigger eseményre, például egy menüpont kiválasztása, meghívja a hozzárendelt utasítást.

#### Beállítások kezelése

Mint minden komplexebb platform esetén, itt is szükség volt egy központosított beállítások kezelőre. Ennek egyik legnagyobb előnye, hogy így a rendszer az összes beállítást egy központi helyen tudja kezelni, mind fizikailag és mind grafikus felületileg. Ahhoz hogy egy plug-in képes legyen a rendszer ilyen szolgáltatását használni a plugins.xml-ben egy osztályt kell beregisztrálni, aminek az ősosztálya AbstractPreferenceInitializer kell hogy legyen. Itt két függvényt kell felüldefiniálnunk. Az initializeDefaultPreferences nevűt a rendszer akkor hívja meg, ha elsőre szeretne hozzáférni az itt kezelt értékekhez, vagy alapbeállításra szeretné a rendszer helyezni az értékeket. A másik függvény, a setProperties pedig akkor hívódik meg, ha rendszer az értékeket szeretnék beállítani. Egy fontos pontja lehet az ilyen beállítás kezelőknek, hogy szinten mindig tartalmaznak egy figyelőt, ami kiértesít bizonyos általunk meghatározott érték változások esetén. Ez azért hasznos mert így kényelmesen, futás időben lehetőségünk nyílik megváltoztatni a modulunk működését, funkcionalitását.

Ahhoz hogy a beállítások megtudjanak jelenni az Eclipse beállítások menüpontjában, szükség van arra, hogy a plugins.xml fájlba beregisztráljunk egy IWorkbenchPreferencePage interfészt megvalósító osztályt. Itt, mint az Eclipse többi grafikus felületénél az SWT könyvtár segítségével építhetünk felületet a beállításainknak.

#### Projekt leíró használata

Ahhoz hogy az Eclipse rendszerén belül komolyabb projektkezelést valósítsunk meg, tipikusan információt kell hozzárendelnünk a projektekhez. Egy lehetőség lehet, hogy az adott információkat a projecten belül fájlokban tároljuk, azonban így nem jól különülnek a projekt meta adatai és a valós erőforrások, amiken dolgozni szeretnénk. Erre adhat megoldást a Project Nature nevű szolgáltatása a rendszernek. Saját projekt környezet definiálása után lehetőségünk nyílik a projektet megjelölni, majd különféle információk társítani a projekthez. E szolgáltatás használata igen gyakori, hisz a rendszer által létrehozott projektről is ennek segítségével derül ki, hogy például java vagy c++ projektek-e. Annak ellenére, hogy a rendszeren belülről egy jól elkülönített adatokról beszélünk, valójában a projektek ilyen jellegű leírója minden egyes projekt gyökér könyvtárában található meg, a „.project” fájlban.

#### Nézet és Perspektíva

Ha megoldásunkat szeretnénk grafikus megjelenítéssel is kiegészíteni, akkor ehhez egy ViewPart osztályból leszármazó osztályt kell megvalósítanunk. A plug-in leírójába való beregisztrálás után az SWT eszközei segítségével felépíthetjük a grafikus felületünket, majd az meg is jelenik az Eclipse „Show View” ablakában kiválasztható opcióként. Azonban ha egy feladat elvégzése során tipikusan több nézetre van szükségünk, azokat könnyen egy úgynevezett perspektívába gyűjthetjük össze. A perspektívát vagy maga a felhasználó is összeállíthatja vagy fejlesztési időben is lehetőség van perspektívákat összeállítani. Ehhez az IPerspectiveFactory megvalósítása majd a modul leíróba való regisztráció szükséges.

#### Grafikus felület és adatok összekötése

A JFace az Eclipse platfrom SWT grafikus könyvtárat kiegészítő MVC az az modell, nézet vezérlő megvalósítása. Lényegében ez lehetőséget biztosít arra, hogy az adott vezérlőhöz, adatokat kössük és bármelyik megváltozása esetén a másik fél követi ezt a módosítást. Többek között ilyen grafikus elem JFace keretein belül a fanézet, lista, táblázat is. Az eszközkönyvtár az SWT-ben lévő elemekre épül rá, a JFace segítségével megvalósított komponenseket az SWT által nyújtott interfészeken keresztül is elérhetőek, így nagy rugalmasságot biztosítva a két API közt.

## Jersey

A Jersey egy olyan server és kliens oldali komponenst nyújt, ami Rest szolgáltatások használatát teszi lehetővé. Kompatibilis implementációt nyújt a JAX-RS szabvánnyal, így más, a szabványt megvalósító komponensekkel is szabadon használható.

### Rest

Hálózati kommunikáció terén az egyik legelterjedtebb architektúra a Rest. A felépítés lényege hogy a kliensek kéréseket küldenek a szerver felé, majd az feldolgozza őket és válasz küld vissza. Tipikusan http protokollt használnak a Rest megvalósítására hisz az általa használt címzési struktúra, az URL nagy kifejező készséget biztosít a kérések megvalósítására. A kliens felé továbbítandó válaszok formátumára nincs megkötés azonban a legtöbbször a hatékonyság miatt valamilyen kompakt formátumot használnak például XML vagy JSON.

Az architektúra fő tulajdonságai:

* Kliens – szerver felépítés: A kliens és a szerver jól elkülöníthetőek egymástól, azok kommunikációja egy egységes interfész segítségével valósul meg.
* Állapotmentesség: A szerver nem tárolja el az egyes kliensek állapotait, nem veszi figyelembe azt, hogy régebben a klienssel mit kommunikált. Épp ezrét a kliensnek mindig pontosan specifikálni kell az ő állapotát a kérése mellett, hogy megfelelő választ kaphasson.
* Gyorsítótárazhatóság: A rendszer lehetőséget biztosíthat arra, hogy az egyes lekérések eredménye eltárolódjon majd újbóli lekérés esetén a régebbi eredmény kerüljön feldolgozásra. Ez nagyban növelheti a válaszidőt, azonban bizonyos esetekben elavult adatokkal való munkához vezethet, így fontos specifikálni, hogy mely adatok kerülhetnek gyórsítótárazás alá és melyek nem.
* Erőforrások kezelése: Minden egyes erőforrásnak egyértelmű, egyedik azonosítója és elérése van. A kliensnek lehetősége van ezen erőforrásokat lekérdezni, illetve azokon műveleteket végezni. A kliens kérései mindig önleírók, egyértelműen határozzák meg a szándékot.

A Rest műveletei a kliens oldaláról:

* Get: Egy erőforrás vagy azzal kapcsolatos adatok lekérése.
* Put: Egy elem tulajdonságainak megváltoztatására.
* Post: Új példány felvétele a rendszerbe.
* Delete: Elem törlésére szolgáló művelet.

A műveletek paraméterei HTTP esetén az elérni kívánt cím végén jelenik meg. Azonban e protokoll használata esetén a művelet megnevezése a kérés fejében, nem pedig a címében szerepel.

