Drools Business Logic integration Platform/

2014. Szeptember

Informatikai Navigátor

Az üzleti szabálymotorok

A Drools Business Rules Management System gyakorlati használata

CreedSoft

11. szám

Tartalomjegyzék

Főszerkesztő: Nyiri Imre (imre.nyiri@gmail.com)

1. Üzleti szabályok és informatikai támogatásuk

Ebben a bevezető részben áttekintjük az üzleti szabályok (*Business Rules*) fogalmát és a hozzá kapcsolódó informatikai megoldásokat, amiket üzleti szabálymotoroknak (*Business Rules Engine=BRE*) nevezünk. Bemutatjuk azokat a főbb alkalmazási területeket, ahol a BRE eszközöket használni lehet, miközben az egész témakört egy olyan megvilágításba helyezzük, ami segíti a további fejezetek könnyebb megértését.

A szabály alapú megoldások nem ismeretlenek az informatikában,hiszen például az egyes programok konfigurációi, a program építéshez használt MAKEFILE sokak számára ismert lehet. Mindezt egy még tágabb fogalmi körbe helyezhetjük, amit *deklaratív* megoldásnak (programozásnak) hívunk. Ekkor specifikáljuk a feladatot az adott specifikációs nyelven (példák: SQL, Prolog, XSL az XML szerkezetek transzformálásához) amire a megoldást a futtató vagy fordító környezet generálja le. A hagyományos programozási eszközöket ebben a kontextusban *imperatív* (utasításokból álló) nyelveknek nevezzük (példák: C, Java, C#). A deklaratív nyelvek általában a tények (*facts*) leírását és az alkalmazható szabályok (*rules*) megfogalmazhatóságát biztosítják számunkra valamilyen erre a célra szolgáló specifikációs nyelven. Ilyen lehet például a matematikai logika nyelve is. A Business Rules motorok is valamilyen deklaratív nyelven megfogalmazott szabályhalmazok fölött dolgoznak, azokat értékelik ki és kezdeményezik az ebből következő akciók végrehajtását vagy események generálását.

Produkciós rendszerek

A szabály alapú szakértői rendszerek a világról alkotott ismereteket egy tudásbázisban (*knowledge base*) tárolják, aminek két része van:

1. Ténybázis (*fact base*): A kész tényállításokat tartalmazza.
2. Szabálybázis (*rule base*): A következtetési szabályokat tartalmazza.

A következtetési szabályok *ha-akkor* alakúak, azaz minden szabály egy a *ha* utáni feltétel és egy következmény részből áll. A feltételt gyakran ilyen neveken emlegetjük: bal oldal (*Left Hand Side=LHS*), előfeltétel (*precondition*). A következményt pedig *postcondition* vagy *Right Hand Side=RHS* néven szoktuk emlegetni. Az üzleti szabályokról szóló szakirodalom leggyakrabban csak az *LHS* és *RHS* megnevezéseket szokta használni.

A szabályok láncolódása

A szabályok láncolódása azt jelenti, hogy egy szabály következmény része is teljesítheti egy másik szabály feltétel részét. Ez azt jelenti, hogy emiatt a kiinduló szabályunknak – más szabályok miatt – további következményei is vannak. A láncolási mechanizmusnak 2 fajtája van:

1. Hátrafelé láncolt következtetés (*backward chaining*): Ebben az esetben a következmények alapján próbáljuk meghatározni az ismeretlen előfeltételeket. Másképpen fogalmazva ez a

láncolási eset egy automatikus tételbizonyítást próbál megvalósítani, azaz adott egy állítás (tétel) és azt akarjuk tudni, hogy az igaz vagy hamis. Az ilyen rendszerre jó példa a *Prolog* nyelv.

2. Előrefelé láncolt következtetés (*forward chaining*): Ekkor a tényekből kiindulva próbálunk újabb tényeket generálni a szabályaink használatával. Ezt a megközelítést nevezzük emiatt *produkciós rendszereknek*, az üzleti szabálymotorok is jellemzően ezt használják.

A szakértői rendszerekben (és a mesterséges intelligenciában) mindkét láncolást használják ugyanis a feladat néha egy állítás igazságának leellenőrzése, máskor pedig új tények generálása.

Egy Prolog nyelven elkészített példa a hátrafelé láncolt működésre

Az 1-1. Programlista tartalmazza a példa prolog programunkat, ami a fájlrendszerben a *csalad.pl* fájlban található.

1-1. Programlista: csalad.pl

```
1 apja("Imre","Imrus").
2 anyja("Saci","Imrus").
3 apja("Imre","Zsazsa").
4 anyja("Saci","Zsazsa").
5 anyja("Saci_anyja","Saci").
6
7 szuloje(SZULO, GYEREK):-anyja(SZULO,GYEREK).
8 szuloje(SZULO, GYEREK):-apja(SZULO,GYEREK).
9 nagyszuloje(NAGYSZULO, UNOKA):-szuloje(NAGYSZULO, SZULO), szuloje(SZULO,UNOKA).
```

Az első 5 sor konkrét tényeket sorol fel, ezek konstans logikai mondatok, azaz igaz állítások (ezeket hívtuk *facts*-nak). Az *anyja("Saci","Zsazsa")* például azt jelenti, hogy *Saci* az anyja *Zsazsának*. Aki tanult logikát, annak ezek ismerős alakú formulák. Van egy halmazunk: $CSALAD := \{Imre, Imrus, Saci, Zsazsa, Saci\_anyja\}$. Az állítások ezen halmaz elemeire vannak megfogalmazva, azonban a prolog nem igényli ennek az explicit megadását, helyette futtatás közben tudja ezt a halmazt saját magától összeállítani. A 7-9 sorok között szabályokat adtunk meg, ahol változókra (ami nagybetű és nincs idézőjelben) hivatkozunk, illetve a *:-* jel a *Ha-Akkor* (logikai implikáció) jelölése. A 7. sor például azt a szabályt fogalmazza meg egy 2 változós logikai formula segítségével, hogy ha valaki (*SZULO* változó) anyja másvalakinek (*GYEREK* változó), akkor ő egyben a szülője is. A 9. sor már egy 3 változós logikai szabály, ami a szülő szabály alapján megadja a nagyszülőnek lenni tulajdonság jelentését. Az 1-2. Programlista már ennek a „kis világnak" a használatát mutatja be, amihez a népszerű és open source *SWI Prolog* (webhely: `http://www.swi-prolog.org/`) rendszert használtuk.

1-2. Programlista: Lekérdezések a Prolog környezetben

```
1 ?- [csalad].
2 % csalad compiled 0.00 sec, 9 clauses
3 true.
4
5 ?- apja("Imre","Imrus").
6 true .
7
```

```
8  ?- anyja("Imre","Imrus").
9  false.
10
11 ?- szuloje("Imre","Zsazsa").
12 true.
13
14 ?- szuloje("Imre","Saci_anyja").
15 false.
16
17 ?- szuloje("Saci_anyja","Imre").
18 false.
19
20 ?- szuloje("Saci_anyja","Saci").
21 true .
22
23 ?- nagyszuloje("Saci_anyja","Saci").
24 false.
25
26 ?- nagyszuloje("Saci_anyja","Imrus").
27 true .
28
29 ?- nagyszuloje("Saci_anyja","Zsazsa").
30 true .
```

Az 1.sorban a családdal kapcsolatos tudáshalmazt töltjük be a prolog futtató környezetbe. Az 5-30 sorok között kérdéseket (tételeket) fogalmaztunk meg és megkérdeztük a rendszertől, hogy az igaz vagy hamis-e. Az 5-9 sorok közötti működés persze még nem meggyőző, hiszen csak logikai konstans állítások, azaz a tényeink, igazságát vagy hamisságát tudakoltuk meg. A 11.sor már érdekesebb, ugyanis ott már a *szuloje* szabályt is használtuk, bár a tények összeillesztése a szabállyal még mindig csak 1 lépésesek. Ez az 1 lépés jelen esetben csak annyit tesz, hogy megnézi, hogy a *szuloje* szabály 2 konkrét input eleme (*Imre* és *Zsazsa*) illeszkedik-e az *apja* vagy *anyja* tényekre. A *szuloje("Imre","Zsazsa")* például visszavezethető az *apja("Imre","Zsazsa")* tényre és így ez a állítás is igaz lett. A *nagyszuloje("Saci_anyja","Imrus")* igazsága (29. sor) ezen a visszafelé láncoláson keresztül bizonyosodik be:

```
nagyszuloje("Saci_anyja","Imrus") ->
szuloje("Saci_anyja","Saci") és szuloje("Saci","Zsazsa") ->
anyja("Saci_anyja","Saci") és anyja("Saci","Zsazsa")
```

A fent bemutatott lépéseket szabály vagy minta illesztésnek szokták nevezni. Az egyes rendszerek jóságát azon is le lehet mérni, hogy ezt az illesztést mennyire hatékony algoritmussal tudja megvalósítani, ami nagyszámú változó és szabály esetén már nagyon fontos szempont.

Egy döntési tábla alapú példa a előre következtető működésre

Az előre láncolásos következtetés az üzleti szabályok megfogalmazása során több alkalommal használatosak, ugyanis jellegénél fogva pont azt tudja, mint ami ott szükséges. A kiinduló tények és a következtetési szabályok alapján levezet újabb tényeket, amelyeknek a hatására rendszerint valamilyen cselekvés (*action*) elvégzésére is sor kerül. A működés a generatív nyelvtanokra hasonlít, hiszen azok is produkciós rendszerek. Vegyük például ezt a 2 szabályt: (1) $S \rightarrow aSb$, (2) $S \rightarrow b$. Az S kiinduló szimbólumból vagy az (1) vagy a (2) szabályt tudjuk használni, amivel ezen karaktersorozatok halmazát tudjuk előállítani: $\mathcal{L} = \{b, abb, aabbb, ...\}$. Azt mondjuk, hogy például a fenti 2 szabály az *aabbb* szót is le tudja generálni, azaz ez is következik a rendszerből. A régóta használt

döntési táblák is pont ezen viselkedés mentén tudják megfogalmazni a szabályokat. Tekintsük a következő táblázatos elrendezésű szabályrendszert:

		Szabályok			
Feltételek	A beszerzés értéke	<100 eFt	>100 eFt	<100 eFt	>100 eFt
	A felhasználás helye	Lokális	Regionális	Lokális	Regionális
	Van rá keretszerződés?	-	Igen	-	Nem
	Korábban már vettünk ilyet?	-	Igen	-	Nem
Teendők	Telepi engedélyezés	X		X	X
	Központi engedélyezés		X		X
	Keretszerződés kötés				X
	Rendelés	X	X	X	X

Az *X* azt jelöli, hogy egy oszlop összes feltételének teljesülése esetén azt a teendőt végre kell hajtani. Ez a szerkezet egy előre következtető keretrendszer számára triviálisan egyszerű forma, így az ilyen szoftverek rendszerint alapból ismerik a döntési táblák használatát. Egy *RACI* mátrix is egyfajta döntési tábla, így egy produkciós rendszer számára elsődleges jelölt lehet, amennyiben azt digitalizált formában szeretnénk kezelni.

A produkciós rendszereket megvalósító szoftverek belső felépítése

A produkciós rendszerek a tényeket a ténybázisban tárolják, ami egy munkamemória (*Working Memory*=*WM*). A WM képes dinamikusan módosulni, a szabályok működése során újabb tényelemek tudnak eltárolódni. Amennyiben egy produkciós szabály minden feltétele teljesül, akkor a szabály aktivált (*activated* vagy *triggered*) állapotba kerül. Mindez azért történik, mert a szabályok LHS részében változók vannak, amik konkrét értékekkel helyettesítődnek és egyes esetekben ettől igazzá válik a szabály LHS, azaz feltétel része. Mindez az RHS részben megfogalmazottak végrehajtását váltja ki. Egy produkciós rendszerben általános alapelv, hogy az előre való következtetés mindaddig folytatódik, amíg van alkalmazható szabály. Mindezen működés a mesterséges intelligencia egy területe, ahol a szabályalapú szakértői rendszereket így valósítják meg. Az elmúlt években megszülettek a hatékony, inkrementális mintaillesztő algoritmusok: RETE, TREAT, LEAPS.

A szabályalapú üzleti logika támogatása

A hagyományos programokban az objektumokra jellemző üzleti logikát a forráskód sok *if-else* utasítása valósítja meg. Ezzel nincs semmi gond, de sok esetben rugalmasabb, áttekinthetőbb és gyorsabban változtatható lenne az alkalmazás, ha ezek a szabályok részben egy külső business rules engine-ben (*BRE*) foglalnának helyet. A BRE képes az üzleti objektumokat figyelni, a rá vonatkozó feltételeket vizsgálni (LHS oldal), valamint megváltoztatni őket (RHS oldal). Ez azt jelenti, hogy programunk ezeken az üzleti objektumokon keresztül tud kapcsolatba lépni egy BRE megoldással és azt akkor használja, amikor néhány *if-else* keretén belül különben is megváltoztatná. Például vegyünk egy banki ügyfelet, akinek számlája van. Amennyiben az ügyfél fiatalabb 25 évesnél és egyetemre jár, úgy ajánljunk neki ifjúsági számlavezetési lehetőséget, ami a banki szolgáltatásokhoz való hozzáférést sokkal kedvezőbbé teszi. A BRE környezetek általában sok komponenssel rendelkező eszközkészletet adnak: BRE szerver, szabályfejlesztő környezet, verziókezelés, esemény

menedzsment. Mindezt együtt *Business Rules Management System* (*BRMS*) néven szoktuk emlegetni, ilyen eszköz a továbbiakban bemutatandó *Drools* is (webhely: `http://www.drools.org/`), ami egy open source és elterjedt BRMS platform. Ehhez hasonló környezet még az IBM ILOG, Blaze Advisor, MS BRE vagy a Tibco iProcess.

Alkalmazási területek

Az első pont befejezéseként röviden összefoglaljuk azokat a területeket és ott alkalmazott jellemző módszereket, ahol a BRMS jól használható.

Szakértői vagy döntéstámogató rendszerek

Minden szakterületnek megvan a rá jellemző tudása, amit az ilyen rendszerek képesek eltárolni és ez alapján a feltett kérdésekre válaszolni. Mindezt öntanuló módon. Egy orvosi alkalmazásban például a tünetek a tények, ezek alapján a rendszer javaslatokat ad a lehetséges betegségekre és terápiákra. Az ismeretreprezentáció a már említett *facts* és *rules* tárolása. A BRMS keretrendszerek a modern szakértői rendszerek elkészíthetőségének alapjait teszik le úgy, ahogy az adatkezelést az SQL szerverek is univerzálisan meg tudták oldani.

A Business Process Management alkalmazások támogatása

A BPM eszközök terjedésével együtt erős igény mutatkozott arra, hogy az üzleti folyamatokat kísérő szabályok ne a folyamatba legyenek bedrótozva, hanem egy külső, akár az üzleti ember által is elérhető helyen legyenek, számára is konfigurálható módon. A számítási és eljárási szabályokon felül itt egy speciális igény a feladatok lehetséges végrehajtóinak (döntéshozó, endorsement, információt kapó, előzetes jóváhagyó) a meghatározása. Ezt a BPMN tervezés nyelvén úgy fogalmazhatjuk meg, hogy a humán taszkok ACTOR halmazát kell meghatároznunk. Itt a szabálymotor több helyről veszi a tényeket: döntés-hatáskör lista szabályzat, HR adatbázis, ami megmondja, hogy egy konkrét szervezeti pozícióban éppen melyik ember (vagy emberek) van(nak). A helyettesítés és eszkaláció meghatározása is igénybe vehet külső szabálytámogatást.

Üzleti szabályok digitalizálása

A mai összetett világban néha a szabályok is elég bonyolultak már. Jól jöhet egy olyan jogtár, ahonnan nem csak a szövegek, hanem a célvezérelt jogszabály használat is elérhetőek. Egy vállalatnál jó példa a döntés-hatáskör lista vagy egyéb más, esetleg technológiai mélységű szabályzatok digitalizálása.

