AGENTFRAME dokumentáció

Készítette:

Kovács Dániel László
dkovacs@mit.bme.hu

Méréstechnika és Információs Rendszerek Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem

2008, február.

Tartalomjegyzék

1 Bevezető ... 3
2 Az AGENTFRAME ... 7
2.1 Szoftverek és dokumentációk ... 7
2.2 JADE telepítése és működ(tet)ése ... 7
2.3 Az AGENTFRAME csomag alapja: KönyvKereskedés ... 9
2.4 JAXB (Java Architecture for Xml Binding) ... 12
2.5 Az AGENTFRAME „csomag” ... 18
2.5.1 AGENTFRAME üzembe helyezése ... 19
2.5.2 AGENTFRAME használata ... 22
3 Összefoglalás ... 33
4 Függelék ... 34
4.1 Eclipse és JADE ... 34
4.1.1 Telepítés ... 34
4.1.2 Beállítások ... 35
4.1.3 FFF ... 37

1 Bevezető

Ez egy belső használatra szánt anyag, amely elsősorban a BME Méréstechnika és Információs Rendszerek tanszékén oktatásra kerülő „*Elosztott Intelligens Rendszerek*" c. B.Sc. szakiránylabor méréseinek kidolgozásában résztvevő tanszéki kollégák számára készült.

Mi az AGENTFRAME? Röviden: egy előre elkészített szoftver-keret, melyet a labor méréseit kidolgozó kollégák kiindulásképp használhatnak saját egyéni szoftvermegoldásaik kialakításához. Bővebben: a labor méréseinek hátteréül a **JADE** (**J**ava **A**gent **DE**velopment Framework; http://jade.tilab.com) keretrendszer szolgál, ami a FIPA (**F**oundation for **I**ntelligent **P**hysical **A**gents; http://www.fipa.org) elnevezésű de facto ágens-szabvány egy referencia implementációja. Az AGENTFRAME minta-alkalmazás a JADE keretrendszerben.

A labor során tehát ezt a keretrendszert használjuk arra, hogy a mérések szoftveres hátterét kialakítsuk. A JADE, mint ahogy nevéből is látszik, **Java** alapú szoftver (http://java.sun.com). Teljeskörű használatához szükséges a megfelelő **JDK** (**J**ava SE Software **D**evelopment **K**it; http://java.sun.com/javase/downloads), és egyéb szoftverfejlesztési csomagok telepítése. Ennek menetéről az anyag megfelelő pontján még bővebben szót ejtünk. Most azonban térjünk még kicsit vissza az AGENTFRAME magyarázatára.

Az AGENTFRAME koncepciója a labort megelőző megbeszélések, és elektronikus levelezés során alakult ki. Végső soron a következő *követelmény specifikációhoz* jutottunk (idézet a *2008. január 8, 14:27*-kor kelt körlevélből).

- *Az AgentFrame egy Java osztály, pontosabban egy JADE-es ágens.*
- *Az ágens – adott platformon való létrehozását követően – beregisztrál a DF (Directory Faciliator) ágensnél. Ennek során kulcsszavak listájának formájában elérhetővé, és kereshetővé teszi a szolgáltatásait (pl. sebész, reumatológus, ápoló, mediátor, bróker, belgyógyász...).*
- *Rendelkezésre fog állni egy opcionális kódrészlet, melynek alkalmas átírásával az ágens képessé válhat adott szolgáltatásokat nyújtó ágensek listájának lekérdezésére (a DF-től).*
- *Az említett kódrészlet adott esetben az ágens GUI (Graphical User Interface) grafikus felületén át is aktiválható lehet (e funkcionalitás GUI-hoz kötése az ágens programozójának feladata). A keretet az AgentFrame fogja biztosítani.*
- *A grafikus felületet egy adott szintaxisú (sémájú) XML írja majd le. Az XML-t nyilván az ágens programozója hozzá létre. Tehát az ágens programozója az ágens GUI-ját egy XML megadásával definiálja majd. Ebben azonban nem feltétlen szerepel a GUI által lefedett funkcionalitás, csak esetleg pusztán maga a megjelenés. Definiálhatók lesznek tehát gombok, szöveges ki/beviteli mezők, stb, melyekbe szám, szöveg, dátum, vagy egyéb atomi típusú objektum kerülhet. Létrehozható lesz például egy olyan GUI, amelyen 1 gomb és 2 szöveges beviteli mező lesz látható. A gomb megnyomásának hatására az ágens az egyik szöveges beviteli mezőben hivatkozott (pl. adott nevű) ágens számára elküldi a másik szöveges beviteli mezőben szereplő string-et.*
- *Az ágenseknek definiáljuk a típusát (orvos, ápoló, stb).*
- *Az ágensek típusához egy-egy XML séma is tartozik majd, amely – túl az ágens által felkínált szolgáltatásokon – meghatározza, hogy az ágens milyen (tartalom)nyelven kommunikál. Arról van tehát szó, hogy az ágensek között cikázó ACL (Agent*

Communication Language) üzenetek tartalma (Content) XML szintaxisú. Tehát XML sémával lehet majd definiálni, hogy éppenséggel milyen XML szintaxissal kommunikál egy adott típusú ágens. Ez a séma nyilván az ágens típusától függ, ami pedig mérésről-mérésre változhat. Ezért, ahogyan az ágensek típusának meghatározása, úgy a kommunikációs (XML) séma meghatározása is a mérést kidolgozó kolléga feladata lesz. (Megjegyzés: erről hamarosan bővebb dokumentációt is összeállítok, azonban addig is javaslom, hogy nézzetek utána a Net-en. Keressetek rá pl. az XSD (Xml Schema Definition) kulcsszóra!)

- *Az ágensek a kommunikáció során tehát csak és kizárólag megfelelő sémához illeszkedő XML tartalmú üzeneteket fogadnak. Az üzenet beérkezését követően kiolvassák az üzenet tartalmát, és ebből egy generikus Java objektumot állítanak elő. Az ágens programozója tehát ezen objektum felhasználásával férhet majd hozzá az üzenetek tartalmához.*
- *Az ágens az üzenetek fogadását adott viselkedés (Behaviour) keretében hajtja végre. A konkrét viselkedéstípuson (pl. OneShot, Simple, Sequential, Complex, Ticker) felül meg kell majd tehát azt is adni, hogy milyen üzenet-mintázatnak (Message Template-nek) megfelelő üzeneteket vár az ágens, továbbá a beszélgetés/párbeszéd (Conversation) azonosítóját is, aminek keretein belül az üzenet beérkezhet.*
- *Az üzenetváltás során adott-kapott XML objektumok kiírását külön DataHandler osztály fogja ellátni. Ezzel pl. az üzenetek tartalma szövegfájlba (vagy később esetleg adatbázisba is) lesz menthető, vagy onnan felolvasható. Várhatóan ez az osztály fogja ellátni az adott XML adott sémának való megfelelésének (Validation) vizsgálatát is.*
- *Végül, kikapcsolásnál az ágens kiregisztrál a DF-nél.*

Ebben a követelmény specifikációban már számos konkrétum megjelenik. Következzen pár idézet a lényegesebb kritikákból, melyek az AGENTFRAME végső alakját megszabták.

> *> Nem errol beszeltunk, hogy egy agens tobb semaval is kommnikalhat?*
> *> (ontologia)*
> *> Hiszen ha o maga is egy adott semat ert a tipusabol kifolyolag, tobb*
> *> tipusu agens fele kuldhet uzeneteket, igy azokat annak nyelven tudnia*
> *> kell kodolni.*
> *> Elkepzelheto, hogy egy agens egy specialis semat ert, de minden agens*
> *> alaphelyzetben az altalanos (adattipus szintu) semat tudja es annak*
> *> ertelmezi, ha nem kap semmi utalast specialis sema hasznalatara.*
>
> *Így van: egy ágens adott sémának megfelelő Content-et fogad el, miközben más ágensekkel az Ő típusuknak megfelelő Content-el kommunikál. Tehát az egyes ágensek a priori ismerik a többi ágens típusát, és az ahhoz tartozó kommunikációs sémákat. Legalábbis az AgentFrame ezt fogja feltételezni a benne foglalt kommunikációs blokk(ok)ban. --- Ha gondolod, akkor lehet egy alapértelmezett séma (számok, string-ek, dátumok, stb, fogadására). Ennek használata opcionális lesz. Az ágens programozója fogja eldönteni, hogy kell-e neki ezen felül saját séma, vagy sem. Ok?*

> *> > továbbá a beszélgetés/párbeszéd (Conversation) azonosítóját is, aminek*
> *> > keretein belül az üzenet beérkezhet.*
> *>*
> *> ez ugye opcionalis?*
>
> *Úgy csinálom meg, hogy kivehető legyen a kódból, rendben?*

> *> > - Az üzenetváltás során adott-kapott XML objektumok kiírását külön*
> *> > DataHandler osztály fogja ellátni. Ezzel pl. az üzenetek tartalma*
> *> > szövegfájlba (vagy később esetleg adatbázisba is) lesz menthető, vagy*
> *> > onnan felolvasható. Várhatóan ez az osztály fogja ellátni az adott XML*
> *> > adott sémának való megfelelésének (Validation) vizsgálatát is.*
> *>*
> *> ez kicsit ugy hangzik, mintha uzenetek tartalmaval csak azt lehetne tenni,*
> *> pedig ez csak raadas. Az uzenetek tartalmat eslodlegesen belso eljarasok*
> *> dolgozzak fel.*
>
> *Ezzel teljes mértékben egyetértek. Így van. Viszont az említett "belső eljárásokat" már - a sémához illeszkedően - a mérés kidolgozójának, azaz az ágens programozójának kell kialakítania. Egy keretet lehet esetleg adni hozzájuk, amit aztán a mérés kidolgozója tölt fel programkóddal.*

Az említett levelezésen túl természetesen párhuzamos szóbeli egyeztetések is zajlottak. Ennek során végül például abban is megállapodtunk, hogy az XML/**XSD** (**X**ml **S**chema **D**efinition; http://www.w3.org/TR/xmlschema-0) dokumentumok Java-ban történő kezelését **JAXB** (**J**ava **A**rchitecture for **X**ml **B**inding; https://jaxb.dev.java.net) alapon oldjuk meg, stb, stb, stb.

Az ilyen, és ehhez hasonló, akár egzakt technikai specifikumok egyeztetését követően gyakorlatilag **RAD** (**R**apid **A**pplication **D**evelopment; http://en.wikipedia.org/wiki/Rapid_application_development) módon indult meg a fejlesztés. A folyamat bemenete a követelmény specifikáció volt, kimenete pedig a(z AGENTFRAME) prototípus, aminek dokumentációját az Olvasó pillanatnyilag épp a kezében tartja.

Gyors alkalmazásfejlesztésre főként az idő szűkössége folytán volt szükség. Nem volt idő arra, hogy a követelmény specifikációt részletes programozói specifikációvá érleljük, amire aztán az implementációs fázis alapozhat. A jelen anyag tehát a szoftver elkészülését követően, illetve azzal egy időben, specifikációs/dokumentációs jelleggel jött létre.

Az előzetes specifikációhoz képest az implementáció főként a következőkben különbözik:

- Az AGENTFRAME nem egyetlen Java-osztály, hanem Java-osztályok egy egész kis csomagja (melyben pl. 2 különböző példa-ágens is helyet foglal).
    - ***OK**: már a kezdetektől fogva nyilvánvaló volt, hogy egyetlen osztályban nem lehet/célszerű megvalósítani a követelmény specifikáció által előírt számtalan funkciót. Szükség van adatkezelési, XML feldolgozási, és egyéb saját fejlesztésű osztályokra. Ezen felül érdemes több (minimum 2), egyszerűbb, egymással együttműködő, **valóban működő** példa-ágenst is előállítani, amik program-szinten szemléltetik a JADE-es ágensközösség/platform működési elveit.*
- Az ágensek típusának explicit deklarációja nem szükségszerű. Az ágens programozója döntheti el, hogy folyamodik-e hasonló „változó” bevezetéséhez.

  - ***OK**: az ágenst megvalósító osztály önmagában véve is egy típusnak felel meg.*
- Az ágensek többféle tartalomnyelven (is) kommunikálhatnak.
  - ***OK**: ez egy opcionális bővítmény. Számos kiegészítésre ad lehetőséget.*
- Nem a DataHandler osztály látja el az XML objektumok validálását.
  - ***OK**: a JAXB csomag szolgáltat erre megfelelő eljárásokat.*
- Az ágensek grafikus felhasználói felületét (GUI-ját) nem XML nyelven írjuk le, hanem – szokványos módon – a programkódban.
  - ***OK**: a méréseket kidolgozó kollégákra nem kívántunk további terhelést helyezni azzal, hogy a JADE-en és JAXB-n felül még egy viszonylag összetett Java GUI leírást is meg kelljen tanulniuk. Továbbá az előállt megoldás semmivel sem összetettebb, vagy rugalmatlanabb, mint a legtöbb XML alapú.*
- Az XSD sémáknak köszönhetően lehetőséget biztosítunk akár több, saját, XML-alapú ágens-adatbázis létrehozására.
  - ***OK**: a DataHandler osztály kiindulásképp egyelőre csak szöveges fájlkonverziót támogat, így célszerűnek mutatkozott valamiféle strukturált adattárolási/hozzáférési lehetőséget is biztosítani az ágensek számára.*

Végső soron tehát **az AGENTFRAME egy Java csomag-gyűjtemény (package-collection), melyben az adatkezelő osztályok mellett 2 példa/csontváz-ágens is helyet foglal, amiket bővítve/kurtítva saját fejlesztésű felhasználói ágensek/ágensközösségek hozhatók létre**.

Ezzel tehát elértük a fejlesztés legfőbb célkitűzését: kialakítottuk a labor méréseinek kidolgozásához alapvetően szükséges szoftveres hátteret. A következőkben e háttér rejtelmeit mutatjuk be részletesen. Először is a JADE rendszerkörnyezet telepítésének és futtatásának alapértelmezett módját ismertetjük, majd egy szabványos JADE-es mintapéldára (és messzemenőkig átfogó, „gyári" dokumentációjára) alapozva rátérünk az AGENTFRAME csomag felépítésére és működ(tet)ésére. A részletes elemzést egy rövid összefoglaló zárja. A függelékben az Eclipse (http://www.eclipse.org) szoftverfejlesztői környezet bemutatásával *további, **opcionális** alternatívát* kínálunk a JADE-es ágensek fejlesztésére és futtatására.

Az anyag tehát lényegében 4 részből tevődik össze: **(1)** Bevezető, **(2)** AGENTFRAME, **(3)** Összefoglaló, és **(4)** Függelék.

1. A bevezetőt mindenkinek ajánlott elolvasni, hogy képbe helyezze magát.
2. Az AGENTFRAME-et ecsetelő részt többféleképp is lehet olvasni attól függően, hogy Eclipse-ben, vagy anélkül szeretnénk fejleszteni. Amennyiben Eclipse nélkül dolgoznánk, úgy a 2-es fejezet teljes egészében szükséges. Ha azonban Eclipse-ben (is) fejlesztenénk, úgy a 2-es fejezet maradéktalan elolvasása már nem feltétlen szükséges. Ami ebben az esetben feltétlen szükséges belőle, az a következő: 2.1, 2.2, 2.3. A 2.4-es és 2.5-ös szakaszt már „szűrve" is olvashatjuk. Azokat a részeket, melyek a sémák és/vagy ágensek fordításáról, vagy futtatásáról szólnak, egyszerűen csak helyettesítsük a függelék 4.1-es alfejezetének megfelelő passzusaival.
3. Az összefoglaló elolvasása nem szükséges, de ajánlott (az anyag elolvasása után).
4. A függelék egyelőre csak az Eclipse-JADE integráció leírását tartalmazza. Ennek elolvasása csak akkor szükséges, ha az Olvasó Eclipse-ben kíván JADE alá fejleszteni.

2 AZ AGENTFRAME

Az AGENTFRAME csomag megértéséhez elengedhetetlen bizonyos alapok megismerése. Célszerű áttekintenünk a csomag működéséhez szükséges szoftver-hátteret, illetve e háttér dokumentációját, a csomag felépítésének és működésének leírása ugyanis erre alapoz.

2.1 Szoftverek és dokumentációk

Az „*Elosztott Intelligens Rendszerek*" c. labor anyagait – így például magát az AGENTFRAME csomagot, illetve a hozzá szükséges szoftvereket és dokumentációt – többféleképp is elérhetjük. Lokális hálózaton keresztül a következő elérésen találjuk őket.

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor

Amennyiben Tanszéken kívülről próbálkozunk, úgy marad a távoli elérés. Vagy egy alkalmas szoftverrel – pl. WinSCP-vel (http://winscp.net) – közvetlenül, vagy web-böngészőnk – pl. Mozilla Firefox (http://www.mozilla-europe.org/hu/products/firefox) – segítségével csatlakozhatunk a mostoha.mit.bme.hu tanszéki szerverhez, melyen az anyagok találhatók. Az előbbi eset értelemszerű. Utóbbi esetben a böngésző címsorába a következőt kell írni.

sftp://username:password@mostoha.mit.bme.hu

A „username" és „password" természetesen felhasználótól függ. Helyes megadásuk, továbbá egy SFTP protokollra alkalmas alkalmazás megléte esetén beléphetünk a MOSTOHA szerverre. Itt alapértelmezésben a

/home/username

könyvtárba kerülünk. A labor anyagait innen pár kattintásnyira találjuk csupán, a következő könyvtárban.

/export/disk1/aigroup/oktatas/ElosztottIntelligensRendszerek-BSclabor

A következőkben ezekre az anyagokra (szoftverekre és dokumentációkra) építve mutatjuk be az AGENTFRAME csomag üzembe helyezéséhez szükséges szoftverháttér kialakítását.

2.2 JADE telepítése és működ(tet)ése

A telepítés során (a következőkben mindvégig) Windows operációs rendszert feltételezünk (lehetőleg XP-t). Az általunk használatos JADE keretrendszer futtatásához a **JDK** (**J**ava **D**evelopment **K**it), és a **JWSDP** (**J**ava **W**eb **S**ervices **D**evelopment **P**ack; http://java.sun.com/webservices/downloads/previous/webservicespack.jsp) telepítése és konfigurációja szükséges. A JDK szolgáltatja a Java programok futtatásához és fordításához szükséges környezetet (márpedig a JADE végső soron egy Java-program), míg a JWSDP a JAXB-hez lesz szükséges. A megfelelő JDK-t (1.6.03) a következő lokális elérésen találjuk.

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\sw\jdk-6u3-windows-i586-p.exe

Innen másoljuk át számítógépünk valamely meghajtójára, és ott futtassuk. Lehetőség szerint az

X:\Program Files\Java\jdk1.6.0_03

könyvtárba telepítsük (ahol X a megfelelő meghajtót jelöli). Sikeres telepítést követően állítsuk be a Windows „Path” és „JAVA_HOME” nevezetű környezeti változóit (Start menü/Vezérlőpult/Rendszer/Speciális/Környezeti változók, Rendszerváltozók/Változó = Path, Szerkesztés). Adjuk hozzá a Path-hoz a következő könyvtárat.

X:\Program Files\Java\jdk1.6.0_03\bin

Ha még nincs JAVA_HOME környezeti változónk, akkor hozzuk létre, ha pedig már van, akkor módosítsuk a JDK telepítési könyvtárára.