### JSON

Teljes nevén JavaScript Object Nation mint a nevéből is mutatkozik egy JavaScriptből átvett leíró szerkezet. Fő erőssége, hogy sikeresen egyesít olyan tulajdonságokat, mint az ember számára is könnyű olvashatóság, kompakt méret és magas ábrázoló készség. Az objektumok mellett még két struktúra ábrázolása is megoldott a formátumban. Ezek a lista, és a párosított kulcs-adat halmazok. Így lényegében alkalmas bármilyen programozási nyelvvel való használatra, hisz ezen adatszerkezetei segítségével nagy lefedést biztosít bármely nyelv elemeire. A formátumot főleg a web világában használják, adatok küldésére és fogadására viszonylag kis méretigénye és gyors feldolgozhatósága miatt.

# Áttekintés

Munkám során több technológiát és módszert is alkalmaztam, így most itt ezek kulcs pontjait mutatom be, majd a következő fejezetben pedig részletesen is áttekintem megoldásom menetét.

## Kliens – Szerver felépítése

Eclipse

Jersey Szerver

Lock tároló

.eiq és .binds  
fájlok

Projekt

.eiq és .binds  
fájlok

Projekt

.eiq és .binds  
fájlok

Projekt

lock  
fájlok

Projekt

Job

.eiq és .binds  
fájlok

Projekt

.eiq és .binds  
fájlok

Projekt

.eiq és .binds  
fájlok

Projekt

Projekt

Eclipse

Jersey Kliens

További kliensek

Pooling

Karbantartás

Munkám lényegében egy verzió kezelő rendszer kiegészítése, így adott volt, hogy kliens – szerver architektúrát kell alkalmaznom a feladat megoldására. A szerver megvalósításom tehát egy kiegészítésének tekinthető a verziókezelő rendszernek, logikusan ugyanazon a szerveren futnak. A munka folyamán csak is zárokat alkotó fájlokkal dolgozom, e végett fontos, hogy kliens oldalon a zárak fájljait kivegyük a más rendszer hatásköréből, és csak az én megoldásom kezelje ezeket a konfliktusok elkerülése végett.

Minkét oldalon adott volt Eclipse futtatókörnyezet az előzetes igények alapján, így ehhez kellett találnom egy kommunikációt megvalósító technológiát. Ehhez a Jersey-t használtam fel. Bár Rest tulajdonságai alapján nem feltétlen a legjobb választás, de az Eclipse OSGi rendszerébe való könnyű integrálhatósága végett mellette döntöttem. A technológiából fakadóan, hogy folyamatos adatcsere történhessen, poolingot az a folyamatos adat lekérdezést alkalmaztam. Ezt kliens oldalon a rendszer által nyújtott Job eszközzel valósítottam meg. Az általam definiált Job objektum feladata az, hogy időközönként ellenőrizze le az adott projektben lévő zárakat és hasonlítása össze őket a szerveren találhatóval, ha a szerveren talál frissebb zárat, akkor töltse azt le a felhasználónak. Ha helyben új zárunk van, azt csak kézzel tudjuk a többi felhasználó részére eljuttatni, ehhez publikálnunk kell az adott projekt zárait a szerver felé, majd a többi kliens letölti azt.

## Kliens részletesebben

Eclipse

EMF példány modell

Szerkesztő nézet

.eiq

.eiq

Incquery Feldolgozó

Példány modell zár

Ahhoz hogy modelleket zárolni tudjunk, a modell mellett szükségünk van a zárat alkotó fájlokra is. Ezek a projekt „lock” nevű könyvtárában helyezkednek el, a modellekkel megegyező néven de „.eiq” és „.binds” fájlkiterjesztéssel. Az eiq végű fájl a modellhez rendelhetők zár mintázatokat tárolja, a binds pedig az e minták változóihoz kötött értékeket. Ezen két fájl együttese adja ki a zárat illetve zárakat, bármely hiánya nélkül a zárolás nem lehetséges. A modellfájl bárhol lehet a projekten belül, egyedül a neve az, ami egyedivé teszi, és a zárakhoz köti azt.

Ahhoz hogy a zárolási funkciót eltudjuk érni, meg kell nyitnunk az egyik modellt egy EMF szerkesztőben, majd a rendszerem ezt érzékeli, és a felderíti a megfelelő lock állományokat. Mivel az EMF szerkesztőknek van egy olyan tulajdonságuk, hogy képesek mások számára kiszolgáltatni az általuk használt modellt, így minden adott lesz ahhoz, hogy a zárainkat alkalmazhassuk.

A már zárolt modell esetén a megállapítása annak, hogy az adott felhasználó művelet sérti-e egy lock alatt lévő objektumokat, az a következő képen történik:

1. a felhasználó a szerkesztőben végrehajtja a műveletet
2. a módosítás megjelenik a szerkesztő belső modelljében (ResourceSet)
3. az IncQuery ezt a módosítást érzékeli, és a változáshoz előre definiált trigger (Job) lefut
4. vizsgálat történik arra nézve, hogy az adott változás érint-e zár alatt lévő objektumot
5. a szerkesztő utasítás végrehajtási kezelője értesül arról, hogy a művelet jóváhagyható-e, vagy visszavonandó

Változástatás a szerkesztőben

ResourceSet változás

IncQuery Job

Ütközés detektálás

CommandStack értesítése

# Megvalósítás

## Server

A szerver egy OSGi keretrendszeren belüli bundle-ként került megvalósításra, így a kiszolgáló könnyen karbantartható illetve igény esetén később egyéb dinamikus szolgáltatásokkal is kiegészíthető majd. Mivel a szerver a Jersey segítségével lett megvalósítva, elindulása után készen várja a feldolgozandó HTTP üzeneteket. A kiszolgáló 3 féle utasításra tud reagálni, ezek pedig:

* projecthez fájl feltöltés
* adott projekt lock fájljainak lekérdezése
* adott projekt egy fájljának a letöltése

A továbbiakban e funkciókat mutatnám be részletesen.

### projecthez fájl feltöltés

Ez a funkció a szerver gyökerének címéből nézve „/upload” címén érhető el, mivel itt a művelet egy szerver felé fájl továbbítás, ezért post típusú üzenetnek határoztam meg. A művelet 3 paramétert vár:

* a projekt egyedi azonosítóját, amit a kliens határoz meg
* a feltöltendő fájl nevét
* a fájlt reprezentáló InputStream típusú objektumként

Végrehajtás során a tárolónak jelölt könyvtárban létrehoz egy könyvtárat a projekt azonosítójával, ha még nincs ilyen, majd elhelyezi, vagy ha már volt egy ugyanolyan nevű fájl akkor pedig felülírja.

### adott projekt lock fájlainak lekérdezése

A funkció a „/projectFiles” címen érhető el és mivel ez egy kliens felé visszaküldött üzenet így get típusú. A választ a kiszolgáló JSON formátumban adja meg, a tartalma pedig egymás után a paraméterben megadott azonosítóhoz tartozó project fájlnevei és a fájlok utolsó módosítási dátumai.