Tudásmenedzsment

Egy szervezet értéke a közös tudás, aminek reprezentálása nem is annyira triviális feladat. A modern kereső-indexelő rendszerek sok mindent megoldanak, de a tudást magát nem reprezentálják. Előrehaladott kutatások vannak a szemantikus web használatával kapcsolatosan, ennek egy fontos része lehet egy BRMS alapú, mesterséges intelligencia rendszer.

2. Drools kezdőlépések

Ebben a részben a Drools BRMS rendszer használatát mutatjuk be, de kitérünk a fejlesztői környezet javasolt kialakítására is. A fő cél az, hogy megtegyük az első lépéseket egy olyan rendszer használata felé, ami lehetővé teszi a produkciós szabályok megfogalmazását és végrehajtását. Azt is látni fogjuk, hogy mindezt hogyan lehet elérni egy külső környezetből.

A Drools környezet fontosabb komponensei

A Drools BRMS részei

A JBoss Drools az üzleti objektumokra vonatkozó szabályok megfogalmazását és azok manipulálását teszi lehetővé. Maguk az objektumok pedig JavaBean-ek. A Drools platform fontosabb részei a következőek.

- *Drools Expert.* Ez maga a Business Rules Engine, azaz ismeri a szabályleíró nyelveket, így képes a megfogalmazott szabályokat végrehajtani. Mindehhez egy kiterjedt API-t biztosít, amin keresztül az expert modullal a külső program kommunikálni tud.

- *Drools Flow.* A szabályalapú motor a sorrendiség és a vezérlési folyam megadásában nem ad kielégítő megoldást, így gyakran ezt a modult is használnunk kell. Segítségével egy folyamat is megadható, aminek csomópontjaiban az üzleti szabályok egy-egy csoportja (*ruleflow group*) hajtható végre. Itt természetesen algoritmust futtató (imperatív) vagy humán csomópontok is lehetnek, amik az eredmény legyártásában közreműködhetnek. Emiatt ez a modul valójában egy workflow motor.

- *Drools Fusion.* Ez egy komplex esemény-feldolgozást (*Complex Event Processing*) végző motor, ami különféle forrásokból esemény objektumokat tud fogadni, amikre az expert modullal együttműködve reagálni tud.

- *Drools Planner.* Különféle optimalizálási és tervezési problémák megoldását támogatja. A feladatok pontos megfogalmazását és megoldását erősen a szabályalapú környezetre támaszkodva igyekszik megoldani. A szabályok egy problémateret tudnak generálni, amiben egy keresési eljárás segítségével tudja megtalálni a lehetséges megoldásokat.

- *Drools Guvnor.* Egy olyan keretalkalmazás, ami összefogja az egyes modulokat, támogatja az erőforrások központi tárolását és azok verziókezelését. Emiatt ez az alkalmazás az, ami BRMS megoldássá integrálja össze a fenti modulokat.

A fenti modulok egyben különböző Drools projektek, amiket itt érhetünk el: `http://www.drools.org/`. A munkánk során az egyes részeket külön-külön is használhatjuk, mindegyik modul telepítési készlete külön is letölthető. Egy jól átgondolt szerver környezet kialakítását a fenti modulokból a JBoss (Red Hat cég) elvégzi helyettünk, így annak a használatát javasoljuk. Regisztráljunk a `https://www.jboss.org` helyen (vagy használhatjuk ezt az accountot: *creedsoft.org/111111*). A

`http://www.jboss.org/downloads/` helyről minden előre összeállított platform megoldást elérhetünk, így egy komplett BRMS szerver környezetet is, amit a *Red Hat JBoss BRMS* címke alatt találunk (az installer jelenlegi letölthető verziójának a neve: *jboss-brms-installer-6.0.1.GA-redhat-4.jar*). Ez egy telepítő program, ami a teljes JBoss Drools BRMS szerver környezetet feltelepíti a gépünkre, az egésznek egy JBoss alkalmazás szerver lesz a futtató környezete. Telepítéskor megkérdezi az *admin* accountot (mi ezt adtuk meg: admin/Imzsa12@). A BRMS webes GUI a szerver indítása után innen érhető el: `http://localhost:8080/business-central`.

A Drools Expert első áttekintése

A Drools Expert a szabálymotor megvalósítása, használható külön Java könyvtárként is. Érdemes ezt a modult önmagában is megérteni és használni, hiszen ez a JBoss Drools BRMS környezet BRE része. Amikor egy tetszőleges Java programból el szeretnénk érni a *BRE* szolgáltatásokat, akkor egy session-t kell a tudásbázishoz (*Knowledge Base*) létrehoznunk. A tudásbázis az a hely, ahol a szabályainkat (*rules*) tároljuk, a *session* pedig közli ezzel a környezettel az éppen vizsgálandónak tartott objektumok halmazát, aminek tulajdonságai adják a ténybázist. A knowledge base része még továbbá az a folyamat leíró lehetőség (*rule flow*), ami azt határozza meg, hogy a szabályainkat milyen sorrendben kell több lépésben alkalmazni. Egy tudásbázis felé természetesen több független session is nyitható, aminek egyébként 2 fajtája létezik:

1. *Állapotmentes session* (*stateless session*): Nem rendelkezik emlékező munkamemóriával (*Working Memory=WM*), de egyszeri végrehajtásra átadható egy vagy több objektum, ez fogja képezni a ténybázist. Minden szabály tüzelni fog, amelyre a feltétel teljesül, de a működés láncolás nélkül befejeződik. Ez a session tipikusan az egyszerű esetekre alkalmazható, ahol csak néhány objektumot adunk át tényként és ezeket egymásra ható szabályok nélkül szeretnénk feldolgozni.

2. *Állapottal rendelkező session* (*statful session*): Itt van WM, amibe menet közben új objektumokat tehetünk be, régieket vehetünk ki. A *fireAllRules()* metódus hívása kiváltja a láncolással végrehajtott szabály alkalmazásokat, miközben a WM rendszerint módosulhat is. A metódus akkor tér vissza, amikor már nincs több tüzelhető szabály. Persze ez a menet több alkalommal ismételhető, azaz a WM a kliens java kódból tovább módosítható és a *fireAllRules()* metódus ismét meghívható.

A Drools inkrementális szabály mintaillesztő algoritmust használ, azaz a WM-beli tények változása során (új objektum, objektum törlése, egy objektum tulajdonságainak megváltozása) erről értesíteni kell a rules engine-t, ebben segít a későbbiekben bemutatandó *FactHandle* referencia, amelynek a tudása az, hogy értesíteni tudja a Drools-t egy-egy objektumváltozásról, amit azután az újraolvas.

A Drools a szabályokat egy *DRL* fájlban (*Drools Language*) tárolja, ez az eszköz legalapvetőbb szabályleíró nyelve. Ugyanakkor létezik egy *DSL* (*Domain Specific Language*) nyelv is, ami egy-egy szakterület nyelvéhez közelebb álló módon tudja a szabályokat definiálni. Fontos lehetőség egy döntési tábla importálása, mint Drools szabályhalmaz. A DRL szabálynyelvet a 3. fejezet mutatja be részletesen.

A Drools egyébként *hybrid reasoning system*, azaz ismeri a már röviden bemutatott előre és hátraláncoló működést is. A Drools 6. verzió a teljes környezetet *KIE* (*Knowledge Is Everything*) néven emlegeti, sőt az API-ban is történtek ennek megfelelően névváltoztatások. Az eddig használt *RETE* algoritmus a tovább lett fejlesztve, amit most *PHREAK* mintaillesztő algoritmusnak néven emlegetnek.

A fejlesztői környezet javasolt kialakítása

A következőkben a fejlesztői környezet javasolt kialakítását írjuk le. Előzetesen néhány ismert komponenst kell a gépünkre telepíteni vagy annak meglétét leellenőrizni: *Java* SDK 1.5 vagy magasabb verzió, *Apache Maven* (webhelye: `http://maven.apache.org/`) és az *Eclipse* valamelyik frissebb változata (ezen leíráskor mi az *Eclipse Luna* kiadást használtuk). A *Drools* alapú fejlesztésekhez van egy Eclipse Plugin, aminek telepítése a következő lépésekből áll:

1. Az *Eclipse Graphical Editing Framework*, azaz a *GEF* hozzáadása az Eclipsehez. Az *Install New Software...* menünél ezt az előre már felvett repository-t keressük meg: `http://download.eclipse.org/releases/luna`. Itt megtaláljuk a *GEF*-et és telepítésre ki tudjuk választani.

2. A következő lépésben a *Drools Expert* csomag letöltése és alkalmas könyvtárba való kicsomagolása történhet meg. A most ismert utolsó, 6.1-es verzió innen érhető el: `http://download.jboss.org/drools/release/6.1.0.Final/drools-distribution-6.1.0.Final.zip`.

3. Térjünk vissza az Eclipse-hez és tegyük fel a Drools plugint is, de előtte a repository-t fel kell venni erre a helyre: `http://download.jboss.org/drools/release/6.1.0.Final/org.drools.updatesite/`). Egy új repository-t az *Install New Software...* menüpont→*Add...* gomb kiválasztásával kezdeményezhetjük.

4. Utolsó lépésként a most feltelepített Drools plugint konfiguráljuk rá a 2. lépésben letöltött *expert* csomagra. A *Window→Preferences* dialógus ablakban fogunk találni egy *Drools* ágat, ott megtaláljuk az *Installed Drools Runtimes* helyet, ahova több futtatási környezetet is felvehetünk, de mi most arra állítsuk, amit kicsomagoltunk. A *binaries* könyvtárat kell kiválasztani.

Ezzel kész a fejlesztői környezet kialakítása. A feltett Drools plugin *maven* alapú, azaz a függőségeket az kezeli.

A Drools kliens alkalmazások felépítése

A szabályok elkészülte után azokat valamely kliens alkalmazás szeretné majd használni, így most egy ilyen program felépítését szeretnénk röviden áttekinteni. Át fogjuk tekinteni azt, ahogy mindezt a Drools 5.x csinálta (ezt *Knowledge API* néven emlegetik), ugyanis az eléggé elterjedten használatos. Ugyanakkor a Drools 6.x kliens API-val (*KIE API*) való használatot is megnézzük. Kompatibilitási okokból a 6. verzió mindkét szabálybázis elérést és használatot támogatja. Nézzük

meg először a Drools 5.x alapú kliens program szerkezetét! Első lépésben kell egy *KnowledgeBuilder* objektum, amelyik a szabályokat, mint erőforrásokat látja. A példában a szabályokat egy *DRL* fájlban tároltuk, emiatt a *ResourceType.DRL* típust adtuk meg.

```
KnowledgeBuilder builder = KnowledgeBuilderFactory.newKnowledgeBuilder();
builder.add(ResourceFactory.newClassPathResource("org/cs/rules/szabalyok.drl"), ResourceType.DRL
    );
```

A Drools ennek legyártásához biztosít egy *KnowledgeBuilderFactory* gyártó metódust. A következő lépés az API használat felépítésekor egy *KnowledgeBase* objektum létrehozása:

```
KnowledgeBase knowledgeBase = KnowledgeBaseFactory.newKnowledgeBase();
knowledgeBase.addKnowledgePackages(builder.getKnowledgePackages());
```

Látható, hogy a tudásbázist reprezentáló *knowledgeBase* objektum az előzetesen legyártott *builder* segítségével építkezik. A következő lépésben a már többször emlegetett állapotőrző session legyártására kerül sor:

```
StatefulKnowledgeSession session = null;
session = knowledgeBase.newStatefulKnowledgeSession();
```

Ezzel a kliens program szerzett egy kapcsolatot a szabálybázishoz, amit elkezdhet használni. Például beteszünk a ténybázisba (azaz a session-re) 4 darab objektumot:

```
Person person = new Person();
FactHandle personFact = session.insert(person);
Profile profile = new Profile();
FactHandle profileFact = session.insert(profile);
Alarm alarm = new Alarm("A");
FactHandle alarm1Fact = session.insert(alarm);
alarm = new Alarm("B");
FactHandle alarm2Fact = session.insert(alarm);
```

Ennyi előkészítés után megkérjük a Rules Engine-t, hogy működtesse a szabályainkat (most azt, ami a *szabalyok.drl* fájlban vannak):

```
session.fireAllRules();
```

A végén a session-t egy *finally* ágban mindenképpen le kell zárni:

```
session.dispose();
```

Ez volt az alapszerkezete egy Drools 5.x kliensnek. Most nézzük meg, hogy az új *KIE API* mindezt, hogy csinálja rövidebben! Az egyes fontosabb osztályokat a következők váltották fel:

- *KnowledgeBase* → *KieBase*
- *KnowledgeSession* → *KieSession*
- *KnowledgeAgent* → *KieScanner*

A 2-1. Programlista egy olyan teljes kódot mutat, amit a Drools plugin generált, ebből teljesen láthatjuk az új KIE alapú kliens felépítését.

2-1. Programlista: *DroolsTest.java* (Drools 6.x módban generálva)

```
1  package com.sample;
2
3  import org.kie.api.KieServices;
4  import org.kie.api.runtime.KieContainer;
5  import org.kie.api.runtime.KieSession;
6
7  /**
8   * This is a sample class to launch a rule.
9   */
10 public class DroolsTest {
11
12     public static final void main(String[] args) {
13         try {
14             // load up the knowledge base
15             KieServices ks = KieServices.Factory.get();
16             KieContainer kContainer = ks.getKieClasspathContainer();
17             KieSession kSession = kContainer.newKieSession("ksession-rules");
18
19             // go !
20             Message message = new Message();
21             message.setMessage("Hello World");
22             message.setStatus(Message.HELLO);
23             kSession.insert(message);
24             kSession.fireAllRules();
25         } catch (Throwable t) {
26             t.printStackTrace();
27         }
28     }
29
30     public static class Message {
31
32         public static final int HELLO = 0;
33         public static final int GOODBYE = 1;
34
35         private String message;
36
37         private int status;
38
39         public String getMessage() {
40             return this.message;
41         }
42
43         public void setMessage(String message) {
44             this.message = message;
45         }
46
47         public int getStatus() {
48             return this.status;
49         }
50
51         public void setStatus(int status) {
52             this.status = status;
53         }
54     }
55 }
```

A működéshez szükséges erőforrásokat deklaratív módon a *kmodule.xml* (2-2. Programlista) fájl tartalmazza, ami a project *META-INF* könyvtára alatt helyezkedik el. A KIE API ezt az XML-t használja arra, hogy kiolvassa a használt erőforrásokat, így a szabályokat is. Esetünkben a fájlrendszerben, a project *rules* könyvtárában van egy *DRL* file, de az itt lévő összes erőforrást az XML-ben *ksession-rules* névre kereszteltük, így ennek segítségével a 17. sorban hozzájutunk

egy *kSession* objektumhoz. Ettől fogva az API használata már a megszokott módon történhet. A 30-54 sorok között egy *Message* class-t alkotunk meg, amelynek egy objektumát a 23. sorban ráteszünk a session-re, mint egy új elemét a tényeknek. A *fireAllRules()* jelentése is megegyezik az 5.x API-ban használt móddal.