Ellenőrizzük, hogy megfelelő-e a telepítésünk! Indítsunk el egy Windows parancssori ablakot (Start menü/Futtatás/cmd, OK), majd a feljövő parancssori ablakba írjuk be a „`java -version`”, majd pedig a „`javac -version`” parancsot. Rendre a következő válaszokat kell(ene) kapnunk.

```
java version "1.6.0_03"
Java(TM) SE Runtime Environment (build 1.6.0_03-b05)
Java HotSpot(TM) Client VM (build 1.6.0_03-b05, mixed mode, sharing)
```

…és…

```
javac 1.6.0_03
```

Ha „mindez volt, ahogy írva”, következhet a JWSDP telepítése. Másoljuk át számítógépünk valamely meghajtójára a következő fájlt, …

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\sw\jwsdp-2_0-windows-i586.exe

…és futtassuk ott. A telepítőben a következőkre kattintsunk: *Tovább, APPROVE, Tovább, Válasszuk ki az 1.6.0_03-as JDK-t, Tovább, „No Web Container”, Tovább, Tovább, Tovább, Tovább, Tovább, Tovább, …ezek után történik meg a valódi telepítés…, Tovább, Tovább, Nem „Register with…”, Tovább, Befejezés*

A következő lépés a JADE (3.5-ös verziójának) telepítése. Ez már jóval egyszerűbb. Fogjuk a

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\sw\JADE-3.5-All.zip

fájlt, és másoljuk át az X meghajtó gyökerébe, majd bontsuk ki ott (pl. PowerArchiver-rel; http://www.powerarchiver.com). Ennek hatására létrejön az

X:\jade

könyvtár, és tartalma. Lényegében tehát ez a könyvtár tartalmazza a JADE-et.[1] A rendszer futtatásához, és alapvető funkcióinak megismeréséhez tanulmányozzuk át a

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\doc\jade\JADE_Lab-Demo-2008.pdf

dokumentumot. Ennél valamivel részletesebb ismertetést nyújt a következő anyag (ám ennek ismerete – az előbbivel ellentétben – már nem szükséges a jelen anyagban való továbbhaladáshoz).

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\doc\jade\JADE_Administrators-Guide.pdf

Tehát a JADE adminisztrátori segédletet egyelőre akár át is ugorhatjuk.

2.3 Az AGENTFRAME csomag alapja: KönyvKereskedés

Az AGENTFRAME csomagban, mint már említésre került, 2 példa-ágens van implementálva. Mindkettő egy eredetileg JADE-hez mellékelt mintapélda, az „`examples.bookTrading`” átdolgozásával/kiegészítésével jött létre[2]. Ennek megfelelően az AGENTFRAME-ben látható programozási megoldások illeszkednek a JADE fejlesztői által sugalltakhoz.

A `examples.bookTrading` csomag tartalmát tehát fizikailag a következő elérésen találjuk.

X:\jade\src\examples\bookTrading*.*

Itt lényegében 3 fájl foglal helyet: *könyvVásárló* ágens, *könyvÁrus* ágens, és a *könyvÁrus* ágens *GUI*-jának a programkódja.

A mintapélda – és a JADE programozási alapjainak – megértéséhez (**NAGYON FONTOS!!!**) a következő olvasmány alapos elolvasása és értelmezése ajánlott.

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\doc\jade\JADE_Programming-Tutorial-for-Beginners.pdf

Az említett anyag a JADE egyik főfejlesztőjének/főtervezőjének, *Caire Giovanni*-nak a műve. Az anyag alapos áttanulmányozása tehát újfent, hangsúlyozottan javasolt. Amennyiben az anyag olvasása során esetleg értelmezési nehézségekbe ütközünk, úgy a Java programozási nyelv alapjainak felelevenítése nagy segítségünkre lehet. Erre szolgál(hat) a következő anyag.

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\doc\java\JAVA_Java-Programming-Language-Basics.pdf

Az anyag elolvasása során megismerhettük a *bookTrading* példában szereplő *BookBuyerAgent* és *BookSellerAgent* ágensek felépítését, és működését. Az említett 2 típusú ágensből tetszőleges számút létrehozhatunk egy-egy ágensplatformon. Ehhez először is le kell fordítanunk mindkettőt.

Lépjünk be az

X:\jade

[1] A gyökérben található tömörített fájlt ezek után akár már le is törölhetjük.
[2] A következő elérésen találhatunk egy igen tömör áttekintést a Java csomagokról, és úgy általában a Classpath-ról: http://www.jarticles.com/package/package_eng.html

könyvtárba Windows parancssorban, és írjuk be a következő 2 parancsot.[3]

```
compile src\examples\booktrading\bookbuyeragent.java
compile src\examples\booktrading\booksellerагent.java
```

Ennek hatására létrejönnek a .class kiterjesztésű, futtatható bytecode Java állományok, melyeket a **JVM** (**J**ava **V**irtual **M**achine) már képes interpretálni. Indítsunk el egy JADE platformot a következő parancs beírásával.

```
runjade -gui
```

Ezek után az RMA ágens GUI-ján kattintsunk a „Start New Agent" gombra (vagy az Action/Start New Agent menüpontra). Indítsunk egy BookBuyerAgent típusú ágenst. Ehhez pl. a következőket írjuk be.[4]

Agent name: `bb1`
Class name: `examples.bookTrading.BookBuyerAgent`
Arguments: `aaa`

Ennek hatására a kiválasztott konténerben létrejön egy bb1 nevű BookBuyerAgent példány. A parancssori ablakban is láthatjuk ennek nyomát. Ágensünk percenként próbál meg kapcsolatba lépni a DF-fel, és lekérdezni tőle a platformon található, „`book-selling`" szolgáltatást nyújtó ágensek listáját.

Ez a lista ekkor még nyilván üres, hiszen még egyetlen könyvÁrus ágenst sem indítottunk a platformon (ne feledjük: egy host-on csak egy platform futhat!). Indítsunk hát 3 könyvÁrus ágenst is! Nyissunk meg egy újabb parancssori ablakot, és írjuk bele a következő 2 parancsot.

```
cd c:\jade
runjade –container bs1:examples.bookTrading.BookSellerAgent
                   bs2:examples.bookTrading.BookSellerAgent
                   bs3:examples.bookTrading.BookSellerAgent
```

`bb1` ágensünk ezek után már rátalál mindhárom `bs` ágensre, azonban mindhárman visszautasítják (REFUSE), mikor ajánlatot kér (CFP) tőlük az „`aaa`" című könyvre vonatkozóan (amit bemeneti argumentumként adtunk meg `bb1` számára). `bb1` ezt az „`Attempt failed: aaa not available for sale`" üzenettel jelzi számunkra. A sikertelenség oka nyilván az, hogy mindhárom `bs` ágens könyvkatalógusa üres még: nincs meg nekik `aaa`. Nekünk kell megadnunk tehát az egyes `bs` ágensek grafikus felületén, hogy milyen könyvekkel rendelkezzenek, és milyen áron kínálják azokat. Tegyünk hát így (adjuk meg több ágensnek is az `aaa` könyvet, lehetőleg különböző áron, esetleg más könyvekkel fűszerezve)!

---

[3] A parancsokat úgy a legegyszerűbb beírni, ha egyszerűen átmásoljuk őket innen, a dokumentumból egyenesen bele a parancssori ablakba. De legyünk tekintettel arra, hogy a Ctrl+V kombináció (a Paste művelet) nem működik a parancssori ablak esetében. A Ctrl+C kombinációval vágólapra helyezett tartalmat az egér jobb gombjával tudjuk belemásolni a parancssori ablakba („Beillesztés" menüpont). Hasonlóan működik a parancssori ablakból történő kimásolás is: az egér jobb gombjával kattintunk egyet az ablak belső régiójára, és a pop-up menüből a „Megjelölés" menüpontot választjuk. Ezek után az egér bal gombját folytonosan nyomva tartva ki tudjuk jelölni a parancssori ablak egy adott részét. A kijelölés végeztével (a bal egérgomb elengedését követően) kattintsunk a kijelölt részre az egér jobb gombjával. Ennek hatására kikerül a vágólapra a kijelölt tartalom, amit immár alkalmazástól függően bárhová bemásolhatunk (pl. az egér jobb gombjával magába a parancssori ablakba is).

[4] **FIGYELEM**: a Java osztályok megadása során mindig vegyük figyelembe a kis- és nagybetűk különbséget!!!

Érdemes észrevennünk, hogy az egyes parancssorok (mint Windows-os task-ok) nem egy-egy ágenst, hanem lényegében egy-egy konténert futtatnak! Az első parancssor, amit létrehoztunk, a `Main-Container` nevű konténert futtatja (s így magát a számítógépünkön host-olt egyetlen platform-ot), míg a másik parancssor, amit az imént indítottunk, a 3 `bs` ágenst tartalmazó mellékkonténert futtatja (amin belül az ágensek mind-mind külön szálon futnak).

Az `aaa` könyv katalógusba helyezését követően a `bb1` ágens már érdemleges ajánlatokat (PROPOSE) is kap – `bs` ágensek felé közvetített – újabb ajánlatkérésére (CFP) válaszul. A `bs` ágensek ajánlatukban a megfelelő (jelen esetben `aaa`) könyv árát közlik `bb1`-el. `bb1` erre kiválasztja a legszimpatikusabbnak tűnő (magyarán legolcsóbb) ajánlatot, és egy megfelelő üzenetben jelzi a legalacsonyabb árat kínáló `bs` ágensnek, hogy ajánlatát elfogadja (ACCEPT_PROPOSAL). A `bs` ágens erre kétféleképp válaszolhat. Ha időközben tfh. egy másik `bb` ágens hamarabb jelzett vissza neki egy ugyanerre a könyvre vonatkozó ajánlatára, úgy a könyvet már eladta volna neki, és így az időközben `bb1`-től beérkezett elfogadást sajnálatos módon kénytelen volna elutasítani (FAILURE). Szerencsére azonban most nem ez a helyzet. `bb1` egymaga testesíti meg a piaci keresletet, nincs kivel versengenie – így a megfelelő `bs` ágens garantáltan el fogja fogadni ajánlatát (és ezt egy megfelelő INFORM üzenettel jelzi is számára). Ezzel tehát a tranzakció lezajlott, megtörtént a „kézfogás”, és a `bb1` ágens – feladata végeztével – befejezi működését.[5]

A szakaszban eddig foglaltak és hivatkozottak alapján elsajátíthattuk a JADE ágensek programozásának főbb fortélyait, és az AGENTFRAME alapját képező mintapéldát. A következőkben tehát rátérhetünk az AGENTFRAME ismertetésére.

Az eddigieken túlmutató, komolyabb JADE programozási útmutatásért a JADE programozói segédletéhez érdemes fordulni.

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\doc\jade\JADE_Programmers-Guide.pdf

Ezen felül a fejlesztés során igencsak hasznosnak bizonyulhat a Java, illetve a JADE **API** (**A**pplication **P**rogramming **I**nterface) részletes ismerete is. A következő URL-eken találhatók.

http://java.sun.com/javase/6/docs/api ...és... http://jade.tilab.com/doc/api

Itt tájékozódhatunk tehát a lehető legrészletesebben arról, hogy összefüggéseiben milyen interfészek, osztályok, konstansok, változók, és metódusok állnak a rendelkezésünkre a fejlesztés során.[6]

---

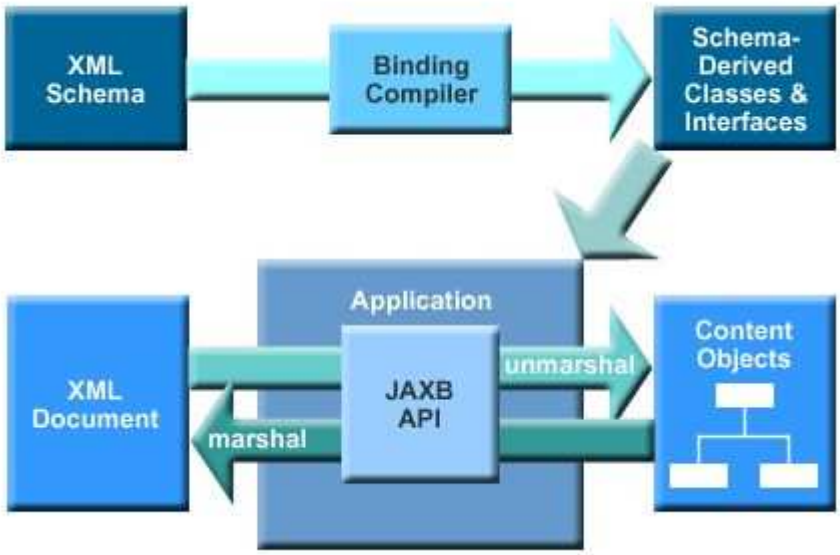
[5] A fenti kommunikációt érdemes egy Sniffer ágenssel végigkövetni (a DF ágenst is, nem csak a piac résztvevőit, monitorozva).

[6] A fejlesztéshez Integrált Szoftverfejlesztési Környezet (IDE – Integrated Development Environment) is igénybe vehető, pl. NetBeans IDE (http://www.netbeans.org), vagy Eclipse IDE (http://www.eclipse.org). Ezek az eszközök számos feladatot (pl. fordítás, build-elés) levesznek a vállunkról. Ily módon hatékonyabbá, és gyorsabbá tudják tenni a fejlesztési folyamatot. Azonban egy hasonló kaliberű szoftver alapos elsajátítása nem csekély feladat. Akinek nincs rá ideje, vagy lehetősége, fejleszthet „mezei” szövegszerkesztőben is (célszerű azonban olyat választani, ami képes legalább a nyitó-záró (kapcsos)zárójeleket összepárosítani, illetve egyéb módon, pl. színekkel támogatni a forráskód jobb átláthatóságát). Az Eclipse IDE jelenlegi legfrissebb verziója a következő elérésről tölthető le: \\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\sw\eclipse-java-europa-fall2-win32.zip A programhoz továbbá számos dokumentáció is tartozik a következő elérésen: \\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\doc\eclipse

2.4 JAXB (Java Architecture for Xml Binding)

Mielőtt rátérnénk az AGENTFRAME csomagban foglalt osztályok, s így az átdolgozott KönyvKereskedés példa ágenseire, célszerű lenne megismernünk a JAXB szoftver-architektúrát (lásd. pl. http://java.sun.com/developer/technicalArticles/WebServices/jaxb), amely az AGENTFRAME-en belüli XML/XSD kezelést biztosítja.

A JAXB külön letölthető formában is elérhető, azonban mi az egyszerűség kedvéért most a JDK-ba integrált (2.0-ás) JAXB-t fogjuk használni. Ezzel gyakorlatilag semmiféle hátrányt nem szenvedünk, hiszen a jelenlegi legújabb (2.1.6-os) verzió sem tud lényegesen többet, sőt.



1. ábra: a Java-XML kötést megvalósító JAXB szoftver-architektúra

Az 1. ábra szemlélteti a JAXB szoftver-architektúra vázlatos felépítését. Jól látható rajta az XML séma központi szerepe. XML sémákról pl. itt olvashatunk bővebben:

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\doc\xsd\...

Tehát a kezdetek kezdetén mindig egy XML sémából indulunk ki. Készítünk belőle Java osztályokat (és interfészeket), melyeket később beépíthetünk alkalmazásunkba. Alkalmazásunk ezen felül a JAXB API (lásd. https://jaxb.dev.java.net/nonav/2.1.6/docs/api) osztályait és metódusait is használhatja. Az egész eljárás célja az úgynevezett „**Marshalling**", avagy „**rendezés**" (a Java objektumok XML-lé alakítása), illetve az „**UnMarshalling**", avagy „**visszarendezés**" (XML-ek Java objektummá alakítása).

A kiindulás tehát mindig egy XSD, amiben XML-ek szintaxisát definiáljuk. Lényegében azt adjuk meg, hogy milyen elemek és attribútumok, milyen struktúrában fordulhatnak elő az XML-ekben. Egy nyelvet definiálunk, magyarán.

Egy XML nyelvi elemet egyszerűnek (Simple) tekintünk, ha nem tartalmaz más elemeket, egyébként összetettnek (Complex) nevezzük. Álljon itt példaképp a következő XSD, amely folyóirat-katalógusokat reprezentáló XML fájlok felépítésének leírására szolgál.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="catalog" type="catalogType"/>
 <xsd:complexType name="catalogType">
  <xsd:sequence>
   <xsd:element ref="journal"  minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="section" type="xsd:string"/>
  <xsd:attribute name="publisher" type="xsd:string"/>
 </xsd:complexType>
 <xsd:element name="journal" type="journalType"/>
 <xsd:complexType name="journalType">
  <xsd:sequence>
   <xsd:element ref="article"  minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:element name="article" type="articleType"/>
 <xsd:complexType name="articleType">
  <xsd:sequence>
   <xsd:element name="title" type="xsd:string"/>
   <xsd:element name="author" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="level" type="xsd:string"/>
  <xsd:attribute name="date" type="xsd:string"/>
 </xsd:complexType>
</xsd:schema>
```

A fenti XSD tehát olyan XML-eket ír le, melyek gyökéreleme szükségképp a „`catalog`”. Ennek 2 attribútuma van: „`section`” és „`publisher`” – mindkettő karakterfüzér. A „`catalog`” elemek tartalma „`journal`” nevű elemek esetleg üres, tetszőlegesen hosszú sora lehet. A „`journal`” elemeknek nincs saját attribútuma, viszont „`article`” elemek esetleg üres, tetszőlegesen hosszú sorát tartalmazhatják. Az „`article`” elemeknek 2 attribútuma van: „`level`” és „`date`” – mindkettő karakterfüzér. Ráadásul az „`article`” elemek egy elemi elem-pár („`title`” és „`date`”) legalább egy-elemű sorozatát is tartalmazzák. Mindkettő karakterfüzér, és nincs külön attribútumuk. Egy ilyen – ennek az XSD-nek megfelelő – XML-re példa a következő.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<catalog section="Java Technology" publisher="IBM developerWorks">
    <journal>
        <article level="Intermediate" date="January-2004">
            <title>Service Oriented Architecture Frameworks</title>
            <author>Naveen Balani</author>
        </article>
        <article level="Advanced" date="October-2003">
            <title>Advance DAO Programming</title>
            <author>Sean Sullivan</author>
        </article>
        <article level="Advanced" date="May-2002">
            <title>Best Practices in EJB Exception Handling</title>
            <author>Srikanth Shenoy</author>
        </article>
    </journal>
</catalog>
```

A JAXB tehát az XSD-ből indul ki. Első lépésben ezt kell Java osztályokká fordítanunk. Erre való az 1. ábrán látható „Binding Compiler”, esetünkben az „`xjc`” program (amely része a feltelepített JDK-nak). Tegyük fel, hogy a fenti XSD elérése a következő.[7]

X:\jade\src\milab\labxy\fooagent\xsd\foo.xsd

[7] Szükség esetén hozzuk létre az XSD fájlt, és a megfelelő könyvtárakat!

Tegyük fel továbbá, hogy az XSD-t az

X:\jade\src\milab\labxy\fooagent

könyvtárba szeretnénk lefordítani. Ehhez először is lépjünk be (pl. parancssorban) az

X:\jade

könyvtárba, és ott írjuk be a következőt.

xsd2java milab.labxy.fooagent milab\labxy\fooagent\xsd\foo.xsd

Az „`xsd2java`" egy batch-fájl, amely az „`xjc`" fordító-program hívását teszi egyszerűbbé. Ez a batch 2 argumentumot vár: **(1)** séma kontextusának (azaz a séma alapján kigenerálandó Java osztályoknak) célhelye Java csomagként megadva, és **(2)** a séma helye könyvtári elérésként.