### adott projekt fájlának a letöltése

A kliensek a „/download” címen érhetik el ezt a get típusú funkciót. Paraméterezése megegyezik a feltöltési funkció paramétereivel. Feldolgozás után válaszként a paraméterben megjelölt elem File objektumként kerül vissza a kérő felé.

## Kliens

### projekt leíró

Ahhoz hogy megfelelően tudjuk használni a modult, a kívánt projektekhez hozzá kell adni a plug-in által biztosított projekt leírót. Ehhez a projekt almenüjében a Team>Locks> Add EMFGit nature opciót kell kiválasztani. Az opció kiválasztása után projekt leíró fájlban megjelenik a mi bejegyzésünk, illetve a projekt gyökerében létrejön egy fájl „.emfgit” névvel, ez a projekt egyedi azonosítóját tartalmazza.

Ahhoz hogy a menüpontok megfelelő helyen jelenjenek meg a plugins.xml fájlban, az „org.eclipse.ui.menus” típusú objektumot kellett deklarálni. Elsőre egy menuContribution típusút, aminek a locationURI paraméterében a következőnek kell szerepelnie „popup:org.eclipse.ui.popup.any?after=additions”. Majd ennek egy menu típusú gyermeket kellett megadni „Team” felírattal és „team.main” azonosítóval. Így az ez alá beszúrt menük mindig a Team API opcióinak megfelelő helyen jelennek meg az előbb említett API használata nélkül.

Miután a projekt leírójába belekerült a mi bejegyzésünk, ugyan már a projekt jelölve van számunkra, azonban mivel kollaboratív környezetben vagyunk így indokolt lenne az adott projekt egyediségét is biztosítani. Mivel a nature csak jelölő információkat engedélyez, ezért egy fájlt kellett létrehoznom, ami majd tárolja az egyedi kulcsot. A fájl neve azért kezdődik ’.’ karakterrel, mert így a rendszer automatikusan elrejti az Eclipse rendszerén belül, így nem zavarja a felhasználót. A fájlt tartalma pedig egy szimpla soros szöveg, amit a Java UUID eszközével generáltam. Ez az eszköz 128bites véletlen azonosítót képes generálni, így alkalmas projektek egyediségének a biztosítására.

Mivel az előbbi biztosítja a projekt egyediségét ahhoz, hogy egy csoport ugyan azon projekten tudjon dolgozni, az adott helyeken a megfelelő UUID-nek kell szerepelnie. Kollaboratív környezetben ideálisan ezt úgy tudják legkönnyebben megvalósítani, hogy valaki létrehozza a kívánt tartalmat, majd azt feltölti egy verziókezelő rendszerbe, a többiek pedig letöltik azt.

### zárak definiálás

A zárak használatához meg kell nyitni egy EMF szerkesztőt, ekkor a rendszer ezt érzékeli, majd feltölti az EMF Lock nevű nézet tartalmát. Ezt legkönnyebben az EMFGit nevű perspektívából tudjuk elérni. A nézet mindig a legutolsó aktív szerkesztő kontextusához tartozik.

A nézet a következő funkciókkal bír:

* Add Row: új, üres sort ad hozzá a listához ezzel
* Delete Row(s): az épp kijelölt sorokat távolítja
* Save: az aktív modellnek megfelelő nevű „.binds” fájlba kisorosítja a zárakat
* Activate: aktiválja a listában lévő zárakat a modellen
* Deactivate: eltávolítja a zárakat a modellről

A táblázat oszlopai:

* Name: a definiált zár neve, a rendszer ezzel hivatkozik rá, üres név esetén a zár figyelmen kívül kerül
* Pattern: a modellel megegyező nevű „.eiq” fájlban található IncQuery mintázatok listája, a kiválasztott alapján fog a zár működni
* Binds: Itt a kijelölt pattern változóira lehet megkötéseket tenni. Az elvárt formátum: patternVáltozó=érték. Az esetlegesen több változó kötése esetén, a megszorítások vesszővel választandóak el. Az itt megadott esetlegesen több változó mindig szűkíti a keresési teret, a változók és viszonyban szűrnek az eredményhalmazon.
* Enable: True/false értékeivel ki illetve bekapcsolhatjuk az adott zárat.

KÉP!!!!!!!

### zárak megvalósítása

#### Nézet detektálás

Ahhoz hogy a számunkra megfelelő szerkesztő jelenlétét érzékelni tudjuk több dologra is szükség van. Elsőre is fel kell iratkoznunk a workbench ablakában történő változásokra. Ezt a következő kódrészlettel lehet megvalósítani:

**final** IWorkbench workbench = PlatformUI.*getWorkbench*();

IWorkbenchPage page = workbench.getActiveWorkbenchWindow()

.getActivePage();

page.addPartListener(listener);

Az első sorban elkérjük a workbench objektumot a rendszertől majd az eszköz aktív oldalára egy IpartListener2 interfészt megvalósító objektummal iratkozunk fel. Az interfésznél a partActivated függvényt valósítjuk meg, a többire nincs szükségünk. A függvényen belül lényegében egy szűrést végzek, azt vizsgálom, hogy az aktívvá lett ablakrész megvalósítja-e az IEditingDomainProvider interfészt. Azért erre vizsgálok, mert az EMF modellek szerkesztőinél bár nem kötelező, de ajánlott ezt az interfészt megvalósítaniuk. Többek közt az alapértelmezett generált modell szerkesztő is megvalósítja ezt. A fő haszna ennek az interfésznek az, hogy a getEditingDomain függvényén keresztül elérhető a szerkesztő EditingDomain objektuma. Ezzel az objektummal ugyanahhoz a modellhez tudunk hozzáférni, mint a szerkesztő maga, így értesülni tudunk a szerkesztési folyamatokról még a modell elmentése előtt. Először a változtatások az EditingDomain által biztosított ResourceSet-en jelennek meg, majd később a változást eredményező művelet a szintén a domén által biztosított CommandStack objektumon jelenik meg. Az utasítás stack egy CommandStackListener típusú objektummal kerül megfigyelésre, illetve szerkesztésre. A ResourceSet objektumot pedig az IncQuery figyeli meg. Így lehetőség nyílik arra, hogy az IncQuery segítségével hatékonyan megfigyeljük a modell változásait, majd ha nem kívánt változás lép végbe, akkor azt az utasítás stacken visszavonhatjuk.

#### IncQuery

Ahhoz hogy az elérhessük a szolgáltatás a következőt kell tennünk a mi esetünkben:

EMFScope emfScope=**new** EMFScope(resourceSet) ;

engine= AdvancedIncQueryEngine.*createUnmanagedEngine*(emfScope);

schedulerFactory = Schedulers

.*getIQEngineSchedulerFactory*(engine);

executionSchema = ExecutionSchemas

.*createIncQueryExecutionSchema*(engine, schedulerFactory);

Először is egy EMFScope objektumot kell létrehozni, erre azért van szükség, mert az újabb IncQuery verziókban már nem EMF modelleken is lehetséges dolgozni, ez egyfajta adapterként működik. Az engine megkapja ezt paraméterként, majd létrehozzuk az ütemezőt, végül pedig az végrehajtási séma jön létre.