2-2. Programlista: *kmodule.xml*

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
3      <kbase name="rules" packages="rules">
4          <ksession name="ksession-rules"/>
5      </kbase>
6      <kbase name="dtables" packages="dtables">
7          <ksession name="ksession-dtables"/>
8      </kbase>
9      <kbase name="process" packages="process">
10         <ksession name="ksession-process"/>
11     </kbase>
12 </kmodule>
```

Amennyiben a Drools plugint arra kérjük, hogy Drools 5.x verziójú kódot generáljon, akkor az eredményt a 2-3. Programlista mutatja. Ez pont ugyanazt a kliens logikát valósítja meg, mint a 2-1. Programlista, ugyanis mindkettő a *Sample.drl fájlban (*2-4. Programlista*)* lévő szabályokat használja, ugyanazzal a logikával.

2-3. Programlista: *DroolsTest.java* (Drools 5.x módban generálva)

```
1
2  package com.sample;
3
4  import org.drools.KnowledgeBase;
5  import org.drools.KnowledgeBaseFactory;
6  import org.drools.builder.KnowledgeBuilder;
7  import org.drools.builder.KnowledgeBuilderFactory;
8  import org.drools.builder.KnowledgeBuilderError;
9  import org.drools.builder.KnowledgeBuilderErrors;
10 import org.drools.builder.ResourceType;
11 import org.drools.io.ResourceFactory;
12 import org.drools.logger.KnowledgeRuntimeLogger;
13 import org.drools.logger.KnowledgeRuntimeLoggerFactory;
14 import org.drools.runtime.StatefulKnowledgeSession;
15
16 /**
17  * This is a sample class to launch a rule.
18  */
19 public class DroolsTest {
20
21     public static final void main(String[] args) {
22         try {
23             // load up the knowledge base
24             KnowledgeBase kbase = readKnowledgeBase();
25             StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
26             KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory.newFileLogger(ksession
                   , "test");
27             // go !
28             Message message = new Message();
29             message.setMessage("Hello World");
30             message.setStatus(Message.HELLO);
31             ksession.insert(message);
32             ksession.fireAllRules();
33             logger.close();
```

```
34          } catch (Throwable t) {
35              t.printStackTrace();
36          }
37      }
38
39      private static KnowledgeBase readKnowledgeBase() throws Exception {
40          KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
41          kbuilder.add(ResourceFactory.newClassPathResource("Sample.drl"), ResourceType.DRL);
42          KnowledgeBuilderErrors errors = kbuilder.getErrors();
43          if (errors.size() > 0) {
44              for (KnowledgeBuilderError error: errors) {
45                  System.err.println(error);
46              }
47              throw new IllegalArgumentException("Could not parse knowledge.");
48          }
49          KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
50          kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
51          return kbase;
52      }
53
54      public static class Message {
55
56          public static final int HELLO = 0;
57          public static final int GOODBYE = 1;
58
59          private String message;
60
61          private int status;
62
63          public String getMessage() {
64              return this.message;
65          }
66
67          public void setMessage(String message) {
68              this.message = message;
69          }
70
71          public int getStatus() {
72              return this.status;
73          }
74
75          public void setStatus(int status) {
76              this.status = status;
77          }
78      }
79  }
```

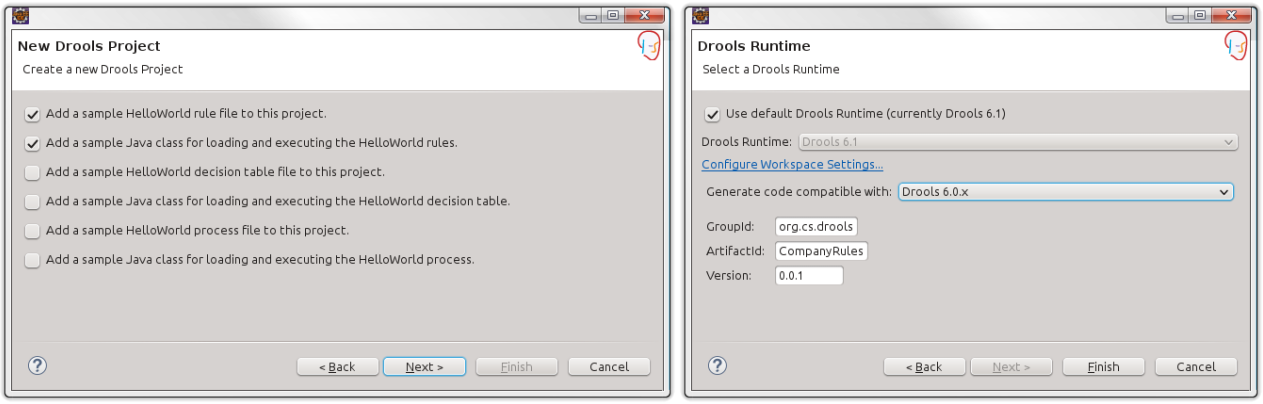
A továbbiakban csak a Drools 6.x kliens felépítést fogjuk használni, de lényegesnek éreztük, hogy bemutassuk, hogy mindez az 5.x verzióban hogy nézett ki. Ez a tudás akkor jöhet jól, amikor migrálni szeretnénk a régebbi programjainkat. A 6.x nem elsősorban azért fontos, mert egy kicsit rövidebb a kliens kiépítése, hanem azért mert többet tud és emiatt többféleképpen használható.

Az első, Hello, worlds Drools alkalmazásunk

A project létrehozása Eclipse segítségével

Most elkészítünk egy tipikus első alkalmazást, ahol a szabályokat is készítünk, ez lesz a szabálybázis. Fejlesztünk hozzá egy kliens programot is, ami bemutatja a szabályok használatát, immár egy teljes példán keresztül. Mindez nem lesz nehéz, mert lényegében az egész project-et a Drools plugin

generálja le. A Drools plugint feltelepítettük, így kapcsoljunk az Eclipse-ben Drools perspektívára, majd válasszuk ki a *File→New→Drools Project* menüpontot. Mi a project-nek a *CompanyRules* nevet adtuk és a következő 2 dialógus ablakon (2.1. ábra) csak annyit kértünk, hogy generáljon le egy példa DRL fájlt (2-3. Programlista) és egy kliens programot (2-1. Programlista). A második, jobb oldali ablakban azt kértük, hogy a Drools 6.x szerint generáljon, illetve a *maven* használathoz szükséges *GroupId* (más néven egy névtér vagy csomag), *ArtifactId* (az elkészített target) és *Version* is meg lett adva.



2.1. ábra. Az új Drools project varázsló 2 dialógus ablaka

Szeretnénk kiemelni, hogy a varázsló ablaka mutatja, hogy a Drools a kliens programjai számára ilyen szolgáltatásokat tud adni, sőt ehhez példa klienst is generál:

- Üzleti szabály (*Business Rules*) készítés (a példánkban most csak ezt használjuk, ahogy azt már említettük).
- Döntési tábla (*Decision Table*) használat (ez szabályhalmazokat tárol).
- Döntési folyamat megadás és használat (*Rule Flow*).

A generált kódot a 2-1. Programlista már mutatta, ez a 2-4. Programlistában lévő 2 darab szabályt használja majd. A szolgáltatást a kliens az *newKieSession("ksession-rules")* metódushívással kéri le (17. sor), aminek használatát a szintén automatikusan generált *kmodule.xml* deklarálja.

A szabályok készítése és működésének értelmezése

Az Eclipse plugin által generált minta *Sample.drl* fájlt nézzük meg egy kicsit alaposabban! Most csak annyit, amennyi feltétlenül szükséges a megértéshez, ugyanis a 3. fejezet teljes egészében a *Drools Rule Language* (*DRL*) nyelv bemutatásáról fog szólni. A szabályok is névtérben vannak, esetünkben ez a csomagnév most *com.sample*. A fájl a következő 2 szabályt definiálja: „*Hello World*" és „*GoodBye*". A csomagon belül minden szabálynak egyedi névvel kell rendelkeznie, amit a *rule* kulcsszó vezet be. A *when* és *then* között a szabály teljesülésének a feltételeit specifikáljuk (ez a már említett *LHS* vagy tények). A *then* és *end* között pedig azokat az akciókat adjuk meg, amiknek

a feltétel teljesülése esetén végre kell hajtódniuk (ez pedig a szintén említett *RHS*). Nézzük először a „*Hello World*” szabályt! Az *LHS* része azt rögzíti, hogy a tények valamely együttállása teljesül-e. Emlékezzünk rá, hogy a tények mindig azon Java Bean objektumokra lesznek vizsgálva, amiket egy korábban felépített *KieSession* (vagy az 5.x verzióban *StatefulKnowledgeSession*) objektumon keresztül beszúrtunk a szabály motor részére. Itt a vizsgálódás annyit jelent, hogy a ténybázison (session-ön) keres egy *Message* objektumot (vegyük észre, hogy a DRL fájlban ezt a Java class-t import utasítással tettük ismertté a Rule Engine és a szabály számára) és megvizsgálja rá a *status == Message.HELLO* feltételt. A „,” a logikai *ÉS* jele, de most a 2. feltételrész (*myMessage : message*) nem ad meg korlátozást a *Message* objektumunkra úgy, hogy annak *message* property-jére bármilyen feltételt adna, azonban itt mégis egy fontos megérteni való dolog van. A motor ugyanis megnézi az éppen vizsgált *Message* objektum *message* jellemzőjét, elvégezné rá a feltétel vizsgálatot, ha volna. A *myMessage* egy un. DRL változó, az a rendeltetése, hogy megjegyezze az éppen vizsgált értéket, azaz esetünkben a *message* adattag pillanatnyi értékét. A sor elején lévő *m* is DRL változó, de ez az éppen vizsgált objektumra vonatkozó referenciát jegyzi meg. Amennyiben a feltétel teljesül, úgy a RHS rész végrehajtódik. A megjegyzett jellemző értéke kiíródik a konzolra (9. sor), az objektum 2 adattagjának az értékét beállítja (10-11 sorok), majd kiad egy *update()* hívást (12. sor) tudatva a Rule Engine-nel, hogy változás keletkezett a tények között is, hiszen egy objektum most megváltozott. A „*GoodBye*” szabály ez alapján már könnyen érthető, ez a *status* mező egy másik értéke esetén teljesül.

2-4. Programlista: *Sample.drl*

```
1  package com.sample
2  import com.sample.DroolsTest.Message;
3
4  rule "Hello World"
5      when
6          m : Message( status == Message.HELLO, myMessage : message )
7      then
8          System.out.println( myMessage );
9          m.setMessage( "Goodbye cruel world" );
10         m.setStatus( Message.GOODBYE );
11         update( m );
12 end
13
14 rule "GoodBye"
15     when
16         Message( status == Message.GOODBYE, myMessage : message )
17     then
18         System.out.println( myMessage );
19 end
```

A szabály tesztelése a 2-1. Programlista használatával

A tesztprogramot már elmagyaráztuk, de a 23. sorban figyeljük meg újra, ahogy a ténybázisra az *insert()* metódus ráteszi a *message* objektumot, majd utána a *fireAllRules()* elindítja a tényekre épülő szabálykiértékeléseket. A futási eredmény a következő 2 üzenet megjelenése a képernyőn:

```
Hello World
Goodbye cruel world
```

Vegyük észre, hogy mindkét szabály lefutott, mert az első átírta a *status* mezőt, így az engine erre az új objektum változatra is teljesülni találta ezután az LHS-t, így ezt a szabályt is érvényre juttatta.

3. Drools szabályok készítése - A DRL nyelv

Ez a fejezet a DRL (*Drools Rule Language*) szabálynyelvet mutatja be részletesen. Elolvasása után képesek leszünk saját összetett szabályokat is megfogalmazni, illetve remélhetőleg egy kicsit elmélyítjük a háttérben lévő működési mechanizmust is.

A Drools, a matematikai logika és 2 osztály ehhez a fejezethez

Mit is jelent a matematikai logika ezen kifejezése: $x \in H : P(x)$? A válasz: vegyünk egy x elemet a H halmazból, ami P tulajdonságú. P egy valamilyen állítás, például az, hogy könnyebb, mint 100 kg. Ez x halmazelemre vagy igaz, vagy hamis. Persze írhatjuk azt is, hogy $x \in H : P(x) \wedge Q(x)$, ami egy olyan állítást jelenti, hogy az x objektumunk P és Q tulajdonságú egyszerre. Mindez egy kicsit más jelölésben, de ugyanígy megfogalmazható a Drools DRL nyelven is. A halmaz egy osztály lesz, amit most *class H* néven nevezünk a fentiek szerint. Az objektumok ezen osztály elemei. A *H()* egyszerűen azt jelenti, hogy egy objektum, ami *H* típusú, de nem írjuk le külön az x-et, hacsak nem akarunk később még erre a konkrét objektumra máshol is hivatkozni a szabályban. A tény adatbázisban lévő objektumok közül néhányan lehetnek *H* típusúak, ezek illeszkednek erre a típusra. Ez a mechanizmus az, ami nagyjából az $x \in H$ résznek felel meg. A *P, Q, ...* állításokat pedig az objektumokra megfogalmazott tulajdonságként kell felfognunk. Például egy *H*-beli objektum ezekkel a mezőkkel rendelkezhet: *m1, m2, m3, ...* Emiatt a tulajdonságot így fogalmazzuk meg: *H(m1-re egy feltétel, m2-re egy feltétel, ...)*. Mindebből azt láthattuk, hogy a matematikai logikai kifejezések az osztályok és objektumok nyelvére meglehetősen természetes módon átfogalmazhatóak. Ezt használja a DRL nyelv is.

A fenti eszmefuttatás és a korábbiak alapján is világos, hogy majd a szabályainkat objektumokkal kell etetni, így a 3-1. és 3-2. Programlisták azokat az osztályokat mutatják be nekünk (*Customer* és *Account*), amelyek objektumait fogjuk a példáinkban használni. Ez 2 JavaBean, azonban szeretnénk kiemelni, hogy az olvasó az *equals()*, *hashCode()* és *toString()* metódusok jó minőségű *override*-olásának a példáját is megtanulhatja innen. Ezekhez az Apache Common lang csomag szolgáltatásait használtuk, amelyek ezen metódusok nagyon korrekt implementálását segítik. Mindez azért szükséges, mert 2 objektum összehasonlításának lehetőségére mindenképpen számítanunk kell, amikor a feltételeket fogalmazzuk meg.

3-1. Programlista: *Customer.java*

```
1  package org.cs.drools.beans;
2
3  import org.apache.commons.lang3.builder.EqualsBuilder;
4  import org.apache.commons.lang3.builder.HashCodeBuilder;
5  import org.apache.commons.lang3.builder.ToStringBuilder;
6
7  public class Customer {
8
9      String name;
10     String country;
11     int age;
12     int category;
13
14     public Customer() {
15        super();
16     }
17
```

```
18      public Customer(String name, String country, int age, int category) {
19          super();
20          this.name = name;
21          this.country = country;
22          this.age = age;
23          this.category = category;
24      }
25
26      public String getName() {
27          return name;
28      }
29
30      public void setName(String name) {
31          this.name = name;
32      }
33
34      public String getCountry() {
35          return country;
36      }
37
38      public void setCountry(String country) {
39          this.country = country;
40      }
41
42      public int getAge() {
43          return age;
44      }
45
46      public void setAge(int age) {
47          this.age = age;
48      }
49
50      public int getCategory() {
51          return category;
52      }
53
54      public void setCategory(int category) {
55          this.category = category;
56      }
57
58      @Override
59      public boolean equals(final Object other) {
60          if (this == other)
61              return true;
62          if (!(other instanceof Customer))
63              return false;
64          Customer castOther = (Customer) other;
65          return new EqualsBuilder().append(name, castOther.name).append(country, castOther.country).append(age, ➥
                castOther.age)
66                .append(category, castOther.category).isEquals();
67      }
68
69      @Override
70      public int hashCode() {
71          return new HashCodeBuilder(-806458195, -1310850617).append(name).append(country).append(age).append(➥
                category).toHashCode();
72      }
73
74      @Override
75      public String toString() {
76          return new ToStringBuilder(this).append("name", name).append("country", country).append("age", age).append➥
                ("category",category).toString();
77      }
78  }
```

Az *Account* egyik adattagja *Customer* típusú objektum, ezt is használni fogjuk még a későbbiekben.

3-2. Programlista: *Account.java*

```
1   package org.cs.drools.beans;
2
3   import org.apache.commons.lang3.builder.EqualsBuilder;
4   import org.apache.commons.lang3.builder.HashCodeBuilder;
5   import org.apache.commons.lang3.builder.ToStringBuilder;
6
7   public class Account {
8
9       long balance;
10      boolean locked;
11      Customer owner;
12
13      public Account() {
14          super();
```