A „`milab.labxy.fooagent`" csomag relatíve épp a megfelelő könyvtárat azonosítja, míg a második argumentum valóban a vizsgált XSD helye. A pirossal jelölt művelet végrehajtásának eredménye a következő.

```
parsing a schema...
compiling a schema...
milab\labxy\fooagent\ArticleType.java
milab\labxy\fooagent\CatalogType.java
milab\labxy\fooagent\JournalType.java
milab\labxy\fooagent\ObjectFactory.java
```

Látható, hogy négy `.java` kiterjesztésű forrásfájl keletkezett: minden összetett elemhez egy-egy külön fájlban egy-egy Java osztály jött létre (amik elnevezése automatikusan az összetett elem típusának a neve). Ezen felül egy `ObjectFactory.java` fájl/osztály is keletkezett. Ez utóbbi főként arra való, hogy az XML elemek JAXB reprezentációjának megfelelő Java objektumokat előállítsa (pl. egy konkrét `article`, vagy `catalog` objektumot). A többi osztály (pl. `CatalogType`) arra való, hogy segítségükkel össze lehessen állítani olyan objektumokat, amiket az `ObjectFactory` megfelelő metódusainak bemenetére adva megkaphatjuk az XML JAXB reprezentációjának megfelelő Java objektumokat.

Az ObjectFactory-nak tehát esetünkben a következő (számunkra érdekes) metódusai vannak:

```
public JAXBElement<JournalType> createJournal(JournalType value) {
        return new JAXBElement<JournalType>(_Journal_QNAME, JournalType.class, null, value);
}

public JAXBElement<ArticleType> createArticle(ArticleType value) {
        return new JAXBElement<ArticleType>(_Article_QNAME, ArticleType.class, null, value);
}

public JAXBElement<CatalogType> createCatalog(CatalogType value) {
        return new JAXBElement<CatalogType>(_Catalog_QNAME, CatalogType.class, null, value);
}
```

Ezek állítják tehát elő az XML elemeket reprezentáló konkrét Java objektumokat. Ezen felül természetesen még olyan „`createXXX`" metódusok is vannak, amikkel egy-egy `ArticleType`-ot, `JournalType`-ot, vagy épp `CatalogType`-ot (objektumot/példányt) állíthatunk elő. A megfelelő objektumok mind-mind rendelkeznek megfelelő `setXXX`/`getXXX` metódusokkal saját belső változóik (azaz attribútumaik, és elemi „elemeik") értékének írására/olvasására.

A felettébb barátságtalan szintaxis ne ijesszen meg senkit – első sorban nem emberi olvasatra szánták! A fentebb említett, – „`xjc`” által – automatikusan kigenerált metódusok célja csupán az, hogy létrehozzák a Marshalling-hoz, és UnMarshalling-hoz szükséges <paraméterezett> `JAXBElement`-eket (objektumokat). Ezek tekinthetők ugyanis, mint mondtam, az XML elemek – Java objektumok formájában adott – reprezentációjának.

Most tehát, hogy immár rendelkezésünkre állnak a megfelelő osztályok és metódusok, próbaképp hozzunk létre egy kis alkalmazást (nevezzük fooagent-nek), amelyik a JAXB API segítségével bemutatja az oda-visszarendezést. Ehhez hozzuk létre a következő forrásfájlt!

X:\jade\src\milab\labxy\fooagent\fooagent.java

A fájlt innen

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\src\fooagent.java

másolhatjuk át. *Megjegyzés: vegyük észre, hogy ez nem egy szokványos JADE ágens forráskódja, pusztán csak egy sima Java program, amely a JAXB API-t használja/szemlélteti.*

Ahhoz, hogy a programot le lehessen fordítani, szükség van 3 további (saját készítésű, és egyébként az AGENTFRAME csomag részét képező) osztályra is. Ezeket innen…

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\src\JAXBMarshaller.java
\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\src\JAXBUnMarshaller.java
\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\src\DataHandler.java

…ide…

X:\jade\src\milab\JAXBMarshaller.java
X:\jade\src\milab\JAXBUnMarshaller.java
X:\jade\src\milab\DataHandler.java

…tegyük át.[8]

A „`JAXBMarshaller.java`”, „`JAXBUnMarshaller.java`”, és „`DataHandler.java`” osztályokkal egyelőre ne foglalkozzunk. Inkább a „`fooagent.java`” kódra vessünk egy pillantást. – A kód jól láthatóan 4 részre tagolódik attól függően, hogy a program a meghívás során milyen argumentumokat kapott a bemeneten.

A `fooagent.java` programnak 2 kötelező parancssori argumentuma van: **(1)** *üzemmód*, és **(2)** *XML fájlra való hivatkozás*. Az első argumentum mondja meg, hogy a másodikkal mit kell tenni. Lényegében a következő üzemmódokat ismeri a program: `writestring`, `writefile`, `readstring`, `readfile`

WRITEFILE üzemmód esetén létrehozunk egy megfelelő Java Objektumot (`JAXBElement<CatalogType> catalogElement`), és ezt Marshall-oljuk bele egy XML fájlba a `JAXBMarshaller` osztály egy megfelelő metódusával.

```
JAXBMarshaller jaxbMarshaller = new JAXBMarshaller();
jaxbMarshaller.generateXMLDocument(xmlDocument, catalogElement, "milab.labxy.fooagent");
```

[8] *Érdemes lehet rápillantani az egyes források első sorában található package deklarációkra.*

A *JAXBMarshaller.generateXMLDocument* metódus 3 értéket kap a bemenetén: **(1)** *xmlDocument* (`String`), ami tulajdonképp a `fooagent` program második argumentuma, **(2)** *catalogElement* (`JAXBElement<CatalogType>`), ami a „`catalog`" XML-elem megfelelő Java reprezentációja, és **(3)** *"milab.labxy.fooagent"* (`String`), ami a séma kontextusát tartalmazó csomagot adja meg. Ez utóbbi ahhoz kell, hogy a megfelelő XML sémához tartozó `ObjectFactory` osztályt meg tudja találni a JAXB… A metódusnak gyakorlatilag nincs kimenete (`void`).

A jobb megértés érdekében fordítsuk is gyorsan le a `fooagent` programot, és ellenőrizzük működését! Ehhez parancssorban, az `X:\jade` könyvtárban a következőket írjuk be.

```
compile src\milab\labxy\fooagent\fooagent.java
run milab.labxy.fooagent.fooagent writefile src\milab\labxy\fooagent\foo.xml
```

Az első sor lefordít mindent, ami szükséges[9], míg a második – adott argumentumokkal – lefuttatja az előállt „`milab.labxy.fooagent.fooagent`" osztályt/programot (`fooagent.class`). A futás eredményeképp a következő XML fájl áll elő.

X:\jade\src\milab\labxy\fooagent\fooagent.xml

Ellenőrizzük, hogy az előállt XML fájl valóban megegyezik-e a fentebb említettel!

Ha igen, akkor minden rendben. Ez volt a „Marshalling" klasszikus esete. Egy megfelelő, adott XSD-ből előállított, gyökérelemet reprezentáló osztályhoz tartozó Java objektumból XML-t generáltunk. De az XML-t nem csak fájlba generálhatjuk, hanem karakterfüzérbe is. Erre szolgál a *JAXBMarshaller.generateXMLString* metódus.

```
jaxbMarshaller.generateXMLString("UTF-8", catalogElement, "milab.labxy.fooagent")
```

Lényegében itt is 3 bemenet van: **(1)** a kimenet karakterkódolását meghatározó String, **(2)** az XML-lé alakítandó Java objektum, és **(3)** a kapcsolódó séma kontextusa. A metódus kimenete egy XML szintaxisú String lesz (amely a Marshall-olt Java objektumnak felel meg).

Az „UnMarshalling", avagy „visszarendezés" ennek épp a fordítottja: adott XML fájlból, vagy String-ből állítja elő a megfelelő Java objektumokat. A *JAXBUnMarshaller.readXMLDocument* metódus az XML fájlok, míg a *JAXBUnMarshaller.readXMLString* metódus a XML String-ek „visszarendezéséért" felelős.

Az előbbinek 3 bemenete van: **(1)** hivatkozás az XML fájlra, **(2)** hivatkozás az XSD fájlra, és **(3)** a séma kontextusa. Az utóbbinak is 3 bemenete van: **(1)** egy XML szintaxisú String, **(2)** hivatkozás az XSD fájlra, és **(3)** a séma kontextusa. Mindkettő kimenete általános Java objektum (`Object`), amit az UnMarshalling-ot hívó alkalmazásnak kell explicite megfelelő típusú objektummá alakítania (casting)[10].

Vegyük észre, hogy az XML szintaxisú String, amit a *JAXBUnMarshaller.readXMLString* metódus bemenetére adunk a `fooagent` kódjában, a *DataHandler* osztály ***DataHandler.readFile2String*** metódusával áll elő (a már meglévő XML fájl előzetes

[9] fooagent.java, ArticleType.java, JournalType.java, CatalogType.java, ObjectFactory.java, JAXBMarshaller.java, JAXBUnMarshaller.java, DataHandler.java – figyeljünk a könyvtárak megadására!

[10] Lásd. a http://java.sun.com/docs/books/tutorial/java/IandI/subclasses.html oldalon a „Casting Objects" szakaszt

beolvasása útján). A *DataHandler* osztály ezen felül egyelőre még csak egyetlen metódust biztosít számunkra: a ***DataHandler.writeString2File***

A metódus bemenete egy-egy `String`. **(1)** a célfájl, amibe a String-et bele kell írnunk (esetleg a fájl felülírásával), és **(2)** maga a String, amit fájlba szeretnénk írni. A metódus kimenete gyakorlatilag semmi (`void`).[11]

Visszatérve: az UnMarshalling során nyilván a beolvasott XML fájl, vagy String validációja is megtörténik. Az UnMarshalling-ot követően a `fooagent` kódjának megfelelően a kapott – adott XML-t reprezentáló – Java objektumokat rendre végigolvassuk, és kiírjuk a megfelelő tartalmakat a kimenetre (a parancssori ablakba). A forráskódban látható lépések tekinthetők általános mintának az UnMarshalling során kapott Java objektumok tartalmához való hozzáférésre.

Próbáljuk ki a következő hívásokat is, hogy lássuk, melyik esetben mit ír ki a program!

```
run milab.labxy.fooagent.fooagent writestring src\milab\labxy\fooagent\foo.xml
run milab.labxy.fooagent.fooagent readfile src\milab\labxy\fooagent\foo.xml
run milab.labxy.fooagent.fooagent readstring src\milab\labxy\fooagent\foo.xml
```

Esetleg módosítsuk az XML fájlt, és vizsgáljuk meg azt is, hogy mi történik olyankor! Ez a fajta vizsgálódás mindenképp tanulságos lehet a JAXB működési mechanizmusának gyakorlati megértésében.

[11] Lényegében a ***DataHandler.readFile2String*** metódus „inverzéről” van szó…

2.5 Az AGENTFRAME „csomag”

Az előzőekben áttekintettük a JADE, és a JAXB használatát és működését. Ennek során az AGENTFRAME „csomag” bizonyos elemeit is érintettük. Nevezetesen szó esett a `milab.JAXBMarshaller`, `milab.JAXBUnMarshaller`, és `milab.DataHandler` osztályokról.

Az említett osztályok, amint teljes megnevezésükből is látszik, a „`milab`” nevezetű csomagban (package-ben) foglalnak helyet. Ennek oka nyilván az, hogy az említett objektumokban megtestesülő funkcionalitás nem kötődik egy-egy konkrét méréshez; a labor bármely mérésen felhasználható kell, hogy legyen; konkrét méréstől független.

A laborokat ezen belül subpackage-ekbe célszerű rendezni. Erre a „`labxy`” elnevezési sémát javasoljuk. Például az első labor a „`milab.lab01`”, a második a „`milab.lab02`” subpackage-ben foglaljon helyet, sit. Az egyes laborokhoz tartozó package-eket ezen belül elvben már tetszés szerint feloszthatjuk (ha szükséges). Az AGENTFRAME keret azonban erre is tesz javaslatot.

Azt javasoljuk, hogy minden egyes ágenst, és a hozzájuk közvetlenül kapcsolódó objektumokat (pl. az ágensek GUI-ját) rendre az ágensekről elnevezett subpackage-ekbe tegyük. Így tehát például a 4. mérés myAgent elnevezésű ágensének kódjait a „`milab.lab04.myAgent`” subpackage-ben helyeznénk el.

Ennek előnye, hogy egy-egy laborgyakorlat kódjai közt különválaszthatóvá válnak az ágensektől független (adott mérésen belül általános) kódok a konkrét ágensek kódjától. Sajnos azonban a felosztásnak itt még nincs vége… Az egyes ágens-package-eken belül is szükség lehet további subpackage-ek létrehozására.

Gondoljunk csak bele abba, hogy mi történne akkor, ha az egyes ágenseknek adott esetben többféle tartalomnyelvet kellene ismerniük (pl. egyet a kommunikációhoz, egyet pedig az XML alapú adatbázisuk kezeléséhez). Ekkor, mikor a JAXB fordítója az egyes nyelvekhez tartozó XML sémákhoz rendre le-legenerálná a kontextusokat (azaz a sémából következő Java osztályokat, és az `ObjectFactory`-t), akkor a kigenerált kontextusok, amennyiben egy könyvtárba kerülnek, összekeverednének (a különböző sémákhoz tartozó `ObjectFactory` osztályok felülírnák egymást, stb). Különböző kontextusok nem kerülhetnek egy könyvtárba.

A fentiekből kifolyólag tehát célszerű a különböző XML sémák kontextusához tartozó package-eket, azaz az ágens által „beszélt/értett” tartalomnyelveket az egyes ágenseken belül különválasztani. Erre a következő konvenciót tudnánk javasolni.

Legyen szó például az „`milab.lab04.myAgent.myAgent`” osztály által implementált ágens tartalomnyelveiről. Tegyük fel, hogy legalább két tartalomnyelvet ismer. Nevezzük ezeket rendre „`language01`”-nek, „`language02`”-nek, „`language03`”-nak,… Ekkor a megfelelő XSD fájlokat is „`language01.xsd`”-nek, „`language02.xsd`”-nek, „`language03.xsd`”-nek, stb, nevezzük, és helyezzük őket rendre a következő könyvtárakba.

X:\jade\src\milab\lab04\myAgent\language01\language01.xsd
X:\jade\src\milab\lab04\myAgent\language02\language02.xsd
X:\jade\src\milab\lab04\myAgent\language03\language03.xsd
…

Az egyes tartalomnyelvekhez tartozó package-ek ekkor a következő elnevezést kapnák.

```
milab.lab04.myAgent.language01
milab.lab04.myAgent.language02
milab.lab04.myAgent.language03
...
```

Tehát például az említett ágens 2-es számú tartalomnyelvéhez/sémájához tartozó Java-kontextus legenerálásához ebben az esetben a következő parancsot kellene kiadnunk az `X:\jade` könyvtárban.

```
xsd2java milab.lab04.myAgent.language02 milab\lab04\myAgent\language02\language02.xsd
```

Ennek hatására az

X:\jade\src\milab\lab04\myAgent\language02

könyvtárban létrejönne a séma kontextusa (azaz a sémából adódó Java osztályok, és a megfelelő `ObjectFactory` osztály is).

2.5.1 AGENTFRAME üzembe helyezése

Az AGENTFRAME-et a következő elérésen találjuk.

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\AgentFrame.zip

A csomag telepítéséhez ezt a fájlt helyezzük az

X:\jade\src

könyvtárba, és ott bontsuk ki. Ennek hatására létrejön egy – a fentebbi ajánlásnak megfelelő – könyvtárstruktúra, benne a megfelelő osztályokkal, és metódusokkal.[12]

Az eddigiekben csak a `milab.JAXBMarshaller`, `milab.JAXBUnMarshaller`, és `milab.DataHandler` osztályokról esett szó. Ez azonban még messze nem minden. Az AGENTFRAME egy kétszereplős minta-ágensközösséget is tartalmaz, melynek alkalmas átdolgozásával a későbbiekben már akár saját testre szabott ágenseinket is kialakíthatjuk.

A `milab.labxy.BookBuyerAgent.BookBuyerAgent`, és a `milab.labxy.BookSellerAgent.BookSellerAgent` osztályok már le vannak fordítva épp úgy, mint ahogy minden más is.

A biztonság kedvéért következzen a fordítás és futtatás lépéseinek felsorolása. Ehhez először is tegyük fel, hogy (csak) a következő fájlokkal rendelkezünk.

[12] Az `AgentFrame.zip` állományt ekkor akár már ki is törölhetjük az `X:\jade\src` könyvtárból.

X:\jade\src\milab\DataHandler.java
Adathozzáférést megvalósító osztály

X:\jade\src\milab\JAXBMarshaller.java
Java→XML konverziót megvalósító osztály

X:\jade\src\milab\JAXBUnMarshaller.java
XML→Java konverziót megvalósító osztály

X:\jade\src\milab\labxy\BookBuyerAgent\BookBuyerAgent.java
Kiegészített KönyvVásárló ágenst megvalósító osztály

X:\jade\src\milab\labxy\BookSellerAgent\BookSellerAgent.java
Kiegészített KönyvÁrus ágenst megvalósító osztály

X:\jade\src\milab\labxy\BookSellerAgent\BookSellerGui.java
Kiegészített KönyvÁrus ágens grafikus felhasználói felületét megvalósító osztály

X:\jade\src\milab\labxy\BookSellerAgent\catalog.xml
Kiegészített KönyvÁrus ágens kiindulási könyv-adatbázisa

X:\jade\src\milab\labxy\BookBuyerAgent\language01\language01.xsd
Kiegészített KönyvVásárló ágens egyetlen kommunikációs tartalomnyelve

X:\jade\src\milab\labxy\BookSellerAgent\language01\language01.xsd
Kiegészített KönyvÁrus ágens egyetlen kommunikációs tartalomnyelve

X:\jade\src\milab\labxy\BookSellerAgent\catalog\catalog.xsd
Kiegészített KönyvÁrus ágens könyv-adatbázisának séma-definíciója

A csomagot akkor, és csak akkor tudjuk lefordítani, ha a fentebb felsorolt összes fájllal rendelkezünk (a „`catalog.xml`” fájlt kivéve – erre ugyanis a fordításhoz nincs szükség). A csomag fordításának és futtatásának lépései ekkor a következők.

1. **A fordítás első lépése az XML sémák Java-kontextusának kigenerálása. Ehhez a következő parancsokat adjuk ki az `X:\jade` könyvtárban (parancssorban).**

```
xsd2java milab.labxy.BookBuyerAgent.language01 milab\labxy\BookBuyerAgent\language01\language01.xsd
xsd2java milab.labxy.BookSellerAgent.language01 milab\labxy\BookSellerAgent\language01\language01.xsd
xsd2java milab.labxy.BookSellerAgent.catalog milab\labxy\BookSellerAgent\catalog\catalog.xsd
```

Mivel a sémák egymástól függetlenek, ezért az előbbi lépések sorrendje tetszőleges lehet. A lépések végrehajtását követően a következő új fájlok keletkeznek.