Ahhoz hogy fenti rendszer működésre tudjuk bírni szükségünk, van még egy query mintára is. Ezt jelen esetben mivel nem domain specifikus megoldást készítettem, futás időben olvassuk be, majd a rendszer készít belőle alkalmazható mintát. Normál esetben a query minta már fordítási időben elérhető, így ebből már fordítási időben elkészül az alkalmazható verzió. Az átalakításra ugye azért van szükség, mert a minta deklaratív módon van leírva, végrehajtani azonban csak imperatív kódot tudunk.

A minta beolvasása után le kell kezelnünk az általa lefedett adathalmaz változásait. Ehhez az IncQuery a Job objektumokat ajánlja fel. Egy jobnál lehetőségünk van meghatározni, hogy az adathalmazon milyen típusú változáshoz szeretnénk triggerelni. Ilyen változás például az új elem megjelenése, elem frissülése stb. Egy példa Job objektumra:

Job<IPatternMatch> jobAppeared = Jobs.*newStatelessJob*(

IncQueryActivationStateEnum.***APPEARED***,

**new** IMatchProcessor<IPatternMatch>() {

@Override

**public** **void** process(IPatternMatch match) {

handlePatternMatch(match);

}

});

A jobok definiálása után már csak egy szabály specifikációt kell létrehozni, ami a mintát és a hozzá kapcsolt Job objektumokat egyesíti, majd hozzá adni ezt a specifikációt a végrehajtási sémához. Ha az összes kívánt szabály specifikáció hozzáadtuk a sémához, fontos hogy a sémán mindenképp hívjuk meg a startUnscheduledExecution függvényt, ugyanis ennek hatására fognak helyesen, megfelelő időben lefutni később a jobok.

#### Lock

A zárakat egy általam létrehozott osztály írja le, 4 paraméterük van a zárak definiálása pontban bemutatottaknak megfelelően. Az osztály egy fontos függvénye az isMatchWithEventAtom ami egy IPatternMatch típusú objektumot vár, az előzőleg bemutatott Job bemenő paraméterének megfelelően. A zárak e függvényét a Job objektumok használják fel, minden egyes Job lefutásakor az összes definiált záron végigmegy, majd a zár visszaadja a függvény által, hogy őt fedi-e az IPatternMatch objektum vagy nem. A fedés eldöntése egyszerű, hisz az IPatternMatch is tartalmazza, hogy pontosan melyik IncQuery minta miatt futott le, és hogy melyik objektumot érinti. A zár esetén pedig pont ezeket az információkat kell megadni, a pattern és a binds mezőkben. Tehát ha a két objektum ezen értékei megegyeznek, akkor az utasítás stack kiértesítendő, ha pedig nincs egyezés, akkor hagyjuk lefutni az utasítást.

#### Helyes IncQuery minták deklarálása zárakhoz

Ahhoz hogy az IncQuery mintát feltudjuk használni zárakhoz egy fontos kitételnek teljesülnie, kell. A deklarált minta paramétereinek tartalmaznia kell azon objektumokat, amiken a zárakat szeretnénk elhelyezni, és azon paramétereiket is szerepeltetni kell, amelyeket zárolni szeretnénk. Egy rossz és helyes példán keresztül mutatnám be hogy mi ennek az oka.

Helyes példa:

**pattern** subsystem(Sys,SysID) {

**Subsystem**.**sysId**(Sys,SysID);

}

Rossz példa:

**pattern** subsystem2(SysID) {

**Subsystem**.**sysId**(\_Sys,SysID);

}

A két példa mindkét esetben a SysID-re illeszkedik paraméter szerint, azonban az első esetben az objektum is megvan nevezve a paraméter listán, aminek a SysID a tulajdonsága. Továbbá feltételezzük, hogy a két query-re ugyanolyan típusú jobok vannak ütemezve, appear,disapper és update trigger feltétellel. Ha lefuttatjuk ugyanazon az eredményhalmazon a két queryt ugyanazokat a SysID értékeket kapjuk meg a jobokon belül, azonban megfigyelhetjük, ha két Sys objektumnak ugyan az a SysID értéke, akkor a második esetben csak egyszer jelenik meg ez a SysID. Ez önmagában még nem jelent problémát, azonban ha például megjelenik egy ugyanolyan értékű SysID-vel rendelkező objektum, ami már szerepelt régebben az eredményhalmazban, akkor a második esetben egyik job sem fut le, míg az első esetben lefut az appear Job. A jelenség oka az, hogy míg az első esetben az új eredménynek a Sys mindenképp egyediséget biztosít, hisz mint objektum egyedi és biztos nem szerepelt eddig az eredményhalmazban, míg a második esetben a SysID-re ez nem teljesül.

A problémát tovább fokozza, ha módosítás történik az eredményhalmazon belül. Az IncQuery kezdetben tartalmazott egy DeltaMonitor nevű eszközt, ami képes volt a műveletek által okozott változásokat megmutatni. Azonban mivel nagyobb modellek esetén a változást kiszámolni nagyon erőforrás igényes is lehet, így az API ezen részét megszüntették. Így azonban nincs eszköz arra nézve, hogy egy művelet miből mibe alakította át az eredményhalmazt. A frissítéseket kezelő job pedig csak már az új megváltozott eredményt kapja meg, a régi eredményhez nem férhet hozzá. Így nem tudjuk eldönteni, hogy a műveletet vissza kell-e hívni az utasítás stackről vagy nem. Erre a problémára szintén megoldást jelent az első verzió, hisz a másodikkal szemben, ha egy módosítás történik az eredményhalmazon, a második esetében csak az update job fut le, míg az első esetben az update mellett az appear és a disapper is biztosan lefut. Ennek az oka szintén az, hogy a tulajdonos objektum a query eredményhalmazába egyediséget biztosít. Jelen esetben pedig az apper és a disapper job azon értékeket fogja megkapni, amik a módosítási művelet során eltűntek vagy megjelentek, így a műveltről eldönthető hogy érint-e zárolt objektumokat vagy nem.

### beállítások

Ahhoz hogy a modulunkhoz beállításokat is tudjunk kezelni a már bemutatott preference eszközöket kell használni. A kliens 3 beállítást kezel. Ezek a zárakat kezelő szerver címe, a pooling engedélyezése illetve a pooling ütemezésének a ciklusa. Ahhoz hogy a rendszer újraindítása nélkül az adott beállítások életbe lépjenek egy konvenciót kell alkalmazni. A beállításokon aktvált IPropertyChangeListener figyelő változás esetén módosít egy központinak jelölt értéket, majd azokon a helyeken ahol ezen beállítás értéke felhasználandó innen hivatkozva éri el az adatot, így mindig a friss értékkel dolgozhat az adott folyamat.

### Jelölök használata

A modul működése során 3 féle jelzést továbbítok a felhasználó felé.

Ezek:

* kapcsolat nem létesíthető a szerverrel
* szerverre történő publikáció hiba
* zár aktív

Megvalósításuk mindegyik esetben hasonló, a hibának megfelelő üzentet társítok a jelölőhöz, majd warning az az figyelmeztető tulajdonságot adok meg neki. Mivel minden jelölőt egy erőforráshoz kell csatolni, így univerzálisan a platform statikusan elérhető workspace objektumához társítom.

