INTELLIGENS RENDSZEREK I. LABORATÓRIUM

1. laborgyakorlat segédlete

Készítette:

Kovács Dániel László
dkovacs@mit.bme.hu

Méréstechnika és Információs Rendszerek Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem

2010, február.

Tartalomjegyzék

1. Bevezető

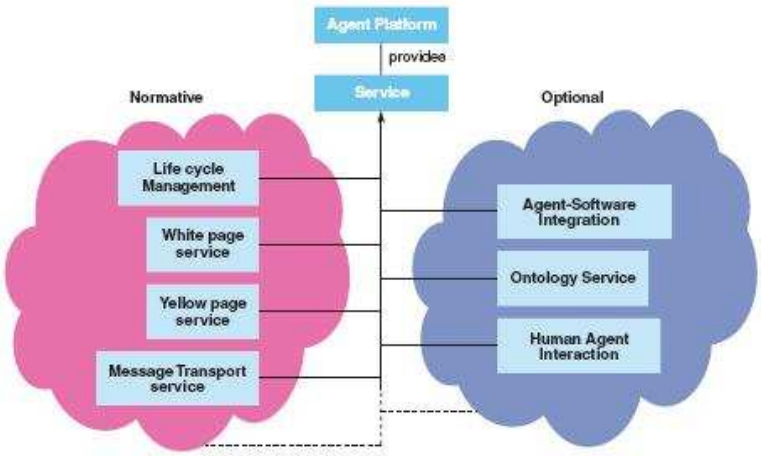
Jelen anyag a JADE (Java Agent DEvelopment Framework) elnevezésű, Java-alapú ágens-keretrendszert bemutató laborgyakorlathoz készült segédletként. A gyakorlat célja megismerkedni egy olyan ágens „middleware" rendszerrel, amely:

- lehetőséget ad ágensközösségek (multi-ágens rendszerek) **gyors** építésére azáltal, hogy a **kommunikáció eszközei szabványosítottak** (**FIPA** szabvány), és **implementáltak**,
- nem kívánja befolyásolni az ágensek feladatvégző és/vagy kognitív képességeit,
- egyszerű grafikus és szöveges módon teszi lehetővé az **ágensközösségek** „munka közben" történő **megfigyelését**.

A rendszer alkalmazásának előnye, hogy az ágensfejlesztés során nem kell feleslegesen a belső állapot, illetve kommunikációs képességek alacsony szintű megvalósításával bajlódni. Ezek mind adottak. A felhasználói fejlesztés az intelligens feladatvégzés, együttműködés, feladatfüggő ember-gép kapcsolat, és egyéb lényeges szempontok megvalósítására összpontosulhat.

2. JADE lényege

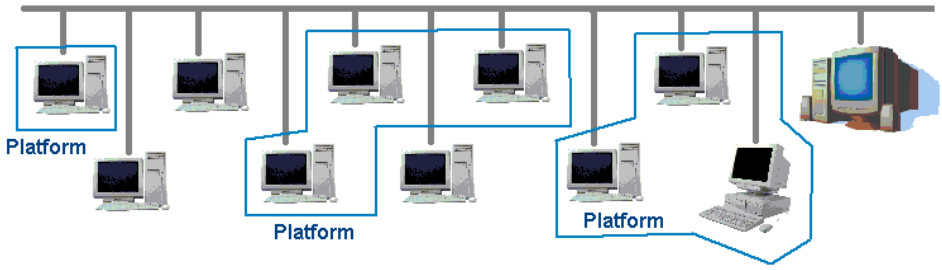
A JADE rendszer (`http://JADE.tilab.com`) a FIPA (Foundation for Intelligent Physical Agents) szabvány (`www.fipa.org`) egy részleges, Java nyelvű *referencia-implementációja*. A FIPA szabvány az ágensközösségre úgy tekint, mint egy *platformra*, amely platform a rajta ténykedő *ágensek* számára meghatározott (általában a kommunikációt és együttműködést megkönnyítő) szolgáltatásokat is biztosít (platform = „ágens-közösség" szervezeti értelemben).



1. Ábra: a JADE-es ágens-platform szolgáltatásai

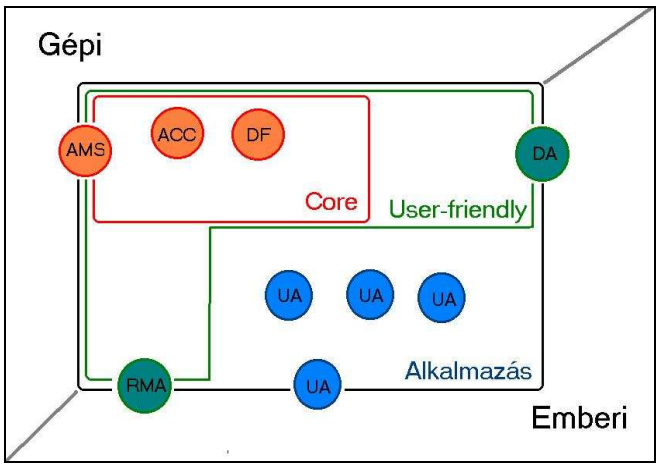
A JADE rendszer (az indítást követően) egy **platform**ot létesít. A platform **konténer**ekből áll. A konténereken belül futnak az **ágens**ek. Az ágensplatform – fizikai értelemben – azon a gépen fut, amelyen a platform **főkonténer**e (Main-Container). Ennek a gazda-számítógépnek (host-nak) a leállása a platform leállását eredményezi. Az ágenseket szokványos **Java-osztály**ok valósítják meg.

Egy platformot más platformok is láthatnak (Remote Platform), illetve más gépeken host-olt konténerek (és bennük ágensek) is csatlakoztathatók hozzá. Ilyen értelemben tehát a platform egy *logikai infrastruktúra*, amely elrejti a konkrét „*fizikai*" infrastruktúrát. Ily módon logikai egységként, összetartozó módon, kiterjedt ágens-közösségeket lehet – akár több gépen is elosztva – üzemeltetni.



2. Ábra: fizikailag elosztott, de logikailag egybefüggő JADE platform-ok

Egy JADE platform több beépített, a platform indulásakor automatikusan (a főkonténerben) létrejövő, vagy a felhasználó által interaktívan (valamely mellékkonténerben) létrehozott, illetve a felhasználó által a platformhoz hozzáfejlesztett (valamely mellékkonténerben indított) ágensből áll.



3. Ábra: a JADE-es ágensek kategóriák szerint

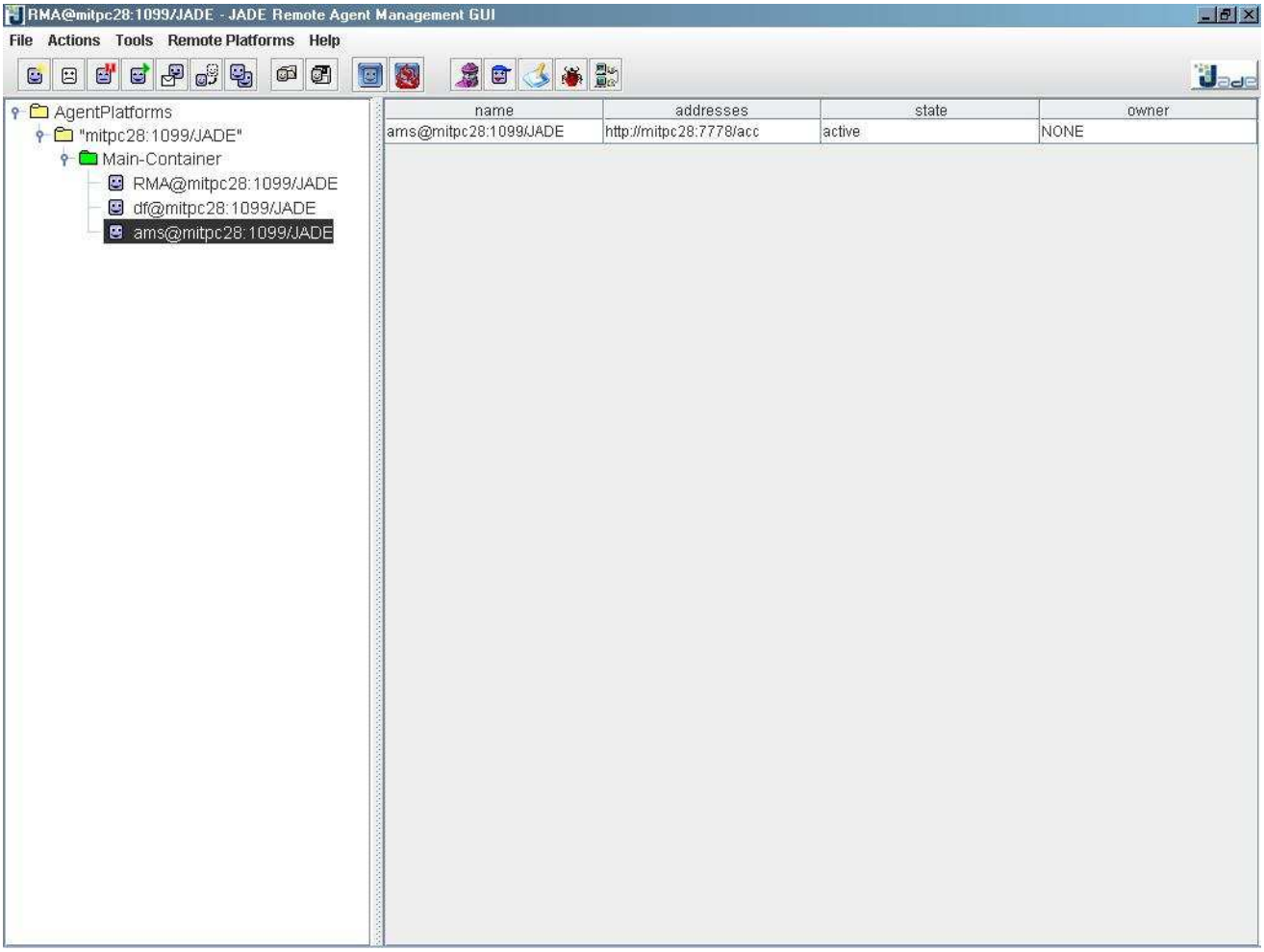
A platformhoz hivatalból az alábbi ágensek tartoznak (melyek a platform indulásakor automatikusan a főkonténerben jönnek létre):

- **AMS** (**Agent Management System**): egyfajta „operációs rendszere” a platformnak; a platform közösségi „kapuőre”. Ez az ágens alapértelmezésben a platform-ot hosztoló gép 1099-es számú port-ján figyel, azon belül egy „JADE” nevű web-alkalmazást valósít meg. Címe ennek megfelelően: **`ams@hoszt_gép_URL-je:1099/JADE`**

- **ACC (Agent Communication Channel):** a kommunikációt lebonyolító (rejtett) ágens. Ez az ágens HTTP protokolon át fogad XML-ben leírt ACL (Agent Communication Language) nyelvű üzeneteket. Alapértelmezésben a platform-ot hosztoló gép 7778-as számú port-ján figyel, tehát elérése alapértelmezésben a következő: **`http://hoszt_gép_URL-je:7778/acc`**

- **DF** (**Directory Facilitator**): a platform képességeinek/szolgáltatásainak „telefonkönyve” (egyfajta Yellow Pages – Sárga Oldalak).

