**Debreceni Egyetem**
**Informatikai Kar**

# Bevezetés a Drools üzleti szabály motor használatába

**Szakdolgozat**

**Készítette:**
Mezei Tamás
programozó matematikus

**Témavezető:**
Dr. Kuki Attila
egyetemi adjunktus

**Debrecen**
2008

# Tartalomjegyzék

# Bevezetés

Az üzleti igényeket hagyományos módon logikai elágazások segítségével valamilyen programnyelven (a nyelvi elemeket felhasználva) tudjuk ábrázolni. Ennek nagy hátránya, hogy ha változik az üzleti igény, akkor a programunkat is módosítani kell, ez pedig fejlesztést igényel. Ennek kapcsán merül fel az az elképzelés, hogy az üzleti igényeket megvalósító logikát külön kellene választani az alkalmazás kódjától, ezzel kiemelve egy másik reprezentációba. A legtöbb esetben az üzleti igények szabályok formájában kerülnek meghatározásra. Így keletkezett a felvetés: a szabályokat ne csak kiemeljük, hanem olyan formában reprezentáljuk melyet nemcsak a technikai emberek, hanem az alkalmazásokat nap, mint nap használók is megértsék; azaz a szabályokat ne valamilyen programnyelv segítségével ábrázoljuk.

Ennek a megoldásnak az a legnagyobb előnye, hogy azok az emberek, akik definiálják az üzleti igényeket, közvetlenül le tudják ezeket írni egy olyan formába, melyet az alkalmazás is megért. Vagyis nemcsak egy felhasználóbarát megoldást készítünk, hanem az alkalmazásunk működését úgy vagyunk képesek megváltoztatni, hogy közben kiválthatjuk az informatikai szakemberek beavatkozását.

Ezért is választottam diplomamunkám céljául az üzleti szabály motorok működését. Az első fejezet betekintést nyújt a szakértő rendszerekben használatos tudásábrázolási módszerekbe, következtetési eljárásokba és ismertet néhány tudásbeszerzési módszert. A második fejezetben megismerkedünk a Drools, Java nyelven íródott üzleti szabály motorral és annak lehetőségeivel. Bemutatásra kerül, hogy a Drools milyen eszközökkel valósítja meg a Java Rule Engine API (JSR94) támogatását.

Választásom azért esett a fent említett Drools szabály motorra, mert üzleti környezetben manapság leggyakrabban használt Java nyelven íródott.

# 1. Ismeretalapú és szakértő rendszerek

## *1.1 Rövid történeti áttekintés*

**1. Az 1960-as évek közepéig**

Ebben az időszakban általános problémamegoldó rendszereket (General-purpose Problem Solver, GPS) próbáltak meg fejleszteni. Ezen rendszerek egy nagy hátrányuk volt, hogy igen gyenge teljesítményt nyújtottak, mivel rengeteg dologra fel kellett készíteni őket.

**2. Az 1960-as évek közepe**

Megjelennek az olyan rendszerek, melyek nem általános probléma megoldására lettek kifejlesztve, hanem egy szűk szakterületen felmerülő problémákra megoldására szakosodtak (DENDRAL).

**3. Az 1970-es évek közepe**

Megjelenik számos szakértő rendszer (MYCIN, META-DENDRAL). Felismerik, hogy a tudás központi szerepét. Rájöttek arra, hogy egy program feladatmegoldó képessége elsősorban a benne foglalt ismeretekből származik, a formalizmus és a következtető rendszer csak másodlagos. Elkezdenek kifejleszteni számos ismeretreprezentációs módszert.

**4. Korai 1980-as évek**

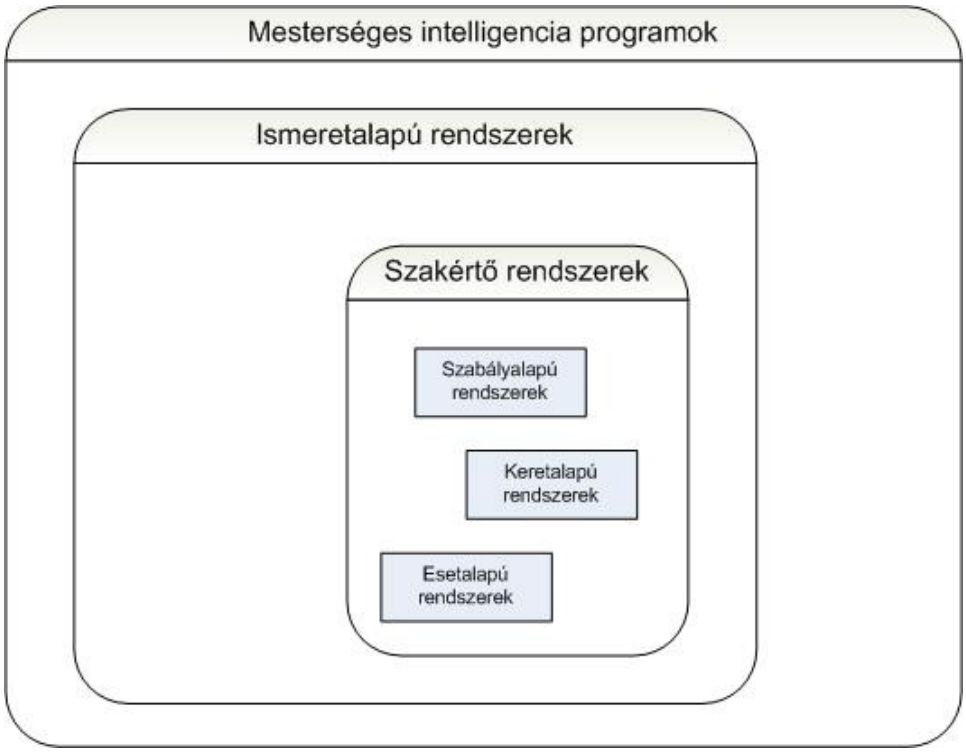
Megjelennek kereskedelmi forgalomban az első szakértő rendszerek (XCON, XSEL, CATS-1). Megjelennek programozási eszközök és keretrendszerek (EMYCIN, EXPERT, META-DENDRAL, EURISKO).

**5. Az 1980-as évek közepe**

Megjelenik a NASA által kifejlesztett CLIPS nevű C-ben írt szakértő rendszer. Ez nagy áttörés volt, mert a CLIPS hordozható a platformok között, gyors és nagyon olcsó.

**6. Az 1990-es évek**

Megjelennek a neurális hálózatokat használó programok (Artificial Neural Systems, ANS).




***1.ábra: Szakértő rendszerek elhelyezése***

Egy ismeretalapú rendszer a rendelkezésre álló információkra/ismeretekre alapozva, valamilyen keresési stratégia szerint javasol a feltett kérdésre egy választ. A rendszer képes arra, hogy megmondja, hogy az adott következtetésre hogyan jutott. Az ismeretalapú rendszerekre általánosan jellemző, hogy szimbolikus adatokkal dolgoznak és nem numerikusakkal.