```
15      }
16
17      public Account(long balance, boolean locked, Customer owner) {
18          super();
19          this.balance = balance;
20          this.locked = locked;
21          this.owner = owner;
22      }
23
24      public long getBalance() {
25          return balance;
26      }
27
28      public void setBalance(long balance) {
29          this.balance = balance;
30      }
31
32      public boolean isLocked() {
33          return locked;
34      }
35
36      public void setLocked(boolean locked) {
37          this.locked = locked;
38      }
39
40      public Customer getOwner() {
41          return owner;
42      }
43
44      public void setOwner(Customer owner) {
45          this.owner = owner;
46      }
47
48      @Override
49      public boolean equals(final Object other) {
50          if (this == other)
51              return true;
52          if (!(other instanceof Account))
53              return false;
54          Account castOther = (Account) other;
55          return new EqualsBuilder().append(balance, castOther.balance).append(locked, castOther.locked).append(➥
                owner, castOther.owner).isEquals();
56      }
57
58      @Override
59      public int hashCode() {
60          return new HashCodeBuilder(-806458195, -1310850617).append(balance).append(locked).append(owner).➥
                toHashCode();
61      }
62
63      @Override
64      public String toString() {
65          return new ToStringBuilder(this).append("balance", balance).append("locked", locked).append("owner", owner➥
                ).toString();
66      }
67  }
```

A szabályillesztés, mint az objektumok végigpróbálása

Nagyon fontos megérteni, hogy a session-re tett objektumok (azaz a tények, *facts*) és a betöltött szabályok miképpen generálják a Rule Engine számára azt, hogy egy szabályt mikor kell végrehajtania. Mielőtt ezt részletesebben megnéznénk, szeretnénk erre egy tökéletesen hasonló példát adni az SQL *select* parancs segítségével. Vegyük ezt a példa parancsot:

```
select table1.oszlop, table2.oszlop from table1, table2
```

Ilyenkor az SQL motor veszi a $D = table1 \times table2$ Descartes szorzatot, azaz olyan $r_1 \in table1, r_2 \in table2 : (r_1,r_2)$ párok jönnek létre. Ez azt jelenti, hogy az eredménytábla sorai az egyes táblák rekordjainak az összeragasztásaként jönnek létre, ezt reprezentálja az előzőekben mutatott pár. A D az eredménytábla halmaz, annyi sora van, amennyi a két tábla sorainak a szorzata. A Drools pont ugyanezzel a Descartes szorzat logikával ellátja össze az összes olyan objektum kombinációt, amit utána megvizsgál abból a szempontból, hogy teljesül-e rá valamely

általa ismert szabály *LHS* része.

A 3-3. Programlista a ténybázisra (session-re) rátesz (18-27 sorok) 3 *Customer* és 2 *Account* típusú objektumot, majd utána kéri a szabályok használatát a Drools motortól (29. sor).

3-3. Programlista: *DroolsTest.java*

```
1  package com.sample;
2
3  import org.cs.drools.beans.Account;
4  import org.cs.drools.beans.Customer;
5  import org.kie.api.KieServices;
6  import org.kie.api.runtime.KieContainer;
7  import org.kie.api.runtime.KieSession;
8
9  public class DroolsTest {
10
11     public static final void main(String[] args) {
12        try {
13
14           KieServices ks = KieServices.Factory.get();
15           KieContainer kContainer = ks.getKieClasspathContainer();
16           KieSession kSession = kContainer.newKieSession("ksession-rules");
17
18           Customer c = new Customer("Nyiri Imre", "Magyarország", 50, 1);
19           kSession.insert( c );
20           c = new Customer("Nyiri Imrus", "Magyarország", 19, 1);
21           kSession.insert( c );
22           c = new Customer("Nyiri Sarolta", "Magyarország", 16, 1);
23           kSession.insert( c );
24           Account a = new Account(100000, false, c);
25           kSession.insert( a );
26           a = new Account(500000, false, c);
27           kSession.insert( a );
28
29           kSession.fireAllRules();
30              kSession.dispose();
31        } catch (Throwable t) {
32           t.printStackTrace();
33        }
34     }
35 }
```

A 3-4. Programlista által mutatott az egyetlen szabályt írtuk csak meg, jelenleg a Drools motor csak ezt ismeri, erre illeszt minden lehetséges objektum kombinációt. A 8-10 sorok között szavakkal a következőket adtuk meg:

1. Vegyél egy *Customer* objektumot, amire most nem adunk semmilyen feltételt (azaz mindegyiket használd). Jegyezd meg, hogy a most éppen rögzített objektum *name* mezőjének értékére a *$name1* változóval szeretnénk majd hivatkozni.

2. Vegyél egy másik *Customer* objektumot is, amire most szintén nem adunk semmilyen feltételt (azaz mindegyiket használd). Jegyezd meg, hogy a most éppen rögzített objektum *name* mezőjének értékére a *$name2* változóval szeretnénk majd hivatkozni.

3. Végül keress a ténybázison egy *Account* objektumot is, amire most nem adunk semmilyen feltételt (azaz mindegyiket használd). Jegyezd meg, hogy a most éppen rögzített objektum *balance* mezőjének értékére a *$balance* változóval szeretnénk majd hivatkozni.

Az első és második *Customer* objektumot a ténybázisról 3 féleképpen vehetjük ki, hiszen ennyien vannak. Az *Account* objektumot pedig 2 féleképpen. Esetünkben a Descartes szorzat a *when* és *then* közötti 3 sor alapján a következő: $Customer \times Customer \times Account$. A megvizsgálandó objektum hármasok száma így $3 \cdot 3 \cdot 2 = 18$. Feltételt nem adtunk meg, ezért ezen 18 kombináció mindegyikében teljesülni fog a „*Test*” szabályunk LHS része, azaz a *DroolsTest* futtatásakor arra

számítunk, hogy a 18 szabály mindegyike lefut, azok pedig a saját RHS (vagy *action*) részük végrehajtásaként kiírnak egy sort a képernyőre.

3-4. Programlista: *CompanyRules.drl*

```
package org.cs

import org.cs.drools.beans.Customer;
import org.cs.drools.beans.Account;

rule "Test"
    when
        Customer($name1:name)
        Customer($name2:name)
        Account($balance:balance)
     then
        System.out.println( "Szabály futott -> " + $name1 + "@" + $name2 + "@" + $balance);
end
```

Örömmel tapasztalhatjuk, hogy valóban a várt 18 sort kaptuk meg futási eredményként:

```
Szabály futott -> Nyiri Sarolta@Nyiri Sarolta@100000
Szabály futott -> Nyiri Sarolta@Nyiri Sarolta@500000
Szabály futott -> Nyiri Sarolta@Nyiri Imrus@100000
Szabály futott -> Nyiri Sarolta@Nyiri Imrus@500000
Szabály futott -> Nyiri Sarolta@Nyiri Imre@100000
Szabály futott -> Nyiri Sarolta@Nyiri Imre@500000
Szabály futott -> Nyiri Imrus@Nyiri Sarolta@100000
Szabály futott -> Nyiri Imrus@Nyiri Sarolta@500000
Szabály futott -> Nyiri Imrus@Nyiri Imrus@100000
Szabály futott -> Nyiri Imrus@Nyiri Imrus@500000
Szabály futott -> Nyiri Imrus@Nyiri Imre@100000
Szabály futott -> Nyiri Imrus@Nyiri Imre@500000
Szabály futott -> Nyiri Imre@Nyiri Sarolta@100000
Szabály futott -> Nyiri Imre@Nyiri Sarolta@500000
Szabály futott -> Nyiri Imre@Nyiri Imrus@100000
Szabály futott -> Nyiri Imre@Nyiri Imrus@500000
Szabály futott -> Nyiri Imre@Nyiri Imre@100000
Szabály futott -> Nyiri Imre@Nyiri Imre@500000
```

Amennyiben a szabályt leegyszerűsítjük úgy, ahogy a 3-5. Programlista mutatja, akkor csak a *Customer* objektumok játszanak szerepet.

3-5. Programlista: *CompanyRules.drl*

```
package org.cs

import org.cs.drools.beans.Customer;
import org.cs.drools.beans.Account;

rule "Test"
    when
        Customer($name1:name)
     then
        System.out.println( "Szabály futott -> " + $name1);
end
```

Ennek megfelelően a futási eredmény is ez a várt 3 sor lesz, azaz 3 alkalommal triggerelődött a *„Test”* szabály.

```
Szabály futott -> Nyiri Sarolta
Szabály futott -> Nyiri Imrus
Szabály futott -> Nyiri Imre
```

A Drools Rule Language (DRL) szabályok alapelemei

Alapvető kulcsszavak

Már láttuk, hogy a Drools szabályok csomaghierarchiába szerveződnek (*package* kulcsszó), a csomagon belül minden szabálynak egyedi névvel kell rendelkeznie. Azt is bemutattuk, hogy van

import utasítás, amivel Java osztályokat tudunk használatba venni, hiszen ne feledjük, a szabályok ezen osztályok objektumai fölött működnek. Ez leginkább 2 dolgot jelent:

1. Az LHS résznél az osztályok segítségével fogalmazzuk meg a feltételeket.
2. Az RHS résznél az objektumok metódusait meghívhatjuk, illetve magát az objektumot használhatjuk.

Szintén láttuk az eddigi példákban, hogy a szabályok definícióit a *rule* szó vezeti be, majd a szabály idézőjelekbe tett neve következik. A *when* és *then* kulcsszavak között az LHS, a *then* és *end* között pedig az RHS szabályrész megadása történik.

A DRL nyelv természetesen támogatja a megjegyzések elhelyezését. Az 1 soros comment # vagy // jelekkel bevezetve adható meg, míg a blokkok megjegyzésbe helyezése a szokásos /* ... */ párral lehetséges.

Feltételek megadása

A feltételeket úgy tudjuk megfogalmazni, hogy az objektum osztályát felhasználva az adattagokra (*property*) fogalmazunk meg tetszőleges összetettségű állításokat. A 3-6. Programlista 8. sorában azt mondjuk, hogy a ténybázis azon objektumaira fusson le a szabály, akik nem idősebbek 20 évesnél és a nevük végződése Sarolta. Ez utóbbit reguláris kifejezéssel is megtehetjük, mert a *matches* logikai művelet kulcsszava ezt támogatja. A vessző operátor a feltételek logikai ÉS műveletei.

3-6. Programlista: *CompanyRules.drl*

```
1  package org.cs
2
3  import org.cs.drools.beans.Customer;
4  import org.cs.drools.beans.Account;
5
6  rule "Test"
7      when
8          Customer(age <= 20, name matches ".*Sarolta", $name:name)
9      then
10       System.out.println( "Szabály futott -> " + $name);
11 end
```

A futási eredmény a várt 1 darab sor lett:

```
Szabály futott -> Nyiri Sarolta
```

Vegyük fel az *Account* objektumot is a *when* feltételek közé (3-7. Programlista), azaz most az LHS vizsgálat történjen a $Customer \times Account$ Descartes szorzat halmaz felett. Tekintettel arra, hogy az *Account* esetén nem fogalmaztunk meg feltételt, így az eredmény, most 2 sor lesz, ugyanis 2 ilyen objektum van az 1 darab *Customer* mellett.

3-7. Programlista: *CompanyRules.drl*

```
1  package org.cs
2
3  import org.cs.drools.beans.Customer;
4  import org.cs.drools.beans.Account;
5
6  rule "Test"
7      when
8          Customer(age <= 20, name matches ".*Sarolta", $name:name)
9          Account($balance:balance)
10     then
11       System.out.println( "Szabály futott -> " + $name);
12 end
```

Változók megadása a szabályokon belül

A 3-8. Programlista 7. sora tartalmaz egy *$c:* részt, ami egy olyan Drools változó, ami az éppen illesztett objektumra való referenciát képes eltárolni. Itt tehát nem egy adattagra (mint a *$name*), hanem az egész objektumra lehet majd hivatkozni a szabály más LHS vagy RHS-beli részeiből.

3-8. Programlista: *CompanyRules.drl*

```
1  package org.cs
2
3  import org.cs.drools.beans.Customer;
4
5  rule "Test"
6      when
7          $c:Customer(age <= 20, name matches ".*Sarolta", $name:name)
8      then
9        System.out.println( "Szabály futott -> " + $c);
10 end
```

A futási eredmény ez lett, ahol szeretnénk kiemelni azt a *toString()* formátumot, amit a *Customer* osztálynak adtunk.

```
Szabály futott -> org.cs.drools.beans.Customer@31fcdb7e[name=Nyiri Sarolta,country=Magyarország,age=16,category=1]
```

Természetesen a *$c* objektum minden metódusa meghívható, például az eredeti *$name* helyett ezt is írhatnánk:

```
System.out.println( "Szabály futott -> " + $c.getName());
```

Az objektumokra ily módon megadott referencia változóval egy nagyon fontos dolgot tehetünk meg, ami az SQL *join* (összekapcsolás) műveletének felel meg. A 3-9. Programlista 8. sorában azt mondtuk, hogy csak azok az *Account* objektumok érdekesek számunkra a *Customer* × *Account* Descartes szorzat halmazból, ahol annak *owner* jellemzője az előzetesen választott *$c Customer* objektum.

3-9. Programlista: *CompanyRules.drl*

```
1  package org.cs
2
3  import org.cs.drools.beans.Customer;
4
5  rule "Test"
6      when
7          $c:Customer(age <= 20, name matches ".*Sarolta", $name:name)
8          Account(owner==$c)
9      then
10       System.out.println( "Szabály futott -> " + $c.getName());
11 end
```

Típusok használata

A szabályok megfogalmazása során bármely Java típus használható. A *String* és számtípusokat eddig is használtuk már. A reguláris kifejezések lehetőségét is láttuk, ez a *java.util.regexp* csomaggal kompatibilis. A *Date* típus használata is gyakori az üzleti szabályok megfogalmazásakor: *Account(dateOpened > "01-Jan-2014")*. Itt szeretnénk kiemelni, hogy a használt dátum konstans formátumát a *drools.dateformat* Java környezeti változóban (*java.lang.System* property) mindig állítsuk be. Ajánlott formátumok: *yyyyy-mm-dd* vagy *yyyyy-mm-dd hh:mm:ss*. A *Boolean* típus esetén a *true* és *false* kiírása szükséges a feltétel vizsgálatnál: *Account(locked==true)*. A Java *enum* típus

használatát egy példán keresztül mutatjuk be. Ez egy felsorolás típus és legyen az *Account* class egyik adattagja (a neve: *accountType*) ilyen típusú:

```
public enum AccountType {
   FOLYO, TANULO, MEGTAKARITASI, KOZOS
}
```

Ekkor például egy ilyen feltétel fogalmazható meg: *Account(accountType==AccountType.TANULO)*.

Globális változók

Ez a kliens program és a szabályfuttató motor közötti globális kommunikációs lehetőség. A 3-10. Programlista egy bean, ami egy ilyen változó típusa lesz.

3-10. Programlista: *GlobalBean.java*

```
1  package org.cs.drools.beans;
2
3  public class GlobalBean {
4     public String name;
5     public String comment;
6     public int    intValue;
7  }
```

Legyen most ez a szabályunk, ahol a 7. sorban adtunk meg egy *global* típusú változót és az RHS mindkét sorában használtuk, sőt a 14. sornál meg is változtattuk az értékét:

3-11. Programlista: *CompanyRules.drl*

```
1  package org.cs
2
3  import org.cs.drools.beans.Customer;
4  import org.cs.drools.beans.Account;
5  import org.cs.drools.beans.GlobalBean;
6
7  global GlobalBean myGlobal;
8
9  rule "Test"
10     when
11         Customer((age==19), $name:name)
12     then
13       System.out.println( "Szabály futott -> " + $name + "@" + myGlobal.intValue);
14       myGlobal.name="Imrus";
15 end
```

A kliens programunk (3-3. Programlista) ezekkel a sorokkal bővült:

```
...
GlobalBean gb = new GlobalBean();
gb.intValue = 19;
kSession.setGlobal("myGlobal", gb);

kSession.fireAllRules();

gb = (GlobalBean)kSession.getGlobal("myGlobal");
System.out.println("Új érték: "+gb.name);
...
```

A *gb* objektumot tesszük rá a ténybázisra, mint globális változó. A szabály action része ezt megváltoztatta, ez a kliens futási eredményén is látható:

```
Szabály futott -> Nyiri Imrus@19
Új érték: Imrus
```

A globális változók arra vannak tervezve, hogy az RHS részben használjuk, segítségükkel külső Java hívásokat tehetünk a szabályunk action részében.

Függvények használata

A függvények használatával a többször alkalmazott kódblokkokat hasznosíthatjuk újra, de előnyös az is, hogy egy találó névvel a szabály is jobban olvasható. Semmit többet nem lehet a *function* írással elérni, mint amit a tiszta Java használattal lehetséges. Most nézzünk egy rövid példát egy függvényre:

```
function String hello(String name) {
    return "Hello "+name+"!";
}
```

És a használata, ahol ez a szabály mindig lefut:

```
rule "using a function"
when
    eval( true )
then
    System.out.println( hello( "Sarolta" ) );
end
```

Mindezt úgy is el tudtuk volna készíteni, hogy Java-ban megírjuk a *Utils.hello()* metódust és így importáljuk be a szabályok közé:

```
import function org.cs.Utils.hello
```

Nyelvi dialektusok

A szabályok leírásánál, az RHS és LHS részben egyaránt a Java nyelv az alapértelmezett, de lehetséges más nyelv is. A Drools még az ismert *MVEL* script nyelvet támogatja, amelynek használatát így lehet kérni:

```
package org.cs
dialect "mvel"
```

Az utolsó pontban az MVEL nyelvről adunk egy áttekintést, a használata időnként célszerű lehet.

A szabályok feltételeihez használható műveletek

Alapműveletek

Az egyes műveleteket egy-egy példával mutatjuk be. Az első az *AND*, amit már használtunk, ez volt a vessző operátor.