### Jersey használata

#### letöltés – feltöltés

Ahhoz a Jersey-t le vagy letöltéshez megfelelően tudjuk használni a következő kódrészletet kell hasznáni:

Client client=Activator.*getClient*();

String url=Preferences.*ServerAddress*;

WebResource resource = client.resource(url).path("/download").queryParam("projectID", projectID)

.queryParam("filename", fileName);

File response=resource.accept(MediaType.***APPLICATION\_OCTET\_STREAM***).get(File.**class**);

Az első sorban elkérem a központosított kliens objektumot a modul Acitvator osztályától, majd pedig a szerver elérhetőségét a beállításokat megvalósító osztálytól. A következő sorban megadom az elérni kívánt erőforrást és felparaméterezem. Ezek után a válasz fogadása jön, itt megadom, hogy fájlt fogok válaszként visszakapni. A feltöltés esetén a WebResource objektumig megegyezik a használat, azonban utána a következő műveleteket kell végrehajtani:

Builder post = resource.entity(fileToUpload,MediaType.***APPLICATION\_OCTET\_STREAM*** );

post.post();

Mint látszik, itt egy post művelet hajtódik végre, jelölésre került a cím az az az erőforrás, a feltöltendő fájl, illetve hogy milyen típusú objektumot töltünk fel.

#### Pooling

A szerver folyamatos ellenőrzése a következő Eclipse Job által történik:

*job*=**new** Job("Check locks on server") {

@Override

**protected** IStatus run(IProgressMonitor monitor) {

**try**{

monitor.beginTask("Check locks ", 1);

refreshActiveProjectsLocks();

monitor.done();

**return** Status.***OK\_STATUS***;

}**finally**{

schedule(Preferences.*PoolingInterval*);

}

}

};

*job*.setPriority(Job.***SHORT***);

*job*.setSystem(**true**);

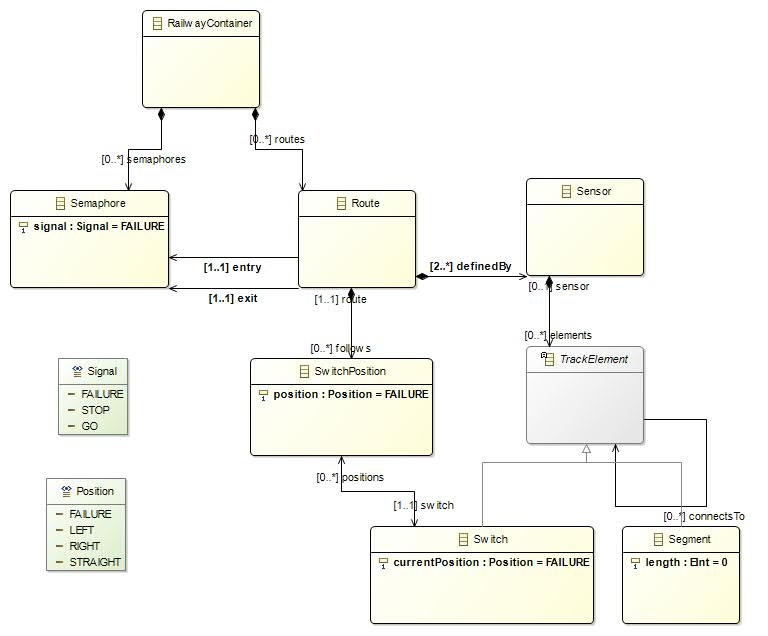
Mint már említettem lényegében egy okosabb szálkezelési eszközt kapunk. Lehetőségünk van jelezni a folyamat állapotát, ezt a monitor objektumon keresztül tehetjük meg, az ütemezést pedig a finally ágban végrehajtott schedule függvény meghívásával ajánlott végrehajtani. Mivel az itt végrehajtott feladat egy viszonylag rövid ideig tartó feladat, így alacsony prioritásra állítottam be és rendszer feladatként állítottam be. Ez azt jelenti, hogy a felhasználó felé a grafikus felületen nem jelenik meg értesítés a munka lefolyásáról.

# Kiértékelés

Az elkészített megoldás mérése

https://github.com/FTSRG/trainbenchmark

Ahhoz hogy az elkészített megoldásom használhatóságát felmérjem méréseket végeztem. Ehhez a tanszéken fejlesztett Train Benchmark framework nevű eszközt használtam fel. Ez az eszköz képes egy vasúti rendszer modelljét legenerálni majd felmérni a modellen végzet műveletek idejét különböző háttértár technológiák esetén. Jelen esetén én csak a modell generáláshoz használtam az eszközt, a méréseket saját magam végeztem el.



1. ábra Generált modell felépítése

A modell generálása esetén a „Repair” nevű opciót használtam, ez a következő paramétereket használta fel a modell elkészítése során:

maxSegments = 5;

maxRoutes = 20 \* SIZE;

maxSwitchPositions = 20;

maxSensors = 10;

posLengthErrorPercent = 10;

switchSensorErrorPercent = 4;

routeSensorErrorPercent = 10;

semaphoreNeighborErrorPercent = 8;

switchSetErrorPercent = 10;

connectedSegmentsErrorPercent = 5;

A felhasznált modellek utjaik száma és fizikai mérete:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SIZE (utak száma) | 50 | 100 | 200 | 500 |
| Méret (MB) | 40 | 78 | 155 | 382 |

A mérés során felhasznál zárak mintái:

**pattern** route(R, RID) {

**Route**.**id**(R, RID);

}

**pattern** sensor(S, SID) {

**Sensor**.**id**(S, SID);

}

A SIZE paraméter 50,100,200 és 500as értékek mellett vizsgáltam meg. Kisebb értékek esetén a Java rendszer nem determinisztikus szemétgyűjtője adta mérési hibák miatt értelmetlen lett volna a mérés, nagyobb modellek használata esetén pedig már kevés volt a memóriám. A zárakat paraméter lekötés nélkül használtam fel, így a route nevű minta viszonylag kicsi, pár ezeres számú eredményhalmazzal dolgozott, míg a sensor pattern az előző halmaz kb. negyvenszeres számosságával volt, így már a legkisebb vizsgált esetben is 44 ezer volt az eredményeinek a száma. A méréseim során a modul használatához szükséges memóriát, a zárak aktiváláshoz szükséges időt és zárak használata közbeni törlés műveletet figyeltem meg.

A méréseimet a következő környezetben végeztem:

* CPU: i5-3210M – 2 mag, 4 szál, 2.5GHz
* RAM: 8GB
* Háttértár: Samsung 840 Evo 250GB
* Operációs rendszer: Windows 8.1 Pro 64bit
* Runtime Eclipse paraméterei: -Xms40m -Xmx6144m

Az Eclipse paramétereinél az –Xms kapcsoló a rendszer kezdetleges memória lefoglalást, az –Xmx pedig a lehetséges maximális memóriát jelenti. Mivel a rendszer alapértelmezésként 40MB és 512MB értékeket használ, ez nagyban korlátozta volna az alkalmazás teljesítményét, így a konfigurációm által biztosított maximumot állítottam be neki.