X:\jade\src\milab\labxy\BookBuyerAgent\language01\ObjectFactory.java
X:\jade\src\milab\labxy\BookSellerAgent\language01\ObjectFactory.java
X:\jade\src\milab\labxy\BookSellerAgent\catalog\BookType.java
X:\jade\src\milab\labxy\BookSellerAgent\catalog\CatalogType.java
X:\jade\src\milab\labxy\BookSellerAgent\catalog\ObjectFactory.java

2. Az XML sémák Java-kontextusának kigenerálását a Java forráskódok fordítása követi. Ehhez a következő parancsokat írjuk be az `X:\jade` könyvtárban.

```
compile src\milab\labxy\bookbuyeragent\bookbuyeragent.java
compile src\milab\labxy\booksselleragent\booksselleragent.java
```

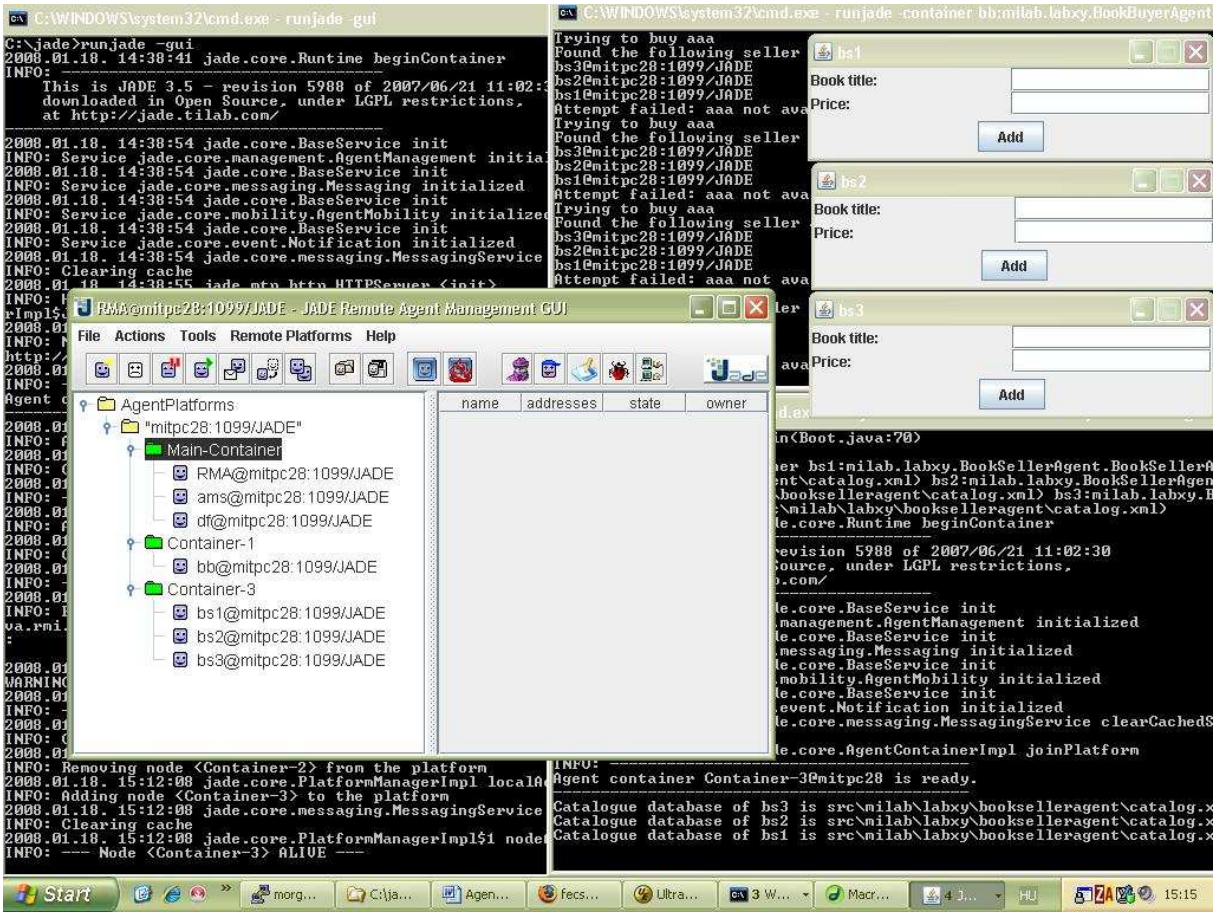
Ez tehát a lényeg: a 2 felhasználói ágens. E két ágens tetszőleges sorrendben történő lefordításával (hiszen az ágensek kódja ideális esetben nem függhet egymástól) minden lefordul, aminek csak le kell fordulnia. Mivel az ágensek által importált osztályok/csomagok egyike sem használja a `milab.DataHandler` osztályt, ezért ennek kivételével minden „`milab`" package-en (és annak subpackage-ein) belül található forráskód automatikusan, és megfelelőképp lefordul.

3. A sikeres fordítást az ágensek futtatása követi. Ehhez a következő lépéseket célszerű megtenni.

a. Nyissunk meg (ha még nincs nyitva) egy szabad parancssori ablakot, és lépjünk be az `X:\jade` könyvtárba.
b. Írjuk bele a következő utasítást: `runjade -gui`
c. Vagy a feljövő RMA ágens GUI-ján keresztül, vagy külön parancssori ablakból, vagy egy, a platformot futtató géptől különböző gazdagép (host) parancssori ablakából indítsunk el egy KönyvVásárló ágenst.
   i. Az RMA GUI-ján keresztül indítva először is kattintsunk a „Start New Agent" menüpontra (vagy ikonra), majd adjuk meg az ágens nevét, osztályát (`milab.labxy.BookBuyerAgent.BookBuyerAgent`), és annak a könyvnek a címét, amit keresnie kell (pl. `aaa`).
   ii. A KönyvVásárló ágens külön parancssori ablakból történő indításához a következőt kell beírnunk az `X:\jade` könyvtárban: `runjade -container bb:milab.labxy.BookBuyerAgent.BookBuyerAgent(aaa)`
   iii. A KönyvVásárló ágens egy, a platformot futtató géptől különböző gazdagép parancssori ablakából történő indításához a következőt kell beírnunk a másik gép parancssori ablakába az `Y:\jade` könyvtárban: `runjade –host platformot_futtato_host_cime -container bb:milab.labxy.BookBuyerAgent.BookBuyerAgent(aaa)`
d. A KönyvVásárló ágens indításához hasonlóan (akár azzal egyidőben, akár csak ugyanabban a konténerben) hozzunk létre 3 KönyvÁrus ágenst is a platformon. Ügyeljünk arra, hogy az ágensek neve egyedi legyen, és helyesen adjuk meg az ágensek osztályát (`milab.labxy.BookSellerAgent.BookSellerAgent`), továbbá ne feledkezzünk meg arról, hogy immár a KönyvÁrus ágensek számára is meg kell adni egy kezdőértéket, a kiindulási adatbázisuk helyét. [13] A fentebb ismertetett felállás mellett ez a köv.: `src\milab\labxy\booksselleragent\catalog.xml`
e. Ezután töltögessük a KönyvÁrus ágensek adatbázisát úgy, hogy a KönyvVásárló által keresett könyv (is) megjelenjen bennük, és így megtörténjen a tranzakció. [14]

[13] Igazság szerint az **ágensek teljes autonómiáját feltételezve** ideális esetben nem is kellene interaktív GUI-t létesíteni számukra. Az ágensek induláskor kapnának egy, vagy több kezdeti paramétert, majd autonóm módon működnének „halálukig". Ebben az esetben egy GUI legfeljebb azt a célt szolgálhatná, hogy emészthető formában tálaljon bizonyos információkat a felhasználó számára. Az ágensek működését azonban, élve a teljes autonómia feltevésével, a felhasználó sohasem befolyásolhatná direkt módon (esetleg csak indirekte, más, nem autonóm ágens „bőrébe bújva", stb). *Ez tehát egyfajta „technokrata deizmus" ágens-analógiája, azt hiszem…* ☺

[14] Valójában, vegyük észre (főleg, ha egyetlen gépen futtatjuk az összes ágenst), hogy a KönyvÁrus ágensek egy, és ugyanazon adatbázist használják elosztottan. Természetesen lehetőség van ágensenként különböző adatbázisok használatára.



2. ábra: példa az AGENTFRAME minta-ágenseinek működésére

Innentől fogva a „felszínen” lényegében minden majdnem ugyanúgy zajlik, mint ahogyan azt a 2.3-as szakaszban láttuk. De „*fecseg a felszín, hallgat a mély*”, mivel a működés bizonyos részei immár jelentős mértékben különböznek attól, amit az előzőekben láttunk (az eredeti mintapéldától). Ilyen például a KönyvÁrus ágens adatbázis-használata, és az XML alapú üzenetváltás. A következőkben főként ezekről a különbségekről/újdonságokról ejtünk szót.

2.5.2 AGENTFRAME használata

Az AGENTFRAME-et nyilván fejlesztési célokra, nem pedig önmagában véve szeretnénk használni, hiszen önmagában – felhasználói szempontból nézve – nem bír több funkcionalitással, mint a JADE-hez eredetileg mellékelt `examples.bookTrading` mintapélda.

Az AGENTFRAME számunkra tehát azért érdekes, mert alapja és irányadója lehet az egyes mérésekhez kialakított, saját fejlesztésű ágenseknek. Ahhoz, hogy az AGENTFRAME-ből kiindulva saját fejlesztésű ágenseket hozhassunk létre, nyilván először meg kell értenünk a csomagban foglalt ágensek (`milab.labxy.BookBuyerAgent.BookBuyerAgent` és `milab.labxy.BookSellerAgent.BookSellerAgent`) kódját. Ebben a szakaszban tehát arra teszünk kísérletet, hogy – az `examples.bookTrading` mintapélda megértését adottnak feltételezve – kihangsúlyozzuk a főbb különbségeket és szempontokat, amikről eddig még nem esett szó.

Az AGENTFRAME minta-ágensei 2 fő szempontból különböznek a 2.3-as szakaszban megismert minta-ágensektől.

1. **A KönyvÁrus ágensek a könyvkatalógust immár nem Hashtable objektumban tárolják, hanem egy XML adatbázisban.** Így ennek kapcsán a teljes adathozzáférési és kezelési rész lecserélődött bennük (JAXB alapú megoldásra).
2. **A KönyvÁrus, és KönyvVásárló ágensek közti üzenetváltásban szereplő üzenetek tartalma immár kizárólag XML alapú.** Így mindkét ágenstípus kódjában kicserélődtek azok a részek, melyek az egymás közti ACL (Agent Communication Language) üzenetek tartalmának olvasását, írását valósították meg.

Az ágensek forráskódján végighaladva láthatjuk, hogy az ACL üzenetek tartalmának beállítása (`setContent`), és olvasása (`getContent`) esetén, előbb-utóbb mindig alkalmas XML feldolgozási lépések történnek. Ezt illusztrálja a következő kódrészlet.

```
1. ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
2. for (int i = 0; i < sellerAgents.length; ++i)
   {cfp.addReceiver(sellerAgents[i]);}
3. milab.labxy.BookBuyerAgent.language01.ObjectFactory factory = new
   milab.labxy.BookBuyerAgent.language01.ObjectFactory();
4. JAXBElement<String> contentElement = factory.createContent(targetBookTitle);
5. JAXBMarshaller jaxbMarshaller = new JAXBMarshaller();
6. cfp.setContent(jaxbMarshaller.generateXMLString("UTF-8", contentElement,
   "milab.labxy.BookBuyerAgent.language01"));
7. cfp.setConversationId("book-trade");
8. cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
9. myAgent.send(cfp);
```

Itt látható az, amiként a KönyvVásárló ágens üzenetet konstruál a megtalált KönyvÁrus ágenseknek. **(1)** létrehozza a megfelelő performatívájú ACL üzenet objektumot; **(2)** az üzenet címzett mezőjébe rendre belehelyezi az ismert KönyvÁrus ágensek AID azonosítóját; **(3)** létrehoz egy, saját `language01` nyelvének megfelelő `ObjectFactory` objektumot; **(4)** a létrejött objektum `createContent` üzenetével létrehozza a `language01.xsd`-ben definiált, „`content`” nevezetű, String típusú elem Java megfelelőjét.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="content" type="xsd:string"/>
</xsd:schema>
```

Vegyük észre, hogy mivel ez az elem elemi (azaz nem összetett), ezért típusa is elemi (most éppen `String`). Tehát létrehozásakor a `factory.createXXX` metódus bemenete `String` lesz. Ez általában is, tetszőleges típusra, és elemre érvényes (legyen akár elemi, akár összetett).

A következő lépésben **(5)** létrehozunk egy `JAXBMarshaller` objektumot, hogy később XML formára tudjuk hozni az iménti `JAXBElement<String>` osztályú objektumot; **(6)** a `JAXBElement<String>` osztályú objektumból kapott XML szintaxisú `String`-et behelyezzük az ACL üzenet tartalmi részébe; **(7)** beállítjuk a beszélgetés azonosítóját; **(8)** beállítjuk az egyedi időbélyeget, aminek alapján majd meg fogjuk tudni különböztetni az üzenetre érkező választ a többi bejövő üzenettől (`MessageTemplate`-ek használatával); végül **(9)** elküldjük az üzenetet.[15]

[15] **FONTOS**: ebben a lépésben célszerű felfigyelni a „`myAgent`” kulcsszóra! Az egyes viselkedés objektumok gyakorlatilag csak így tudnak hivatkozni az őket létrehozó ágensre. **EZ NAGYON FONTOS!!!** …a `myAgent` változó egyébként a `Behaviour` osztály saját belső változója, amit paraméterezett konstruktora inicializál.

Az üzenetek fogadása során a küldő nyelvéhez igazodva, az előzőekhez hasonló módon UnMarshall-oljuk a beérkezett (és `MessageTemplate` szűrőn átesett) üzeneteket. Ennyi.

A másik fő különbség – az üzenetek tartalmának XML (de)kódolásán túl – a KönyvÁrus ágensek könyvkatalógusának kezelése. A könyvkatalógus immár egy XML fájlban foglal helyet, amit egy alkalmas XSD ír le. Ez az XSD tekinthető az XML adatbázis séma-definíciójának (az „adatbázis" struktúráját adja meg).

Az AGENTFRAME jelen verziójában a KönyvÁrus ágens könyvkatalógus-adatbázisának séma-definíciója a következő.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="catalog" type="catalogType"/>
 <xsd:complexType name="catalogType">
  <xsd:sequence>
   <xsd:element ref="book" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:element name="book" type="bookType"/>
 <xsd:complexType name="bookType">
   <xsd:sequence>
   <xsd:element name="title" type="xsd:string"/>
   <xsd:element name="price" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="rating" type="xsd:string"/>
 </xsd:complexType>
</xsd:schema>
```

Ez a séma valamivel egyszerűbb, mint az előbbi, JAXB-t bemutató szakaszban ismertetett `fooagent`-es séma, ezért részletes elemzésétől most eltekintünk.

Visszatérve: a lényeg, hogy – az `examples.bookTrading` mintapéldához képest – a `BookSellerAgent` osztály belső változói megváltoztak, továbbá `updateCatalog` metódusának implementációja is módosult. Sőt, immár külön `searchCatalog`, és `removeCatalog` metódus is tárult hozzá.

Vegyük észre, hogy az `updateCatalog`, és a `removeCatalog` metódusok nem csak az ágens belső változóit módosítják, hanem magát az XML adatbázist is.

A `searchCatalog` és `removeCatalog` teljesen szokványos metódusa a `BookSellerAgent` osztálynak. Viszont az `updateCatalog` metódus továbbra is egy újabb (`OneShotBehavior` típusú) viselkedés létrehozásával valósítja meg feladatát. Erre azért van szükség, mert ezt a függvényt az ágens GUI-ja, avagy a `BookSellerGui` osztály példányai hívogatják.[16]

Lényegében csakis olyankor történik meg az `updateCatalog` metódus hívása, amikor a GUI-n egy adatbeviteli eseményt érzékel az operációs rendszer, azaz amikor a

```
JButton addButton = new JButton("Add");
```

gombhoz hozzáadott

[16] Az ágensek viselkedése, és a GUI külön szálon (az eventDispatcher thread-en) fut.

```
addButton.addActionListener( new ActionListener() {

    public void actionPerformed(ActionEvent ev) {

            try {

                    String title = titleField.getText().trim();
                    String price = priceField.getText().trim();
                    myAgent.updateCatalog(title, Integer.parseInt(price));
                    titleField.setText("");
                    priceField.setText("");

            } catch (Exception e) {

                    JOptionPane.showMessageDialog(BookSellerGui.this,    "Invalid    values.
"+e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);

            }

    }

});
```

esemény-figyelő (ActionListener) lefut. Vessünk most egy mélyebb pillantást a GUI kódjára! Haladjunk végig, sorról-sorra, és értelmezzük az egyes utasításokat. A `BookSellerGui` osztály deklarációját a következő sorok előzik meg.

```
1. package milab.labxy.BookSellerAgent;
2. import jade.core.AID;
3. import java.awt.*;
4. import java.awt.event.*;
5. import javax.swing.*;
```

(1) a jelen GUI-t tartalmazó csomag nevének megadása; **(2)** a JADE keretrendszer AID ágensazonosító-osztályának betöltése; **(3)** az **AWT** (**A**bstract **W**indow **T**oolkit) csomag által közvetlenül tartalmazott összes osztály betöltése, melyek rajzokat és képeket megjelenítő grafikus felhasználói felületek készítésére alkalmasak; **(4)** az AWT csomag eseménykezelő al-csomagjában található interfészek és osztályok betöltése, melyek lehetővé teszik az AWT komponensek által generált különböző típusú események (pl. gombnyomások, egérkattintások) kezelését; **(5)** a Java Swing elnevezésű[17], platform-független[18], AWT-re építő, „pehelysúlyú" API-jának a betöltése.

Lényegében tehát az AWT és a Swing csomagok osztályainak betöltése ad lehetőséget a GUI létrehozására. A Swing-ről bővebben például a következő elérésen lehet olvasni.