```
Customer(name="valaki", age=20)
```

Mindezt az ismertebb *&&* jellel is készíthettük volna, de nem szokásos. Viszont a logikai vagy, azaz az *OR* esetén már használjuk a // jelet:

```
Customer(name="valaki" || age=20)
```

A relációs jelek is használhatóak, például ha azon *Customer* objektumokra akarunk szabály akciókat végrehajtani, amelyeknél az életkor nem 19 év, azt így fogalmazhatjuk meg:

```
Customer(age!=19)
```

Emlékezzünk rá, hogy a tesztprogramunk 3 *Customer* objektumot tett a session-re, amiből az egyik pont 19 éves, emiatt ez akkor 2 objektum kiválasztását fogja eredményezni. Itt szeretnénk kiemelni a *not* kulcsszó jelentését, ami teljesen más hatást eredményez. Vajon mikor hajtódik végre ez a szabály és mit jelent:

```
rule "Test"
    when
        not Customer()
    then
        System.out.println( "Szabály futott -> ");
end
```

A *when* és *then* között egy logikai állítás van és ez a szabály mindig csak 1 alkalommal fog lefutni, ugyanis ez most nem konkrét objektumokra vonatkozik, hanem arra, hogy az összes közül találtunk-e legalább 1 illeszkedőt. A *not* kvantor tulajdonképpen azt állítja, hogy az utána következő feltétel kielégíthetetlen. Amennyiben igen, akkor ennek a logikai értéke *true* lenne, de a *not* miatt *false* lesz. Ilyenkor nincs szabály végrehajtva. A fenti példában a *Customer*-re nem adtunk semmilyen korlátozást, így 3 olyan objektum is van, ami ezt teljesíti, így a szabály nem hajtódik végre. Írhatnánk nemes egyszerűséggel ezt is:

```
rule "Test"
    when
        not Customer(false)
    then
        System.out.println( "Szabály futott -> ");
end
```

Ezt a feltételt egyetlen *Customer* sem fogja teljesíteni, így az egésznek *false* lesz a visszatérési értéke, a *not* után pedig *true*, azaz a szabály RHS része 1 alkalommal le fog futni.

Az *exists* a *not* működésének az inverze és egyben az egzisztenciális kvantor. Itt nem számít, hogy hányféleképpen elégíthető ki a feltétel, csupán az a lényeg, hogy kielégíthető. A fenti magyarázat után könnyen érthető a használata, de azért nézzünk egy példát, amikor a szabály lefut:

```
rule "Test"
    when
        exists Customer(age==19)
    then
        System.out.println( "Szabály futott -> ");
end
```

Az utolsó kvantor, amit meg kell ismernünk a *forall*. A példát a Drools tudástárból vettük. A *then* utáni rövid mondat alapján érthető is a működése:

```
rule "All english buses are red"
when
    forall( $bus : Bus( type == 'english')
                   Bus( this == $bus, color = 'red' ) )
then
    # all english buses are red
end
```

A fenti szabály szavakkal elmondva azt jelenti, hogy minden olyan busz, amelynél a *type== 'english'* igaznak kell lennie, hogy az egyben piros is. Ez a logika univerzális kvantora, emiatt ennek az egész állításnak van egy *true* vagy *false* értéke.

A mi *Customer* objektumainkra egy ilyen uneverzális kvantort alkalmazhatunk, ami a *Mind magyar* szöveget megjeleníti a képernyőn, ugyanis emlékezzünk rá, hogy ez mindhárom *Customer*-re igaz. Ezt a matematikai logika nyelvén így írtuk volna: $\forall c \in Customer : c.country = Magyarország$.

```
rule "Test"
    when
        forall ( Customer(country == "Magyarország"))
    then
        System.out.println( "Mind magyar");
end
```

Az *eval* egy olyan script futtató utasítás, ami a beállított dialektustól függően *Java* vagy *MVEL* kódot tud futtatni, de azoknak logikai értékkel kell visszatérniük. Példánkban a *Customer* osztályunkat kiegészítettük egy *isYoung()* metódussal:

```
... // Egy logikai értéket visszaadó metódussal kiegészült Customer class
public boolean isYoung()
{
        return age < 20 ? true : false;
}
...
```

A használat pedig így néz ki, a *when* részbe bekerült a logikai metódus, ami természetesen a 3 lehetséges objektumból azt a 2 szabály fogja kiváltani, ahol az $age<20$ igaz.

```
rule "Test"
    when
        $c:Customer($name:name,$age:age)
        eval($c.isYoung())
    then
        System.out.println( "Név: "+$name+", Kor: "+$age);
end
```

A szabály a tesztelés eredményeképpen valóban 2 alkalommal futott le:

```
Név: Nyiri Imrus, Kor: 19
Név: Nyiri Sarolta, Kor: 16
```

A *this* mindig az éppen nézett objektumra hivatkozik, használatára itt van egy kifejező példa, ami megakadályozza, hogy a $Customer \times Customer$ párnál az első tag megegyezzen a másodikkal.

```
$customer1 : Customer( )
$customer2 : Customer( this != $customer1 )
```

A kollekciók használata

A tartalmazás vizsgálat célja egy kollekcióba tartozás leellenőrzése, amihez a *contains* kulcsszó használatos a következő példában mutatott módon:

```
$account : Account( )
Customer( accounts contains $account )
```

Itt most az *accounts* a *Customer* objektum kollekció típusú adattagja. A vizsgálat azt ellenőrzi le, hogy az előző sorban megjegyzett *$account* objektumot ez tartalmazza-e. Ennek a tagadása így írható:

```
$account : Account( )
Customer( accounts not contains $account )
```

A *memberOf* operátor az objektumra vonatkozó vizsgálat, azt nézi meg, hogy az eleme-e egy kollekciónak:

```
$customer : Customer( $accounts : account )
Account( this memberOf $customer.accounts )
```

A tagadása:

```
$customer : Customer( $accounts : account )
Account( this not memberOf $customer.accounts )
```

A *from* kulcsszó azért hasznos, mert olyan *Account* objektumra tudunk vizsgálni, amit nem mi tettünk rá a session-re, hanem egy kollekcióból származik.

```
$customer : Customer( )
Account( ) from $customer.accounts
```

A szabályok tevékenység (RHS vagy action) részének kialakítása

Amikor egy szabály teljes feltételrendszere teljesül, akkor az aktiválódik (*activated*), de még nem kerül végrehajtásra. Mindennek az a vége, hogy az összes szabály kiértékelése után azok közül néhány aktivált állapotba kerül. A következő lépésben a Rule Engine veszi egymás után az aktív állapotban lévő szabályokat és végrehajtatja azok RHS részét. Amikor egy szabály lefutott, akkor mondjuk azt, hogy a szabály *fired* (azaz elsült) állapotba került. A következő aktív szabály futtatásakor néha konfliktuskezelés is szükséges (*conflict resolution strategy*). Miután egy szabály végrehajtódott, a szabálymotor frissíti az aktív szabályok listáját, egyeseket aktívvá tesz, másokat passziválhat. Mindez azért szükséges, mert minden szabály maga is változtathatja a tényeket és emiatt azt, hogy egyes szabályokat kell-e még futtatni vagy sem. A Rule Engine ezt a folyamatot mindaddig végzi, amíg el nem fogynak az aktív, azaz a végrehajtásra váró szabályok. Persze a következő *fireAllRules()* hívásnál mindez elölről kezdődhet. Az ismertetett működés során a szabálymotor a session-t is kezeli. A *modify* kulcsszó arra szolgál, hogy a szabály jelezze egy tényobjektum megváltozását. Erre nézzünk egy példát:

```
rule "interest calculation"
no-loop
when
    $account : Account( )
then
    modify($account) {
end
      setBalance((long)($account.getBalance() * 1.15) )
    };
```

A *modify* blokk több, vesszővel elválasztott kifejezést is tartalmazhat természetesen. Az *insert* szolgál arra, hogy a session-re egy új tényobjektumot helyezzünk fel:

```
insert(new Account())
```

A *retrack* egy objektumot leszed a tények közül, amire például szolgáljon ez a szabály:

```
rule "Lower the alarm when all the fires have gone"
when
   not Fire()
   $alarm : Alarm()
then
   retract( $alarm );
   System.out.println( "Lower the alarm" );
end
```

A szabályok megadható attribútumok

Nem beszéltünk még a szabály opcióiról, amelyeket a *when* kulcsszó előtt lehet megadni. Az egyik legfontosabb opció a *salience*, amelyet egy egész szám követ. A megadott szám lesz a szabály

prioritása (alapértelmezésben 0), és több aktivált szabály esetén a legmagasabb prioritású fog tüzelni.

Egy másik fontos opció az alapértelmezésben hamis értékű *no-loop*. Volt már arról szó, hogy a szabálypéldány végrehajtásakor a saját aktovációja automatikusan eltűnik, hogy azok a szabályok se kerüljenek végtelen ciklusba, amelyek nem érvénytelenítik saját előfeltételeiket (lásd. fentebb a *modify* kódját). Ha azonban az érintett szabály módosítja a feltételrészében szereplő valamelyik objektumot, akkor a Drools újra észleli az aktováció érvényességét és visszateszi azt a végrehajtandók közé. Ez kerülhető el *no-loop true* megadásával. Tartsuk észben, hogy a több szabályon keresztül lezajló kölcsönös rekurzió ellen ez sem hatásos, az igazi megoldás az, ha minden szabály valamilyen módon érvényteleníti saját aktovációját, azaz az LHS részének hamisra állításáról gondoskodik ilyenkor.

A *dialect* az utolsó fontos attribútum, ami a szabályhoz használt nyelvet adja meg. Ez lehet *java* vagy *mvel*, de tekintettel arra, hogy az első a default, így csak az *mvel*-t szoktuk kiírni. Nézzük meg a használatot is:

```
rule "rule attributes"
salience 100
dialect "mvel"
no-loop
when
  //conditions
then
  //consequence
end
```

Az MVEL script nyelv áttekintése

A Drools környezetben jelentős szerepe van az *MVEL* (*MVFLEX Expression Language*) script nyelvnek, ezért nézzük át egy kissé részletesebben is (webhelye: `http://mvel.codehaus.org/`). Kedvcsinálónak lássuk a *quicksort* rendezés implementációját:

```
import java.util.*;

// the main quicksort algorithm
def quicksort(list) {
    if (list.size() <= 1) {
        list;
    }
    else {
        pivot = list[0];
        concat(quicksort(($ in list if $ < pivot)), pivot, quicksort(($ in list if $ > pivot)));
    }
}

// define method to concatenate lists.
def concat(list1, pivot, list2) {
    concatList = new ArrayList(list1);
    concatList.add(pivot);
    concatList.addAll(list2);
    concatList;
}

// create a list to sort
list = [5,2,4,1,18,10,15,1,0];

// sort it!
quicksort(list);
```

Az *mvel* parancssorban is használható, így kell elindítani:

```
java -jar mvel2-2.2.0.Final.jar
```

Egy mvel scriptet tartalmazó fájlt így lehet lefuttatni:

```
java -jar mvel2-2.2.0.Final.jar <mvel file>
```

Alapvető *mvel* szintaxis

Legyen egy *Customer* objektum neve *cust*. Ekkor így lehet hivatkozni a jellemzőire:

```
cust.name
cust.age
```

Egy logikai kifejezés a jellemzők használatával a következő:

```
cust.name=="Nyiri Sarolta"
```

Az mvel kifejezéseket „,”-vel választhatjuk el egymástól. Amennyiben van egy visszatérési érték, úgy az mindig az utolsó kifejezés értéke, de írhatunk *return* utasítást is. A nyelv rendelkezik a szokásos relációs operátorokkal: *==*, *!=*, *<*, *<=*, ... Összehasonlíthatunk a *null* értékkel (a *nil* konstans is használható). Két eltérő típusú érték is összehasonlítható: *"123" == 123* (ez *true* lesz). A logikai ÉS (*&&*) és VAGY (*||*) használata a Java nyelvvel egyezik meg, de ezen felül van még 2 érdekes operátor. Az *or* művelet több taggal rendelkezik, amin végigmegy és az első nem üres értéket adja vissza:

```
alma or korte or szilva
```

A *~=* a reguláris kifejezés operátor. Ellenőrzi, hogy a bal oldalon lévő érték illeszkedik-e a jobb oldalon lévő mintára:

```
str ~= "[0-9]+"
```

Az számtani műveletek szintén a Java-beliekhez hasonlítanak, a *%* operátor is létezik (*5%3==2*). A *+* jel képes 2 sztringet összefűzni. A típusnevek úgy használhatóak, ahogy a Jávában vannak: *java.util.HashMap*. Egy beágyazott osztályra pedig így hivatkozhatunk: *org.proctor.Person$BodyPart*.

Az mvel nyelv az ismert literál típusok megadását így támogatja, egy-egy példán keresztül:

```
"This is a string literal"
'This is also string literal'

//
// String escape sequences
//
\\ - Double escape allows rendering of single backslash in string.
\n - Newline
\r - Return
\u#### - Unicode character (Example: \uAE00)
\### - Octal character (Example: \73)

125    // decimal
0xAFF0 // hex
10.503 // a double
94.92d // a double
14.5f  // a float
104.39484B // BigDecimal
8.4I       // BigInteger
```

Adatszerkezetek inline megadása

Ez egy *Map* objektum *inline* megadása:

```
["Foo" : "Bar", "Bar" : "Foo"]
['Bob' : new Person('Bob'), 'Michael' : new Person('Michael')]
```

A 2. sor ezzel a kóddal is létrehozható, ezzel ekvivalens.