Memória igény

E mérés során azt vizsgáltam, meg hogy az Eclipse mennyi memóriát foglal le magának, ha a modelleket az EMF alapértelemzett szerkesztőjével nyitom meg, majd a zárak elhelyezése utáni állapotot is megvizsgáltam. Az előbb bemutatott zárak közül mindkettőt aktiváltam, a méréseket minden egyes modell esetén egy újonnan megnyitott Eclipseben vizsgáltam meg. Az eredmények mindig a folyamat végeztével beálló értékeket jelentik. A modellek mellett más nem volt megnyitva, más munkafolyamat nem történt. Az üres, szerkesztői nézetet nem tartalmazó Eclipse kb. 295MB memóriát igényelt, mindig ebben az üres nézetben történt az egyetlen aktuális modell megnyitása.

Az eredmények alapján azt láthatjuk a szerkesztő és a zárolt modell memória igénye a modell nagyságával exponenciálisan nő. A mérést elvégeztem több esetben is zárak alkalmazása nélkül, pusztán csak az IncQuery minta illeszkedésre, ekkor nem történt lényegi változás az eredmények terén. Az 500as modell esetében bár még a zárak elhelyezése sikeres volt, ezek után azonban a rendszer már nem volt működőképes, nem reagált semmire, így a mérés tovább részéből ezt a modellt már kihagyom.

Zárak aktiválásához szükséges idő

A következő mérésemben azt vizsgáltam, meg hogy a különböző méretű modelleken a zárak elhelyezése mennyi időt vesz igénybe. Ezt a végrehajtási séma aktiválása, az az a startUnscheduledExecution függvény meghívása előtti és utáni rendszeridőpontok differenciája alapján mértem. Ez azért megfelelő pont a mérésre, mert a függvény meghívására az összes, mintára illeszkedő objektum regisztrációra kerül, és a neki megfelelő appear Job meghívódik. Külön teszteltem azt az esetet, amikor csak a route, és amikor a route és sensor zárak együtt aktiválódtak. Így lehetőségem van megfigyelni hogyan, alakulnak az eredmények egy kis és egy nagy minta illeszkedési eredményhalmaz esetén.

Figyelembe véve, hogy a generált modell lényegében lineáris, tehát nagyobb számú út esetén a modell többi része is ezzel párhuzamosan nő. Így kijelenthető hogy a kezdeti zár lerakási idő nem a modell őszméretétől, ha nem a modellben található illeszkedések számától függ. Akár csak a memória használata esetén, itt is exponenciális növekedést figyelhetünk meg, az illeszkedő objektumok száma exponenciálisan növeli a zárak lehelyezései idejét. Arra hogy nem a modellméret, ha nem csak az illeszkedő objektumok száma okozza ezt, arra az magyarázat, hogy az IncQuery már a memóriában lévő modellen dolgozik, ezen megvizsgálni egy illeszkedést szinte jelentéktelen költségű, azonban ha illeszkedést talált a rendszert azt nyílván kell tartania már, ami mint látszik, nagy költséggel jár.

Művelet végrehajtása zárolt modellen

Kérdés volt még hogy már zárolt modell esetén egy zárat sértő művelet feloldása mennyi időt vesz igénybe. Ennek a mérésére az EMF Notification szolgáltatását használtam fel. Ez az eszköz még az IncQuery előtt képes jelezni egy modell változását, így lényegében az itt jelzett változást tekinthetjük a művelet kezdetének. A művelet végét az utasítás stackre való eljutásig mértem.

A méréseket végrehajtottam csak a route illetve a route és sensor pattern páros együttes használatával is. A route pattern alkalamzása esetén egy véletlen kiválasztott Route elemet töröltem, a minta páros esetén egy Route és egy Sensor objektum törlését is megvizsgáltam. A kapott eredmények alapján a már zárolt modellen a műveletek végrehajtási ideje minimális.

Ezek után mindegyik modell esetén 100, 200 és 500 véletlenszerű elem törlését is megkíséreltem úgy, hogy legyen közöttük mindenképp Route és Sensor típusú elem. Az alkalmazott záraktól és modell mérettől függetlenül, 100, 200 és 450 ms körül értékeket kaptam. Tehát a műveletek végrehajtása jól skálázódik, lineáris.

Összegzés

Mint kiderült a módszer hátrányai a hatalmas memória igény, és a modellek zárolási ideje. A hatalmas memória igényt ugyan minimálisan lehetne csökkenteni egy másik szerkesztő, vagy testreszabott Eclipse használatával, azonban a zárak alkalmazása így is jelentős terhelést jelent. A zárak elhelyezésének az ideje annak ellenére, hogy egy memóriában történő művelet, igen hosszú ideig is eltarthat nagy illeszkedési halmaz esetén. Sajnos ezt tovább csökkenteni nem igen lehet. Másik oldalról, viszont ha a rendszerünk már egyszer elindult, a rajta végzett műveletek folyamatosak, elvárható időn belül végrehajtódnak.

# Kapcsolódó munkák

Mint minden mérnöki munka esetén, itt is érdemes lehet megnézni, felkutatni már esetlegesen meglévő megoldásokat az adott problémára.

## Zárak megvalósítása általában

A zárakat több minőségben is vizsgálhatjuk, ilyen főbb tulajdonságok a művelet zársértésének detektálása, granularitás, zárak egymásra való hatásai, illetve hierarchiákon végzett zárolások mikéntjei.

Azt, hogy egy adott művelet egy zárat sértett-e, kétféleképpen detektálhatjuk, vagy a művelet végrehajtása előtt vagy után detektáljuk az esetleges zársértést. Az első eset a pesszimista, hisz feltételezzük, hogy zár van az adott objektumon, a második eset az optimista eset, megpróbáljuk a műveletet végrehajtani és csak aztán detektálunk.

Granularitás szerint a zárak sokrétűek lehetnek. Verziókezelő rendszerek vagy fájlrendszer esetén a zár a fájlra szól, vagy épp egy adott könyvtárra, de egy modell szerkesztőben már például nem feltétlen. Ott már tipikusan a modell elemeire, részeire helyezhetünk el zárakat, így egy komolyabb zárolási logikát alkalmazva.

Zárak esetén gyakori az, hogy az adott műveleteket kiéheztetik, az erőforrások kölcsönösen blokkolják. Így megoldást kell találni arra nézve, hogy a rendszer tovább működhessen. Erre tipikusan a relációs adatbázisok területén láthatunk megoldásokat, ilyen például a kétfázisú zárolás is.

Fontos kérdés még, hogy ha egy adatszerkezetben zárt helyezünk egy csomópontra, akkor mi történik a leszármazottaival, kapcsolódó elemeivel. Itt szintén az relációs adatbázisok területéről példa lehet a fa vagy a figyelmeztető protokoll megvalósítása.