- **RMA (Remote Monitoring Agent)**: a platformba beépített menedzser-ágens. Ennek létrehozása igazából nem egészen automatikus, de beépített módon biztosított. Az RMA ágens mindent, és mindenkit számon tart, aki a platformon elhelyezkedik. Ráadásul külön grafikus felülete van mindezek megjelenítésére, és menedzselésére. Lényegében tehát az RMA ágens ad lehetőséget az ágensplatformok felhasználói menedzselésére. FONTOS: *Vegyük észre, hogy a JADE indításakor a GUI (Graphical User Interface) gyakorlatilag a főkonténerben létrejövő RMA ágens saját grafikus felülete.*



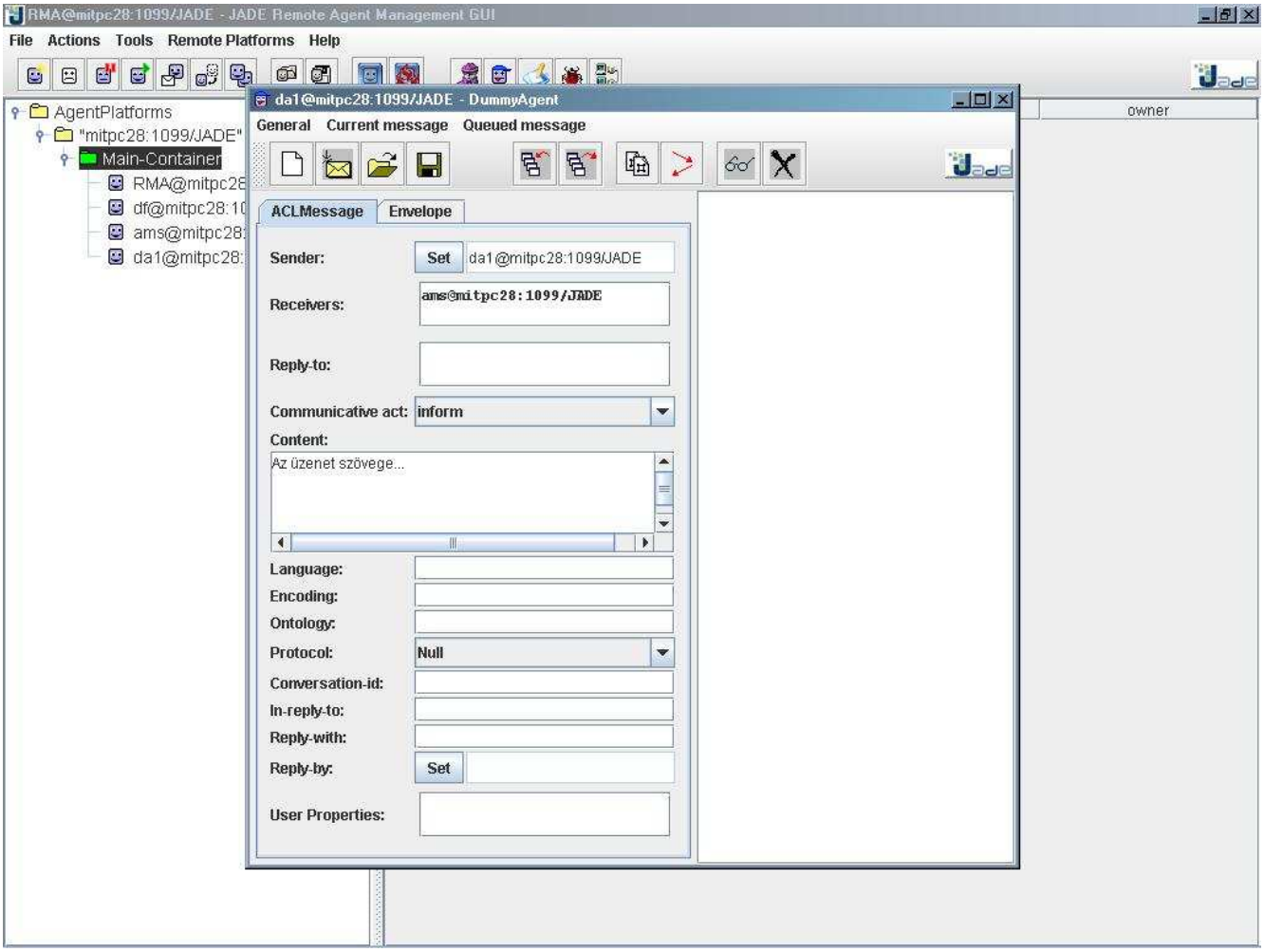
4. Ábra: egy JADE platform a MITPC28-as host-on, indítás után, az RMA ágens GUI-ján megjelenítve

A platform felhasználói felületéhez tartoznak még a következő ágensek:

- **SNIFFER**: amely a felhasználó által megadott ágensközi kommunikációt figyeli és jeleníti meg,
- **INTROSPECTOR**: amely az ágensek életútját kiséri végig, benső állapotukba enged bepillantást.

Az előbb említett két ágens számára konkrétan meg kell adni, hogy mely ágens(eke)t figyelje. Ezen túl létrehozható még az úgynevezett **DA** (**Dummy Agent**) ágens is, amely gyakorlatilag egy „ágensbőrbe bújtatott” emberi felhasználónak felel meg.

A DA-t szokás „embert csomagoló”, avagy wrapper-ágensnek is nevezni. A DA ágens semmiféle reaktív, vagy proaktív funkcionalitással nem bír; nincsenek beépített viselkedései, nem autonóm, stb. Csupán egy GUI-t biztosít, amelyen keresztül az emberi felhasználó bizonyos alapvető ágens-funkciókat aktiválhat (pl. üzenetek küldése/fogadása más ágenseknek/ágensektől).



5. Ábra: a DA ágens GUI-ja

Az előbb említett ágenseken felül természetesen még tetszőleges számú egyéb ágens indítható, többek közt saját, úgynevezett felhasználói **UA (User Agent)** ágensek. Ilyenek pl. az **`src\examples`** és a **`demo`** könyvtárban található, előre elkészített ágensek. De ilyenek lesznek a laborgyakorlat keretein belül megírt ágensek is.

3. JADE használata

Figyelem!

1. A JADE nem ipari, hanem **kísérleti/egyetemi fejlesztés**, amire nem győznek figyelmeztetni a rendszer fejlesztői. Ebből kifolyólag számos hiba (feltárt, és fel nem tárt) található még a rendszerben, amiket a fejlesztők természetesen folyamatosan próbálnak felszámolni. Még közel sem járnak a végén… Viszont minden ésszerű észrevételért hálásak.
2. A JADE Java-alapú, a Java pedig közismerten igen **erőforrás-igényes**. Ezért szegényesebb hardver konfiguráció esetén érezhető lassulásra lehet számítani.
3. Az **idő-szinkronizálás** még nem tökéletesen megoldott. Ebből kifolyólag pl. a Sniffer ágens az ágensközösségből „kivadászott" üzenetforgalmat nem szükségképp mutatja szigorú időrendben.
4. Számos opció csak az **egér jobb gombjával** működik!
5. Egy platformon belül az **ágensek lokális neve mindig egyedi**!

Tanulmányozzuk a „`c:\jade`" könyvtár tartalmát!

A c:\jade könyvtár közvetlenül tartalmazza a futtatáshoz szükséges batch-fájlokat. Ezek a következők:

`run.bat`	*Java osztályok paraméterezhető indítása* *Egyetelen paramétere a futtatni kívánt Java-osztály teljes neve*
`runjade.bat`	*JADE-es ágensek paraméterezhető indítása* *Paraméterei megegyeznek a **jade.Boot** osztály paramétereivel*
`compile.bat`	*Java osztályok (pl. JADE-es ágensek) paraméterezhető fordítása* *Egyetelen paramétere a fordítani kívánt Java-osztály elérése*
`xsd2java.bat`	*XML sémák Java-osztályokra történő, paraméterezhető fordítása* *Két paramétere van: **(1)** mi legyen a kigenerált Java-kontextus package-neve (hová kerüljön); **(2)** a fordítandó XSD fájl elérése*

Vegyék észre a „**`run…`**" kezdetű batch fájlokban található azon részeket, melyek…

1. A **CLASSPATH** környezeti változó beállításáért felelősek,
2. A **jade.Boot** osztályt indítják

***Főkonténer** indítása*: **java JADE.Boot [options] [Agent list]**
***Mellékkonténer**ek indítása* **java JADE.Boot -container [options] [Agent list]**

A **jade.Boot** osztály legalapvetőbb **[options]** parancssori opciói a következők:

-container	mellék konténer indítása,
-host <hostname>	melyik hoszton fusson a főkonténer,
-port <portnumber>	port specifikálása az RMI leíró részére,
-gui	RMA ágens indítása a főkonténerben a platform indulásakor
-help	összes elérhető parancssori opció listája

Az indítani kívánt ágensek **[Agent list]** listájának parancssori szintaxisa a következő:

[Agent list] = [Ágensnév:Csomagnév.ÁgensOsztályNév(arg1 arg2 … argN)]

Például egy GUI-val megtoldott JADE platform indítása 3 további ágenssel a következőképp történik:

```
runjade –gui pa0:examples.PingAgent.PingAgent
             rma1:jade.tools.rma.rma
             da0:jade.tools.DummyAgent.DummyAgent
```

Ágensnév: ez gyakorlatilag egy „becenév (nickname)”, ami platformon belül egyedi kell, hogy legyen. **Csomagnév:** az ágenst megvalósító Java-osztály C:\JADE\SRC könyvtárból (mint CLASSPATH-ból) indulóan vett könyvtári elérése. **ÁgensOsztályNév:** az ágenst megvalósító Java-osztály neve (nyilván `.class` kiterjesztés nélkül). **argK:** az indított ágensek opcionális paraméterei szóközökkel elválasztva. Ha az ágenst paraméterek nélkül indítjuk, akkor ez a rész nem is kell.