`http://java.sun.com/docs/books/tutorial/uiswing`

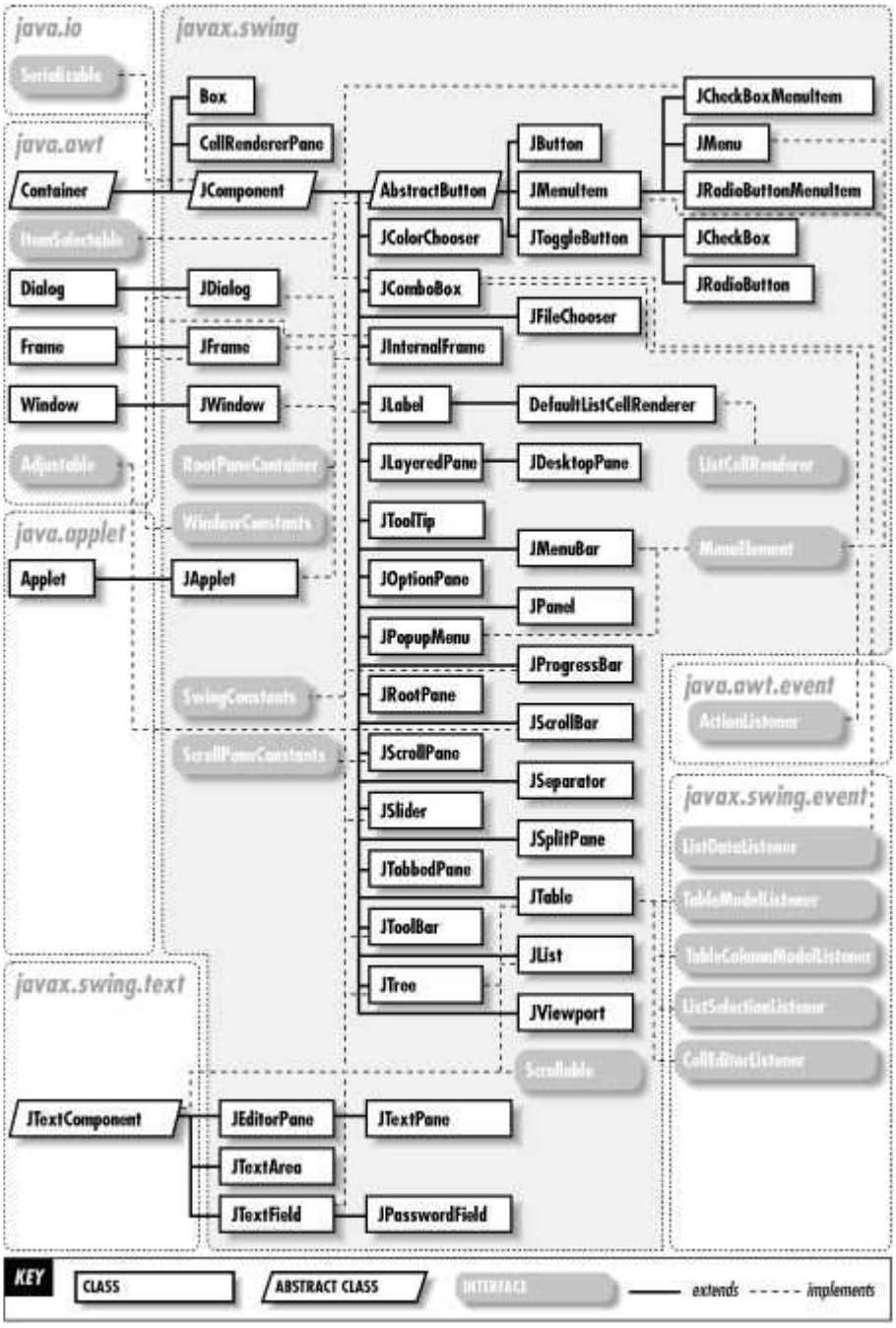
A különböző csomagok és osztályok betöltését a `BookSellerGui` osztály deklarációja követi. Látható, hogy ez az osztály a Swing-es `JFrame` osztály leszármazottja. Az összes Swing-es komponens (osztály, interfész, stb) egzakt leírása a Java API dokumentációjában (`http://java.sun.com/javase/6/docs/api`) olvasható a `javax.swing` csomagra kattintva.

A `JFrame`–et lényegében egy Windows-os program/alkalmazás ablakaként célszerű elképzelni (habár nyilván platform-független komponensről van szó). Egy alkalmazás adott esetben akár több ablakból is állhat, számunkra azonban ez most nemigen lényeges, hiszen a kidolgozandó laborgyakorlatok során nem a grafikus felületé lesz a főszerep…

[17] A Swing elnevezés nem rövidítés – pusztán csak a Jazz egy irányzatáról nevezték el a projektet.
[18] …és itt nyilván nem JADE-es ágens-platformra, hanem operációs rendszerre célszerű gondolni.

A Swing-es komponensek az AWT komponenseitől – elnevezés tekintetében – többnyire csak annyiban térnek el, hogy „`J`” betűvel kezdődik a nevük. Példának okáért a `JFrame` osztály az AWT-s `Frame` osztály leszármazottja. Például, mikor egy `BookSellerGui` objektumot hozunk létre, akkor gyakorlatilag egy `JFrame` objektum keletkezik, avagy egy AWT-s `Frame`.

Az AWT widget-jei (gombok, beviteli mezők, stb) a Swing-től eltérően „nehézsúlyúak” (nem mind tisztán Java alapúak), ezért se nagyon foglalkozzunk most ezekkel. Koncentráljunk inkább a Swing-re! A `javax.swing` csomag elemei a következők.

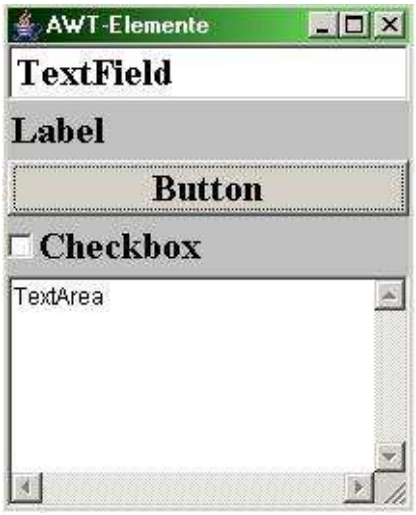


3. ábra: a `javax.swing` csomag elemei

Ebből természetesen nincs szükség az összes elem (osztály) ismeretére. A főbb osztályok, melyek ismerete kiindulásképp elegendő, a következők.

- `JFrame` (alkalmazás ablaka)
- `JPanel` (alkalmazás ablakának egy része)
- `JButton` (gomb)
- `JLabel` (felirat – nem szerkeszthető)
- `JTextField` (szöveges ki/beviteli mező – sor)
- `JTextArea` (szöveges ki/beviteli terület – téglalap)
- `JCheckBox` (kipipálható négyzet – egyszerre több is bejelölhető)
- `JRadioButton` (rádiógomb – egyszerre csak egy jelölhető ki)

Ezek közül szemléltet néhányat a következő ábra.



4. ábra: példa néhány fontosabb widget-re

A `BookSellerGui` osztály, mint látni fogjuk, még ezeknek is csak egy kis töredékét használja csak fel. Az osztály implementációja 3 fő részből tevődik össze: **(1)** belső változók megadása; **(2)** konstruktor; és **(3)** a `show` metódus implementációja.[19]

Az osztály belső változói a következők.

```
1. private BookSellerAgent myAgent;
2. private JTextField titleField, priceField;
```

Az első változó **(1)** a `myAgent`. Látható, hogy ennek típusa/osztálya `BookSellerAgent`. Viszont a `BookSellerAgent` osztályt nem importáltuk be. Szerencsére ez nem jelent gondot, mivel ugyanabban a csomagban van, mint a jelenleg vizsgált kód, a GUI kódja. A változót egyébként a `BookSellerGui` osztály konstruktora fogja incializálni. A következő két változó **(2)** két szöveges beviteli mezőnek felel meg (egyelőre ugyancsak inicializálatlanul).

A `BookSellerGui` osztály konstruktora felelős a grafikus felület elemeinek megkonstruálásáért, és azok elrendezéséért.

[19] Itt gyakorlatilag a `JFrame.show` metódus újra-implementálásáról van szó. Polimorfizmus…

```
1. BookSellerGui(BookSellerAgent a) {
2.        super(a.getLocalName());
3.        myAgent = a;
4.        JPanel p = new JPanel();
5.        p.setLayout(new GridLayout(2, 2));
6.        p.add(new JLabel("Book title:"));
7.        titleField = new JTextField(15);
8.        p.add(titleField);
9.        p.add(new JLabel("Price:"));
10.       priceField = new JTextField(15);
11.       p.add(priceField);
12.       getContentPane().add(p, BorderLayout.CENTER);
13.       JButton addButton = new JButton("Add");
14.       addButton.addActionListener(new ActionListener() {...});
15.       p = new JPanel();
16.       p.add(addButton);
17.       getContentPane().add(p, BorderLayout.SOUTH);
18.       addWindowListener(new WindowAdapter() {...});
19.       setResizable(false);}
```

Az első sorban **(1)** láthatjuk, hogy a konstruktor bemenete egy `BookSellerAgent` objektum(ra mutató referencia[20]). Ha belepillantunk a `BookSellerAgent` kódjába, láthatjuk, hogy a

```
private BookSellerGui myGui;
```

belső változót a `setup` metódus hívja meg – a kezdetek kezdetén – a következőképp.

```
myGui = new BookSellerGui(this);
myGui.show();
```

Magyarán az ágens az önmagára mutató referenciát (`this`) adja át a GUI objektum konstruktorának. Ennek felhasználásával **(2)** megtörténik a GUI ablak (alkalmazás/program ablak felső peremének) elnevezése.

Az itt látható `super` hívás a szülő (most éppen a `JFrame` objektum, amiből a GUI objektum leszármazik, amit kiterjeszt (`extends`)) konstruktorára vonatkoznak.[21] Tehát az `a.getLocalName()` `String`-gel, mint bemenettel meghívjuk a szülő konstruktorát (amit ilymódon örököltünk).

A következő sor **(3)** ugyancsak a konstruktor bemenetén kapott ágens objektumot használja. Most éppen arra, hogy beleírja a `BookSellerGui` saját `myAgent` változójába.

Ezek után **(4)** létrehozunk egy új panelt, majd **(5)** beállítjuk az elrendezését. Itt most éppen 2*2-es GridLayout-ra állítottuk, ami azt jelenti, hogy egy 2*2-es táblázat-szerű elrendezésben adhatjuk hozzá az egyes widget-eket a panelhez. A GridLayout-ról bővebben itt lehet olvasni: `http://java.sun.com/j2se/1.4.2/docs/api/java/awt/GridLayout.html`

A GridLayout egyébként nem az egyetlen lehetséges elrendezési fajta, és ráadásul nem is Swing-es, hanem AWT-s. A további fontosabb Layout-ok a következők.

[20] A Java-ban ugyebár nincsenek explicit pointer-ek (ellenben pl. a C++-szal)…
[21] Lásd. http://java.sun.com/docs/books/tutorial/java/IandI/super.html

- `BorderLayout` (http://java.sun.com/j2se/1.4.2/docs/api/java/awt/BorderLayout.html)
- `CardLayout` (http://java.sun.com/j2se/1.4.2/docs/api/java/awt/CardLayout.html)
- `FlowLayout` (http://java.sun.com/j2se/1.4.2/docs/api/java/awt/FlowLayout.html)
- `GridBagLayout` (http://java.sun.com/j2se/1.4.2/docs/api/java/awt/GridBagLayout.html)

A felsoroltakból talán a `FlowLayout` a legegyszerűbb (`LayoutManager`). Balról jobbra rendezi az újabb, és újabb hozzáadott komponenseket (pl. widget-eket). Ennél talán egy fokkal bonyolultabb a `CardLayout`, amely egymásra pakolja a komponenseket, amikből csak a legfelső (mint egy kártyapakli legfelső lapja) látszik csupán. Természetesen lehet állítani a kártyák sorrendjét. A `BorderLayout` valamivel összetettebb elrendezést is megenged már. Itt északra, keletre, délre, nyugatra, és középre helyezhetünk komponenseket. A legösszetettebb, és egyben legnagyobb tudású `LayoutManager` pedig talán a `GridBagLayout`. Lényege, hogy különböző dimenziójú (magasságú és szélességű) komponensek szisztematikus elhelyezését teszi lehetővé.

Visszatérve a `BookSellerGui` kódjára **(6)**, a hatodik sorban azt láthatjuk, amit `p` panelünkhöz hozzáadjuk az első komponenst: egy `JLabel` címkét. Ez tehát jelen esetben a panel bal felső sarkába kerül, de (ITT HÍVJUK FEL A FIGYELMET – **FONTOS!!!**) a panel módosítása egyelőre semmiféle vizuális „mellékhatással" nem jár. A képernyőn e műveletek hatására semmi sem jelenik meg. Arról van szó ugyanis, hogy ennek az egész megjelenítési architektúrának a „filozófiája" az **MVC** (**M**odell-**V**iew-**C**ontrol) elgondoláson alapul, amely különválasztja a GUI modelljét, megjelenítését, és vezérlését. Ennek haszna nyilvánvaló, nem szorul további indoklásra. A modell tehát, mint látjuk, például egy `JPanel` objektum kialakítása, egy `JFrame` objektum „benépesítése", az alkalmazás logikájának „behuzalozása", stb. Ezek után, ha már kész a „modell", jöhet a megjelenítés. Ez lesz a `show` metódus feladata. Végül pedig a megjelenített GUI a felhasználói interakció során vezérel, és vezéreltetik.

A hetedik sorban **(7)** azt láthatjuk, hogy a létrehozott widget, pontosabban a 15 karakter hosszú szöveges beviteli mező az előbbi címkével ellentétben programszinten nevesítve van. Ennek oka abban keresendő, hogy itt most a `BookSellerGui` osztály `titleField` belső változójának inicializálásáról van szó. Erre a változóra nyilván a későbbiekben (ha valaminek a hatására változik a GUI, vagy ha a GUI hatására kell valaminek megváltoznia az említett grafikus komponens tartalmától függően) még szükségünk lehet.

A komponens létrehozását **(8)** annak panelre helyezése követi. Ezt **(9)** a panel bal alsó szekciójának címkével való ellátása követi. A **(10)-(11)** az előbbi (7)-(8) lépésekhez hasonló, így erre most nem térünk ki. Viszont a következő lépés **(12)** már sokkal érdekesebb. Itt a szülő `JFrame` objektumtól örökült `getContentPane` metódus kerül meghívásra. Sőt, nem is (csak) ez a metódus, hanem az általa visszaadott `Container` objektum `add` metódusa. Ez a metódus helyezi el ugyanis az előbbiekben előállított panelünket a program-ablak közepére (lásd. `BorderLayout.CENTER`).

Felmerülhet a kérdés, hogy: *miért nem a* `JFrame` *objektum* `add` *metódusát használtuk, minek bonyolítani?* A válasz egyszerű: használhattuk volna azt is, de nem célszerű. Bővebben itt olvashatunk erről: `http://java.sun.com/docs/books/tutorial/uiswing/components/toplevel.html`

Az imént elkészült panelünk `Frame`-re helyezését **(13)** egy `JButton` gomb-objektum létrehozása követi. A gomb azért van nevesítve, hogy a következő lépésben **(14)** hozzá lehessen adni egy `ActionListener` objektumot. Ez az objektum hordozza magában azt az

információt, amely megmondja, hogy mit kell tenni a gomb lenyomásakor. Erre kicsit később, a szakasz vége felé még kitérünk.

Van tehát egy gombunk (`addButton`), de mit kezdjünk vele? Netán pakoljuk fel az előző panelre? De hát azt a panelt már elhelyeztünk a `Frame`-en… Hozzunk esetleg létre egy újabb panelt, és arra tegyük? A válasz: igen, ezt láthatjuk a **(15)-(16)** számú lépésekben.

Ezek után **(17)** már-már szokás szerint feltesszük ezt a(z egyetlen egy gombot tartalmazó) panelt a `Frame` `contentPane`-jére, ráadásul déli fekvéssel (`BorderLayout.SOUTH`).

A következő lépésben **(18)** a szülő `JFrame` objektumtól örökölt `addWindowListener` metódust használva egy – a `WindowListener` interfészt implementáló – `WindowAdapter` esemény-figyelő objektumot adunk a GUI-hoz, hogy az operációs rendszer program-ablakától érkező különböző eseményeket meg tudjuk fogni, és le tudjuk őket kezelni (hasonlóan a (14) lépésben látottakhoz). Pillanatokon belül erről is szót ejtünk.

A `BookSellerGui` objektum/osztály konstruktorának utolsó utasítása **(19)** a GUI átméretezhetőségét állítja hamisra.

Ezzel tehát át is tekintettük a `BookSellerGui` konstruktorát. Az áttekintés során azonban mellőztük az esemény-kezelés részleteit, pedig hát talán ez a legfontosabb rész (az MVC C-betűje). Az eseménykezelés köti össze az alkalmazás logikáját a grafikus felülettel. Lássuk először is a (13) számú lépésben létrehozott `addButton` gombhoz (14) számú lépésben „hozzáhuzalozott” eseménykezelést.

```
addButton.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent ev) {

            try {

                    String title = titleField.getText().trim();
                    String price = priceField.getText().trim();
                    myAgent.updateCatalog(title, Integer.parseInt(price));
                    titleField.setText("");
                    priceField.setText("");

            } catch (Exception e) {

                    JOptionPane.showMessageDialog(BookSellerGui.this,    "Invalid    values.
"+e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);

            }

    }

});
```

Itt tehát a GUI `addButton` gombjának `addActionListener` metódusával egy `ActionListener` objektumot adunk a gombhoz. Az `ActionListener` objektumot nem nevesítjük, mivel a későbbiekben úgysem fogunk rá hivatkozni – csak most, ebben a pillanatban van rá szükség.

Ebben a névtelen `ActionListener` objektumban egyetlen különlegesség van csupán: az `actionPerformed` metódusának a felüldefiniálása/implementációja. E metódusnak nincs kimenete (`void`), viszont „mellékhatásai” annál inkább. Bementére egy `ActionEvent` osztályú esemény érkezik. Most ez nyilván csakis a gomb lenyomásából eredhet.

Ha tehát az `ActionListener` eseményt „érzékel”, akkor lefut ez az `actionPerformed` metódusa. Először is a `try` résszel próbálkozunk, aztán hiba (`exception`) esetén jöhet a `catch` rész.

Az első két lépésben létrehozunk két – az `actionPerformed` metódus számára – lokális, `String` típusú változót, melyeket nyomban inicializálunk is a `BookSellerGui` megfelelő belső változóinak értékével. Pontosabban nem is e változók értékével, hanem e – `JTextField` típusú – változók `getText` metódusa által visszaadott `String` típusú objektumainak `trim` metódusa által visszaadott értékével, magyarán azokkal a – kezdő és záró szóközöktől mentesített – karakterfüzérekkel, melyek a GUI megfelelő szöveges beviteli mezőiben a program futása során aktuálisan szerepelnek. Röviden: beolvassuk a GUI-ról a megfelelő mezők tartalmát.

A beolvasott tartalommal immár meghívható a GUI-t létrehozó `BookSellerAgent` ágens-példány `updateCatalog` metódusa. Persze arra azért figyelni kell, hogy e metódus második argumentuma `Integer` osztályú kell, hogy legyen. Ezért aztán a `price String`-et a fent látható módon `Integer`-ré alakítjuk.

A következő két utasítás pedig egyszerűen csak üresre állítja a GUI megfelelő két szöveges beviteli mezőjét, lévén be lett olvasva a tartalmuk.

Ha a fentiek során semmiféle hiba sem történt, úgy az `actionPerformed` metódus végrehajtása véget ér. Egyébként a `catch` ágba kerülünk, ahol teljesen szokványos módon lekezeljük a kivételt. Ezen, ha már majd a saját ágensünket konstruáljuk, gyakorlatilag csak minimálisan kell majd változtatni (értelemszerűen át kell írni a megfelelő objektumok nevét).

Ezzel tehát a GUI-hoz kötöttük a szükséges alkalmazás-logikát. Mindazonáltal ezzel még nem vagyunk kész. A GUI konstruktorában még egy esemény-kezelés látható. Előbb már erről is esett szó. A (18) számú lépés:

```
addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
            myAgent.doDelete();
      }
});
```

Itt tehát a `BookSellerGui` objektum `JFrame`-től örökölt `addWindowListener` metódusának felhasználásával adunk egy új `WindowAdapter` esemény-figyelőt a GUI-hoz. Erre azért van szükség, mert az eddigi események nem fedtek le minden eshetőséget. Eddig csak az alkalmazás logikájával kapcsolatos eseményeket kezeltük. Most viszont az operációs rendszer azon eseményeit is megpróbáljuk lekezelni (és alkalmazásunk logikájához igazítani), amelyek majd a program futása során adódhatnak.