Ismeretalapú rendszernek nevezünk egy olyan számítógépes alkalmazást/rendszert, amelyben az ismeretek a működés során használt algoritmusoktól jól elkülöníthetőek. A következtetést megvalósító algoritmus legfőbb tulajdonsága, hogy a működése során nem veszi figyelembe, hogy milyen adatokkal kell dolgoznia. Ez a fajta megközelítés több célt is szolgál. Sok esetben egy ismeretalapú rendszer hatalmas mennyiségű információval dolgozik, és ha nem választanánk ketté az ismereteket és a következtetést, akkor egy apró módosítás is

lehetetlen feladat lenne, mind az algoritmusban, mind az ismeretekben. De másrészről ez könnyebbséget is nyújt a fejlesztés során, mivel két jól elkülöníthető részre lehet bontani.

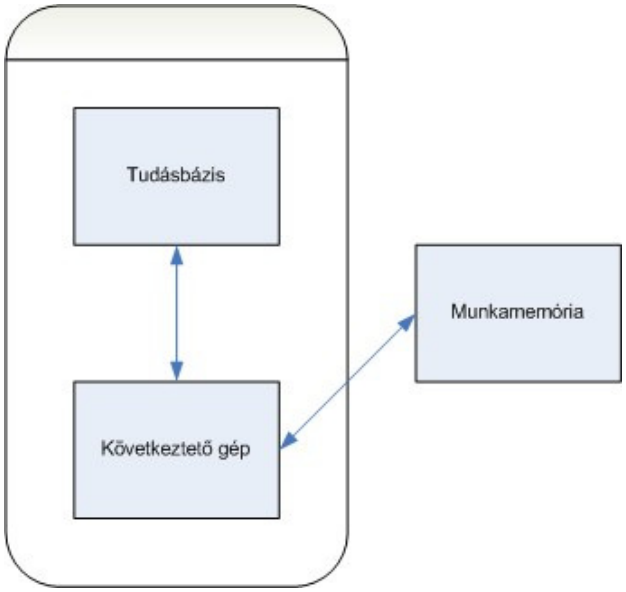
Az ismeretalapú és szakértő rendszereket szoros szálak fűzik össze, lehet őket egymás szinonimájaként is használni, de általánosságban egy szakértő rendszer, az egy speciális ismeretalapú rendszer. Vagyis az olyan ismeretalapú rendszereket, melyek egy szűk problémakör kezelésében, szakértői ismeretek segítségével az emberrel összemérhető teljesítményt nyújtanak nevezzük szakértő rendszereknek.

Ha csak technológiai megoldásként tekintünk ezen rendszerekre, akkor gyakorlatilag a kettő egy és ugyanaz, mert ugyanazokat az algoritmusokat és ugyanazokat az ismeretreprezentációs módszereket alkalmazzák. Ha a felhasználási területeikről, a bennük tárolt ismeretek jellegéről, illetve a hozzájuk szorosan kapcsolódó ismeret-beszerezési és ismeretreprezentációs módszerekről beszélünk, akkor hasznos lehet külön választani őket.

Az ismeretalapú rendszereket két nagy csoportra oszthatjuk, aszerint, hogy az alkalmazási területén mennyire támaszkodnak, illetve milyen mértékben támaszkodhatnak az általuk szolgáltatott eredményekre.

Egyes rendszereknél a javasolt megoldás minden további nélkül elfogadhatóak. Azon rendszerek amelyek ilyenek, nevezzük **döntéshozó rendszer**eknek. Az ilyen típusú rendszerek lehetővé teszik a döntést a nem tapasztalt szakembereknek, olyan területeken is, melyek meghaladják szakképzettségüket. Ezen rendszerek lehetőséget biztosíthatnak arra, hogy szükség esetén a döntést felül tudja bírálni a felhasználó.

A másik típusú ismeretalapú rendszer a **döntéstámogató rendszer**. Ezen rendszerek feladata, hogy a szakértők munkáját segítsék, a lehetséges megoldásokat, mint alternatívákat felvázolva.




***2. ábra: Ismeretalapú rendszerek általános felépítése***

## *1.2. Szakértő rendszerek (Expert Systems)*

A szakértő rendszer egy olyan ismeretalapú számítógépes program, ami az ember problémamegoldó képességét modellezi. Ez a problémamegoldás sajnos korlátozott. Határt szab a problémakör mérete, és az, hogy csak egy jól körülhatárolt körben képes problémamegoldásra. A problémakör egy szűk szakterület, a megoldás pedig valamilyen tanács, értékelés vagy szakvélemény.

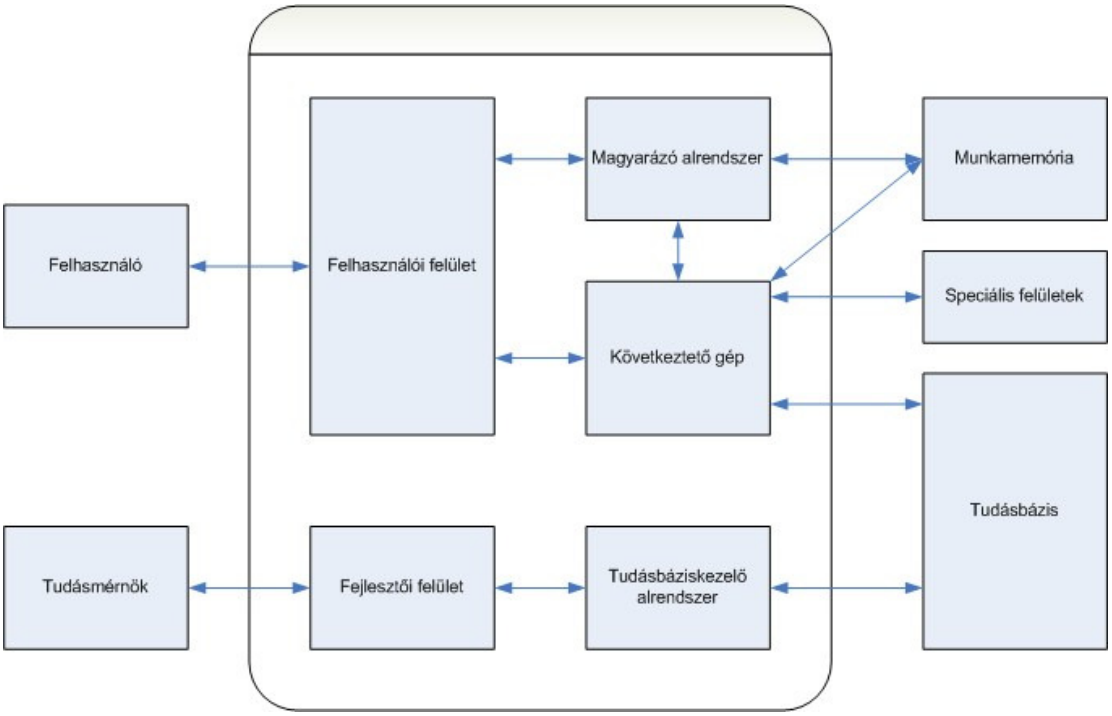
Mint korábban már megismertük, egy szakértő rendszer működéséhez két lényeges elem szükséges: a **tudásbázis** és a **következtető gép**.