## CDO

A CDO az az Connected Data Objects egy olyan megoldás, amely képes az EMF modellek tárolására magas testreszabhatóság mellett. Egyik fő tulajdonsága, hogy az adatok tárolására számos lehetőségünk van. A rendszer képes használni SQL, NoSQL, fájl vagy akár memória (Ram) alapú háttértárat is. Felépítéséből adódóan ugyan lehetőséget biztosít kollaboratív munkára, de az esetleg konfliktusok megoldásához tipikusan külső eszközöket kell használnunk pl EMF Diff/Merge.

## EMFstore

Az EMFStore egy olyan tároló EMF modellek számára, ami képes verzió követni az egyes modellek változásait. Létjogosultsága nyilvánvaló, hisz a többi verziókövető rendszer kizárólag szöveg alapú, ezáltal nem tudnak megfelelően kezelni olyan fájlokat, amikben szöveg nem program kódot, ha nem egy modellt testesít meg. Ezáltal a rendszer hatékony megoldást tud nyújtani nem csak verzió kezelésre, ha nem kollaboratív munkára is, hisz a modellek összefésülése illetve konfliktus feloldása könnyen megoldható.

## EMFstore és CDO összehasonlítása

### Adattárolás

A CDO számos tárolási technológiát fel tud használni egészen az egyszerű fájl alapútól a komplexebb adatbázisokig. Előnye abban rejlik, hogy a tárolási mechanizmusa szabadon megvalósítható adapterek útján, így akár eddig nem létező technikákkal is kiegészíthető. Fontos megemlíteni, hogy a rendszer támogatja a lazy loading az az a késleltetett betöltést is. Ennek a haszna főleg akkor jelentkezik, amikor egy nagyobb modellt szeretnék használni. A technika lehetőséget biztosít arra, hogy ne az egész modellt olvassuk be a háttértárról, ha nem csak egy részét, épp amit használunk.

http://en.wikipedia.org/wiki/XML\_Metadata\_Interchange#Integration\_of\_industry\_standards

Ezzel szemben az EMFStore csak is kizárólag XMI fájl alapú perzisztenciát támogat. Az XMI lényegében 3 technológiát egyesít, ezek pedig az XML, a fájl struktúráját rögzíti, az UML és MOF pedig az ábrázolt adatok formátumát adja meg. Bár a formátum akár egy ember számára is jól olvasható struktúrához vezet, azonban a gép számára ez nehézkes feldolgozást jelent. A CDO lazy loading megoldásához képest, ahhoz hogy itt dolgozni tudjunk egy modellel, a háttértárolóról teljes egészen be kell azt olvasni, és a memóriában kell tárolni. Így a modellhez való első hozzáférés nehézkes lehet és nagy memória igény is jelentkezhet egy komplexebb modell esetén. Cserébe viszont az első interakció után a modellen végzett műveletek késleltetése kicsi lesz, hisz a memóriában lévő modellen történnek a műveletek.

### Kollaboráció

<http://www.slideshare.net/koegel/emfstore-a-repository-for-emf-models>

A CDO mint egy központosított tároló, elérhetősége miatt egyértelmű hogy alkalmas lehet megosztott, több felhasználót igénylő munkák kiszolgálására. Azonban mivel, komolyabb modell műveletek, mint pl a merge nem támogat így egyértelműen csak online munka esetén lehet hatékony. Bár a rendszer támogat offline módot is, ekkor azonban csak összegyűjti a változtatásokat, majd amikor a kiszolgáló elérhető lesz, végrehajtja azokat. Mivel ez a mód gyakran konfliktus kezelést kíván, külső eszközöket kell bevonni amik, nehezítik a munkát.

Az EMFStore ezzel szemben a több felhasználós környezet igényeit figyelembe véve készült. Műveletei az SVN rendszerhez hasonlóan működnek, ezáltal a mindig offline történnek. Ennek ellenére a rendszer képes online üzenetek közlésére a modell változásairól. Figyelembe véve hogy verziókövetést, magas fokú modell összefésülést, probléma megoldásokat is tartalmaz, egyértelműen hogy kollaboratív munkára ez a megoldás hatékonyabb biztosított eszközei által.

### Jogok kezelése, zárolás

Felhasználók azonosítása terén, a CDO több lehetőséget is kínál. Lehetőségünk van egyszerűen akár egy fájlban is megadni a felhasználókat, illetve komolyabb LDAP (Lightweight Directory Access Protocol) is támogatott, ami már komolyabb strukturáltabb felhasználó kezelést tesz lehetővé. Az objektumok zárolása esetén is több lehetőség van, objektumokat akár egyesével is zárolhatunk, de lehetőségünk van magasabb szintű szűrő kifejezéseket is deklarálni.

Az EMFStore ezzel szemben szegényesebb funkcionalitással bír. A felhasználók azonosításánál külső megoldást nem lehet alkalmazni, és lényegében lock funkcióval sem rendelkezik a rendszer. Az egyetlen, amit tehetünk, hogy az adott projekt estén jogot adunk felhasználónak vagy egy felhasználók csoportjának, és ezzel kontrollálhatjuk a hozzáférésüket.

### Konfliktusok megoldása

Mint azt az eddigiekben már említettem, a CDO esetén külső megoldásokhoz kell folyamodni tipikusan, míg az EMFStore magas szintű összefésülést, modell commit pontra való visszaállítást tesz lehetővé.

### Skálázhatóság

Mivel magas háttértár testreszabhatósággal rendelkezik a CDO ezért skálázhatósága nagyban függ a felhasznált technológiától. Mivel főként SQL alapú adatbázist használata az elterjedt így az ilyen rendszerekre alapú megoldást veszem most figyelembe. Tekintettel arra, hogy általában egy SQL adatbázis jól skálázódik nagyobb adatmennyiség esetén, számos helyen gyorsító tárazást lehet alkalmazni a műveletek során, és hogy a CDO lazy loading megoldást is támogatja, egy viszonylag jól skálázható rendszer kaphatunk.

Ellentétben ezzel, az EMFStore pusztán XMI alapú tárolásra ad lehetőséget, és a modellen való művelet esetén mindenképp be kell tölteni a teljest modell a memóriába. Ez az első művelet végrehajtása előtt sok időt tehet ki, azonban később a műveleti idők viszonylag kicsik lesznek. A megoldás hátránya hogy egy méret fölött biztosan kifutunk a memóriából, és a rendszer szinte használhatatlanná válik, hisz a java szemét gyűjtője folyamatosan „szenvedni” fog.

### Összefoglalás

Figyelembe véve a két eszköz tulajdonságait, mindkettőnek vannak célterületet meghatározó tulajdonságaik, amik egyértelműen eldöntik, hogy mire is használhatóak. Az EMFStore esetében egyértelmű hogy a készítők célja a verziókezelés megvalósítása volt, azonban más funkcióval nem igen rendelkezik a rendszer. Ezzel szemben az CDO sokrétű szolgáltatásai, mint a felhasználó kezelés, zárak megvalósítása, rugalmas háttértár kezelés véleményem szerint szélesebb igényeket fed le és szélesebb réteg által használt megoldás, míg az EMFStore egy szűk réteg eszköze lehet csak.