Ilyen esemény például az, amikor a felhasználó a program-ablakot az ablak jobb felső sarkában található X gombra kattintva bezárja. A `WindowAdapter` objektum `windowClosing` metódusának fentebbi felüldefiniálása/implementációja épp ezt az esetet kezeli tehát. Ilyenkor egész egyszerűen az ágens `jade.core.Agent` osztálytól örökölt `doDelete` metódusa fut le.

A `doDelete` metódus gyakorlatilag leállítja (megsemmisíti) az ágens objektumot. A JADE API-ja a következőt írja: *„Make a state transition from* active*,* suspended *or* waiting *to* deleted *state within Agent Platform Life Cycle, thereby destroying the agent."* Ennél részletesebben a JADE programozói specifikációiból tájékozódhatunk a metódust illetően.

Végezetül térjünk rá a `BookSellerGui` osztály utolsó, harmadik szakaszára: a megjelenítésre.[22] Ezt gyakorlatilag az osztály `show` metódusa valósítja meg. Ezt hívja meg maga a `BookSellerAgent` ágens is egyébként (mint már láthattuk).

```
public void show() {

        pack();
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int centerX = (int)screenSize.getWidth() / 2;
        int centerY = (int)screenSize.getHeight() / 2;
        setLocation(centerX - getWidth() / 2, centerY - getHeight() / 2);
        super.show();

}
```

A metódus első utasítása a `pack`. Ez az utasítás automatikusan, a `LayoutManager`-nek megfelelően méretezi a grafikus komponenseket. Természetesen ezt mi magunk is, egyenként, direkt módon megtehetnénk az alkalmas `setSize` metódusok meghívásával. Ez utóbbi megoldás azonban nem javallott (túl azon, hogy sokkalta kényelmetlenebb, mint az előbbi).

A második utasítás gyakorlatilag – szabványos módon – egy `java.awt.Dimension` osztályú objektum-példányt hoz létre. Ez az objektum a képernyő pixelekben vett szélességét és magasságát tartalmazza. A következő lépésekben ezen értékek kiolvasásával meghatározásra kerül a képernyő középpontja. Ezt követi a `JFrame`-től örökölt `setLocation` metódus alkalmas meghívása. Ezen belül a `getWidth` és `getHeight` metódusok ugyancsak a `Frame`-hez tartoznak, és rendre a program-ablak szélességét és magasságát adják vissza pixelekben.

Tehát a `setLocation` metódus fentebbi meghívása valóban a képernyő közepére helyezi a program-ablakot. Ezek után az utolsó lépés a valódi megjelenítés. Ezt az úgynevezett konstruktorok láncolásával (constructor chaining) érjük el: `BookSellerGui.show` → `javax.swing.JFrame.show` → `java.awt.Frame.show` → `java.awt.Window.show` → `java.awt.Container.show` → `java.awt.Component.show` – láncban egymást hívják a konstruktorok…

Számunkra azonban mindez csupán egyetlen, végső utasítás: `super.show();`

[22] Ez van – a „látványos(abb)" dolgok mindig a végére maradnak az összképet meghatározandó…még ha esetleg a teljes egész nyúlfarknyi részéről is van szó csupán… Sajnos azonban legtöbbször hiányzik az idő/energia/motiváció, hogy az ember a javát (vagy Java-t?) is megértse. – Reméljük ezegyszer nem így lesz! ☺

3 Összefoglalás

Ebben az anyagban az „Elosztott Intelligens Rendszerek” c. B.Sc. szakirány-laboratórium alapját képező AGENTFRAME csomag-gyűjteménnyel ismerkedhettünk meg, amely kiindulásként szolgál a labor méréseit fejlesztő kollégák számára.

A csomag megismeréséhez először is röviden áttekintettük keletkezésének történetét (a bevezetőben), és a csomaggal szemben előzetesen támasztott követelményeket. Ezt követően rátértünk a csomag futtatásához szükséges szoftveres és dokumentációs háttér vizsgálatára, majd a szoftverek telepítésére, és futtatására.

Ezeket az alapozó jellegű szakaszokat további két szakasz követte, melyekben rendre megismerkedhettünk az AGENTFRAME alapját képező „gyári” mintapéldával, illetve az XML-ek feldolgozásához használt JAXB csomag működ(tet)ésének elveivel.

Végül, ezen összefoglaló előtt, az előző szakaszokra alapozva, igen részletesen áttekintettük az AGENTFRAME felépítését és működ(tet)ését. Az összefoglalót a függelék követi, ahol jelenleg az Eclipse-JADE integráció kerül górcső alá.

4 Függelék

4.1 Eclipse és JADE

Ebben a szakaszban az előzőekben megismert „fapados” ágensfejlesztési módszer kiegészítéseképp bemutatunk egy *kényelmesebb, opcionális alternatívát*: az **Eclipse IDE** (**I**ntegrated **D**evelopment **E**nvironment) integrált szoftverfejlesztési környezet JADE-del történő integrációját.

Az Eclipse (http://www.eclipse.org) számtalan egyéb szolgáltatása mellett az „Elosztott Intelligens Rendszerek” labor gyakorlataihoz fejlesztendő JADE-es ágensek egyszerű és kényelmes megírására, futtatására, és javítására (is) ad lehetőséget.

A következőkben szó lesz tehát az Eclipse (és számunkra szükséges kiegészítői) telepítéséről, beállításáról, és használatáról. A vázolt megoldás *szándékosan* úgy lett kialakítva, hogy teljes egészében illeszkedjen az anyagban bemutatott „fapados” megoldáshoz – annak egy kiegészítése legyen.

Megjegyzés (FONTOS): természetesen az Eclipse-et – univerzalitásának köszönhetően – a bemutatott megoldástól eltérően is lehet használni. Sőt, használata (a laborgyakorlatok kidolgozói számára) nyilván még csak *nem is kötelező*. Mindazonáltal az előálló forráskódokat és futó állományokat javallott az anyag 2.5-ös szakaszának elején vázoltak szerint strukturálni. Ez azért **nagyon fontos**, mert így a labor anyagai végül mind a „`milab`” nevezetű Java package-ben, szisztematikusan egyesülnek majd. Ennek előnyei nyilvánvalók.

4.1.1 Telepítés

Az JADE-Eclipse integráció első lépése a megfelelő szoftverek, így az Eclipse, és szükséges kiegészítői (plugin-jei) telepítése. Feltételezzük, hogy a JADE és az AGENTFRAME az anyag 2.5.1-es szakaszáig bezárólag látható módon immár telepítve lett.

Az Eclipse telepítése nagyon egyszerű. Bontsuk ki a

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\sw\eclipse-java-europa-fall2-win32.zip

fájlt az X: meghajtó főkönyvtárba. Ennek hatására létrejön egy

X:\eclipse

könyvtár, benne az Eclipse minden alapértelmezett összetevőjével. Ennyi.

Ezen felül 2 kiegészítőre lesz még szükségünk: **(1)** az *EJADE plugin*-re (http://dit.unitn.it/~dnguyen/ejade), és **(2)** az *XJC plugin*-re (https://jaxb-workshop.dev.java.net/plugins/eclipse/xjc-plugin.html).

Az előbbi arra való, hogy a JADE-et, és a JADE-es ágenseket gombnyomásra futtatni tudjuk Eclipse-ből. Az utóbbi hasonlóképp arra való, hogy a 2.4-es szakaszban látottakhoz híven, gombnyomásra futtatni tudjuk az XJC (**X**ML-**J**ava **C**ompiler) fordítót.[23]

A plugin-ek telepítéséhez bontsuk ki a

\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\sw\eclipse-plugin-it.fbk.sra.ejade_0.7.1.zip
\\mostoha\aigroup\oktatas\ElosztottIntelligensRendszerek-BSclabor\sw\eclipse-plugin-org.jvnet.jaxbw_1.0.0.zip

fájlokat az

X:\eclipse\plugins

könyvtárba. Ezzel tehát készen is volnánk – következhet az Eclipse futtatása és konfigurálása.

4.1.2 Beállítások

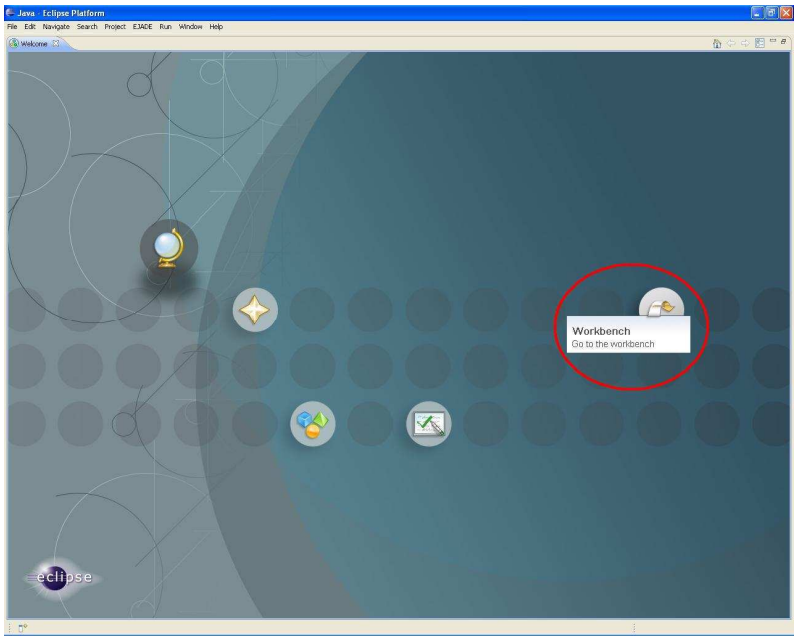
Az Eclipse futtatásához és konfigurálásához az

X:\eclipse\eclipse.exe

fájlt kell elindítanunk. Az egyszerűség kedvéért érdemes lehet egy rá mutató link-et helyezni az asztalra (pl. Windows Intézőben jobb egér-klikk a fájlra, Küldés, Asztal (parancsikon létrehozása)).

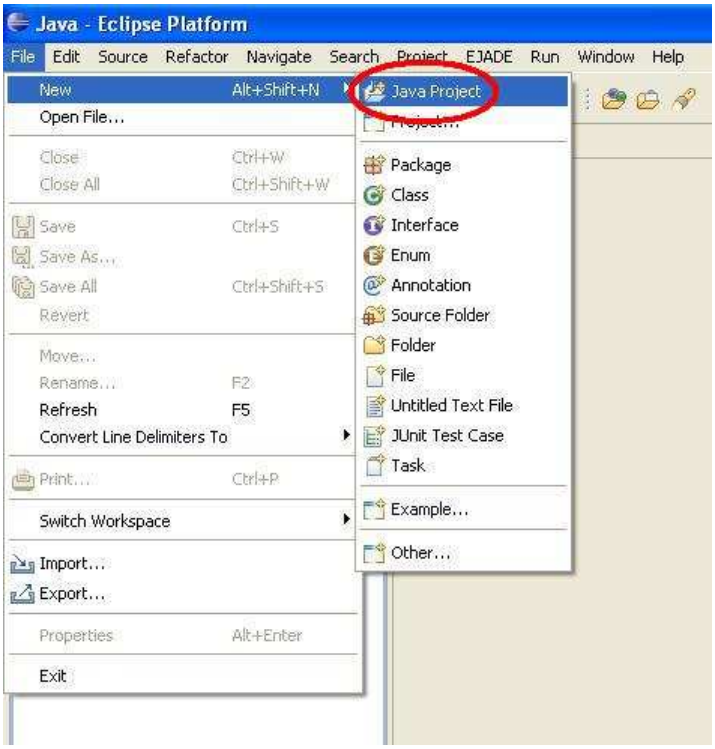
Az indítást követően felugrik egy „*Workspace Launcher*” elnevezésű képernyő. Itt kattintsuk be a „*Use this as default and do not ask again*” feliratú checkbox-ot, és kattintsunk az *OK*-ra.

Ezt követően, rövidesen elindul az Eclipse. A nyitóképernyőn kattintsunk a „*Workbench*”-re.

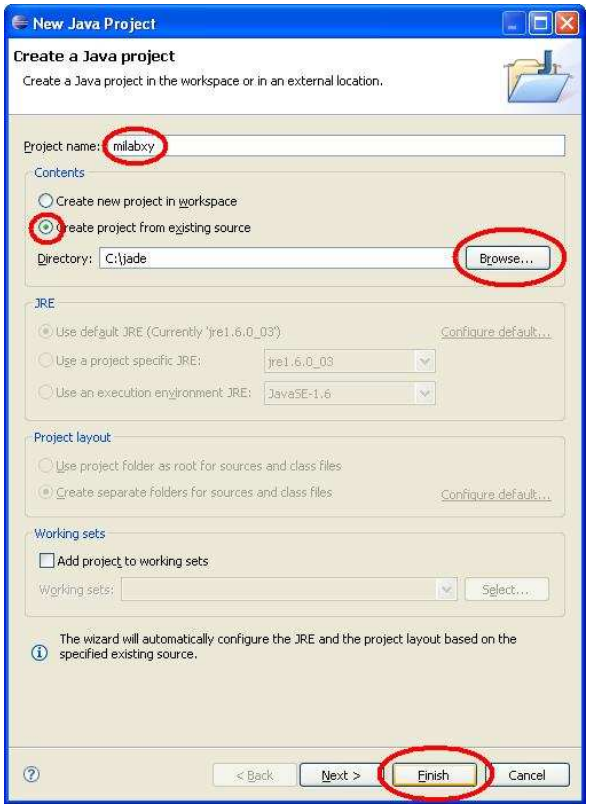


[23] Emlékezzünk csak vissza, hogy ez a fordító szolgált arra, hogy XSD sémákból megfelelő Java-kontextust (azaz lényegében Java-osztályokat) generáljon.

Ennek hatására bejön a munkafelület, ahol a későbbiekben is dolgozni fogunk. Ahhoz, hogy megkezdhessük az érdemi munkát, először is létre kell hoznunk egy projektet, méghozzá egy Java projektet. Kattintsunk a File/New/Java Project menüpontra!

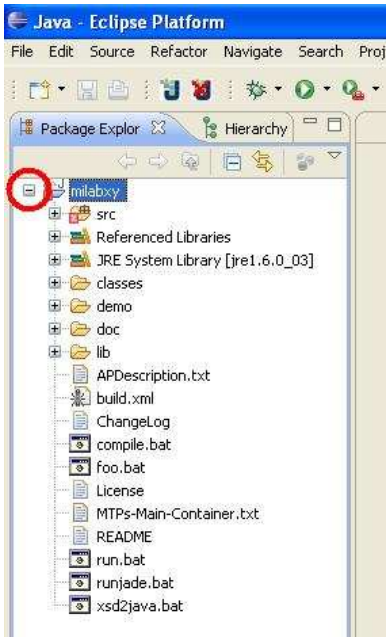


A feljövő képernyőn írjuk be a projekt nevét (pl. *milabxy*), válasszuk ki a „*Create project from existing source*” opciót, böngésszük ki az X:\jade könyvtárat (*Browse*), majd kattintsunk a „*Finish*” gombra.



Ezek után várni kell egy darabig (*Building workspace…*). Az Eclipse megpróbálja feltérképezni, és összefüggéseiben ellenőrizni és lefordítani a kijelölt mappa tartalmát.

Miután beolvasásra kerültek a projektünkhöz szükséges adatok, bal oldalt, a „*Package Explorer*” fül alatt máris láthatóvá válik új projektünk (pl. *milabxy*). Bontsuk is ki a „+” jelre kattintva! Nagyjából a következő látvány tárul elénk.



Gyakorlatilag tehát az X:\jade könyvtár tartalma jelenik meg a „*milabxy*” projekt alatt. **FONTOS**: vegyük észre, hogy az Eclipse elrejti az automatikusan lefordított CLASS fájlokat!

4.1.3 FFF

A szakasz címe nem egy hexadecimális számot sejtet, hanem egy rövidítés: **F**ejlesztés, **F**ordítás, **F**uttatás – ezekkel fogunk most foglalkozni.

Térjünk vissza előbbi projektünkhöz! A projektek forráskódja általában a projekten belül egy „*src*” nevezetű mappába szokott kerülni, míg a lefordított állományok egy „*bin*” mappába generálódnak. Jelen esetben az „*src*” stimmel, azonban nincs „*bin*”. A lefordított állományok ugyanis most alapértelmezésben, rendre a forráskódjuk mellé kerülnek. Ez összhangban van az anyag előző részeivel, ahol az ún. „fapados” JADE/AGENTFRAME használatot ecseteltük.[24]

A „*milabxy*”, és az „*src*” mappa mellett balra egy kis pirosan keretezett fehér X látható. Ez azt jelzi, hogy az adott mappán belül valamely forrásállomány fordítása során probléma adódott. Ezek szerint az Eclipse az előbb, mikor a projektet beolvastuk, azért „gondolkodott” annyit, mert az adatok beolvasása és ellenőrzése mellett lehetőség szerint le is fordította azokat.

De vajon mi lehet a gond az „*src*” mappában? Bontsuk ki, és járjunk utána! …de egyelőre csak a „*milab*” kezdetű elemekkel foglalkozzunk!

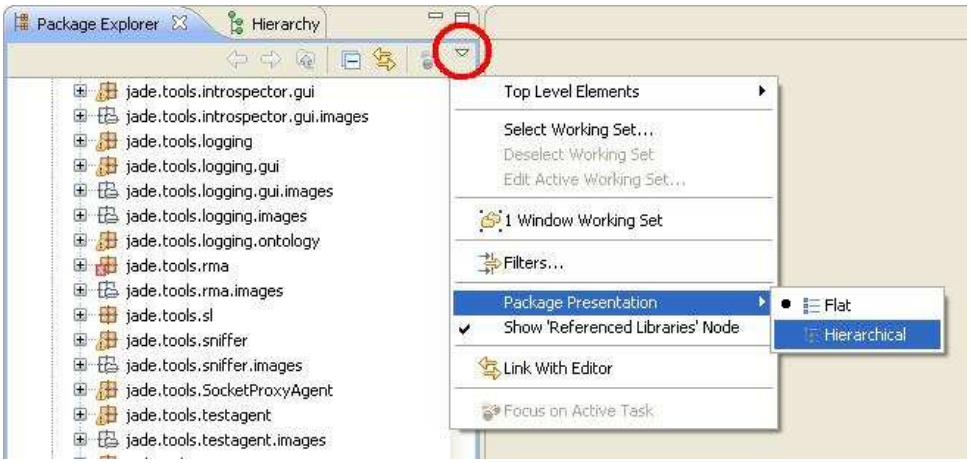
[24] Ebből következik, hogy a „fapados” módszer az Eclipse-szel kombinált módon is használható. Az Eclipse nem feltétlenül szükséges. Nem feltétlen várjuk el a laborgyakorlatokat fejlesztő kollégáktól az Eclipse használtatát.

A barna csomagok a Java csomagokat (és alcsomagokat), míg a szürkék a közvetlenül csak egyéb adatokat magukba foglaló mappákat (amikben közvetlen nincs Java forráskód) jelölik.

Rövid keresgetés után hamar rálelhetünk a „*milab*” kezdetű elemekre, és – amennyiben még nincsenek lefordítva – láthatjuk, hogy a „*milab.labxy.BookBuyerAgent*” és „*milab.labxy.BookSellerAgent*” csomagok mellett is ott szerepel a kis pirosan keretezett X.

Bontsuk tehát most ezeket is ki!