```
Map map = new HashMap();
map.put("Bob", new Person("Bob"));
map.put("Michael", new Person("Michael"));
```

A *List* inline formátumára nézzünk ismét egy példát:

```
["Jim", "Bob", "Smith"]
```

Az *Array* adatszerkezet pedig így néz ki:

```
{"Jim", "Bob", "Smith"}
```

A *coercion* (kényszerítés) azt jelenti, hogy az mvel automatikusan megfelelő típusra alakít valamit, de a kódban ezt nem kell feltüntetni. Egy ilyen tömb típus nélküli: *{1,2,3,4}*. Ugyanakkor néha szükséges lehet egy helyen ezt int[] típusúnak látni, ezt automatikusan elvégződik a háttérben.

Property navigáció

A Bean property elérése történhet hagyományosan (példa: *user.getManager().getName()*), de az mvel a minősített hivatkozást javasolja: *user.manager.name*. A Null-Safe (null biztos) navigáció azt támogatja, hogy ne kelljen ilyen hosszú sorokat írni:

```
if (user.manager != null) { return user.manager.name; } else { return null; }
```

A fenti kód mvel nyelven így fogalmazható meg röviden: *user.?manager.name*. Egy *user* lista eleme tömbcímzéssel is elérhető: *user[5]* (ezzel a Java kóddal ekvivalens: *user.get(5)*). Egy *customer Map* elérése: *customer["key"]* (ezzel a Java kóddal ekvivalens: *customer.get(„key")*). Amikor a *Map* kulcsa sztring, támogatott az ilyen elérési formátum: *customer.key*. Egy *String* típus karaktereit is el lehet érni tömb indexeléssel.

Vezérlőszerkezetek

Az egyes vezérlőszerkezeteket szintén csak egy-egy példán keresztül mutatjuk be, felhasználva az mvel dokumentáció példáit. A hagyományos *if-then-else* természetesen használható:

```
if (var > 0) {
    System.out.println("Greater than zero!");
}
else if (var == -1) {
    System.out.println("Minus one!");
}
else {
    System.out.println("Something else!");
}
```

Létezik a már ismert 3 operandusú kifejezés, szintén a megszokott szemantikával:

```
var > 0 ? "Yes" : "No";
// Egymásba is ágyazhatjuk:
var > 0 ? "Yes" : (var == -1 ? "Minus One!" : "No")
```

A *foreach* ciklus, ahol a *name* az indexváltozó, ami végigmegy a *people* kollekción:

```
count = 0;
foreach (name : people) {
    count++;
    System.out.println("Person #" + count + ":" + name);
}

System.out.println("Total people: " + count);
```

Érdekes és esetenként hatékony használati lehetőség a *foreach* ciklust sztringre alkalmazni:

```
str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

foreach (el : str) {
   System.out.print("[" + el + "]");
}
```

Az mvel a *for (item : collection) { ... }* szintaktikát is a fenti *foreach* ciklussal megegyezően használja. Az ismert *for* ciklus pedig a megszokott módon használható:

```
for (int i =0; i < 100; i++) {
   System.out.println(i);
}
```

Hátultesztelős ciklus, ahol a bennmaradási feltételre vizsgálunk:

```
do {
   x = something();
}
while (x != null);
```

Hátultesztelős ciklus, ahol a kilépési feltételre vizsgálunk (mint a Pascal nyelven):

```
do {
   x = something();
}
until (x == null);
```

Elöltesztelős ciklus, csináld amíg igaz a ciklusfeltétel:

```
while (isTrue()) {
   doSomething();
}
```

Elöltesztelős ciklus, csináld amíg hamis a ciklusfeltétel:

```
until (isFalse()) {
   doSomething();
}
```

Listákra alkalmazható projekciók és filterek

Képzeljük el, hogy van egy *customers* változónk, aminek ez a típusa: *List<Customers>*, továbbá a *Customer* class tartalmaz pontosan 1 darab *address* mezőt, aminek van egy *zipcode* adattagja is. Ekkor a lenti példa a *customers* kollekcióból minden vevőre listába szedi azok *zipcode*-ját és az eredményt a *zipcodes* listaváltozóba helyezi.

```
zipcodes = (address.zipcode in customers);
```

A fentiekben leírt lehetőség miatt hívjuk mindezt projekciónak. Amennyiben egy *Customer* objektum több *address*-t is tartalmazna, úgy a fenti kód például így változhatna meg:

```
zipcodes = (postCode in (addresses in $customers));
```

A *$* jelenti azt a szimbólumot (*placeholder*), amellyel egy filterezett lista elemeire hivatkozhatunk. Nézzünk néhány példát, amelyek teljes listákon végeznek műveleteket:

```
(toUpperCase() in ["foo", "bar"]); // returns ["FOO", "BAR"]
(($ < 10) in [2,4,8,16,32]);       // returns [true, true, true, false, false]
($ in [2,4,8,16,32] if $ < 10);    // returns [2,4,8]
```

Funkció definíciók és Lambda kifejezések

A *def* kulcsszó egy új függvény létrehozását teszi lehetővé, amire egy egyszerű példa a következő:

```
def hello() { System.out.println("Hello!"); }

// Hívása:
hello();
```

Ez a példa már paramétereket is fogad és van visszatérési értéke, ami $a+b$, ugyanis ez az utolsó kifejezés sor:

```
def addTwo(a, b) {
   a + b;
}
```

Most pedig nézzünk egy lambda kifejezést és annak használatát:

```
threshold = def (x) { x >= 10 ? x : 0 };
result = cost + threshold(lowerBound);
```

A *threshold* változó egy végrehajtható függvényt tárol el, ez maga a lambda kifejezés. A lényeg ott van, hogy egy függvényt egy változó tud eltárolni, így ez menet közben más és más értékre állítható, így esetenként eltérő algoritmus fut le ugyanazon a kódhelyen.

Egy rövid és teljes mvel script példa

Az mvel nyelv ismertetését az ismert számkitalálós programmal zárjuk, tanulmányozzuk ezt befejezésül:

```
import java.io.*;
//
// Seed the random number
//
$num = (int) Math.random() * 100;
$guesses = 0;
$in = -1;
//
// Setup the STDIN line reader.
//
$linereader = new BufferedReader(new InputStreamReader(System.in));
System.out.print("I'm Thinking of a Number Between 1 and 100... Can you guess what it is? ");
//
// Main program loop
//
while ($in != $num) {
if ($in != -1) {
System.out.print("Nope. The number is: " + ($num < $in ? "Lower" : "Higher") + ". What's your next guess? ");
}
if (($in = $linereader.readLine().trim()) == empty) $in = -2;
$guesses++;
}
System.out.println("You got it! It took you " + $guesses + " tries");
```

Az mvel nyelv integrációja a Java környezettel

Ebben az alpontban az mvel Java integrációját tekintjük át. Az előző pontokban megtanultuk a nyelv alapjait, most átnézzük azt, hogy a Java nyelvből miként használhatjuk azt. Már most megjegyezzük, hogy az integrációnak 2 működési módja van: interpretált és fordított. Az előző állapotmentes, az utóbbi pedig gyorsabb működésű. Az együttműködést az *org.mvel2.MVEL* Java osztály valósítja meg. A 3-12. Programlista egy Java kódot mutat, ahol egy mvel nyelven megfogalmazott logikai kifejezést (4. sor) röptében futtatunk le az *MVEL.eval()* metódus (10. sorban) segítségével. A Java és az mvel közötti változó mappinget egy *Map* adatszerkezet biztosítja (6.

sor), az mvel számára egy *myVar* változót adunk át, *Integer(100)* értékkel (7. sor). Azt tudjuk, hogy a kifejezésünk logikai, ezért használhattuk a *result* változót (ez a beépített változó a kifejezés visszatérési értéke) az *if* utasításban.

3-12. Programlista: Interpretált on the fly kifejezés végrehajtása

```
1  ...
2  @Test
3  public void test1() {
4          String expression = "myVar > 100;";
5
6          Map vars = new HashMap();
7          vars.put("myVar", new Integer(100));
8
9          // We know this expression should return a boolean.
10         Boolean result = (Boolean) MVEL.eval(expression, vars);
11
12         if (result.booleanValue()) {
13            System.out.println("Nagyobb");
14         } else { System.out.println("Kisebb vagy egyenlő"); }
15 }
```

A 3-13. Programlista az előző kifejezés kiértékelés lefordított változatát mutatja, de a működés eredménye ugyanaz.

3-13. Programlista: Fordított (compiled) on the fly kifejezés végrehajtása

```
1  ...
2  @Test
3  public void test2() {
4
5          String expression = "foobar > 99";
6
7          // Compile the expression.
8          Serializable compiled = MVEL.compileExpression(expression);
9
10         Map vars = new HashMap();
11         vars.put("foobar", new Integer(100));
12
13         // Now we execute it.
14         Boolean result = (Boolean) MVEL.executeExpression(compiled, vars);
15
16         if (result.booleanValue()) {
17            System.out.println("It works!");
18         }
19 }
```

A 3-14. Programlista a *c1* objektum name adattagjának lekérdezését mutatja meg, a kifejezés kiértékelése után a result értéke *Nyiri Imre* lesz, így a képernyőn is az jelenik meg.

3-14. Programlista: Egy objektum adattagjának elérése

```
1  ...
2  @Test
3  public void test3() {
4
5          Customer c1 = new Customer();
6
7          c1.setAge(20);
8          c1.setCategory(1);
9          c1.setCountry("Magyar");
10         c1.setName("Nyiri Imre");
11
12         String result = (String) MVEL.eval("name", c1);
13         System.out.println(result);
14 }
```

A 3-15. Programlista bemutatja, hogy egy függvényt (a neve *foo*) miként lehet létrehozni és futtatni. A képernyőn *6.75* fog megjelenni.

3-15. Programlista: Függvény létrehozás és lefuttatás

```
...
@Test
public void testFunctions5() {
       String exp = "def foo(a,b) { a + b }; foo(1.5,5.25)";
       System.out.println(MVEL.eval(exp, new HashMap()));
}
```

A 3-16. Programlista egy példát ad arra, hogy külső fájlból miképpen tudunk gyorsan betölteni egy mvel scriptet (*loadFromFile()*).

3-16. Programlista: A quicksort használata

```
...
import static org.mvel2.util.ParseTools.loadFromFile;
...
public void testQuickSortScript() throws IOException {
       Object[] sorted = (Object[]) test(new String(loadFromFile(new File("samples/scripts/quicksort.mvel"))));
       int last = -1;
       for (Object o : sorted) {
          if (last == -1) {
             last = (Integer) o;
          } else {
             assertTrue(((Integer) o) > last);
             last = (Integer) o;
          }
       }
}
```

Végül a 3-17. Programlista azt szemlélteti, ahogy X és Y változókat átvesszük, az mvel script Z-nek ad egy értéket és végül Z változót visszakérjük. A képernyőn 300 fog megjelenni.

3-17. Programlista: Változók változtatása

```
...
@Test
public void test() {
       String expression = "Z=X+Y;";

       Map vars = new HashMap();
       vars.put("X", new Integer(100));
       vars.put("Y", new Integer(200));
       vars.put("Z", null);
       MVEL.eval(expression, vars);
       System.out.println(vars.get("Z"));
}
```

Template mechanizmus. A Java, mvel és a beépített template motor

Hatékony és érdekes lehetősége a Java és mvel együttműködésének a beépített template lehetőség kombinálása a script nyelv adta lehetőségekkel. A 3-18. Programlista a legegyszerűbb eset, amikor átadunk egy értéket Java-ból és azt helyettesítjük.

3-18. Programlista: Template használati lehetőség

```
...
@Test
public void templateTest() {
       String templateStr = "Hello, @{nev} kolléga!";

       Map vars = new HashMap();
       vars.put("nev", "Nyiri Imre");
       vars.put("out", "");

       String output = (String) TemplateRuntime.eval(templateStr, vars);
       System.out.println( output );
}
```

4. Az adatok (objektumok) validálása

Folytatjuk a Drools bemutatását egy adatellenőrző keretrendszer kidolgozásával. Közben néhány új dolgot is megtanulunk, illetve egy nagyobb példát láthatunk arra, hogy ezt a környezetet miképpen tudjuk bevetni az eddigi mindennapi feladatainkba, emelve annak színvonalát.

Ebben a fejezetben bemutatunk egy olyan Drools-ra épülő keretmegoldást, ami képes egy szakterületi adatmodell konkrét példányának a validálására. Ez lehet, hogy csak egy-egy objektum ellenőrzését jelenti, de lehetséges a szabályainkat a teljes adategyüttesre is lefuttatni.

Az adatellenőrző keretrendszer tervezett felépítése

Elsőként tekintsük át a keretrendszer által használt 3 interface-t! Természetesen ezeket komplexebbre is lehetne tervezni, de már ebben a funkcionalitásukban is elég jól használhatóak. A *Notification* (4-1. Programlista) feladata az, hogy a validálás (adatellenőrzés) során keletkezett információs objektum felületét definiálja. A tervezés során azt a döntést hoztuk, hogy egy ellenőrzés során felmerült minden problémát ebbe a 2 kategóriába sorolunk be: *ERROR* és *WARNING*. Mindezt az informatikában elterjedt szokásoknak megfelelően lehet értelmezni, azaz a figyelmeztetés még nem jelenti azt, hogy az ellenőrzés megakadályozná a további feldogozást. Ezt a 2 lehetőséget a *MessageType* típus reprezentálja, ami egy *enum*. A *getMessageType()* visszaadja, hogy ez az információs rekord melyik a 2 lehetőség közül. A *getMessageKey()* metódus egy Stringet ad vissza, ami az üzenet kulcsa lehet egy Java *properties* fájlban. Ezzel a keretrendszerünk nyelvfüggő működése is könnyen megoldható. Végül a *getContextOrdered()* metódus az objektumok egy listáját szolgáltatja, ami az információs üzenethez kísérő objektumok társítását teszi lehetővé.

4-1. Programlista: *Notification* interface

```
1  package org.cs.drools.validation;
2
3  import java.util.List;
4
5  public interface Notification {
6
7      public enum MessageType {
8          ERROR, WARNING
9      }
10
11     public MessageType getMessageType();
12     public String getMessageKey();
13     List<Object> getContextOrdered();
14 }
```

A *NotificationReport* interface (4-2. Programlista) adja azt a szolgáltatási felületet, ahol egy teljes ellenőrzési menet végeredménye lekérdezhető. A *getNotifications()* az összes felderített ERROR és WARNING információt visszaadja, míg a *getNotificationsByType()* csak az egyik fajtát, amelyiket megadtuk. A *contains()* azt vizsgálja meg, hogy az ellenőrzés eredményeképpen keletkezett-e ilyen kulcsú üzenet objektum. Az *addNotification()* pedig a riport összeépítésekor használható metódus, azaz új probléma üzenet esetén ezzel regisztráljuk azt a jelentés objektumban. Amint az jól látható, az első 3 metódus lekérdező, azaz ezt használhatja majd az adatellenőrzést kérő kliens. A 4. metódus pedig az ellenőrzés során lesz használatos.

4-2. Programlista: *NotificationReport* interface

```
1  package org.cs.drools.validation;
2
3  import java.util.Set;
4
5  public interface NotificationReport {
6
7      Set<Notification> getNotifications();
8      Set<Notification> getNotificationsByType(Notification.MessageType type);
9      boolean contains(String messageKey);
10     boolean addNotification(Notification notification);
11 }
```

A *NotificationObjectsFactory* interface (4-3. Programlista) létrehozása egy programozási jógyakorlat elengedhetetlen része. Egy ilyen interfészt implementáló objektum képes lesz legyártani olyan objektumokat, amik az előző 2 interface valamelyikét valósítják meg.

4-3. Programlista: *NotificationObjectsFactory* interface

```
1  package org.cs.drools.validation;
2
3  public interface NotificationObjectsFactory {
4
5      Notification createNotification(Notification.MessageType type, String messageKey, Object... context);
6
7      NotificationReport createNotificationReport();
8  }
```

Az interface-ek implementálása

Következzen a bemutatott 3 interface implementálása!

A *Notification* interface alapértelmezett implementációja

A *DefaultNotification* osztály (4-4. Programlista) egy alapértelmezett implementációja a *Notification* felületnek. Tekintettel arra, hogy az interface 3 információt kezel, emiatt az implementáció 3 adattagot tartalmaz (12-14 sorok). A konstruktor lényegében nem is tesz mást, mint ezeket eltárolja, persze előtte egy minimális vizsgálatot csinál arra nézve, hogy a kapott paraméterei használhatóak-e. A 26-29 sorok között lévő *getMessageType()* egyszerűen visszaadja a *type* privát adattag értékét, ezzel majd bárki tudni fogja, hogy ez az objektum hiba vagy figyelmeztetés tételt tartalmaz-e. A másik 2 metódus is ugyanilyen egyszerű. Az osztály felírja az *equals()*, *hashCode()* és *toString()* metódusokat, mert látni fogjuk, hogy 2 *Notification* objektum összehasonlítására szükség lesz, ezért azt rendesen, logikailag helyesen kell megtenni. Ilyenkor nem elég az örökölt *Object*-beli megvalósítás. Itt is követtük azt a design pattern-t, amit Joshua Bloch *Effective Java* című könyvében lehet olvasni és az Apache Commons lang segíti az implementációját.

4-4. Programlista: *DefaultNotification* osztály