A tudásbázis "HA-AKKOR" típusú szabályok, objektumok, tények, heurisztikák segítségével megfogalmazva tartalmazza a problémakör ismeretanyagát.

A következtető gép a logika következtető módszerei segítségével a tudásbázisból újabb tényeket vezet le, és esetleg a felhasználótól újabb adatokat kérdez. A következtetési lánc végén megadja a "megoldást", vagy azt, hogy nincs "megoldás". A megoldáshoz

általában nem rendelkeznek konkrét megoldó algoritmussal, a megoldást a tudásbázis segítségével, a probléma és a tárgykör függvényében építik fel.

Ahhoz, hogy a rendszer kommunikálni tudjon a felhasználóval, szükség van valamilyen felhasználói felületre. Ez lehetővé tesz egy fajta párbeszédet. Tehát egy szakértő rendszer kiegészül egy nagyon fontos elemmel a **felhasználói felület**tel.




***3. ábra: Szakértő rendszerek felépítése***

Ahogy az ábrán is látható, több különálló modulból épül fel a szakértő rendszer. Minden egyes modulnak más és más a feladata, viszont a probléma megoldásánál ezek szorosan együtt működnek.

Az egyes modulok:

- **Ismeretszerző modul**

  Ez a következő részekből áll: fejlesztői felület és tudásbáziskezelő alrendszer. Feladata lehetőséget biztosítani a tudásmérnöknek, hogy az ismereteket (tudást) formalizál módon vigye fel a tudásbázisba.

- **Felhasználói felület**

  Feladata, hogy lehetőséget biztosítson a felhasználónak a rendszerrel való kommunikációra. Ezen keresztül teheti fel a kérdéseit a szakértő rendszernek a felhasználó és fordítva.
- **Következtető gép**

  A logika következtető módszerei segítségével a tudásbázisból automatikusan újabb tényeket vezet le és interaktív módon újabb adatokat kér a felhasználótól.
- **Tudásbázis**

  Ez a szakértő rendszer hosszútávú memóriája. Az adott problémakör ismeretanyagát tartalmazza.
- **Magyarázó alrendszer**

  Ennek segítségével pillanthatunk be a szakértő rendszer következtetési folyamatába. Tájékoztatja a felhasználót a megoldás aktuális állapotáról, megmagyarázza például a felhasználónak feltett kérdéseket, illetve a feladatmegoldás után megindokolja a rendszer javaslatát, megmutatja, hogy milyen összefüggéseket használt a megoldás során a rendszer.
- **Munkamemória**

  Mondhatjuk azt, hogy ez a szakértő rendszer rövid távú memóriája. A felhasználótól kapott és/vagy a következtetések során nyert új tények kerülnek ide.
- **Speciális felületek**

  Ezen keresztül tudjuk a következtető gép működését befolyásolni, paraméterezni.

Felmerülhet bennünk a kérdés, hogy érdemes-e szakértő rendszereket fejlesztenünk és használnunk, ha egyszer vannak emberi szakértőink. Nézzük meg az előnyeit és hátrányait.

A szakértő rendszerek előnyei:

- Több szakértő tudásával rendelkezik, ezért jobb döntéseket hozhat.
- Gyorsabban szolgál megoldással, mint az ember. Egy szakértő rendszer probléma megoldásának ideje percekben mérhető, míg az ember órákat is eltölthet egy hasonló problémával.
- Mindig ugyanazt a megoldást szolgáltatja ugyanarra a problémára, tehát nem befolyásolják a külső körülmények.

- Gyorsan egymás után sok problémát képes megoldani, az átállási ideje minimális, ellentétben egy emberi szakértő átállási idejével.
- Használatának nincsenek helyi és időbeli korlátai. Egy emberi szakértő munkaidőben, megszokott helyen dolgozik.
- Egyszerűen, olcsón többszörözhető.

A szakértő rendszerek hátrányai:

- Egy szűk szakterületen, speciális problémák megoldására alkalmas.
- Nem tud "józan ésszel" gondolkodni, csak a megadott szabályok szerint működni.
- Mivel csak szabályokat követve hoz döntést, olyan helyzetekben amelyekre nincsenek definiálva szabályok, rossz döntéseket hozhat.

Az előnyöket és hátrányokat figyelembe véve miért is érdemes szakértő rendszereket készítenünk:

- A szakértői tudás és a problémamegoldás formalizálása előnyös.
- A szaktudás egy része állandóan változik.
- A felhasznált szabályok, előírások bonyolultak.
- Ha az emberi szakértők nagyon túlterheltek.
- A munka nagy része rutinfeladat, vagy elvégzésük kevesebb szakértelmet kíván.
- A szakértők gyakran váltanak munkahelyet.

### 1.2.1 A szakértő rendszerek alkalmazási területei

| Problématípus | Feladat |
|---|---|
| Irányítás | Egy adott rendszer működésének szabályozása az előírt specifikációk alapján. |
| Konfigurálás | Objektumok összeállítása adott feltételek szerint. |
| Diagnosztika | Megfigyelhető jellemzők alapján a működési hibákra való következtetés. |
| Értelmezés | Adatok alapján egy eset leíráséra való következtetés. |

| | |
|---|---|
| Szimuláció | Egy rendszer komponensei közti kölcsönhatások modellezése. |
| Szelekció | Adott lehetőségek közül a legjobb kiválasztása. |
| Előrejelzés | Következtetés adott szituáció következményeire. |

## *1.3 Tudásbázis (Knowledge-base), következtető gépek (Inference engines)*

Eddig szakértő rendszerekről, és ismeretalapú rendszerekről volt szó általánosságban. Megnéztük előnyeiket, hátrányaikat, hasznukat, korlátaikat, és szerkezetüket is nagy vonalakban. Most nézzük meg részletesen, hogy milyen részekből épülnek fel. Azt tudjuk, hogy egy ismeretalapú rendszerhez kell tudásbázis, következtető gép, és ahhoz, hogy használatba is tudjuk venni, egy felhasználói felület.

Itt a tudásbázisra és a következtető gépre fogunk koncentrálni, megnézzük ezek fajtáit, módszereit.