Újabb fajtájú elemek jelennek meg a fában… A „**J**” a Java forrásokat jelöli, míg az „**X**” az XML, vagy XSD állományokat. Valójában azonban, mint látjuk, ez nem egy fa. A különböző hierarchia-szinten elhelyezkedő csomagok és alcsomagok mind egy szinten vannak. Ez az úgynevezett flat nézet. Mielőtt tovább folytatnánk a hibák okának felderítését, érdemes lehet hierarchikus nézetre kapcsolni.



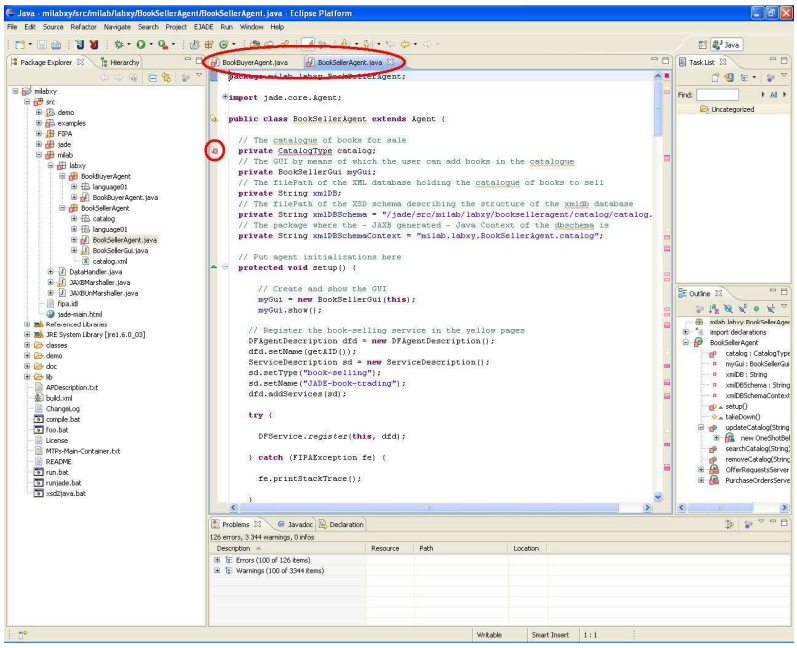
Ehhez az előbbi ábrán látható módon kattintsunk a „*Package Explorer*” jobb felső sarkában látható kicsi, lefelé mutató nyilacskára, aminek hatására egy pop-up menü jelenik meg, amiben a *Package Presentation/Hierarchical* menüpontot válasszuk!

Az eredmény magáért beszél. Immár sokkal áttekinthetőbb a csomagok hierarchiája (pl. a JADE beépített csomagjai, amiket most épp úgysem használunk, össze vannak húzva, és csak a számunkra aktuálisan érdekes „`milab`” csomag van kibontva).

Jól láthatóan a *KönyvVásárló*, és *KönyvÁrus* ágensek forráskódjában nem stimmel valami. De vajon mi lehet a gond, hiszen eddig (a „fapados” szisztéma szerint) minden rendben volt?

Nézzünk csak bele a forráskódokba! Kattintsunk az egér bal gombjával duplán mindkét kis „**J**” ikonkára!

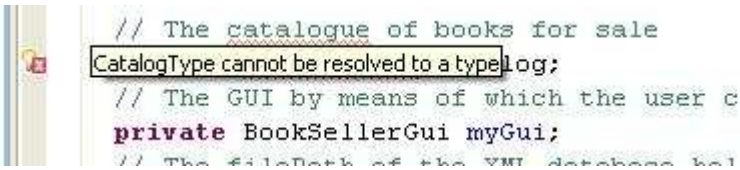
Kattintásaink nyomán a képernyő középső részén egymás után/mellett megjelenik a két forráskód szerkeszthető, szöveges formában.



A képen éppen a `BookSellerAgent` forráskódja látszik. Túl azon, hogy a forráskódban különböző színek jelölik a különböző program-elemeket (pl. változókat, konstansokat, függvényeket), az Eclipse egyéb támogatásokat is nyújt életünk egyszerűbbé tétele érdekében.

Ezek közül számunkra most elsősorban a forráskódtól balra elhelyezkedő kis jelek érdekesek. Ezek jelölik ugyanis a figyelmeztetéseket (*Warning*) és hibákat (*Error*).

A `BookSellerAgent` osztály privát változóinál, a „`catalog`” változó megadásánál például máris látható egy hiba. Vigyük az egérkurzort a hibát jelző X jel fölé!

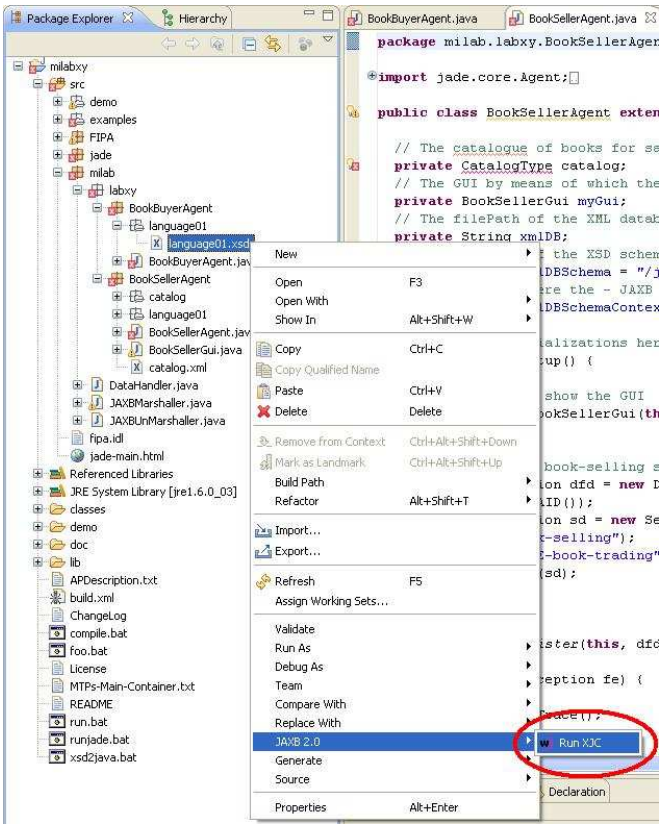


Egy kis pop-up üzenet jelzi számunkra a hiba okát: „*CatalogType cannot be resolved to a type*”. Rövid gondolkodást követően rájöhetünk arra, hogy az a baj, hogy nem tudunk `CatalogType` típusú objektumot létrehozni, mivel nincs `CatalogType` osztályunk! De miért?

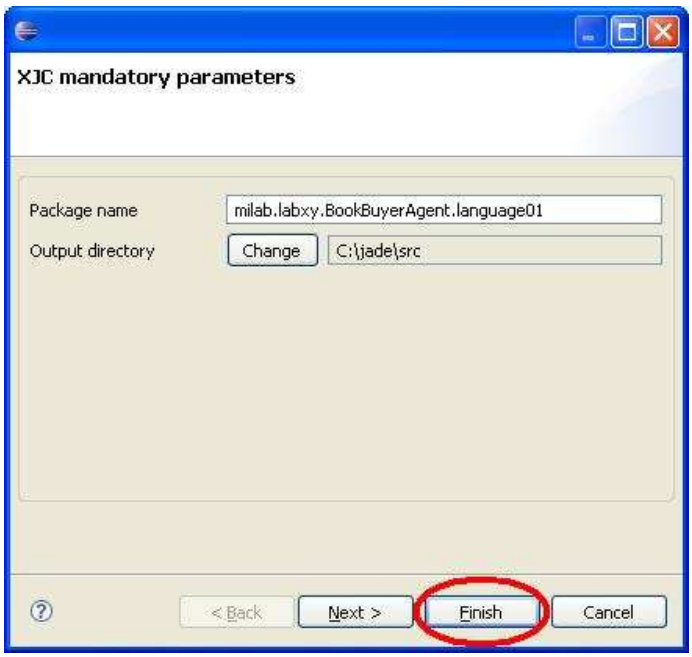
A válasz egyszerű: mivel <u>még nem fordítottuk le a megfelelő XSD sémát</u> (XJC-vel).

Ha tovább nézelődünk a két forráskódban, láthatjuk, hogy csak XSD sémákhoz generált Java-osztályok hiánya okozza a hibákat. Legfontosabb teendőnk tehát most az, hogy lefordítsuk az összes XSD sémát, és akkor (elvben) az ágenseknek is le kell fordulnia.

Az XSD sémák fordítása a 2.5.1-es szakaszban látottakhoz híven zajlik most is, csak nem parancssorból (konzolból), hanem Eclipse-ből, gombnyomással, a következőképp.

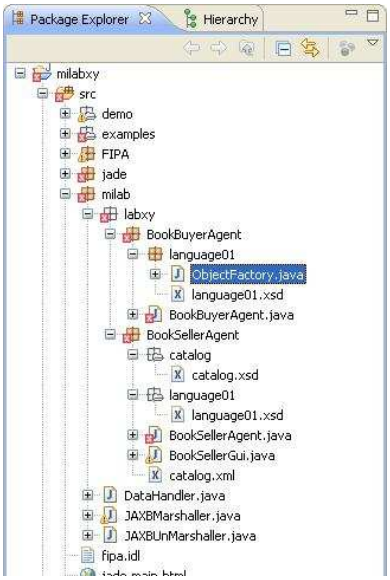


A fenti ábrának megfelelően először is böngésszük ki a sémát a „*Package Explorer*”-ben (pl. a `\milab\labxy\BookBuyerAgent\language01\language01.xsd` sémát), majd kattintsunk rá az egér jobb gombjával. A felugró menüből válasszuk a „*JAXB 2.0/Run XJC*” opciót!



A megjelenő képernyőn töltsük ki a megfelelő mezőket: a „*Package name*” mezőbe az XSD-t tartalmazó csomag nevét írjuk (esetünkben `milab.labxy.BookBuyerAgent.language01`) épp úgy, ahogyan eddig is (a 2.5.1-es szakaszban), míg az „*Output directory*” mezőbe böngésszük ki a megfelelő könyvtárat, ahonnan a csomag-hierarchia indul! Ez esetünkben mindig, sémától függetlenül X:\jade\src lesz. Végezetül kattintsunk a „*Finish*” gombra.

Rövid gondolkodás után lefut a fordító. Viszont egyelőre nem látható a futás eredménye (nem látszanak ez előállt Java források). Ahhoz, hogy lássuk az eredményt, először is az egér bal gombjával jelöljük ki a `milab.labxy.BookBuyerAgent.language01` csomagot, majd nyomjuk meg az **F5** gombot a billentyűzeten. Ennek hatására frissül a képernyő, és az alább láthatóaknak megfelelően megjelenik benne a kigenerált kontextus (esetünkben egy `ObjectFactory.java` fájl).

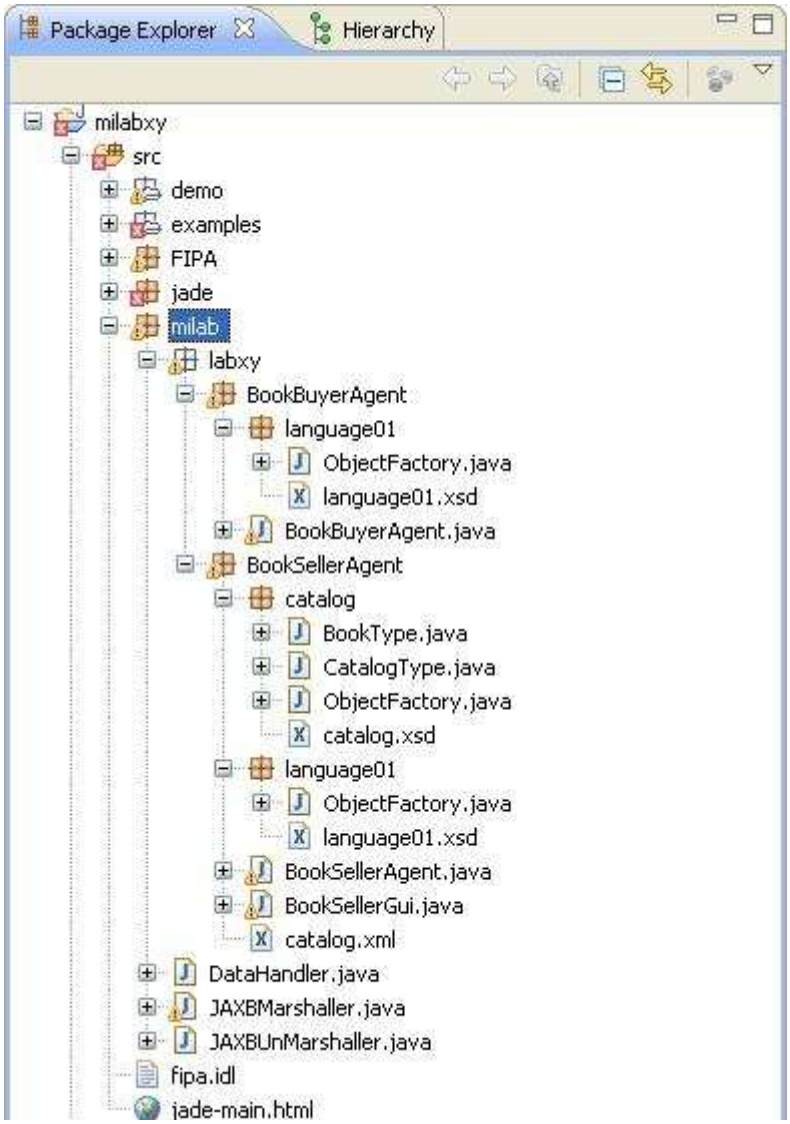


Érdemes észrevennünk, hogy a `milab.labxy.BookBuyerAgent.language01` csomag színe szürkéről barnára változott. Ennek oka nyilván az, hogy immár ebben a mappában (legalább) egy Java forráskód *is* helyet foglal.

Most pedig folytassuk a többi XSD – tetszőleges sorrendben történő – fordításával! A megfelelő lépések végrehajtását követően a következő új forrásfájloknak kellene előállni.[25]

X:\jade\src\milab\labxy\BookBuyerAgent\language01\ObjectFactory.java
X:\jade\src\milab\labxy\BookSellerAgent\language01\ObjectFactory.java
X:\jade\src\milab\labxy\BookSellerAgent\catalog\BookType.java
X:\jade\src\milab\labxy\BookSellerAgent\catalog\CatalogType.java
X:\jade\src\milab\labxy\BookSellerAgent\catalog\ObjectFactory.java

Ez teljes mértékben megfelel a 2.5.1-es szakaszban bemutatott módszer eredményének. Ráadásul a sémák lefordításának eredményeképp, csodák csodája, végre valóban maradéktalanul megszűntek a `milab` csomag „hibai”.

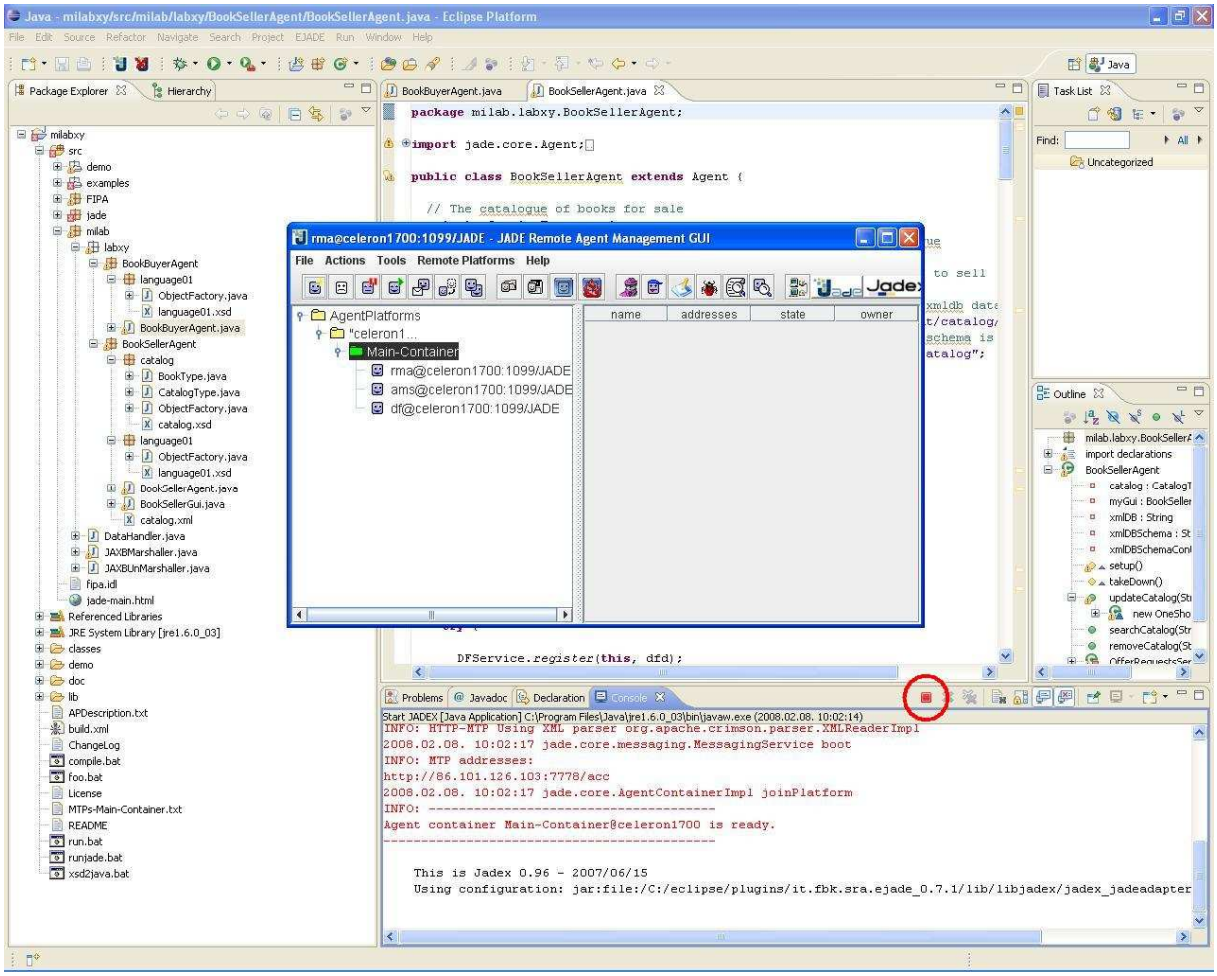


[25] A fordítás során különösen ügyeljünk a csomag nevének (*Package name*) megadására!

Ezek szerint végre megpróbálhatjuk elindítani az ágenseket (remélve, hogy a fordítási hibákon túl nem lesz se futás közbeni hiba, se pedig egyéb rendellenesség). Mindazonáltal, ha mindent az eddigieknek megfelelően tettünk, úgy elvileg semmi gond nem adódhat.