```
1  package org.cs.drools.validation;
2
3  import java.io.Serializable;
4  import java.util.List;
5
6  import org.apache.commons.lang3.builder.EqualsBuilder;
7  import org.apache.commons.lang3.builder.HashCodeBuilder;
8  import org.apache.commons.lang3.builder.ToStringBuilder;
9
10 public class DefaultNotification implements Notification, Serializable {
```

```
11
12      private Notification.MessageType type;
13      private String messageKey;
14      private List<Object> context;
15
16      public DefaultNotification(Notification.MessageType type, String messageKey, List<Object> context) {
17          if (type == null || messageKey == null) {
18              throw new IllegalArgumentException("Type and messageKey cannot be null");
19          }
20
21          this.type = type;
22          this.messageKey = messageKey;
23          this.context = context;
24      }
25
26      @Override
27      public Notification.MessageType getMessageType() {
28          return type;
29      }
30
31      @Override
32      public String getMessageKey() {
33          return messageKey;
34      }
35
36      @Override
37      public List<Object> getContextOrdered() {
38          return context;
39      }
40
41      @Override
42      public boolean equals(final Object other) {
43          if (this == other)
44              return true;
45          if (!(other instanceof DefaultNotification))
46              return false;
47          DefaultNotification castOther = (DefaultNotification) other;
48          return new EqualsBuilder().append(type, castOther.type).append(messageKey, castOther.messageKey).append(
                context, castOther.context).isEquals();
49      }
50
51      @Override
52      public int hashCode() {
53          return new HashCodeBuilder(98587969, 810426655).append(type).append(messageKey).append(context).toHashCode
                ();
54      }
55
56      @Override
57      public String toString() {
58          return new ToStringBuilder(this).append("type", type).append("messageKey", messageKey).append("context",
                context).toString();
59      }
60
61  }
```

A *NotificationReport* interface alapértelmezett implementációja

Az implementáló osztály (4-5. Programlista) *messagesMap* adattagja tárolja a problémás tételeket. Ennek a *Map* objektumnak a kulcsa *Notification.MessageType* típusú, azaz vagy ERROR vagy WARNING lehet, azaz a Map maximum 2 elemű, de ezek maguk is ilyen halmazok: *Set<Notification>*, azaz *Notification* elemekből álló *Set*. Azt tudni kell, hogy egy halmazba ugyanaz az elem csak egyszer kerül bele, emiatt volt fontos például a *Notification equals()* precíz megvalósítása. A *getNotifications()* metódus mindegyik problémás tételt visszaadja, ezért a 23. sorban azokat egy közös halmazba tesszük. Itt nem veszítünk elemet a halmaz tulajdonság miatt, ugyanis a maximum 2 különböző kiinduló halmazban különbözőek a *Notification* objektumok, de az egyesítésnél a *type*-juk egymástól is különböző. A 28-37 sorok közötti *getNotificationsByType()* implementációja triviálisan egyszerű, hiszen eleve szétszeparáltuk őket. A *contains()* egy egyszerű keresési feladat a korábban bemutatott *getNotifications()* halmazon. Az *addNotification()* úgy adja hozzá a *messagesMap* adattaghoz a most megkapott *Notification* példányt, hogy az a típusának megfelelő halmazba kerüljön.

4-5. Programlista: *DefaultNotificationReport* osztály

```
1  package org.cs.drools.validation;
2
3  import java.io.Serializable;
4  import java.util.Collection;
5  import java.util.Collections;
6  import java.util.HashMap;
7  import java.util.HashSet;
8  import java.util.Map;
9  import java.util.Set;
10
11 import org.cs.drools.validation.Notification.MessageType;
12
13 public class DefaultNotificationReport implements NotificationReport, Serializable {
14
15    private static final long serialVersionUID = -7834838929377314070L;
16    protected Map<Notification.MessageType, Set<Notification>> messagesMap = new HashMap<Notification.MessageType
          , Set<Notification>>();
17
18    @Override
19    public Set<Notification> getNotifications() {
20       Set<Notification> messagesAll = new HashSet<Notification>();
21
22       for (Collection<Notification> messages : messagesMap.values()) {
23          messagesAll.addAll(messages);
24       }
25       return messagesAll;
26    }
27
28    @Override
29    public Set<Notification> getNotificationsByType(MessageType type) {
30       if (type == null)
31          return Collections.emptySet();
32       Set<Notification> messages = messagesMap.get(type);
33       if (messages == null)
34          return Collections.emptySet();
35       else
36          return messages;
37    }
38
39    @Override
40    public boolean contains(String messageKey) {
41       for (Notification message : getNotifications()) {
42          if (messageKey.equals(message.getMessageKey())) {
43             return true;
44          }
45       }
46       return false;
47    }
48
49    @Override
50    public boolean addNotification(Notification notification) {
51       if (notification == null)
52          return false;
53       Set<Notification> notifications = messagesMap.get(notification.getMessageType());
54       if (notifications == null) {
55          notifications = new HashSet<Notification>();
56          messagesMap.put(notification.getMessageType(), notifications);
57       }
58       return notifications.add(notification);
59    }
60 }
```

A *NotificationObjectsFactory* interface alapértelmezett implementációja

A *DefaultNotificationObjectsFactory* (4-6. Programlista) megvalósítása nem szorul magyarázatra, de szeretnénk kiemelni az elméleti fontosságát annak, hogy egy ilyen gyártó osztályt mindig érdemes elkészíteni, amikor több interface is van. A későbbiekben sokkal rugalmasabban tudjuk majd a programunkat változtatni, illetve az a környezetétől függően más-más implementáló objektumot tud szolgáltatni, aminek az igénye gyakoribb, mint gondolnánk egy ilyen keretrendszer esetében. Másfelől nagy előny, hogy az adott interface-eket megvalósító objektumok példányait létrehozó gyártó metódusok rendelkeznek néhány előnnyel, ahogy azt szintén megtanulhattuk az *Effective Java* című könyvből:

1. A gyártómetódusnak van neve, ellentétben a konstruktorral, ami örökli az osztály nevét. Ez sok esetben előnyös, mert utalhat arra, hogy milyen módon történik az objektum létrehozása.

2. A gyártómetódus nem feltétlenül hoz létre új objektumot, lehet, hogy csak valamelyik meglévőt osztja ki újra. Ezt a konstruktor nem tudja így megvalósítani, ő mindig egy új objektumot akar létrehozni.

3. A gyártómetódus képes olyan objektumot is visszaadni, ami egy subclass-beli objektum, de mint ilyen, szintén implementálja az interface-t. A konstruktor csak egyetlen fajta osztályba tartozó objektum legyártására képes.

4-6. Programlista: *DefaultNotificationObjectsFactory* osztály

```
1  package org.cs.drools.validation;
2
3  import java.io.Serializable;
4  import java.util.Arrays;
5
6  import org.cs.drools.validation.Notification.MessageType;
7
8  public class DefaultNotificationObjectsFactory implements NotificationObjectsFactory, Serializable {
9
10     @Override
11     public Notification createNotification(MessageType type, String messageKey, Object... context) {
12        return new DefaultNotification(type, messageKey, Arrays.asList(context));
13     }
14
15     @Override
16     public NotificationReport createNotificationReport() {
17        return new DefaultNotificationReport();
18     }
19 }
```

Szabályok készítése az adatok validálására

A *NotificationHelper* osztály

Az eddigiekben a keretkörnyezetet készítettük el, most elkészítjük a szabályokat, eközben rámutatunk néhány olyan, új dologra, amiket érdemes megjegyezni. A *NotificationHelper* osztály csupán 2 statikus metódust tartalmaz: *error()* és *warning()*. A használatuk mindkét esetben a DRL fájlból fog történni, azaz itt most megtanuljuk az egyik lehetséges módját annak, hogy miképpen lehet olyan Java kódot írni, amit a szabály RHS részéből meghívhatunk. A 2 metódus implementációja hasonló, a különbség csak az közöttük, hogy az egyiket hiba, a másikat figyelmeztetés esetén hívjuk meg a szabály akciójából. Tekintsük át az *error()* működését és megvalósításának koncepcióit!

4-7. Programlista: *NotificationHelper* osztály

```
1  package org.cs.drools.validation;
2
3  import org.kie.api.runtime.KieRuntime;
4  import org.kie.api.runtime.rule.RuleContext;
5
6  public class NotificationHelper {
7
8     public static void error(RuleContext kcontext, Object... context) {
9
10       KieRuntime runtime = kcontext.getKieRuntime();
11       NotificationReport report = (NotificationReport) runtime.getGlobal("notificationReport");
12       NotificationObjectsFactory factory = (NotificationObjectsFactory) runtime.getGlobal("➥
             notificationObjectsFactory");
13
```

```
14          Notification.MessageType t = Notification.MessageType.ERROR;
15          String k = kcontext.getRule().getName();
16          Notification notification = factory.createNotification(t, k, context);
17          report.addNotification(notification);
18      }
19
20      public static void warning(RuleContext kcontext, Object... context) {
21
22          KieRuntime runtime = kcontext.getKieRuntime();
23          NotificationReport report = (NotificationReport) runtime.getGlobal("notificationReport");
24          NotificationObjectsFactory factory = (NotificationObjectsFactory) runtime.getGlobal("➥
                notificationObjectsFactory");
25
26          Notification.MessageType t = Notification.MessageType.WARNING;
27          String k = kcontext.getRule().getName();
28          Notification notification = factory.createNotification(t, k, context);
29          report.addNotification(notification);
30      }
31  }
```

Már a metódus paramétereinél van 2 megmagyarázandó dolog. A *RuleContext* osztály képviseli az éppen futó szabály környezetét, azaz ennek segítségével férhetünk hozzá további információkhoz. Itt fontos megjegyezni, hogy *kcontext* néven a DRL automatikusan létrehoz egy ilyen típusú változót, ezt fogjuk majd használni, amikor hívjuk a metódust. Az *Object... context* paraméter egy objektumok listája, azonban ezt nem kötelező megadni, azaz, amikor nincs egyetlen átadandó objektum sem, akkor nem szükséges ezt a 2. paraméter még megadni sem, azaz például nem kell bajlódni ilyenkor egy 0 elemmel rendelkező lista átadásával sem. A metódus megvalósításának a 10. sora mutatja, ahogy a *KieRuntime* objektumot lekérdezzük a *kcontext*-től, így a 11. és 12. sorban mutatott módon visszakérdezhetjük a *report* és *factory* objektumokat, amiket – amint majd látjuk – a rendszert használó kliens ad át a DRL részére, mint *global* típusú szabály objektumok. Ezek használatát a korábbiakban már megtanultuk. A 15. sor is tartogat érdekességet, mert megtanulhatjuk azt, hogy a *kcontext.getRule()* hívással miképpen juthatunk hozzá a végrehajtás alatt lévő szabályt reprezentáló objektumhoz, aminek most csak a nevére van szükségünk. Ez a név lesz az, amit *messageKey* értékként fogunk használni. Itt kihasználtuk azt, hogy a szabály neve egy csomagon belül mindig egyedi kell legyen, így ezt a karaktersorozatot felhasználhatjuk majd egy properties fájl kulcsaként is. Végül a 17. sorban az *addNotification()* hívással hozzáadjuk a most kezelt tételt a riport objektumhoz. A *warning()* megvalósítása teljesen hasonló, egy 3. paraméterrel ezt a 2 metódust könnyen kiválthatnánk 1 metódussal is, de a későbbi rugalmasabb fejleszthetőség érdekében megtartottuk a jelenlegi megoldást.

A *Validation.drl* szabályfájl

Elérkeztünk oda, hogy elkészíthetünk néhány példa szabályt, hiszen ez az egész keretmegoldásunk központi eleme. A 4-8. Programlista a példa kedvéért most csak 2 szabályt (*customerNameRequired* és *youngerThanNeeded*) tartalmaz és csak a *Customer* objektumokat vizsgálja. A 7-8 sorokban láthatjuk a már ismert 2 *global* típusú objektum deklarációját, ami azt jelenti, hogy a szabályt használó kliens képes lesz egy-egy *NotificationReport* és *NotificationObjectsFactory* objektumot átpasszolni a szabályfájlnak. Ezen objektumok neve azért fontos, mert az input objektumátadásnál is ezeket fogjuk használni. A 10-11 sorok közötti 2 darab *import* utasításnál is tanulhatunk valami újat, ugyanis láthatjuk azt, ahogy a *NotificationHelper* osztály 2 statikus metódusát beemeljük a DRL fájlba, utána ezeket a Java metódusokat pedig a 18. és 27. sorokban használjuk is. Megjegyezzük, hogy itt a háttérben a Java statikus import lehetősége van használva. A *customerNameRequired* szabály azt ellenőrzi le, hogy a *name* mező *null* értékű-e, mely esetben egy

WARNING rekord generálódik. Ez utóbbit már a beimportált *warning()* függvény állítja össze és adja hozzá a vizsgálat riportjához, ahogy azt láttuk a 4-7. Programlistánál. A *warning() kcontext* paramétere onnan jön, hogy egy DRL fájlból mindig el tudjuk érni ezt a változót, ami egy *RuleContext* osztálybeli objektum. A 2. (opcionális) paraméter az az objektum referencia, amivel most valami gond volt. A *youngerThanNeeded* szabály megírása sem nehezebb, itt azt vizsgáljuk, hogy elég idős-e a vevő.

4-8. Programlista: *Validation.drl* rule fájl

```
1  package org.cs.drools.validation
2
3  import org.kie.api.runtime.rule.RuleContext;
4  import org.cs.drools.beans.Customer;
5  import org.cs.drools.beans.Account;
6
7  global NotificationReport notificationReport;
8  global NotificationObjectsFactory notificationObjectsFactory;
9
10 import function org.cs.drools.validation.NotificationHelper.error;
11 import function org.cs.drools.validation.NotificationHelper.warning;
12
13
14 rule "customerNameRequired"
15 when
16     $c:Customer(name == null)
17 then
18     warning(kcontext, $c);
19     //System.out.println("Név: "+$c.getName());
20 end
21
22
23 rule "youngerThanNeeded"
24 when
25     $c:Customer($age:age < 20)
26 then
27     warning(kcontext, $c);
28     //System.out.println($c.getName() + $age);
29 end
```

Az adatérvényességet vizsgáló keretrendszer használata

A 4-9. Programlista a szolgáltatást implementáló logikát mutatja be, egyelőre csak egy *JUnit* teszt esetként. A megvalósításhoz a Drools 6. verzióban bevezetett korszerű *KIE* API-t használjuk, így annak további megismeréséhez is jó például szolgál ennek a kódnak a megértése.

A stateful megvalósítás lépései

A 25-27 sorok egy általánosan használt lépéssorozat szerint működnek, aminek a vége az, hogy megkapjuk a *kSession* objektumot.

4-9. Programlista: *ValidationServiceTest - test()* method

```
1
2  package org.cs.drools;
3
4  import java.util.ArrayList;
5
6  import org.cs.drools.beans.Account;
7  import org.cs.drools.beans.Customer;
8  import org.cs.drools.validation.DefaultNotificationObjectsFactory;
9  import org.cs.drools.validation.Notification;
10 import org.cs.drools.validation.NotificationObjectsFactory;
11 import org.cs.drools.validation.NotificationReport;
12 import org.junit.Test;
13 import org.kie.api.KieBase;
14 import org.kie.api.KieServices;
```

```
15 import org.kie.api.runtime.KieContainer;
16 import org.kie.api.runtime.KieSession;
17 import org.kie.api.runtime.StatelessKieSession;
18
19 public class ValidationServiceTest {
20
21    @Test
22    public void test() {
23
24       try {
25          KieServices ks = KieServices.Factory.get();
26          KieContainer kContainer = ks.getKieClasspathContainer();
27          KieSession kSession = kContainer.newKieSession("ksession-rules");
28
29          // set globals
30          NotificationObjectsFactory factory = new DefaultNotificationObjectsFactory();
31          NotificationReport report = factory.createNotificationReport();
32          kSession.setGlobal("notificationReport", report);
33          kSession.setGlobal("notificationObjectsFactory", factory);
34
35          // set facts
36          Customer customer = new Customer("Nyiri Imre", "Magyarország", 50, 1);
37          ArrayList<Object> facts = new ArrayList<Object>();
38          facts.add( customer );
39          facts.add( new Customer("Nyiri Imrus", "Magyarország", 19, 1) );
40          facts.add( new Customer(null, "Magyarország", 16, 1) );
41          facts.add( new Account(100000, false, customer) );
42          facts.add( new Account(500000, false, null) );
43
44          for (Object o : facts) { kSession.insert(o); }
45
46          kSession.fireAllRules();
47
48          for (Notification n : report.getNotifications())
49          {
50             System.out.println( n.getMessageKey() + " (" + n.getMessageType() + ")" );
51             System.out.println( n.getContextOrdered().get(0) );
52          }
53       } catch (Throwable t) {
54          t.printStackTrace();
55       }
56       kSession.dispose();
57    }
58 }
```

A *newKieSession()* hívásnál a *ksession-rules* értéket adtuk meg, ezt a 2-2. Programlistában is mutatott *kmodule.xml* fájl deklarálja, ami a *META-INF* könyvtárban található. Az új *KIE* API-ban egy-egy ilyen session-t ebben az XML-ben deklarálhatunk, nem kell programozottan létrehozni minden részletét, ahogy azt a Drools 5. API-jában tettük. A következő XML részlet azt mutatja, hogy deklaráltunk egy stateless session-t is, *ksession-rules-sl* néven. Ez utóbbit még a későbbiekben használni fogjuk, de egyelőre most nézzük meg a stateful használati módot.