### 1.3.1 Tudásbázis

Az ismeretalapú/szakértő rendszer tudásbázisa sok szempontból hasonlít egy ember tudására. Itt is információ található, ez sok kis különálló részből épül fel, ezek között van valamilyen kapcsolat. Az emberi gondolkodás is ezeknek a kapcsolatoknak a segítségével jut el egyik dologtól a másikhoz. Az, hogy egyik gondolattól éppen melyikhez jutunk el, lehet szabályozott, de nagyon sok múlik a heurisztikán, tehát valamilyen ötleten vagy korábbi tapasztalaton. Tehát az emberi tudás és a rendszer tudása is valamilyen struktúrába van szervezve. Ez, tehát maga a rengeteg információ, a felszínes tudás. A mély tudás azt jelenti, hogy azt is tudjuk, hogy milyen információval rendelkezünk, vagyis mit tudunk, és ez a tudás milyen struktúrát, rendszert alkot.

A szakértő rendszerben az adott szakterület ismereteit nevezzük tudásnak. A tudás a problémamegoldásban játszott szerepe alapján osztályozható a következő módon: ad-e

instrukciókat a problémamegoldáshoz, csak az ismert tények, objektumok leírására szorítkozik, vagy a tudás struktúráját, absztrakt modelljét építi fel, összekapcsolva esetleg problémamegoldó instrukciókkal is.

A három fő tudástípus a következő:

- **Procedurális tudás**

  Megadja, hogy hogyan kell a problémát megoldani, hogy hogyan hajtsuk végre az egyes lépéseket. Általában szabályok, eljárások, függvények segítségével szokás leírni az ilyen fajta tudást.
- **Deklaratív tudás**

  A problémához tartozó azon ismereteket írja le, amelyeknél nem áll rendelkezésre semmilyen utasítás arra, hogy hogyan kell őket alkalmazni, milyen összefüggés van közöttük. Ezek általában egyszerű logikai kifejezések, tények, de a problémához tartozó fogalmak, objektumok leírása is ide tartozik.
- **Strukturált tudás**

  A fogalmak, objektumok közti kapcsolatok modelljét adja meg. A kapcsolatok leírása általában grafikus formában történik, a fogalmak, objektumok pedig valamilyen adatstruktúrával jellemezhetőek.

A tudást szakterületenként eltérő formában és struktúrában kell leírnunk. Azonban kialakult néhány leírási forma, amely a szakterületek többségénél alkalmazható. Ezt nevezik **tudásábrázolás**nak vagy **tudásreprezentáció**nak, de elterjedt elnevezés még az **ismeretreprezentáció** is.

A tudásreprezentáció egy tárgykörről szerzett ismeretek ábrázolása olyan szerkezetben, amely a tárgykörben felmerülő feladatok számítógépes megoldását megkönnyítik, vagyis olyan technika, amely lehetővé teszi a szaktudás leírását a tudásbázisban.

Egy probléma megoldása során alapvető szerepe van annak, hogy milyen módon írjuk le, hogyan reprezentáljuk a megoldáshoz szükséges ismereteinket. Többféle ismeretreprezentációs technikát dolgoztak ki, melyek közül a felhasználó kiválaszthatja a

problémájához legjobban illeszkedő reprezentációs módszert.

### 1.3.2 Tudásábrázolási technikák

Egy szakértő rendszer tudásbázisában kevesebb a kapcsolat az egyes információrészek között, mint egy ember memóriájában, de a kapcsolatok pontosabban meghatározottak. Fontos itt megjegyeznünk, hogy az alkalmazott tudásábrázolási módszer elválasztathatatlan az alkalmazott következtetési mechanizmustól. Vagyis, ha valamelyik ábrázolási mód mellett döntünk, akkor azzal be is határoljuk, hogy mely következtetési algoritmusokat használhatjuk. Egy adott reprezentációs nyelvnek számos feltételnek kell megfelelnie, annak érdekében, hogy biztosítsa a hatékony felhasználást.

A következő feltételeknek kell megfelelniük:

- Legyen tömör és kifejező.
- Legyen egyértelmű.
- Legyen független az általa ábrázolni kívánt tudástól.
- Legyen az általa leírt információ felhasználható következtetésre.

A következőkben megvizsgálunk néhány tudásábrázolási technikákat.

#### *1.3.2.1 Szabályalapú*

Ez a leggyakoribb tudásábrázolási forma. A tudásbázis pontosan definiált szabályokból épül fel, és ezek alkotják a kapcsolatot is az információrészek között. A szabályalapú ábrázolási mód a logikán alapul, mégpedig az elsőrendű logikán, más néven a predikátumkalkuluson.

A szabályalapú rendszerek tudásbázisa két fő részből áll: a **munkamemóriából** és **szabálybázisból**. A munkamemória az elemi tényállítások halmaza, ez rögzíti azokat a tényeket a világról, melyeket a problémamegoldás pillanatában ismerünk. Azok a tények/állítások, amelyek nincsenek benne a munkamemóriában, hamisnak tekintjük.

A tudásbázis másik fő része a szabálybázis, ebben szabályok vannak, mely szabályok általában **HA-AKKOR** vagy **AKKOR-HA** alakúak. Ez nagyon hasonlít a programozási nyelvekben használatos IF-THEN szerkezetekhez. A szabályalapú ábrázolás az ELSE ág fogalmát nem ismeri, ezt általában egy másik szabály írja le.

A szabály egy feltétel-esemény pár, a feltétel (premissza) az első rész a HA után, az esemény (következmény, konklúzió) pedig a második (az AKKOR után), ami végrehajtódik, ha a feltétel teljesül. A feltétel részben tények logikai (ÉS, VAGY) kapcsolata definiálható, illetve tagadás fogalmazható meg. A feltétel rész minden esetben egy munkamemóriabeli tényre/tényekre vonatkozik.
Az esemény azt mondja meg, hogy mi történjék a szabály aktivizálódása után. Ennek kétféle hatása lehet: módosíthatja a munkamemóriát újabb tények hozzáadásával vagy törlésével, illetve mellékhatásként input/output jellegű feladatokat láthat el.

A szabályalapú ábrázolás előnyei:

- **Modularitás**. A szabályok, amelyek az ismeretanyag egy-egy egységét képezik egymástól függetlenül hozhatók létre, törölhetők vagy módosíthatók.
- **Univerzális megjelenés**. Ismereteinket kizárólag szabályok formájában fogalmazunk meg.
- **Természetesség**. A hétköznapi életben is igen sok helyzetben feltételes szabályokkal fejezzük ki magunkat.
- **Bizonytalanságkezelés**i lehetőségekkel a rendszer könnyen kiegészíthető.

A szabályalapú ábrázolás hátrányai:

- **Végtelen láncolás**. Mind adatvezérelt, mind célvezérelt esetben – éppen a modularitás miatt – könnyen írhatók olyan szabályok, melyek esetében a következtető mechanizmus a feladatmegoldás során végtelen szabályláncot generál.
- **Új, a korábbiakkal ellentmondó ismeret beépítése lehetséges**, mivel nincsen általános módszer a szabályok esetleges ellentmondásainak ellenőrzésére, így a modularitás adta lehetőség miatt – egy új szabály könnyen ellentmondásossá teheti a szabályrendszert.
- Meglévő **szabályok módosításánál** is fennáll az előző két lehetőség.