Az ágens, miután elindítjuk, kap egy **azonosító**t. Ezen ágens-azonosító (pl. `pa0@milab11:1099/JADE`) felhasználásával lehet kapcsolatba lépni az ágenssel. Érvényes ágens-azonosító lehet például:

```
rma@milab01:1099/JADE
pa0@aipc11:1099/JADE
```

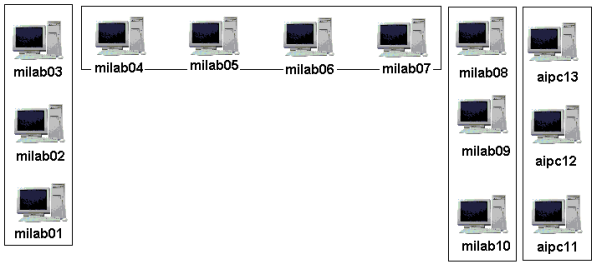
Nyilván ezek lokális hálózaton (LAN-on), pontosabban azon belül adott domain-ben érvényesek (esetünkben: *mit.bme.hu*). Részletesen kiírva az előbbiek így néznének ki:

```
rma@milab01.mit.bme.hu:1099/JADE
pa0@aipc11.mit.bme.hu:1099/JADE
```

Az ágens-azonosító tehát egészen pontosan azt adja meg, hogy mi az ágens beceneve (pl. `rma`), mi az őt logikailag tartalmazó platform főkonténerét futtató host-gép URL-je (pl. `milab01.mit.bme.hu`), és azon belül melyik port-on figyel a főkonténer (alapértelmezésben ez mindig `1099`).

Az ágensek-azonosítójához hasonlóan a JADE platform is rendelkezik egyfajta „azonosítóval”, pontosabban leírással, melyet induláskor kiír egy fájlba (`\jade\APDescription.txt`).

Labor térképe



6. Ábra: a laboratórium gépeinek elrendezése

Az ábrán az I.E.316-os teremben található MI labor gépeinek lokális hálózaton belüli azonosítója és topológiája látható. Az gépek mindegyike a „`mit.bme.hu`” domain részét képezi.

4. JADE programozása

Az előbbiekben áttekintettük a JADE keretrendszer alapjait, és használatának módját. A JADE használatával kapcsolatos további részletes információ (pl. összes parancssori opció, RMA+Dummy+DF+Sniffer+Introspector ágens GUI-jának részletes leírása) a labor weblapjáról letölthető ***JADE-adminisztrátori útmutató***ban található. Ebből a **6. fejezet elolvasása KÖTELEZŐ!**

A JADE használatát, és a Java programozási nyelv ismeretét adottnak feltételezve belefoghatunk az ágensek programozásába. JADE alá (JADE-hez) ágenseket fejleszteni többféleképp is lehet. Fejleszthetjük az ágenseket sima „fapados" módon, szövegszerkesztőben, parancssorból fordítva és futtatva őket (ahogy az eddigiekben is láttuk), vagy használhatunk valamiféle szoftverfejlesztési környezetet (pl. Eclipse, NetBeans). A labor során mindkét módszerrel megismerkedünk. Az utóbbi lehetőségről a jelen segédlet következő, „*JADE és Eclipse*" c. fejezete ír bővebben.

Ebben a fejezetben vázlatosan, fejlesztési eszköztől függetlenül kívánjuk bemutatni a JADE-es ágensek programozásának főbb szempontjait. A fejezet rövid áttekintést ad tehát a JADE-es ágensek programkódjának felépítéséről, és e felépítés mögött húzódó főbb elgondolásokról. Részletesebb magyarázatot a labor weblapjáról letölthető ***Kezdő JADE-programozói segédlet***ben találunk, aminek **alapos elolvasása és megértése KÖTELEZŐ!!**[1]

A JADE ágenseket többnyire egy-egy Java osztály valósítja meg. Ez az osztály azonban annyiban nem tekinthető szokványosnak, hogy éppenséggel a JADE **API**-ját (**A**pplication **P**rogramming **I**nterface-ét) használja. A JADE API lényegében a JADE-es ágensek fejlesztéséhez elengedhetetlen osztályok és metódusok gyűjteménye (a `\jade\lib` könyvtárban található **JAR** (**J**ava **AR**chive) fájlok).

Például már magát az ágenst (avagy az általunk fejlesztett ágens-osztályt) is az API előre elkészített `jade.core.Agent` osztályából származtatjuk (`extends`). Az ágensek belső változóira ez semmiféle megkötést nem tesz, azonban van néhány örökölt metódus, és úgynevezett viselkedés(osztály), amit mindenképp célszerű implementálnunk.

A JADE-es ágensek működését ugyanis valójában viselkedések valósítják meg. Ezek a viselkedések takarhatják belső változók értékének módosítását, más ágensek számára történő üzenetküldést, üzenetek fogadását, és feldolgozását, egy algoritmus végrehajtását, vagy akár valamiféle grafikus megjelenítést, vagy bármilyen egyéb Java-kódot.

A viselkedéseket a JADE keretrendszer aktiválja az ágens állapotától függően. Az egészet úgy érdemes elképzelni, mint egy szabály-alapú szakértői rendszert. Az aktív, elsüthető szabályok közül a keretrendszer – adott szisztéma szerint – mindig kiválasztja valamelyiket, és végrehajtja (elsüti). Esetünkben ez azt jelenti, hogy az ágens aktuálisan aktív viselkedései közül a JADE keretrendszer – adott szisztéma szerint – mindig kiválaszt egyet, és azt végrehajtja. A viselkedés végrehajtásának befejeződéséig más viselkedésre nem kerül sor.

A viselkedés végrehajtásának befejeződését követően a JADE keretrendszer újabb aktív viselkedés után néz az adott ágens kapcsán. Ha nincs ilyen, akkor altatja az ágenst addig, amíg szükséges (pl. amíg nem kap üzenetet egy másik ágenstől, vagy nem állítjuk le, vagy a GUI-ra nem érkezik egy olyan esemény, ami visszahat az ágens állapotára, stb). Itt tehát valójában arról van szó, hogy minden egyes ágens egy-egy szál (`thread`) a **JVM**-ben (**J**ava **V**irtual **M**achine). Az ágensek működése „egyszálú" (`single threaded`).[2] A GUI-t külön (`eventDispatcher`) szálon célszerű megvalósítani.

[1] Az említett segédletet a JADE főprogramozója, Caire Giovanni írta kifejezetten olvasmányos, és közérthető stílusban (angolul). Ennél jobb programozói segédletet nehéz lenne elképzelni a JADE-hez. További JADE-programozói tudásra a *JADE-programozói útmutató*ból tehetünk szert, ámde ennek elolvasása már nem kötelező (opcionális).

[2] Természetesen van lehetőség ennek több-szálú kiterjesztésére is, azonban nem javallott.

Az ágensek létrehozásakor mindig egy úgynevezett `setup()` metódus fut le. Ez valójában a `jade.core.Agent` osztály `setup()` metódusának általunk felüldefiniált változata szokott lenni (`polimorfizmus`). A `setup()` metódusban szokás megoldani az ágens belső változóinak inicializálását, és a kezdeti viselkedések aktiválását/létrehozását. Erre szolgál a `jade.core.Agent` osztálytól örökölt `addBehaviour()` metódus.

Magukat a kezdeti viselkedéseket a `jade.core.behaviours.Behaviour` osztályból származtatott osztályok reprezentálják. A konkrét viselkedések tehát ezen leszármazott osztályok példányai.

A viselkedéseket reprezentáló osztályokat több helyen is deklarálhatjuk/implementálhatjuk. Szokás például az ágenst reprezentáló osztály belső, privát osztályaként (`private class`) megadni őket, de közvetlenül is létrehozhatjuk őket az `addBehaviour()` metódus meghívásakor is (a metódus bemenetén). Mindkét esetben ugyanolyan jellegű viselkedés-objektum-példányok keletkeznek. Az `addBehaviour()` metódus bemenete tehát egy `jade.core.behaviours.Behaviour` osztályú, vagy abból leszármaztatott objektum.

A `jade.core.behaviours.Behaviour` osztály tehát a legáltalánosabb viselkedést írja le. Az ebből leszármaztatott „speciális" viselkedéseknek tetszőleges belső változói lehetnek. Van viszont néhány metódusuk, amiket felül kell definiálni. Az egyik ilyen metódus az `action()`, a másik pedig a `done()`. Az előbbiben valósítjuk meg a viselkedés végrehajtása során kívánt funkcionalitást (pl. üzenetküldést), míg az utóbbiban egy egyszerű ellenőrzést hajtunk végre (pl. a kapcsolódó ágens valamelyik belső változójának vizsgáljuk meg az értékét). Az ellenőrzés eredményét `boolean`-ként visszaadjuk. Ha a visszaadott érték igaz, akkor a JADE keretrendszer deaktiválja a viselkedést. Hamis visszatérési érték esetén azonban a viselkedés (az `action()` metódus végrehajtását követően is) aktív marad, és később újra végrehajtásra kerül. Nyilván a JADE keretrendszer – a viselkedés végrehajtásakor – először mindig az `action()` metódust hívja meg, és csak utána következik a `done()`.

Természetesen vannak előre létrehozott, speciális viselkedések is a JADE API-ban. Ilyen pl. a `jade.core.behaviours.OneShotBehaviour`, melynek `done()` metódusa mindig igaz értékkel tér vissza, és így egyetlen egy futtatás után deaktiválódik. A `jade.core.behaviours.CyclicBehaviour` ennek épp az ellenkezője. Ennek `done()` metódusa mindig hamis értékkel tér vissza, és így sosem deaktiválódik. De vannak másféle, előre definiált viselkedésfajták is, mint pl. a `jade.core.behaviours.TickerBehaviour`, amely csak megadott időszakonként, periodikusan aktiválódik. Ennek `onTick()` metódusa valósítja meg a periodikus funkcionalitást.