## MetaEdit

Az iparban egy igen népszerű és sokfunkcionalitású modellező eszköz a MetaCase vállalat MetaEdit szoftvercsomagja. A szoftver népszerűségének az igen szerteágazó funkcionalitási sokszínűségének köszönheti Ilyen fontosabb funkciók például:

* modellek központi szerverről való elérése
* modellből kód és eszközök generálása
* validáció
* sok felhasználós kollaboratív környezet biztosítása
* modellek verziókövetése
* magas szintű integrálhatóság szinte bármilyen nyelvvel és környezettel

Összehasonlítva Az Eclipse által nyújtott eszközökkel az első különbség, hogy ez egy fizetős megoldás, míg az előbbiek mind ingyenesek és nyíltak. Szolgáltatások terén elég jól lefedi a CDO és EMFStore nyújtotta lehetőségeket, azonban néhány, tipikusan testreszabhatósági tulajdonságokban csorbákat szenved.

### Modell szerkesztés

http://www.metacase.com/cases/index.html#reviews

A munka folyamán jól elkülönülő meta modell (Workbench) és példány modell szerkesztővel (Moduler) rendelkezik. Ezen szerkesztők specifikusan csak az egyik feladat elvégzésére vannak kiélezve, így jóval hatékonyabb módon lehet egy adott problémát megoldani. Állításuk szerint nem csak hogy könnyebben tanulható az ő megoldásuk, mint az Eclipse eszközei, de azonos rendszer elkészítése akár 10x is gyorsabb lehet az ő eszközük segítségével.

Annak ellenére, hogy a rendszer nagy integrálhatóságot biztosít számos fejlesztői környezetbe, fejlesztés közben lényegében a külső eszközök használata problémássá válhat annak ellenére, hogy egy jól definiált kiegészítő interfésszel rendelkezik. Az eszközök olyan szintű integrálhatóságát, mint az Eclipse rendszere nem tudja megvalósítani. Épp ezért egy komplexebb, szerteágazóbb eszköztárat igénylő fejlesztés során az Eclipse biztosan kényelmesebb és hatékonyabb munkára képes.

### Kollaboratív munka támogatása

Rendszerük lényegében a bemutatott két Eclipse eszköz tulajdonságait ötvözi. Képes verziókezelésre, bár ennek a granularitása egész más felfogású, mint az EMFStore esetében. A leírások alapján a változásokról pillanatképeket készít, és ezekre lehet esetlegesen visszatérni. Így bár szerényebb lehetőségekkel, de megvalósítja a verziókezelést. Azonban magas fokú zárolási lehetőségekkel is bír, külön objektumokat és struktúrák zárolását is engedélyezi, felhasználóknak pedig széles körű jogok szabhatók meg. Így egyedüli hátránya ebben az aspektusa az offline munka lehetősége.

### integrálhatóság, más környezetekkel való kapcsolat

Az eszköz bár rendelkezik kiegészítő interfésszel, ám ez közel sem közelíti meg az Eclipse funkcionalitását. Ilyen funkciók lehetnek például más aspektusú szerkesztők használata egy időben ugyanarra a modellre, vagy az adatokon végzett több lépcsős feldolgozás nehézkes lehet. Másik oldalról viszont, az Eclipse eszközei csak a Java környezetre redukálódnak, míg a MetaEdit lényegében bármely programozási nyelvvel együtt tud működni, ez pedig igen nagy előnyt jelenthet egy olyan környezetben ahol az adott problémához a legjobb programozási nyelvet, rendszert használják.

# Továbbfejlesztési irányok

Az elkészített alkalmazásom több lehetséges fejlesztési ponttal is rendelkezik, ezek közül néhány fontosabb:

* Komplexebb lock definíciók lehetősége sokat tudna javítani az eszköz használhatóságán. Bár az IncQuery minta illeszkedési leírása igen szerteágazó funkcionalitással bír, azonban a paraméterek megadásánál semmilyen funkcionalitás nincs a további paraméter lekötésének lehetőségén túl. Itt mindenképp lehetne bővíteni a funkcionalitást logikai műveletekkel, esetleg pár függvénnyel. Távlati célként pedig akár egy külön zárak definiálására szolgáló nyelv megalkotása lenne a legideálisabb, például az Eclipse Xtext eszközével.
* Zárat szegű műveletek esetén jelenleg a felhasználó érdemi visszajelzést nem kap arról, hogy nem sikerült a kívánt művelete, csak azt tapasztalhatja, hogy minden maradt a régiben. Itt egy felhasználóbarát megoldás lenne, ha tudatnánk a felhasználóval, hogy pontosan miért nem tudja az akcióját végrehajtani, melyik zár vagy zárak akadályozzák őt.
* Miközben a felhasználó dolgozik az adott modellen kollaboratív környezetben sokat segítene, ha a rendszer magától tudná zárolni a modell egyes részeit pusztán a felhasználó akció alapján. Így sok konfliktus lenne elkerülhető illetve hatékonyabb munkát eredményezne, hisz az adott művelet végrehajtása előtt nem kellene a zár elhelyezésével foglalkozni.
* Bár a feladat során használt Jersey REST megoldása funkcionalitásban elég volt a követelményeknek, ha a program funkcionalitása tovább nőne, bizonyossággal a technika korlátaiba ütköznénk. A REST alapvetően egy tartalom szolgáltatási eszköz, valós idejű vagy titkosított adatátvitel már nagyon nehézkesen vagy egyáltalán nem lenne megvalósítható segítségével. Jelenleg talán a legjobb eszköz a WebSocket lehetne, hisz egyaránt nyújt magas szintű szolgáltatások, de alacsony szintű adat átvitelre és lehetőséget biztosít így a kommunikációt teljesen igényünk szerint alakíthatnánk.
* Az Eclipse rendelkezik egy úgynevezett Team API-val. Ez az interfész lehetőséget biztosít arra, hogy kollaboratív eszközt valósítsunk meg platformon belül. Ezt használja többek között a Subversive (SVN) és EGit (Git) eszköz is. Az elkészített alkalmazásom használhatósága nagyban növekedett volna, ha az SVN vagy Git rendszerekkel kombinálni tudtam volna úgy, hogy ezt az interfészt megfelelően implementálom. Azonban e feladat komplexitása miatt erre nálam nem került sor.

# Összegzés

Irodalomjegyzék

1. Levendovszky, J., Jereb, L., Elek, Zs., Vesztergombi, Gy.: Adaptive statistical algorithms in network reliability analysis, Performance Evaluation - Elsevier, Vol. 48, 2002, pp. 225-236
2. National Istruments: LabVIEW grafikus fejlesztői környezet leírása, <http://www.ni.com/> (2010. nov.)
3. Fowler, M.: UML Distilled, 3rd edition, ISBN 0-321-19368-7, Addison-Wesley, 2004
4. Wikipedia: Evaluation strategy, <http://en.wikipedia.org/wiki/Evaluation_strategy> (revision 18:11, 31 July 2012)

Függelék