- A **stratégiát módosító ún. meta-szabályokkal a beépített vezérlési stratégiát módosítani lehet**. Ezek azonban formailag nem különböznek a tárgyterületi ismeretanyagot leíró szabályoktól, ami megtévesztő lehet. Egy szabály jól-strukturáltságát erősen lerontja az, ha feltétel részében keverednek e kétféle ismeretanyagra utaló állítások.
- **Nincs szabványosítva a szabályok nyelve**, ez implementációnként nagyon eltérhet, ami a tudásbázisok másik rendszerbe való átvitelét jelentősen megnehezíti.

Példák szabályalapú ábrázolásra:

- **HA** esik az eső **AKKOR** használj esernyőt
- **HA** beteg vagy **AKKOR** menj orvoshoz
- **AKKOR** lesz jó a lágytojás **HA** sót teszel a hideg vízbe **ÉS** a vizet felforralod **ÉS** beleteszed a tojást **ÉS** 4 percig főzöd **ÉS** hideg vizet engedsz rá

#### *1.3.2.2 Struktúrált vagy keretalapú*

A gondolkodásunk tárgyakhoz, objektumokhoz és azok közös jellemzőit megragadó fogalmakhoz, ún. osztályokhoz illetve prototípusokhoz kötődik. Ezt használja ki a keretalapú reprezentáció, amely lehetővé teszi osztályok, osztály hierarchiák és adott osztályba tartozó egyedek definiálását és ezen objektumok hatékony manipulálását. Minsky a 70-es években fejlesztette ki a keretalapú reprezentációs módszert. Szerinte minden átélt szituáció a hozzátartozó viselkedéssel együtt egy gondolati egységet képez, amelyet agyunk keret formájában tárol. Új szituációba kerülve a már átélt szituációhoz illesztve alakítjuk ki viselkedésünket, elvárásainkat, miközben a korábbi keretet esetleg módosítjuk vagy új keretekkel bővítjük. Ezen ábrázolási forma elterjedéséhez nagyban hozzájárult az objektum-orientált programozás elterjedése, melynek néhány jellegzetességét megtalálhatjuk a keretek között is.

A keret olyan információ tárolási egység, amely összefogja azokat a tulajdonságokat, melyek egyetlen objektumra jellemzőek és azokat a relációkat, amik más objektumokhoz kötik. Mind a tulajdonságokat, mind a relációkat, rekeszekkel (slot) és a hozzájuk tartozó értékek párosaival lehet leírni. A keretek mind osztályok, mind egyedek leírására

szolgálhatnak. Amennyiben a keret osztályt határoz meg, akkor az **is-a** reláció segítségével megadható, hogy milyen más osztály alosztályának tekinthető. Így osztály hierarchia adható meg. Ha a keret egy egyed leírására szolgál, akkor meg kell adni, hogy melyik osztályhoz tartozik. Erre szolgál az **instance-of** reláció. Ahogy egy objektumot többféle nézőpontból is tekinthetünk, ugyanúgy megadhatjuk, hogy egy objektum egyszerre több osztály egyede is lehessen.

A keretalapú rendszerekben alapvető következtetés az öröklődés: az egyes osztályok megöröklik a hierarchiában felettük elhelyezkedő osztályok rekeszeit és ha vannak akkor azok értékeit is. Ehhez hasonlóan az egyes egyedek is megöröklik azon osztályok tulajdonságait és értékeit, amelyekhez tartoznak.

A több helyről való öröklődés lehetősége konfliktushoz vezethet. Ennek feloldására több lehetőség van:

- Ha a konfliktus egy öröklődés mentén jött létre, akkor a legspecifikusabb osztály meghatározásából származó értéket szokás elfogadni.
- Ha a konfliktus forrása több ágról való öröklődés, akkor ennek feloldásához különbséget kell tenni szükséges és tipikus tulajdonságok között.

Eddig a pontig a keretalapú reprezentáció is lényegében logikai reprezentáció, csak más, az emberi olvasatot megkönnyítő strukturáltabb formában. Megváltozik a helyzet, ha megengedjük, hogy az egyes rekeszekhez ne csak értékeket, hanem számítási eljárásokat is, démonokat vagy metódusokat lehessen rendelni. A démonok vagy metódusok akkor lépnek működésbe, ha a keretet manipuláljuk. A metódus típusa határozza meg az aktivizálódás feltételét. Az egyik akkor aktivizálódik, amikor a rekesz értékére szükség van, de az még nem ismert. A másik, akkor lép működésbe, amikor a rekesz értéke megváltozik. Ebben az esetben a metódus előre meghatározott módon vagy az új vagy a régi értékkel dolgozhat.
A metódusok ugyanúgy öröklődnek, mint a rekeszek és értékeik.

A keretalapú ábrázolás előnyei:

- **Eseményvezérelt végrehajtás**. A démonok, olyan eljárások, amelyek végrehajtása eseményvezérelt. A démonok csak akkor aktivizálják a rendszert ha az adott rekeszben

előre specifikált értékváltozás következik be.

- **Az ismeretek szervezése**. A keretek ismereteinket egy világos, áttekinthető struktúrába szervezik, ahol az egyes rekeszek tartalmának elérése közvetlenül történik.
- **Önvezérlés**. A keretek úgy vannak strukturálva, hogy az adott helyzetben képesek meghatározni saját alkalmazhatóságukat.
- **Dinamikus értékek elhelyezése**. Az egyes rekeszek értéke a problémamegoldás során dinamikusan, könnyen változtatható.
- **Deklaratív és procedurális ismeretek együttes ábrázolása**. A deklaratív ismereteket a rekeszek neve és tartalma, míg a procedurális ismereteket az egyes rekeszekhez kapcsolódó démonok testesítik meg.

A keretalapú ábrázolás hátrányai:

- **Prototípusoktól való eltérés**. Nagyon sok valós jelenség eltér a megszokott sémától. Ha nem megfelelő absztrakciós szinttől indítottuk a keretek kibontását, a kivételek miatt nagyon elbonyolódhat a rendszer.
- **Új szituációkhoz való alkalmazkodás problémás**. Ha olyan új szituációt vagy objektumot kell reprezentálni, amelyet nem tudunk a hierarchiába beilleszteni. Ez korlátozza a keretalapú reprezentáció alkalmazhatóságát.
- **A heurisztikus ismeretek ábrázolása problémás**. Míg a szabályalapú rendszereknél könnyen írhatunk heurisztikákat itt nem. Keretekkel például könnyen le lehet írni egy tipikus betegséget, egy általánosat, valamely egyedi betegség specifikus szimptómáit, azonban azt, hogy az egyes szimptómák hogyan hatnak egymásra, így hogyan befolyásolják a diagnózist, inkább szabályok segítségével lehet megadni.