A többi viselkedésről, és ezek részleteiről a labor weblapjáról letölthető ***Kezdő JADE-programozói segédlet***ből tájékozódhatunk (**elolvasása és megértése KÖTELEZŐ**)!

Az ágens funkcionalitását tehát viselkedések valósítják meg. Felmerülhet a kérdés, hogy: *vajon az ágens minden lehetséges viselkedését már a kezdetek kezdetén (a* `setup()` *metódusban) aktiválni kell?* A válasz: nyilván nem. Az egyes viselkedések újabb viselkedéseket hozhatnak létre az `addBehaviour()` metódus segítségével. Mindazonáltal érdemes jól meggondolni, hogy mikor-melyik viselkedés milyen másik viselkedést hoz létre, hogy véletlenül se adódjon nem kívánt működés (pl. végtelen ciklus).

Az ágens működése végeztével leáll. Leállítását kezdeményezhetjük akár a `setup()` metódusban (pl. a végén, vagy egy feltételtől függően), vagy akár egy viselkedésben is. Erre szolgál a `jade.core.Agent` osztálytól örökölt `doDelete()` metódus hívása.

FIGYELEM: Viselkedésből (amennyiben nem az ágens privát osztályaként valósítjuk meg) úgy tudjuk meghívni az ágens `doDelete()` metódusát, hogy hivatkozunk a `jade.core.behaviours.Behaviour` osztálytól örökölt `myAgent` változóra, amely a `Behaviour` osztály konstruktorának átadott ágens-objektumot (`this`) kapja kezdeti értékként a viselkedés létrejöttekor. Tehát: `myAgent.doDelete()`

A `doDelete()` metódus hatására a JADE keretrendszer végrehajtja az ágens `takeDown()` metódusát. Ezt a metódust, hasonlóan a `setup()`-hoz, nekünk magunknak, mint az ágens fejlesztőinek kell implementálnunk. Ide helyezhetjük szükség esetén a különböző „rendrakó” eljárásokat, stb. De már az is elég, ha kiírunk egy üzenetet (a konzolra), amiben jelezzük, hogy az ágens szabatosan leáll. A `takeDown()` metódus végrehajtását követően a JADE keretrendszer valóban megszünteti az ágenst.

Felmerülhet még a kérdés, hogy: *létezésük során hogyan kommunikálnak az ágensek? Egyáltalán, hogyan szereznek tudomást egymásról?*

A válasz (nagy vonalakban) a következő: a kommunikáció során ugyebár a FIPA által definiált **ACL** (**A**gent **C**ommunication **L**anguage) nyelvnek megfelelő üzenet-objektumok kerülnek átvitelre. Minden ágensnek van egy saját kis üzenet-sora (MessageQueue-ja) – ebbe kerülnek az ágens számára küldött üzeneteknek megfelelő `jade.lang.acl.ACLMessage` üzenet-objektum-példányok.

Az üzenetek átvitelét tehát a JADE keretrendszer oldja meg. Ezzel nekünk nem kell foglalkoznunk Számunkra elsősorban csak a `jade.core.Agent` osztálytól örökölt `send()` és `receive()` metódusok érdekesek. Az előbbi hatására egy (vagy több) címzett ágens MessageQueue-jában landol az üzenetünk, míg utóbbi esetben **FCFS** (**F**irst **C**ame **F**irst **S**erved) alapon kiveszünk egy üzenetet a MessageQueue-ból (na persze csak ha van benne).

Üzenet-küldést/fogadást az ágens bármely metódusában kezdeményezhetünk (akár `setup()`, akár `takeDown()`, vagy akár valamelyik viselkedésben is, stb). *De vajon kinek küldhetünk, vagy kitől kaphatunk üzenetet?*

A válasz: arról, hogy aktuálisan mely ágensek vannak (beregisztrálva) a platformon, a DF (Directory Faciliator) ágenstől szerezhetünk tudomást. Amint azt már a „*JADE lényege*” c. fejezetben említettük, ez lényegében egy „Yellow Pages” szolgáltatást megvalósító ágens. Nála lehet megérdeklődni, hogy kik vannak a platformon, illetve különböző kereséseket is lehet indítványozni (pl. „Kik azok az ágensek, akik egy adott szolgáltatást nyújtanak?”, stb). Mindezt akár egy viselkedésben is megvalósíthatjuk. Erre mutat példát a következő kódrészlet.

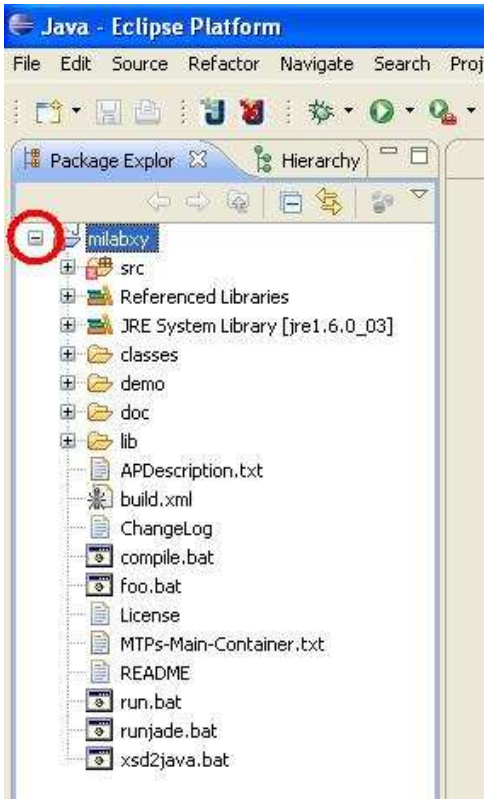
```
addBehaviour(new TickerBehaviour(this, 10000) {

  protected void onTick() {

    System.out.println("Trying to buy "+targetBookTitle);
    DFAgentDescription template = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("book-selling");
    template.addServices(sd);

    try {

            DFAgentDescription[] result = DFService.search(myAgent, template);
            System.out.println("Found the following seller agents:");
            sellerAgents = new AID[result.length];

            for (int i = 0; i < result.length; ++i) {

            sellerAgents[i] = result[i].getName();
                    System.out.println(sellerAgents[i].getName());

            }

    } catch (FIPAException fe) {

            fe.printStackTrace();

    }

    myAgent.addBehaviour(new RequestPerformer());

}});
```

A kódrészlet magyarázatába most inkább nem fognánk bele, mivel jelentős mértékben túlmutat a jelen segédlet keretein. Amennyiben megértési nehézségekbe ütköztünk a kód olvasásakor (nem éppen a Java nyelv ismeretének hiánya miatt), úgy mindenképp alaposan olvassuk el, és értsük meg a labor weblapjáról letölthető ***Kezdő JADE-programozói segédlet***et! Az ágensek programozásával kapcsolatos további útmutatásért tehát az említett segédlethez forduljunk (**elolvasása KÖTELEZŐ**)!

5. JADE és Eclipse

Ebben a szakaszban JADE-es ágensek Eclipse **IDE** (**I**ntegrated **D**evelopment **E**nvironment) szoftver-fejlesztői környezet alatti fordításával, és futtatásával foglalkozunk. Az Eclipse a desktop-on található ikonra kattintva indítható.

Az indítást követően 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!

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.

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

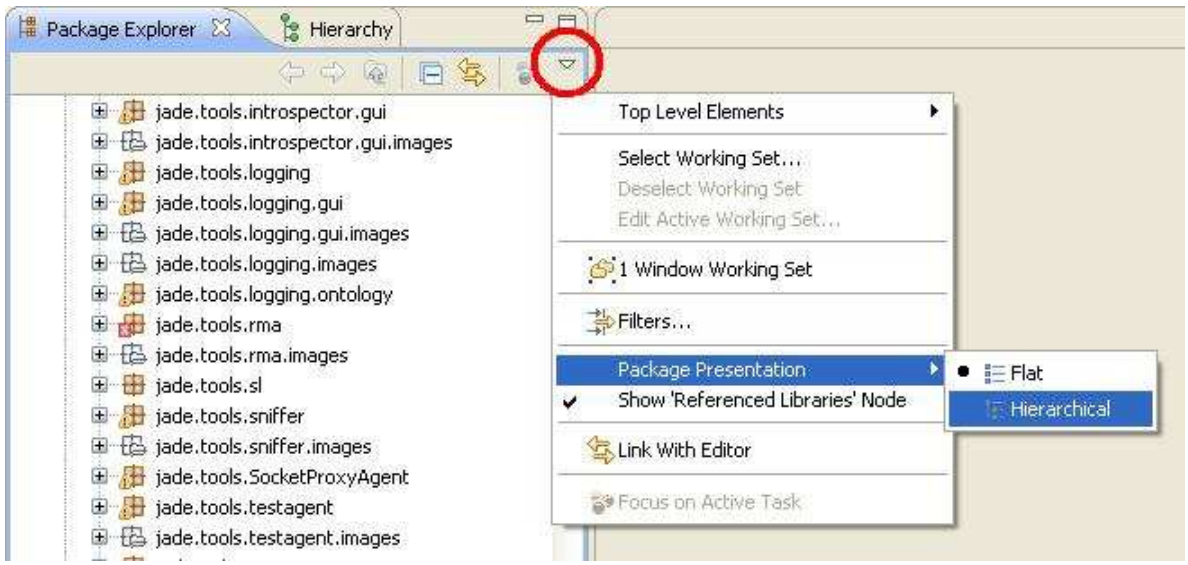
A barna csomagok a Java csomagokat (és alcsomagokat), míg a szürkék a közvetlenül csak egyéb adatokat magukba foglaló mappákat (amikben közvetlen nincs Java forráskód) jelölik.

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.[3]

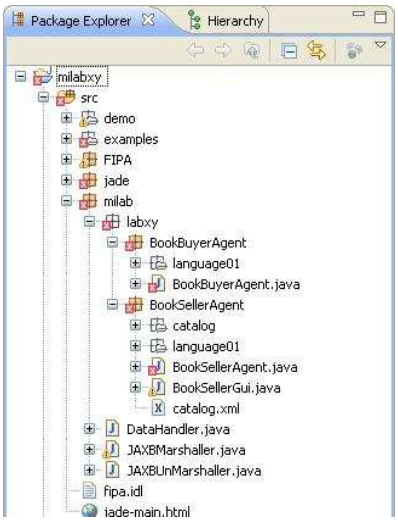
[3] Itt az általánosság kedvéért szerepel „*labxy*”. A jelen laborgyakorlat esetében ez nyilván „*lab01*” lesz, azaz „*xy*”=”*01*”.