```
...
<kbase name="rules" packages="rules">
                <ksession name="ksession-rules" type="stateful" />
                <ksession name="ksession-rules-sl" type="stateless" />
</kbase>
...
```

A 30-33 sorokban létrehozzuk a 2 globálisnak szánt objektumot és a session segítségével át is adjuk azt a DRL részére. Ez a lépés teszi lehetővé, hogy amikor az *error()* vagy *warning()* metódust használjuk, akkor azok implementációja használhatta ezt a 2 globálist. A 36-42 sorok között legyártjuk az ellenőrzendő objektumokat (a tény objektumokat), majd egy *for* ciklussal (44. sor) az egészet betesszük a szabály session-be (más néven a *working memory*-ba).

A következő lépés a *fireAllRules()* hívása, ami elkezdi a szabályok alkalmazását. A metódus lefutása után a *report* változónk – lévén az globális és az *error()* vagy *warning()* metódusok írhatták – tartalmazza a vizsgálati eredményt, amit a 48-52 sorok között meg is jelenítünk.

A futási eredmény így néz ki:

```
youngerThanNeeded (WARNING)
org.cs.drools.beans.Customer@3039ff4c[name=<null>,country=Magyarország,age=16,category=1]
```

```
customerNameRequired (WARNING)
org.cs.drools.beans.Customer@3039ff4c[name=<null>,country=Magyarország,age=16,category=1]
youngerThanNeeded (WARNING)
org.cs.drools.beans.Customer@247d03c0[name=Nyiri Imrus,country=Magyarország,age=19,category=1]
```

A stateless megvalósítás lépései

A Drools sateless session egyszerűen végigfutttatja a szabályokat, de nem alkalmaz semmilyen előzetes állapotra vonatkozó működést és az RHS akciók által elvégzett változtatások miatt nem aktivál újra egy szabályt. Egy validációt végző keretrendszer esetén egy ilyen kliens (4-10. Programlista) szolgáltatása hagyományosan jobb, mert nincs szükség ilyenkor állapot megőrzésre. Most a már említett *ksession-rules-sl* deklarált session-t fogjuk használni.

4-10. Programlista: *ValidationServiceTest - test2()* method

```
1
2  package org.cs.drools;
3
4  import java.util.ArrayList;
5  import java.util.List;
6
7  import org.cs.drools.beans.Account;
8  import org.cs.drools.beans.Customer;
9  import org.cs.drools.validation.DefaultNotificationObjectsFactory;
10 import org.cs.drools.validation.Notification;
11 import org.cs.drools.validation.NotificationObjectsFactory;
12 import org.cs.drools.validation.NotificationReport;
13 import org.junit.Test;
14 import org.kie.api.KieServices;
15 import org.kie.api.command.Command;
16
17 import org.kie.api.runtime.KieContainer;
18 import org.kie.api.runtime.KieSession;
19 import org.kie.api.runtime.StatelessKieSession;
20 import org.kie.internal.command.CommandFactory;
21
22
23 public class ValidationServiceTest {
24
25    @Test
26    public void test2() {
27
28       try {
29
30          KieServices ks = KieServices.Factory.get();
31          KieContainer kContainer = ks.getKieClasspathContainer();
32
33          StatelessKieSession kSession = kContainer.newStatelessKieSession("ksession-rules-sl");
34
35          NotificationObjectsFactory factory = new DefaultNotificationObjectsFactory();
36          NotificationReport report = factory.createNotificationReport();
37
38
39          System.out.println(report.getNotifications().size());
40
41          // set facts
42          Customer customer = new Customer("Nyiri Imre", "Magyarország", 50, 1);
43          ArrayList<Object> facts = new ArrayList<Object>();
44          facts.add(customer);
45          facts.add(new Customer("Nyiri Imrus", "Magyarország", 19, 1));
46          facts.add(new Customer(null, "Magyarország", 16, 1));
47          facts.add(new Account(100000, false, customer));
48          facts.add(new Account(500000, false, null));
49
50          //
51          List<Command> cmds = new ArrayList<Command>();
52
53          cmds.add(CommandFactory.newSetGlobal("notificationReport", report));
54          cmds.add(CommandFactory.newSetGlobal("notificationObjectsFactory", factory));
55
56          cmds.add(CommandFactory.newInsertElements(facts));
57
58          //for (Object o : facts) { kSession.insert(o); }
59
60          kSession.execute( CommandFactory.newBatchExecution( cmds ));
61
62          //kSession.fireAllRules();
```

```
63
64          for (Notification n : report.getNotifications()) {
65              System.out.println(n.getMessageKey() + " (" + n.getMessageType() + ")");
66
67              System.out.println(n.getContextOrdered().get(0));
68
69          }
70
71
72
73      } catch (Throwable t) {
74          t.printStackTrace();
75      }
76  }
77  ...
78  }
```

A 30-31 sorokban nincs semmi változás, de a 33. sornál fontos észrevenni, hogy a *kSession* objektum most *StatelessKieSession* típusú és a *kContainer* objektum *newStatelessKieSession("ksession-rules-sl")* hívásával hoztuk létre. A 42-48 sorok között itt is létrehozzuk a tény objektumokat, azonban azokat az előzőektől eltérő módon tesszük a session-re. A *cmds* egy *List<Command>*, ami lehetővé teszi, hogy a *Command design pattern* szerint hajtsuk végre azokat a lépéseket, amik a working memory előkészítésére szolgálnak. A 53-54 sorok mutatják, ahogy a 2 global objektum a session-re tevő parancsát készíti el (*CommandFactory.newSetGlobal()*). A 43. sorban létrehozott *facts* listát a 44-48 sorokban feltöltöttük a már ismerős tény objektumainkkal. Mindezt azért tettük, hogy az 56. sorban azt a parancsot is megalkothassuk, ami majd ezen objektumokat is a session-re teszi (*CommandFactory.newInsertElements(facts)*). A végeredmény az, hogy a *cmds* változó tartalmaz minden parancsot a működéshez, ami a szabályok előkészítését és futtatását jelenti a 60. sorban látható módon. A *execute()* eltér a már megismert *fireAllRules()* metódustól, ugyanis stateless esetben ezt kell használni.

A kliens programunk utolsó sorai megegyeznek a 4-9. Programlista utolsó soraival, a képernyős kimenet is ugyanaz.

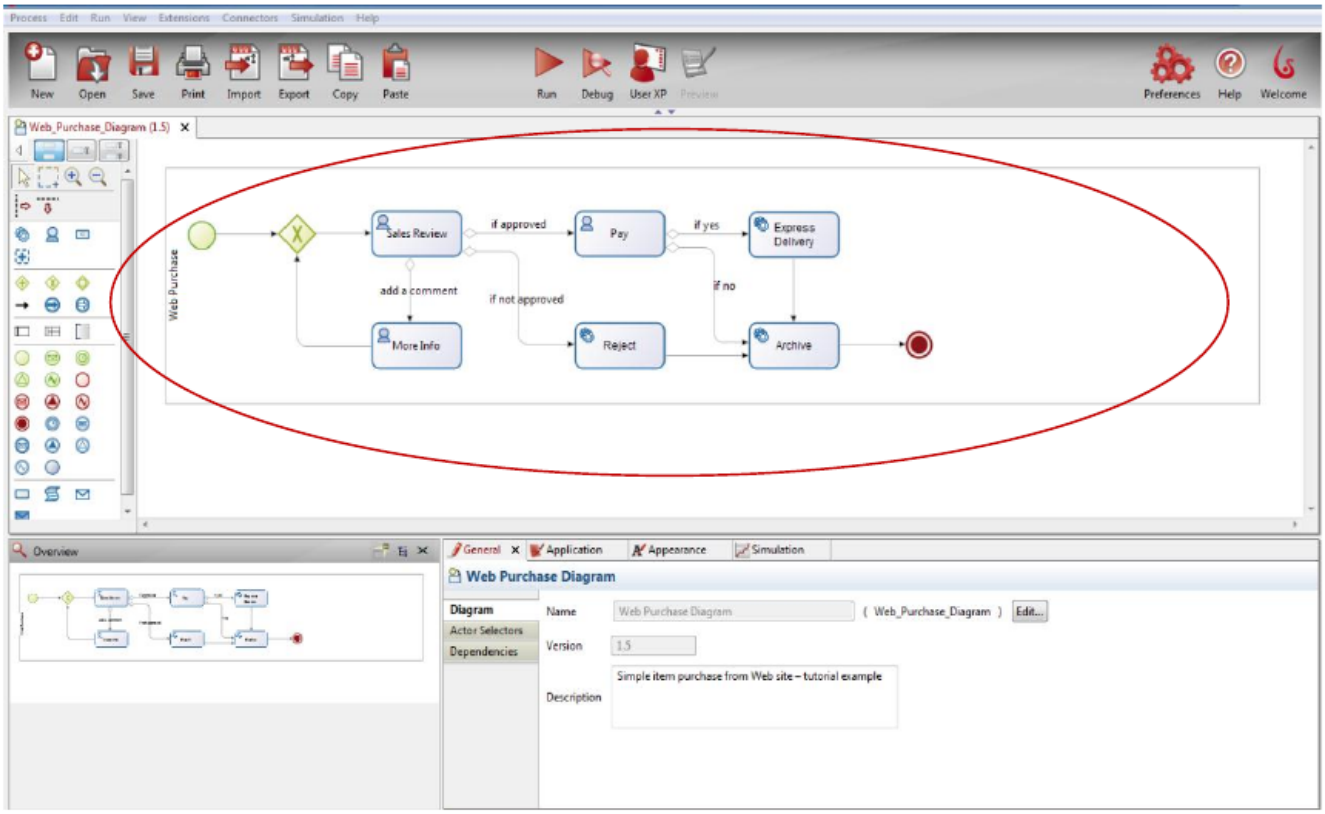
Az adatérvényességet vizsgáló szolgáltatás elkészítésének vázlata

A bemutatott kódok a Drools használatával alapjai lehetnek egy univerzális validációt végző keretrendszernek. A DRL szabályokat kívülről lehet konfigurálni, így az egész működés rugalmasan alakítható a mindennapi változásokhoz. A szabályok sokkal összetettebbek is lehetnek, több objektumot is vizsgálhatnak és egy egész szakterületi domain modell pillanatnyi értékegyüttesét képesek konzisztencia vizsgálat alá vetni. A szabályok megfogalmazása deklaratív és letisztult logikai állításokon alapulnak, így sokkal áttekinthetőbb lehet, mint a sok *if* utasítás a programjainkban. A szükséges teendők (a szabály RHS része) is kívülről konfigurálhatóak, ami adott esetben nagyfokú szabadságot ad a megvalósításban. A keretrendszer előnye, hogy a feltérképezett összes problémát egy adatszerkezetben képes tárolni, azt egyszerre is feldolgozhatjuk. A hagyományos esetben ritkán teszünk ilyen hiba/figyelmeztetés kollektort az ellenőrző rutinokhoz, így azok kezelése sem szokott annyira egységes lenni, mint lehetne. Itt olyanokra is gondolunk, hogy a logger fájlban a bejegyzések szépek legyenek, az összes hibáról csak 1 e-mail értesítés menjen ki.

A validációs szolgáltatást érdemes valamilyen szerviz felületen is elérhetővé tenni, így azt sok alkalmazás is tudja használni. Kezdeti lépésként persze egy célklienst is lehet használni, de ez esetben csak akkor lesz újrahasználható az egész keretrendszer, ha ezt egy hordozható adapter osztály segítségével valósítjuk meg.

5. Az adatok transzformálása a Drools használatával

Ebben az összefoglalóban elkezdünk megismerkedni a Bonita nevű BPM motorral, ugyanis az eddig tanult elméleti tudásunkat csak akkor tudjuk fejleszteni, ha valamilyen konkrét eszközzel ki is próbáljuk. A Bonita egy nagyon fejlett és kimagasló képeségekkel rendelkező szoftver (webhely: `http://www.bonitasoft.com/`). A szerző először 2004. évben találkozott ezzel az eszközzel, ami akkor már stabil és produktív verzióval rendelkezett. Akkorriban az ObjectWeb (webhely: `http://www.ow2.org/`) community keretében fejlődött ez az alkalmazás, amit mindenkinek jó szívvel ajánlok bármilyen nagyvállalati produktív megoldáshoz is.



5.1. ábra. Bonita Studio

Mi a Bonita?

A Bonita egy olyan szoftver, ami a BPM megoldások (workflow alkalmazások) számítógépes automatizálásához szükséges minden szerver oldali és fejlesztő eszközt megad. Három nagy részből áll:

A ??. ábra mutatja a Confirmation template and messages beállítását, ami lehet Pool vagy Task szintű is. Ez mindig valamely művelet megerősítését, visszanyugtázását teszi lehetővé.

6. Szakterület közeli szabályok készítése (DSL)

A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

7. A komplex eseményfeldolgozás

A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

8. Drools Flow

A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

9. KIE

A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

10. JBoss BRMS 6.0

A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

11. KIE

A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

12. KIE

A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

13. KIE

A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

14. KIE

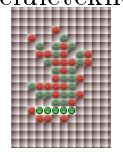
A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

adasda	sadad

Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle

taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez. A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez. A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez. A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez. A Bonita User Experience az egységes munkakörnyezet és task kosár korszerű megvalósítása. Többnyelvű használatot támogat, a magyar nyelv is elérhető. A végfelhasználó jól szervezett esetben csak ezt a környezetet használja, ugyanis itt láthatja a különféle taszkjait, aminek kezelő és megjelenítő felületei automatikusan beépülnek ebbe a keretkörnyezetbe. A UserXP-vel most ismerkedők azt vehetik észre, hogy kinézete és használata hasonlít a népszerű webmail felületekhez.

		aaaa ssssss fsdfds fdfsd fdsfs dsfsf
qqq		
		aaaa ssssss fsdfds fdfsd fdsfs dsfsf
qq		
qqq	qqq	qq