#### *1.3.2.3 Logika*

A logika olyan leíró nyelv, mely lehetővé teszi, hogy a világ tényeire vonatkozó állításokat mondatok formájában fogalmazzuk meg. A mondatokból azután mechanikus eljárások segítségével újabb mondatok származtathatók. Ezt a folyamatot hívjuk következtetésnek. A tények, amelyek a világ részei, lehetnek igazak és hamisak. Mivel a világ tényei nem vihetők be a gépbe, a tényeket ábrázolni kell, azaz valamilyen nyelven ki kell azokat fejezni. A logikai reprezentációban a tényekre vonatkozó állításokat mondatok fejezik

ki. A világ és reprezentációja az értelmezés réven kapcsolódik egymáshoz, amely megfeleltetést biztosít a tények és mondatok között. Általános esetben több értelmezés is lehetséges.

A következtetés nem más mint adott szintaxis és szemantika mellett új mondatok származtatása a már ismert mondatokból. A következtetéstől elvárjuk, hogy ugyanazon értelmezésben az új mondatok is a világ ugyanazon tényeire utaljanak.

Egy mondat igaz, ha olyan tényre utal, amely valóban előfordul a világban, azaz a mondat igazsága függ a világ állapotától és az értelmezésétől. Vannak olyan, ún. érvényes mondatok, amelyek igazsága sem a világ állapotától, sem az értelmezéstől nem függ. Ezeket nevezzük tautológiának. Bizonyos mondatok kielégíthetetlenek, azaz olyannak írják le a világot, amilyen sohasem lehet.
Úgy tűnhet, hogy az érvényes és kielégíthetetlen mondatok feleslegesek, hiszen csak nyilvánvalóan igaz illetve hamis dolgokat fejeznek ki, ám éppen létüknek köszönhető, hogy a számítógépekkel logikai következtetéseket tudunk végrehajtani. A következmények gépi kiszámításakor ugyanis a világról csak annyi tudható, ami az tudásbázisban van, de az már nem hogy, miként feleltethetők meg egymásnak a reprezentációs nyelv elemei és a világ tényei. Ahhoz, hogy automatikusan eldönthető legyen, hogy egy konklúzió következik-e a tudásbázisból, ki kell mutatni, hogy a "ha a tudásbázis igaz, akkor a konklúzió igaz" összetett állítás érvényes, azaz minden értelmezésben igaz.

#### *1.3.2.4 Esetalapú*

Az esetalapú tudásábrázolás ötlete onnan származik, hogy a szakértők az esetek nagy részében nem egyedi problémákkal kerülnek szembe, hanem ugyanazon problémák különböző variációival. Ha rendelkezésünkre állnak olyan esetek, melyek nagy hasonlóságot mutatnak a megoldandó problémával, akkor az aktuális problémára könnyebb megoldást találni. Ezt alátámasztja az is, hogy a szakértők is a saját, korábbi tapasztalataikra támaszkodnak a problémák megoldásánál.

A fentebb leírt okok miatt ez egy népszerű ábrázolási forma. Az esetalapú ábrázolást

használva a tudásbázis nagyon hasonlít egy lexikonhoz, ami konkrét eseteket tárol. Előnyei közé tartozik, hogy sokkal közelebb áll az emberi gondolkodáshoz, mint mondjuk a szabályalapú ábrázolási módszer. A tudásbázist korábbi esetek feldolgozásával, kielemzésével kell feltölteni. Minden egyes esetnek tartalmaznia kell a probléma pontos leírását, a körülményeket, amelyek a probléma kezdetekor fennálltak. Ezeken kívül tartalmaznia kell a probléma megoldását és eredményét is. Az a folyamat azonban nincs letárolva, hogy az adott megoldás hogyan született, vagyis a következtetési folyamat nincs letárolva. Ez ellentétben áll a szabályalapú ábrázolásnál megismertekkel vagyis, hogy nyomon lehet követni, hogy az adott a következtetés hogyan született.

#### *1.3.2.5 Hibrid*

A legtöbb rendszerben nem határolódnak el ilyen tisztán a tudásreprezentációs módszerek, hanem valamilyen módon ötvözik ezeket. A keretalapú ábrázolásnál említett okok miatt, inkább hibrid rendszereket alkalmaznak, amelyek ötvözik a keret- és a heurisztikus ismeretek leírására alkalmas szabályalapú reprezentálási módokat. A keret attribútumok közötti kapcsolatokat, heurisztikus ismereteket lehet szabályokkal leírni. Természetesen egy hibrid rendszer következtető gépében ezért a keret-struktúra feladatmegoldásra való mozgósítását ellátó öröklődési-, démon és egyéb mechanizmusok mellett megtalálható a szabályok végrehajtását biztosító mechanizmusok is.

### 1.3.3 Következtető gép

A szakértő rendszer másik része (a tudásbázis mellett) a következtető gép. Ez ad értelmet a tudásbázisban tárolt információnak, ez kapcsolja össze az információdarabokat, és vonja le a megfelelő következtetéseket. A megfelelő kérdéseket feltéve a felhasználó felé, aktivizálja a szükséges szabályokat, ismereteket, vagy újabb kérdéseket tesz fel, vagy ad valamilyen választ. De nemcsak a következtetéseket tudja elvégezni, hanem vissza is tud következtetni a saját maga által megtett lépésekre, így meg is tudja magyarázni, hogy miért hozta épp az adott döntést.

A következtető rendszer kiindulhat bizonyos adatokból, tényekből, és ezekből hoz egy

döntést. Ez az **előre haladó következtetés**.
Példa: ha tudjuk egy betegség tüneteit, és a betegségre vagyunk kíváncsiak.