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.



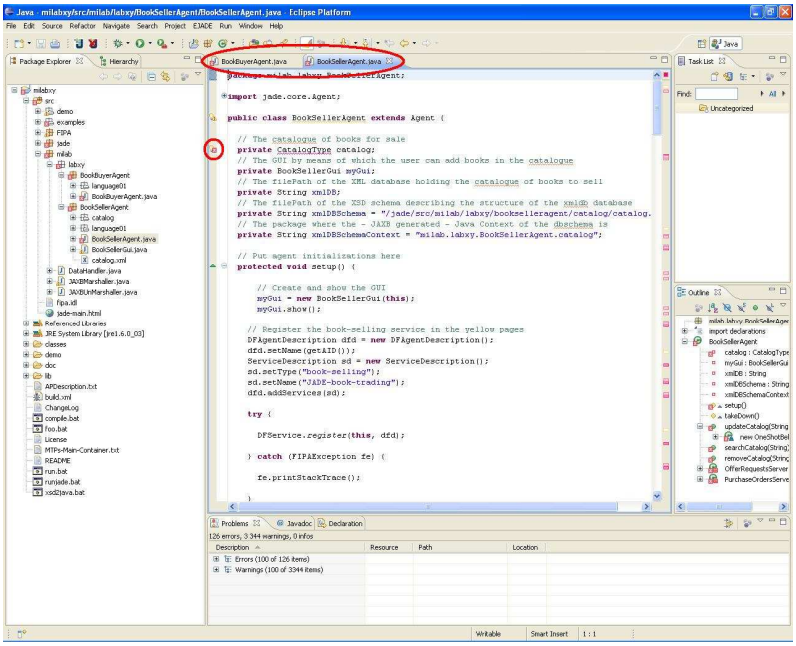
Ehhez az előbbi ábrán látható módon kattintsunk a „*Package Explorer*” jobb felső sarkában látható kicsi, lefelé mutató nyilacskára, aminek hatására egy pop-up menü jelenik meg, amiben a *Package Presentation/Hierarchical* menüpontot válasszuk!



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).

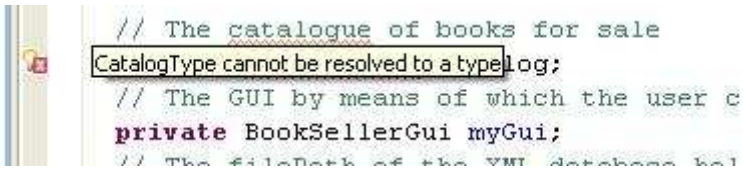
Jól láthatóan a *KönyvVásárló*, és *KönyvÁrus* ágensek forráskódjában nem stimmel valami. De vajon mi lehet a gond? Nézzünk csak bele a forráskódokba! Kattintsunk az egér bal gombjával duplán mindkét kis „**J**” ikonkára!

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. konstansokat, változókat, osztályokat, metódusokat), 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!

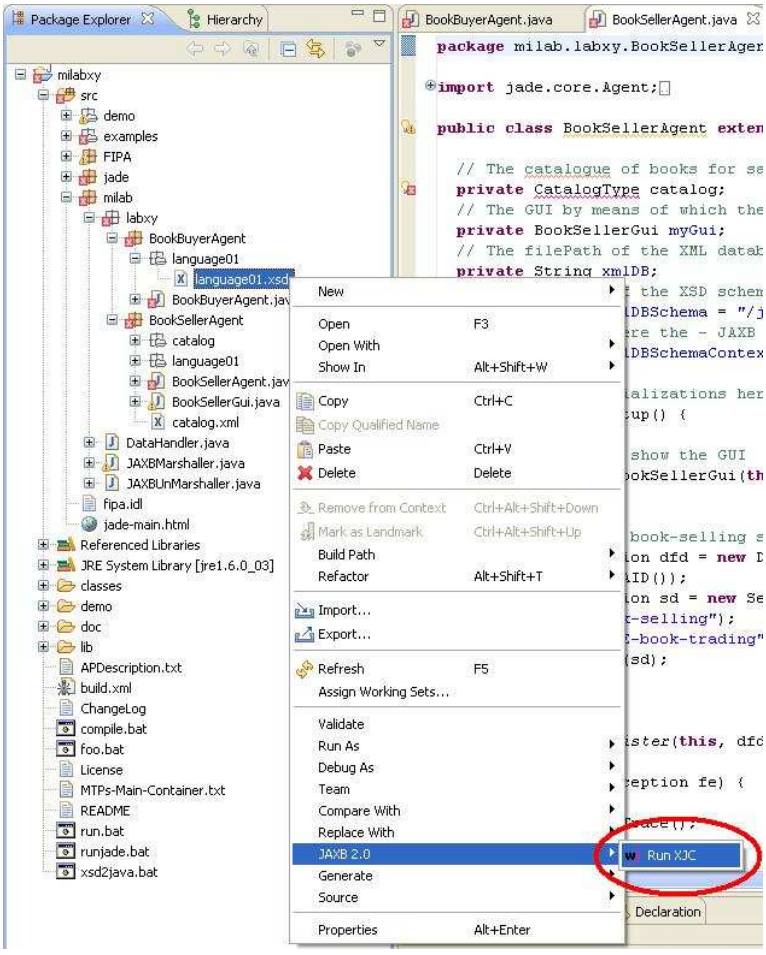


Egy kis pop-up üzenet jelzi számunkra a hiba okát: „*CatalogType cannot be resolved to a type*”. Rövid gondolkodást követően rájöhetünk arra, hogy az a baj, hogy nem tudunk `CatalogType` típusú objektumot létrehozni, mivel nincs `CatalogType` osztályunk! De miért?

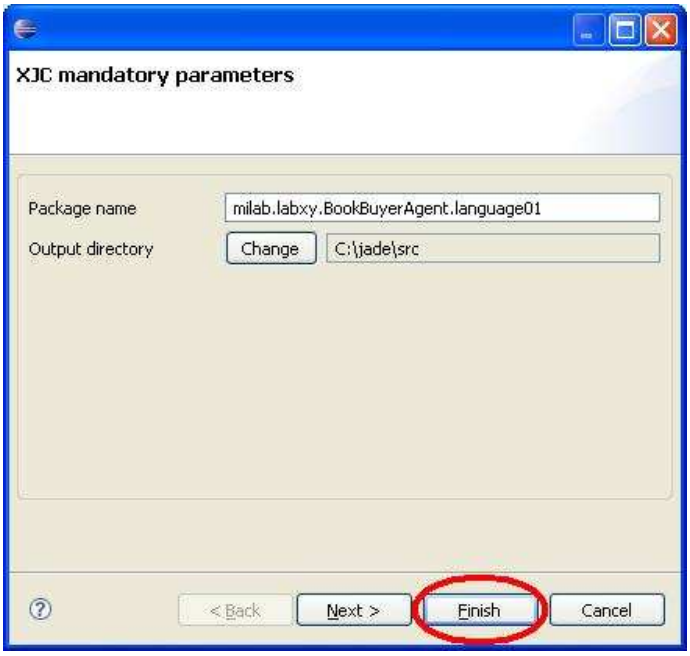
A válasz: mivel még nem fordítottuk le a megfelelő XSD sémákat (Xml-Java Compiler-rel), azaz nem állítottunk elő a megfelelő Java osztályokat az ágensek tartalomnyelveit definiáló XML sémákból (XSD fájlokból). Erről bővebben a Függelékben olvashatunk (lásd. **7.A fejezet**).

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 Függelékben leírt módon zajlik**, csak most nem parancssorból (konzolból), hanem Eclipse-ből, kényelmesen, néhány gombnyomással, a következőképp.

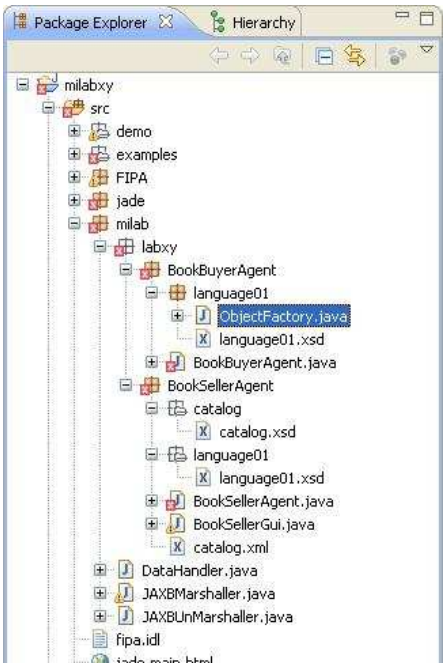


A fenti ábrának megfelelően először is böngésszük ki a fordítandó sémát a „*Package Explorer*”-ben (pl. a `\milab\labxy\BookBuyerAgent\language01\language01.xsd` sémát), majd kattintsunk rá az egér jobb gombjával. A felugró menüből válasszuk a „*JAXB 2.0/Run XJC*” opciót!



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, 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 (vagy nyomjunk egy Enter-t).

Rövid gondolkodás után lefut a fordító. Viszont egyelőre nem látható a futás eredménye (nem látszanak ez előállt Java források, stb). 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).