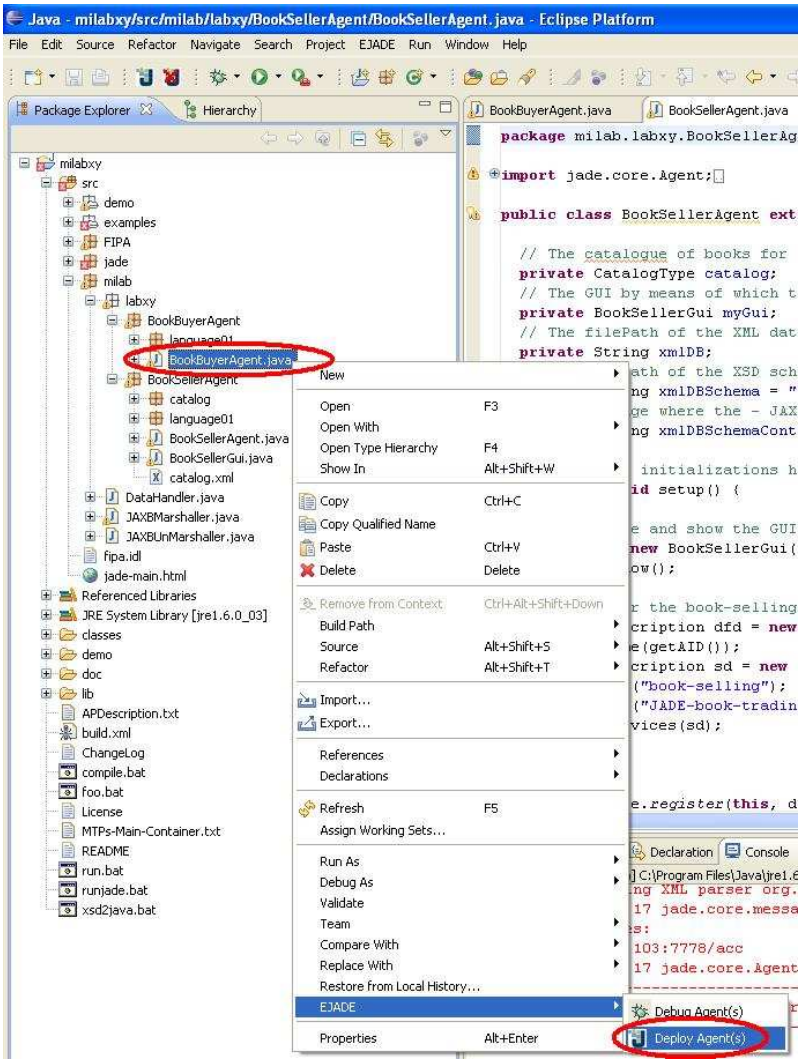
Az ágensek indítása nagyon egyszerű: először is el kell indítanunk egy JADE-es ágens-platformot. Ezt az „*EJADE/Start EJADE RMA*” opcióval, vagy a megfelelő ikonra kattintva tehetjük meg.

Az eredmény a már jól ismert RMA ágens GUI-jának megjelenése, és egy „*Console*” feljövetele a képernyő jobb alsó részében.

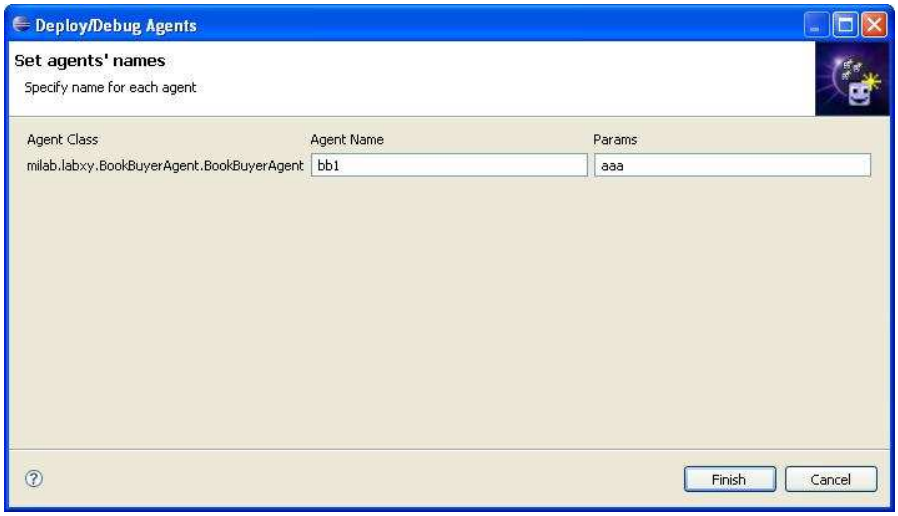


A „*Console*” gyakorlatilag a Windows-os parancssori ablaknak felel meg azzal a különbséggel, hogy nem interaktív (azaz nem írhatunk bele semmit sem a billentyűzetről). A „Console” azt is jelzi, hogy fut-e a benne foglalt folyamat. Ha igen, akkor az előbbi ábrán körbekarikázott *STOP gomb* piros (kattintható), egyébként szürke (nem kattintható).

Most már tehát valóban elindíthatjuk az ágenseket – akár az Eclipse-ből, akár az RMA ágens GUI-ján keresztül. Mivel az utóbbi módszert már jól ismerjük, tekintsük az előbbi esetet, az ágensek Eclipse-ből történő indítását!



Az előbbi ábrának megfelelő módon kattintsunk az indítani kívánt ágens osztályára az egér jobb gombjával, majd a felugró menüből válasszuk ki az „*EJADE/Deploy Agent(s)*” opciót!

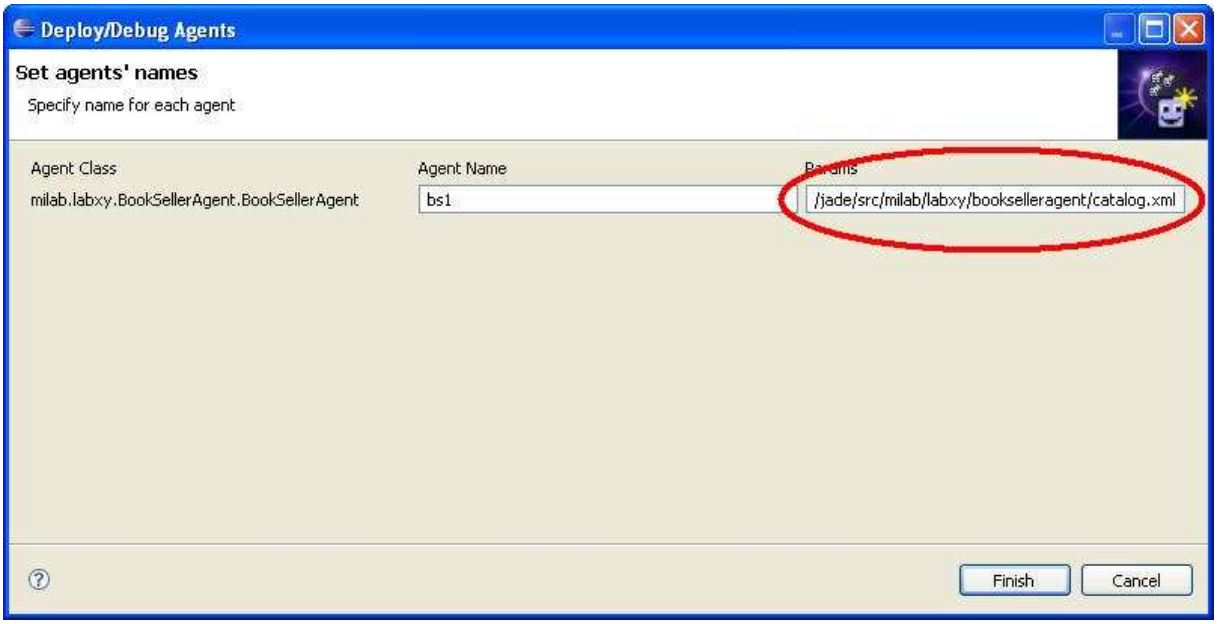


A felugró ablakban gépeljük be a létrehozni kívánt ágens nevét (ügyelve arra, hogy a platformon belül egyedi legyen), írjuk be az ágens parancssori paramétereit (szóközökkel elválasztva), majd kattintsunk a „*Finish*” gombra!

Az esetlegesen feljövő „*Errors in Workspace*” hibaüzenettel most ne foglalkozzunk, mivel biztosan nem az általunk használt `milab` csomagra utal. Kattintsunk csak a „*Proceed*”-ra!

Ha mindent jól csináltunk, elindul egy *KönyvVásárló* ágens (esetünkben „`aaa`” című könyvre vadászva). GUI-ja ugyebár nincs, de a konzolon nyomon követhetjük működését.

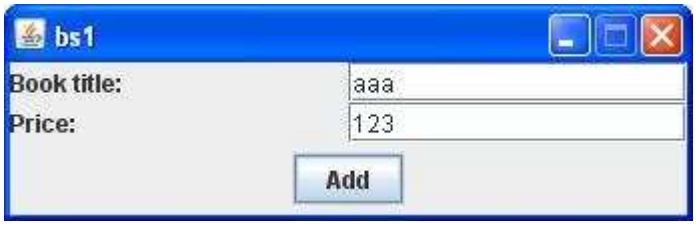
Indítsunk most el egy *KönyvÁrus* ágenst is! Legyen a neve „`bs1`”.



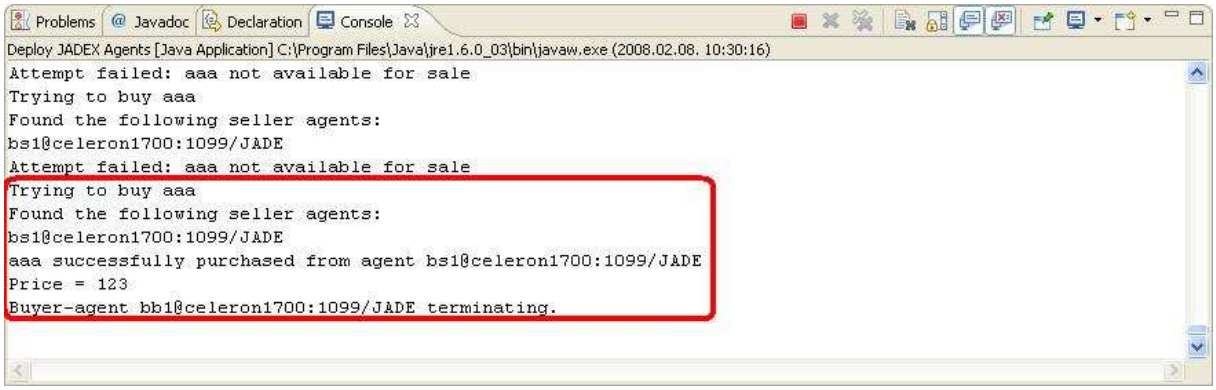
FONTOS: az ábrán látható módon (gyökérhez viszonyítva) adjuk meg egyetlen paraméterét, XML könyv-adatbázisának fájlrendszerbeli elérését.

A *KönyvÁrus* ágens indulását követően megjelenik a GUI-ja, és máris kapcsolatba lép vele a *KönyvVásárló* ágens. Mindezt a konzolon is nyomon követhetjük.

Kezdetben elvileg a *KönyvÁrus* ágensnek csupa „`aaa`”-tól eltérő könyve van a katalógusában, ezért – az ágens-interakció kedvéért – célszerű bevinni az „`aaa`”-t is adatbázisába.



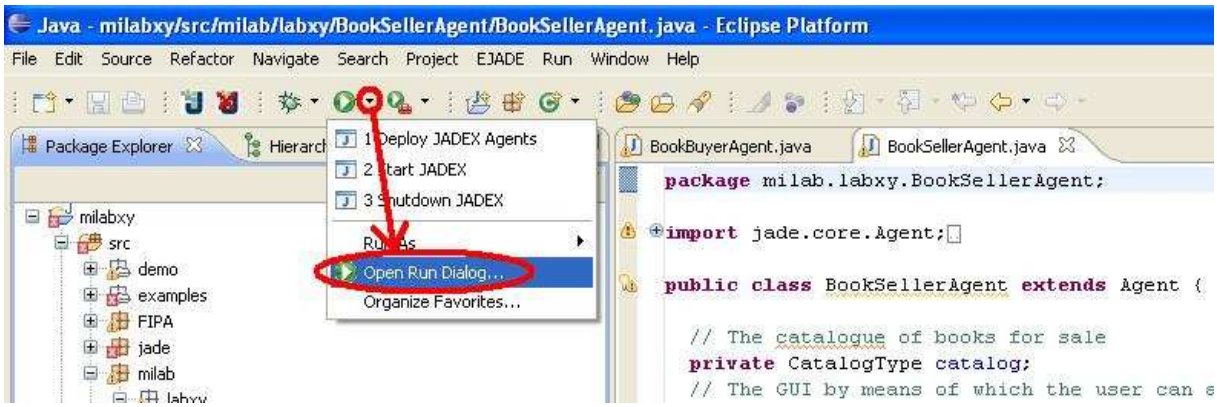
A könyv bevitelét követően – a konzolon jól látható módon – megtörténik az adás-vétel…



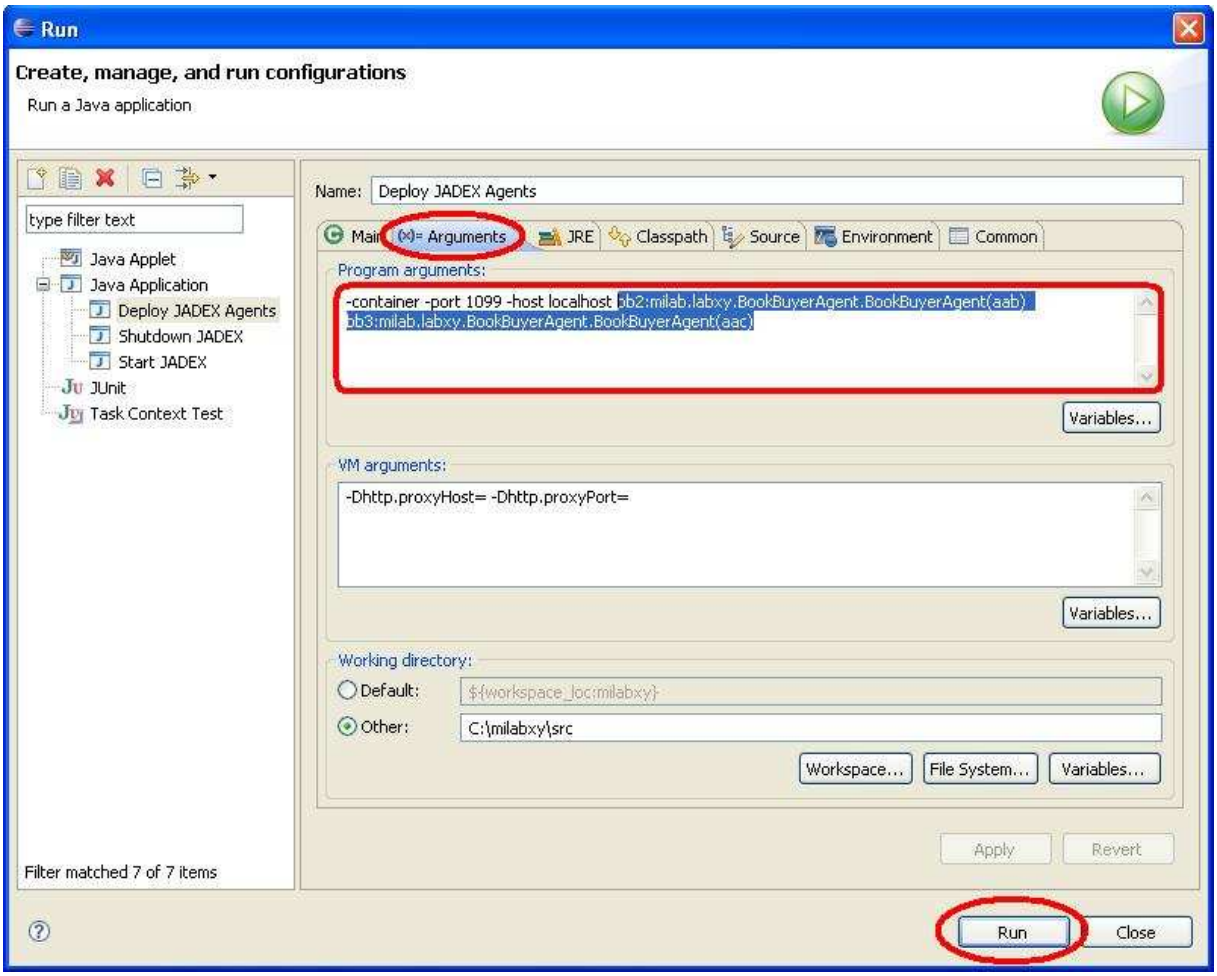
Az adás-vétel lebonyolítását követően a *KönyvVásárló* ágens szabatos módon leáll. Ezt az RMA ágens GUI-ján is megfigyelhetjük.

Eddig csak egyszerre egy ágens indításával foglalkoztunk. Ekkor minden indított ágens külön konténerben jött létre. Természetesen lehetőségünk van arra is, hogy egy konténerben egyszerre több ágenst is indítsunk. A „*Ctrl*” billentyű-gombot nyomva tartva jelöljünk ki néhány ágenst az egér bal gombjával, majd a szokásos módon, az egér jobb gombjával kattintva valamely kijelölt ágensre, indítsuk el őket (de csak miután rendre elneveztük, és – esetleg – felparamétereztük).

Esetünkben csak kétféle ágens van, ezért többet nem is igen választhattunk. Amennyiben azonban egy konténerben egy *adott típusú* ágensből többet (is) szeretnénk indítani, úgy valamivel kényelmetlenebb eljárásnak kell magunkat alávetni. Vagy az RMA ágens GUI-ját használjuk, és egyenként indítjuk az ágenseket a „*Start New Agent*” opcióval, vagy Windows-os parancssorból indítunk egy konténert több (akár azonos típusú) ágenssel, vagy Eclipse-ből tesszük ugyanezt egy kicsit egyszerűbben. Ez utóbbihoz kattintsunk először is az „*Open Run Dialog…*” menüpontra (az alábbi ábra szerint).



A megjelenő ablakban kattintsunk az „*Arguments*” fülre, töltsük ki a „*Program arguments*” részt (úgy, ahogyan parancssorból szoktuk paraméterezni a `runjade.bat`-ot), majd kattintsunk a „*Run*” gombra!



Az ábrán jól látható, hogy 2 további *KönyvVásárló* ágenst indítunk egy új konténerben. Az ábráról az is kitűnik, hogy ily módon lehetőségünk nyílik arra, hogy távoli gépeken host-olt platformokon indítsunk konténereket/ágenseket. Ez tehát az ágensek indításának egy jóval kötetlenebb, ámde egyben komplikáltabb módja.

Ezzel gyakorlatilag a tudnivalók végére értünk. Ennyi elég kell, hogy legyen ahhoz, hogy akár Eclipse-ben, akár anélkül ki tudjuk fejleszteni a laborgyakorlatokhoz szükséges ágenseinket.

A tanulás azonban, ahogy mondani szokták, „holtig tart”. Mi azonban ne várjunk addig (a „tanulmányi út” végéig)! ☺ A jelen anyag elolvasását követően, amennyiben van hozzá kedvünk és időnk, próbáljuk meg még jobban megismerni az Eclipse fejlesztői környezetet!

Példának okáért érdemes megismerni az XML és XSD fájlok Eclipse-es szerkesztésének módját. Ilyen, és ehhez hasonló ismeretekre például Eclipse tutorial-okból tehetünk szert. A következő weblap kiindulásul szolgálhat mindehhez.

`http://del.icio.us/tag/eclipse+tutorials`