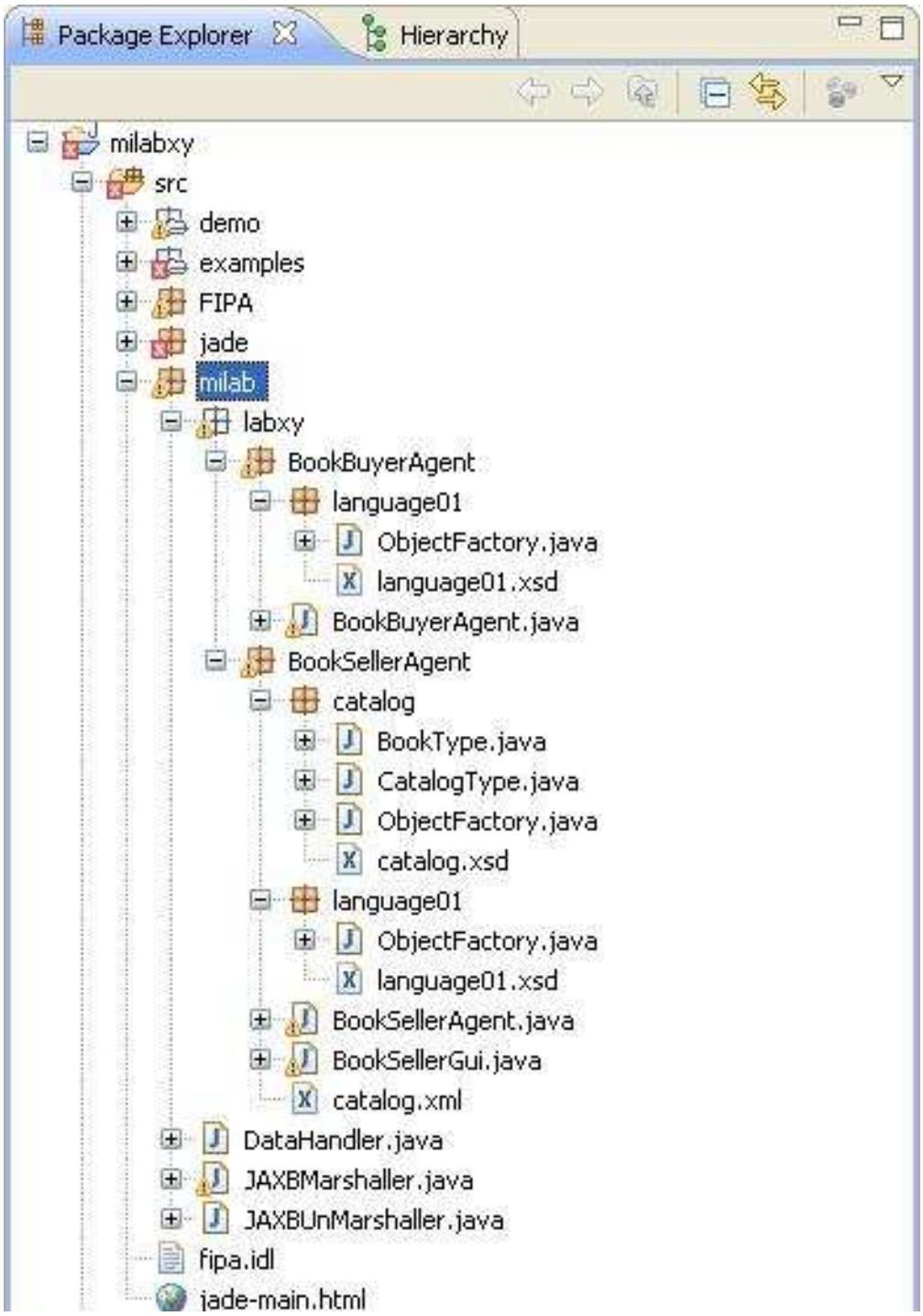


Érdemes észrevennünk, hogy a `milab.labxy.BookBuyerAgent.language01` csomag színe szürkéről barnára változott. Ennek oka nyilván az, hogy immár ebben a mappában (legalább) egy Java forráskód *is* helyet foglal.

Most pedig folytassuk a többi XSD – tetszőleges sorrendben történő – fordításával! A megfelelő lépések végrehajtását követően a következő új forrásfájloknak kellene előállni.[4]

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 az anyag előző részében bemutatott „fapados” módszer eredményének. Ráadásul a sémák lefordításának eredményeképp végre valóban maradéktalanul megszűntek a `milab.labxy` csomag „hibai”.

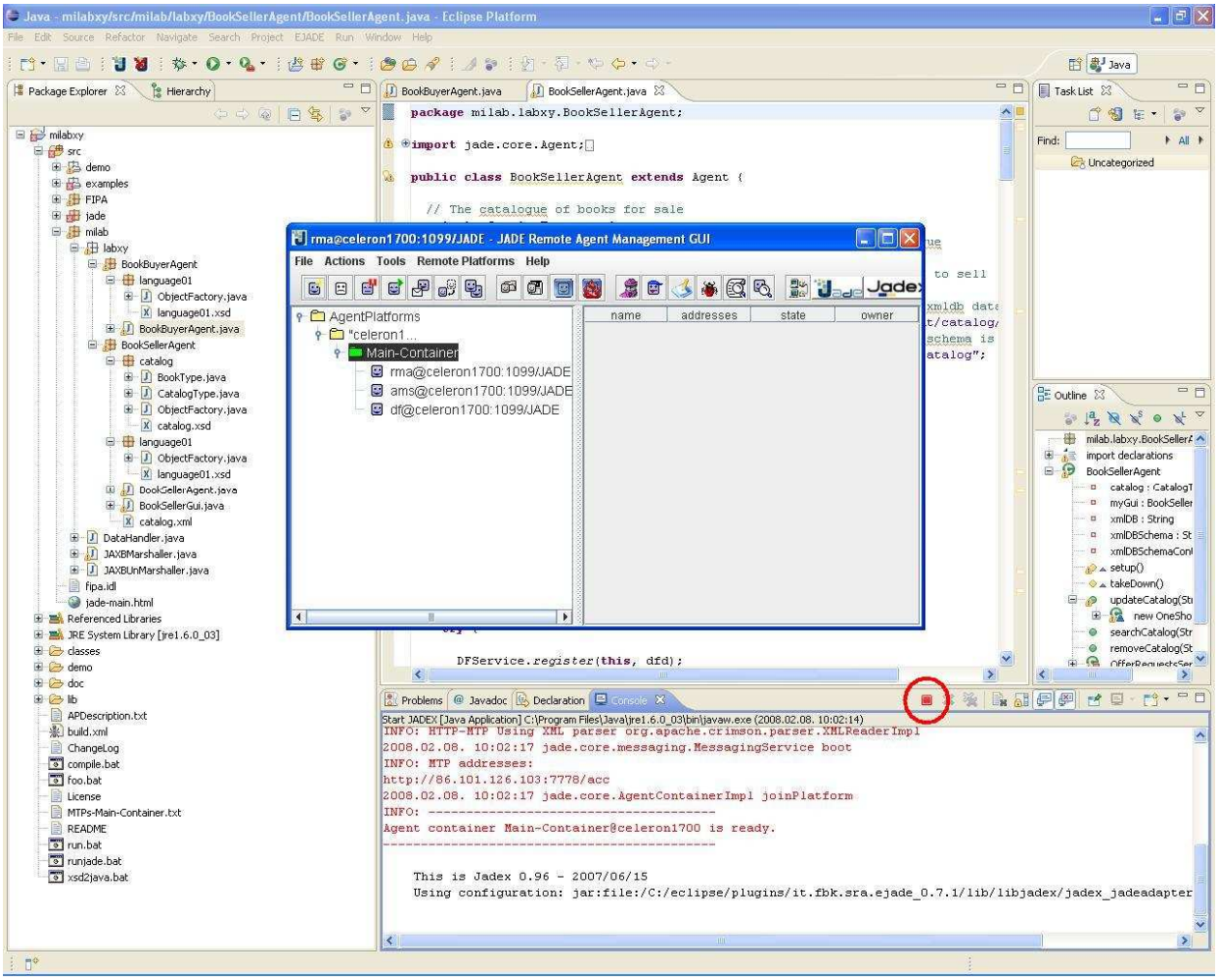


[4] A fordítás során különösen ügyeljünk a csomagok nevének (*Package name*) megadására!

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.[5]

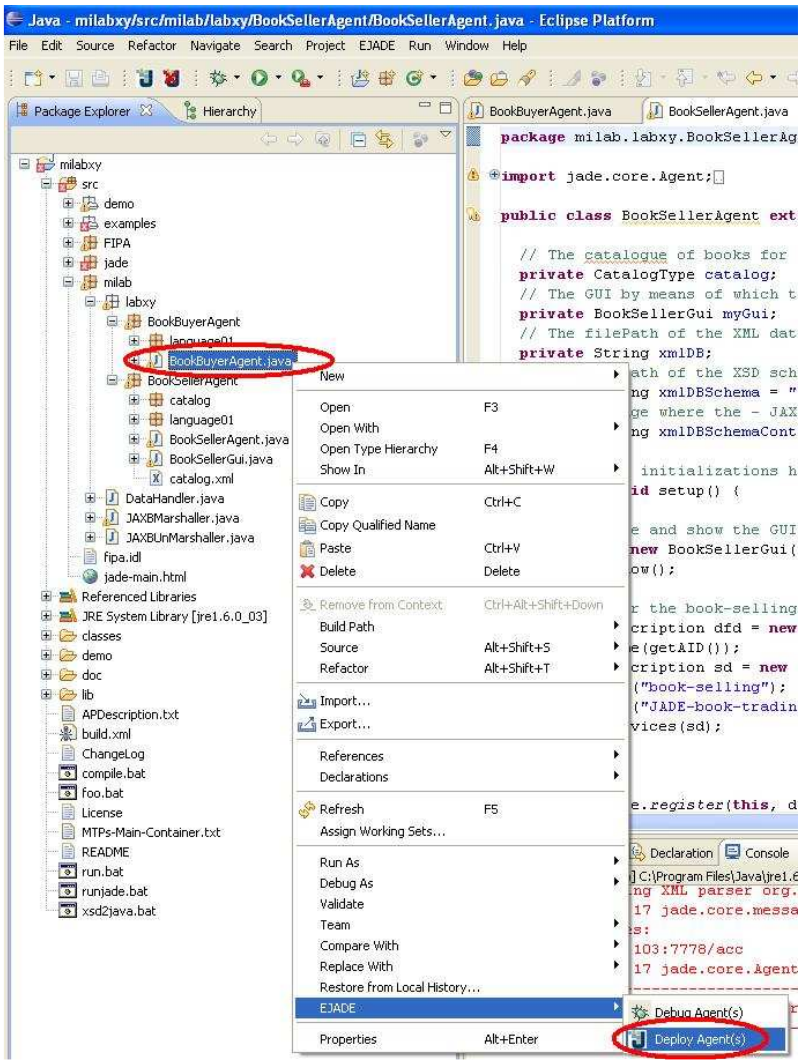
Az eredmény a már jól ismert RMA ágens GUI-jának megjelenése, és egy „*Console*” feljövetele a képernyő jobb alsó részében.



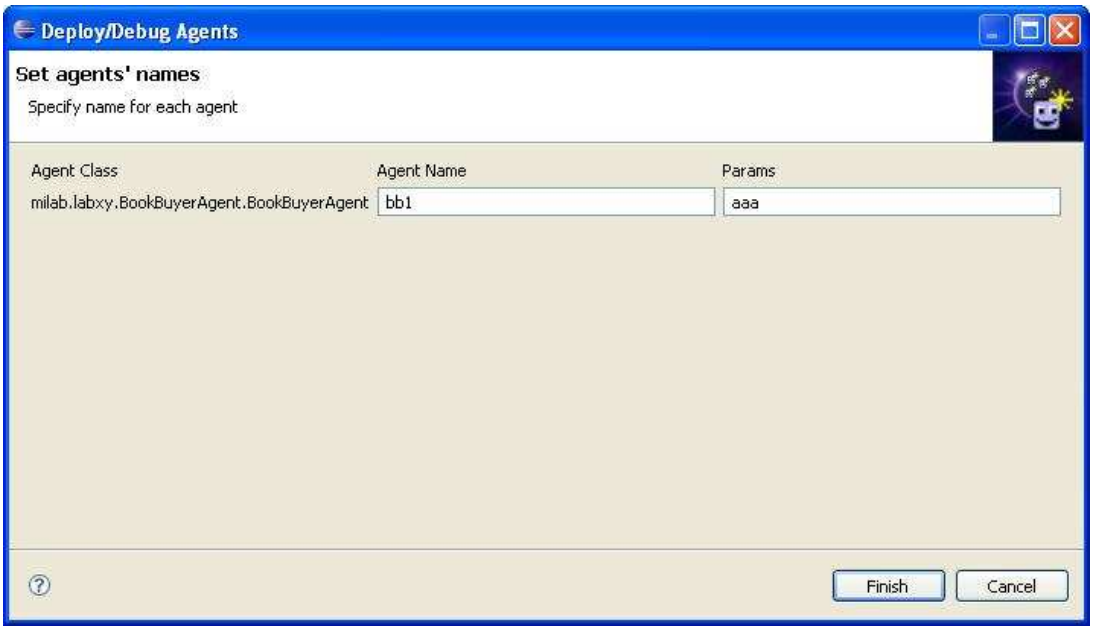
A „*Console*” gyakorlatilag a Windows-os parancssori ablaknak felel meg azzal a különbséggel, hogy nem interaktív (azaz nem írhatunk bele semmit sem a billentyűzetről). A „Console” azt is jelzi, hogy fut-e a benne foglalt folyamat. Ha igen, akkor az előbbi ábrán körbekarikázott *STOP gomb* piros (kattintható),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

[5] Egy gazdagépen (host-on) egyszerre csak egy ágens-platform futhat. Tehát esetleg állítsuk le a futó platformot, és indítsunk el egy újat!

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!

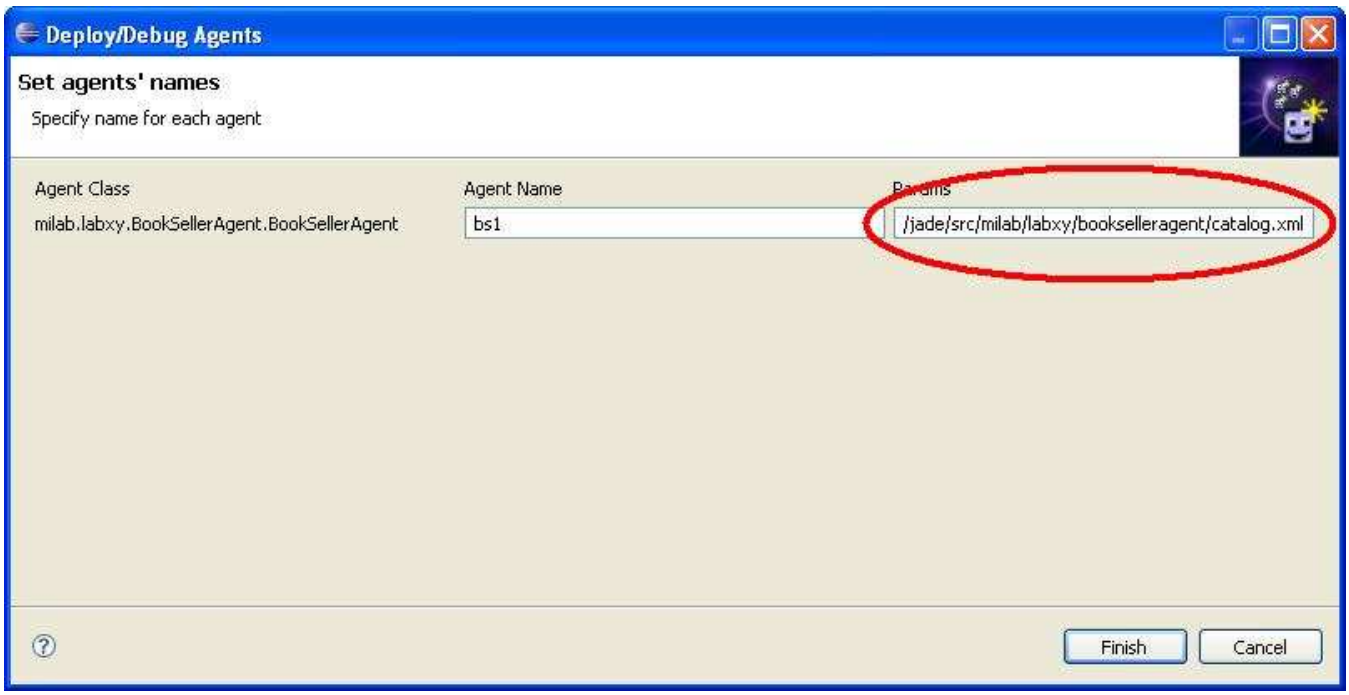


A felugró ablakban gépeljük be a létrehozni kívánt ágens nevét (ügyelve arra, hogy a platformon belül egyedi legyen), írjuk be az ágens parancssori paramétereit (szóközökkel elválasztva), majd kattintsunk a „*Finish*” gombra!

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.

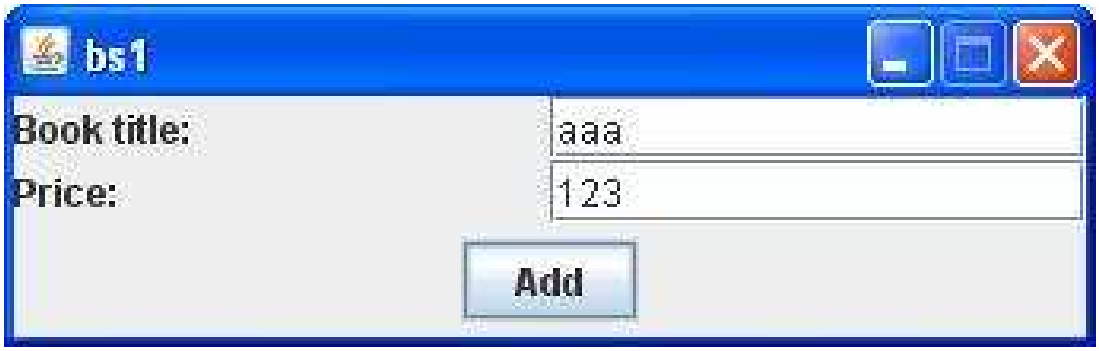
Indítsunk most el egy *KönyvÁrus* ágenst is! Legyen a neve „`bs1`”.



FONTOS: az ábrán látható módon (gyökérhez viszonyítva) adjuk meg az ágens 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 (ellenőrizzük le – kattintsunk az egér bal gombjával duplán az XML-re!), ezért – az ágens-interakció kedvéért – célszerű bevinni az „`aaa`”-t is adatbázisába.

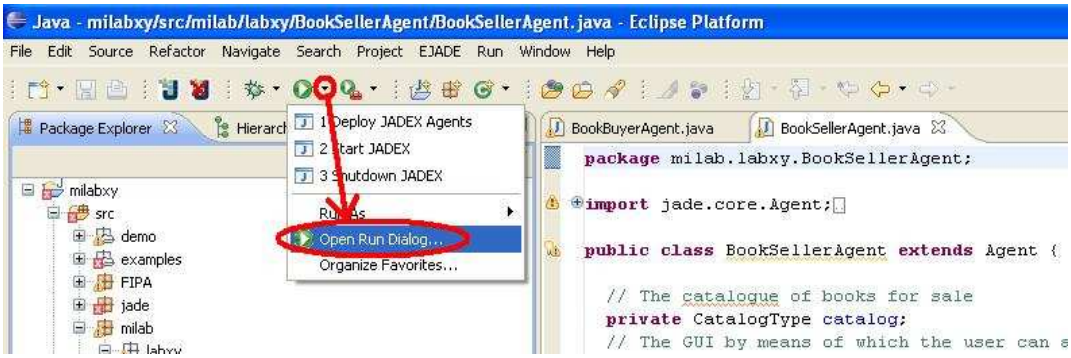


A könyv bevitelét követően – a konzolon jól látható módon – megtörténik az adás-vétel…

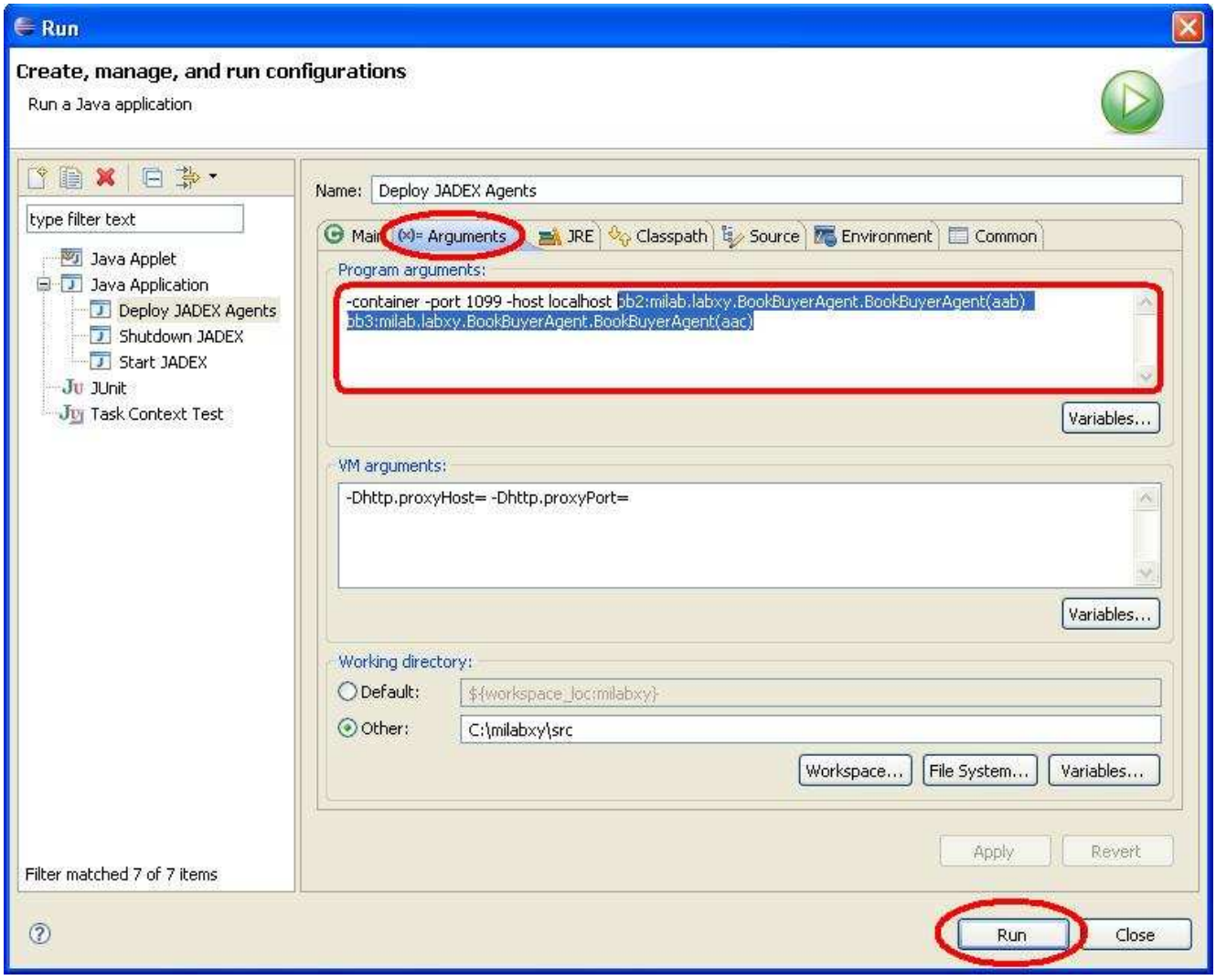
Az adás-vétel lebonyolítását követően a *KönyvVásárló* ágens szabatos módon leáll. Ezt az RMA ágens GUI-ján is megfigyelhetjük.

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. Ha 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 valamivel komplikáltabb módja.

6. Befejezés

A segédlet eddigi részében megismerkedhettük a JADE elnevezésű, Java-alapú, univerzális ágens-keretrendszer koncepciójával, használatának módjával, és (vázlatosan) programozásával. A segédlethez két további kiegészítő-anyag is tartozik:

1. **JADE-adminisztrátori útmutató (6. fejezete)**
2. **Kezdő JADE-programozói segédlet**

Mindkét anyag letölthető a labor weblapjáról! Elolvasásuk és megértésük KÖTELEZŐ a laborgyakorlat teljesítéséhez.

A segédlet hátralévő részében, a Függelékben a **JAXB** (**J**ava **A**rchitecture for **X**ml **B**inding) szoftver-architektúrával foglalkozunk. A labor során ezt használjuk majd arra, hogy XML/XSD fájlokat írjunk, olvassunk, módosítsunk.

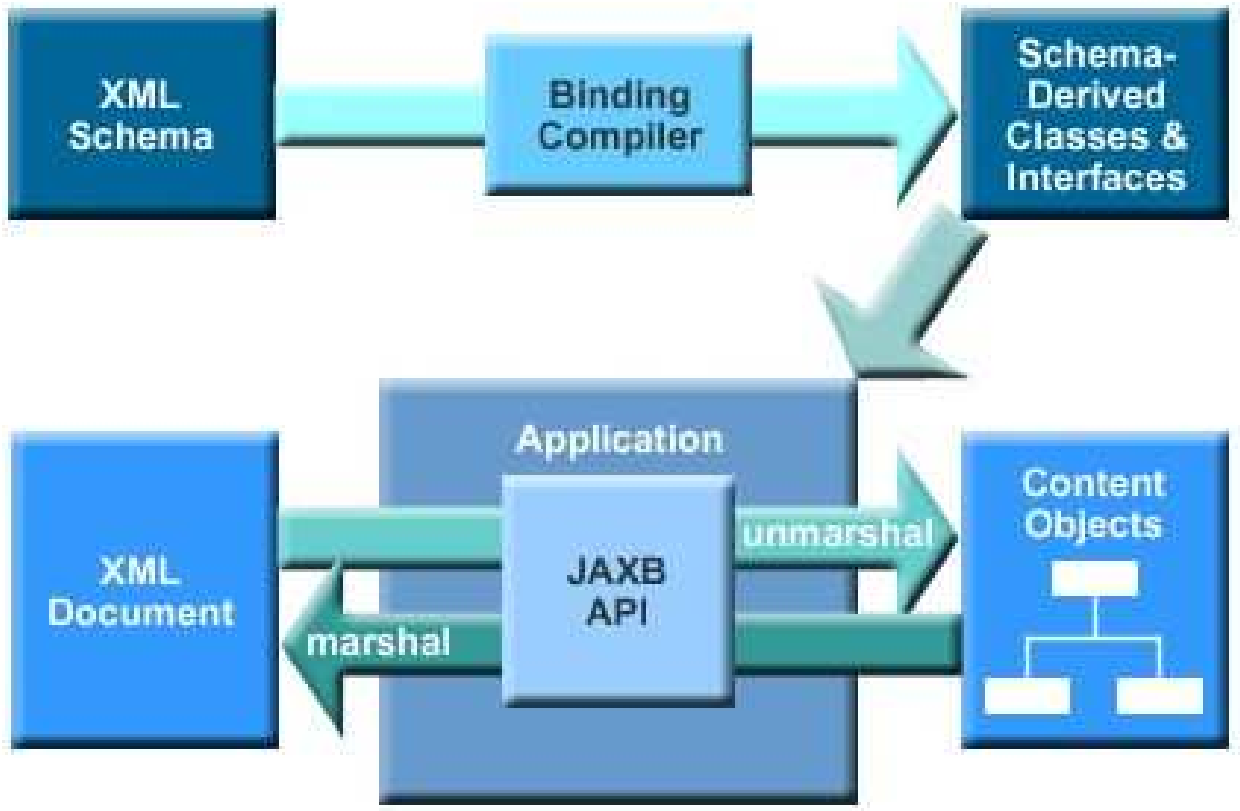
A jelen segédlet (a Függeléket is beleértve) elolvasása és megértése KÖTELEZŐ a laborgyakorlat teljesítéséhez.

7. Függelék

A. *Java Architecture for XML Binding (JAXB)*

A laborgyakorlatok során használt JADE ágensek XML tartalmú ACL üzenetekkel kommunikálnak egymással. Az üzenetek tartalmát JAXB (Java Architecture for Xml Binding) felhasználásával állítjuk elő, illetve dolgozzuk fel (lásd. pl. http://java.sun.com/developer/technicalArticles/WebServices/jaxb). A JAXB tehát egy szoftver-architektúra, amely XML/XSD kezelést biztosítja a laborgyakorlatok ágensei számára.

A JAXB külön letölthető formában is elérhető, azonban mi az egyszerűség kedvéért 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 (XSD) központi szerepe. 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, amivel 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 „`author`”) 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.[6]

X:\jade\src\milab\lab01\fooagent\xsd\foo.xsd

Tegyük fel továbbá, hogy az XSD-t az

X:\jade\src\milab\lab01\fooagent

könyvtárba szeretnénk lefordítani. Ehhez először is lépjünk be (pl. parancssorban) az X:\jade könyvtárba, és ott írjuk be a következőt.

xsd2java milab.lab01.fooagent milab\lab01\fooagent\xsd\foo.xsd

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

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.lab01.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\lab01\fooagent\ArticleType.java
milab\lab01\fooagent\CatalogType.java
milab\lab01\fooagent\JournalType.java
milab\lab01\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 is létre egy kis alkalmazást (nevezzük fooagent-nek), amelyik a JAXB API segítségével bemutatja az oda-visszarendezést (azaz a Marshalling-ot, és UnMarshalling-ot). Ehhez először is tekintsük a következő forrásfájlt!

X:\jade\src\milab\lab01\fooagent\fooagent.java

Vegyük észre, hogy ez nem egy szokványos JADE ágens forráskódja, pusztán csak egy sima Java program, amely a JAXB API használatát szemlélteti. Ahhoz, hogy a programot le lehessen fordítani, szükség van még 3 további osztályra is.

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

A „`JAXBMarshaller.java`”, „`JAXBUnMarshaller.java`”, és „`DataHandler.java`” osztályokkal egyelőre ne foglalkozzunk többet. 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.lab01.fooagent");

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.lab01.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\lab01\fooagent\fooagent.java
run milab.lab01.fooagent.fooagent writefile src\milab\lab01\fooagent\foo.xml
```

Az első sor lefordít mindent, ami szükséges[7], míg a második – adott argumentumokkal – lefuttatja az előállt „`milab.lab01.fooagent.fooagent`” osztályt/programot (`fooagent.class`). A futás eredményeképp a következő XML fájl áll elő.

X:\jade\src\milab\lab01\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.lab01.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)[8].
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 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`).[9]

[7] fooagent.java, ArticleType.java, JournalType.java, CatalogType.java, ObjectFactory.java, JAXBMarshaller.java, JAXBUnMarshaller.java, DataHandler.java – figyeljünk a könyvtárak megadására!
[8] Lásd. a http://java.sun.com/docs/books/tutorial/java/IandI/subclasses.html oldalon a „Casting Objects” szakaszt
[9] Lényegében a ***DataHandler.readFile2String*** metódus „inverzéről” van szó…

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.lab01.fooagent.fooagent writestring src\milab\lab01\fooagent\foo.xml
run milab.lab01.fooagent.fooagent readfile src\milab\lab01\fooagent\foo.xml
run milab.lab01.fooagent.fooagent readstring src\milab\lab01\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.