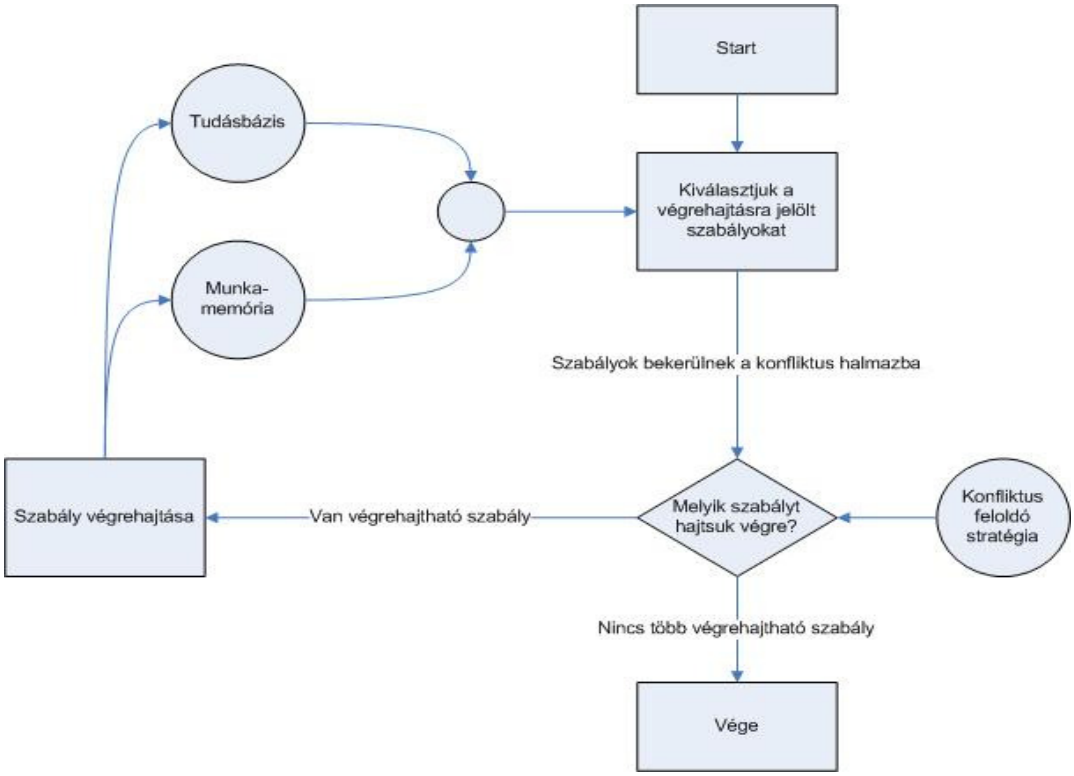
Ha egy célból, válaszból halad visszafelé az indokokhoz, kiváltó okokhoz, azt **visszafele haladó következtetés**nek hívjuk.
Példa: elvesztettük az összes pénzünket a tőzsdén, és szeretnénk tudni ennek az okát.

### 1.3.4 Következtetési eljárás

A következtetés modellje az emberi következtetésen alapul, miszerint az ember a hosszú és rövid távú memória segítségével határoz meg új dolgokat. A szakértő rendszernél a hosszú távú memória szerepét a tudásbázis, a rövid távúét a munkamemória, a következtetési eljárását pedig a következtető gép veszi át.
A következtetési eljárás a szakértő rendszerekben a korábbi ismeretek alapján határoz meg új ismereteket.

### *1.3.4.1 Szabályalapú következtetés*

#### 1.3.4.1.1 Szabályalapú adatvezérelt következtetés




***4. ábra: Szabályalapú adatvezérelt következtetés***

Ezt előre haladó következtetésnek is nevezik. Elve az, hogy egy adott kezdeti állapotból kiindulva valamely célállapotot megalkot, illetve oda eljut. A rendszer először a felhasználótól kéri a kiindulási adatokat, majd ezekből a tényekből indul ki, és a célállítás felé halad. Minden lépésben először az egyes szabályok **feltételrész**ét illeszti az aktuális tényekhez, azután azok közül a szabályok közül, amelyek feltételrésze igaz, tehát illeszkednek, kiválaszt egyet, és végrehajtja az adott szabály **következmény** részében előírt műveleteket. Ez az adat bekerül új tényként a munkamemóriába, és ennek segítségével új szabályokat alkalmaz. A következtetés akkor fejeződik be sikeresen, ha a célállítás megjelenik a tények között. Ezt az eljárást **adatvezérelt következtetés**nek (data driven) vagy **előre láncolásnak** (forward chaining) nevezik.

Itt a szabályok természetes formája: HA feltétel, AKKOR tevékenység.

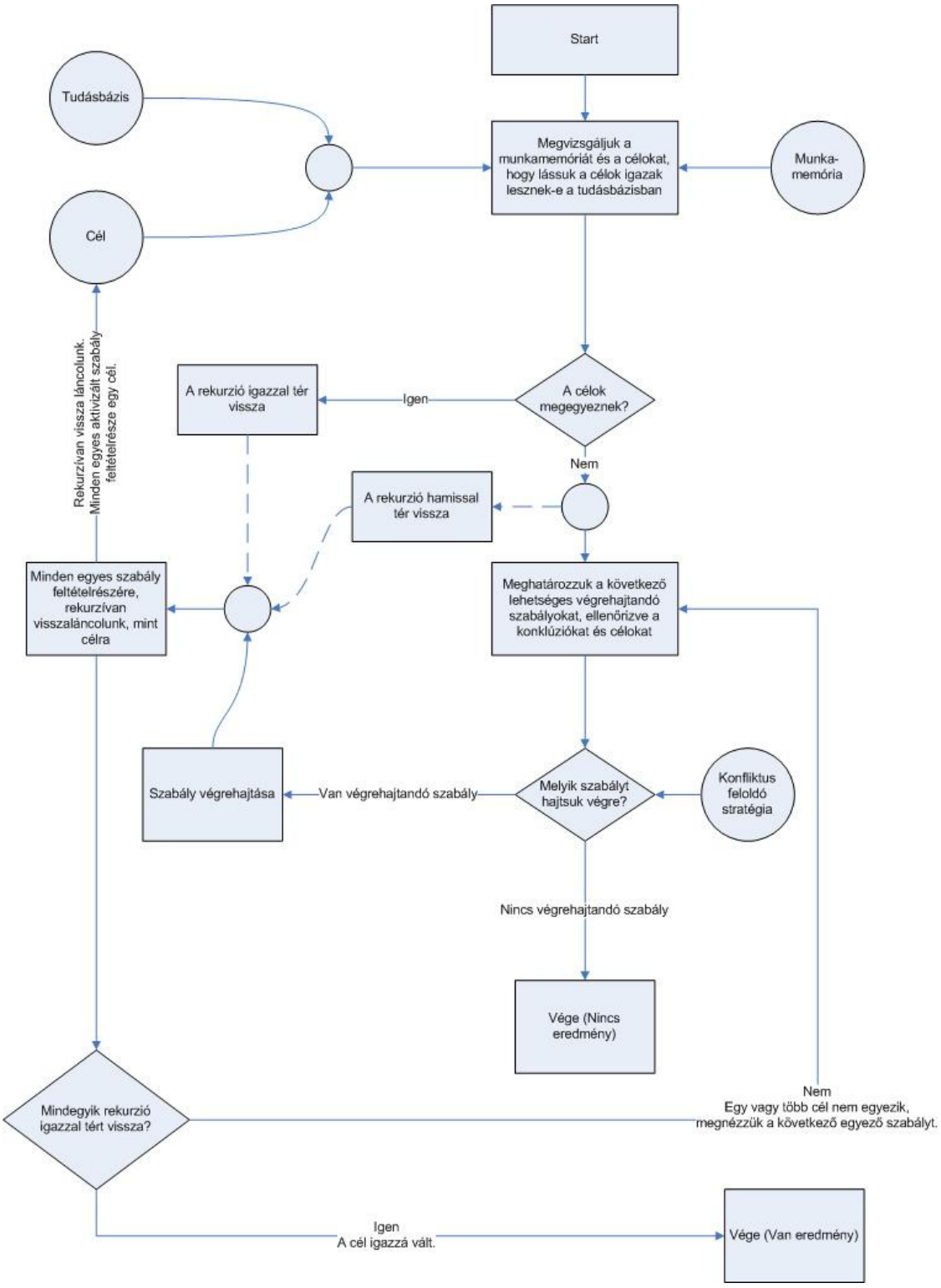
Példa szabályalapú adatvezérelt következtetésre:

- Tudásbázis:
  - SZ1: **HA** Z **AKKOR** teljesül = Igaz
  - SZ2: **HA** F v B **AKKOR** Z
  - SZ3: **HA** G ^ D **AKKOR** E
  - SZ4: **HA** E v J **AKKOR** F
  - SZ5: **HA** A **AKKOR** D
- Munkamemória:
  - A, G, H = Igaz
  - B, C, J = Hamis
  - D, E, Z, F = Ismeretlen

Az adatvezérelt következtetés lépései:

- 1. lépés: Végig nézzük az összes szabályt, hogy melyik feltételrészben van olyan tény, ami igaz.
  - SZ1: (Z = Ismeretlen) = Hamis
  - SZ2: (F = Ismeretlen) v (B = Hamis) = Hamis
  - SZ3: (G = Igaz) ^ (D = Ismeretlen) = Hamis
  - SZ4: (E = Ismeretlen) v (J = Hamis) = Hamis
  - SZ5: (A = Igaz) = Igaz -> Aktivizálódik, D = Igaz, ez egy új tény a munkamemóriába.
- 2. lépés:
  - SZ3 aktivizálódik, E = Igaz
- 3. lépés:
  - SZ4 aktivizálódik, F = Igaz
- 4. lépés:
  - SZ2 aktivizálódik, Z = Igaz
- 5. lépés:
  - SZ1 aktivizálódik, cél teljesül!

### 1.3.4.1.2 Szabályalapú célvezérelt következtetés




***5. ábra: Szabályalapú célvezérelt következtetés***

Visszafelé haladó következtetésnek is hívják. Egy célállítás érvényességére következtet a kezdetben érvényes tényekre támaszkodva. Egy célállításból indul ki, és a tények irányába halad.

A célállítás igazolását visszavezeti már ismert tényekre. Minden lépésben egy szabály következményrészét illeszti a célhoz, vagy valamelyik részcélhoz, és visszavezeti a cél vagy részcél igazolását a szabály feltételrészét alkotó tények igazolására. A következtetés akkor ér véget sikeresen, ha végül minden részcél illeszthető valamely tényhez. Ha a célhoz vagy részcélhoz az eljárás nem talál illeszthető tényt vagy szabályt, akkor az eljárás visszalép, és új szabály illesztésével próbálkozik. (Ha ilyet sem talál, megkérdezheti a felhasználót.) Ezt **célvezérelt következtetés**nek (goal driven) vagy **visszafelé láncolás**nak (backward chaining) nevezik.

Itt a szabályok természetes formája az AKKOR tevékenység, HA előzmény, mivel a szabályok kidolgozása a következményrész illesztésével kezdődik.

Példa szabályalapú célvezérelt következtetésre:

- Tudásbázis:
    - SZ1: **HA** Z **AKKOR** teljesül = Igaz
    - SZ2: **HA** F v B **AKKOR** Z
    - SZ3: **HA** G ^ D **AKKOR** E
    - SZ4: **HA** E v J **AKKOR** F
    - SZ5: **HA** A **AKKOR** D

Az adatvezérelt következtetés lépései:

- Cél: ismeretlen
- 1. lépés: Keressünk egy célszabályt: SZ1 konklúziója
- 2. lépés: SZ1 premissza vizsgálata: Z ismeretlen
- 3. lépés: Z melyik szabályban szerepel konklúzióként (célszabály keresés), ez lesz a célszabály: SZ2
- 4. lépés: SZ2 premissza vizsgálat: F ismeretlen
- 5. lépés: Az újabb részcél F lesz, keresése konklúzióban: SZ4-ben szerepel, ez lesz a

célszabály

- 6. lépés: SZ4 premissza vizsgálata: E ismeretlen, a részcél ez lesz
- 7. lépés: E az SZ3-ban szerepel konklúzióként, ez lesz a célszabály
- 8. lépés: A részcél G lesz, újabb célszabály keresése, G nem szerepel sehol
- 9. lépés: Megkérdezzük a felhasználót: G = Igaz
- 10. lépés: G az SZ3-ban szerepel, ezért SZ3 premissza vizsgálata: D ismeretlen
- 11. lépés: D keresése konklúzióban: SZ5
- 12. lépés: SZ5 premissza vizsgálata: A ismeretlen
- 13. lépés: A keresése konklúzióban: nincs ilyen, felhasználót megkérdezni
- 14. lépés: A = Igaz, SZ5 aktivizálódik, konklúzió: D = Igaz
- 15. lépés: SZ3 aktivizálódik, konklúzió: E = Igaz
- 16. lépés: SZ4 aktivizálódik, konklúzió: F = Igaz
- 17. lépés: SZ2 aktivizálódik, konklúzió: Z = Igaz
- 18. lépés: SZ1 aktivizálódik, konklúzió: teljesül = Igaz
- 19. lépés: A cél értéke ismertté vált

#### *1.3.4.2 Logika*

A legegyszerűbb formális logika az ún. ítéletkalkulus, amelynek szintaxisa csak ítéletváltozókat és két ítéletkonstanst (IGAZ, HAMIS) enged meg, amelyek a szokásos logikai műveletekkel fűzhetők mondatokba. Ezen műveletek halmaza a logikai és (^), a logikai vagy (v), a tagadás (¬), az implikáció (→) és az ekvivalencia (↔). Műveleti jelek és zárójelek segítségével összetett mondatok is konstruálhatók. Az ítéletkalkulus szemantikájának alapja, hogy az értelmezés egy mondat ítéletváltozóihoz a logikai igaz (I) vagy a logikai hamis (H) értékeket rendeli minden lehetséges módon. Az IGAZ ítéletkonstanshoz csak I, a HAMIS-hoz pedig csak H rendelhető. Egy mondatnak tehát több értelmezése is lehetséges. A mondatok igazságértékének megállapítása a műveleti jelek jelentése alapján történik, amelyeket az alábbi igazságtábla foglal össze:

| **A** | **B** | **¬A** | **A^B** | **AvB** | **A→B** | **A↔B** |
|---|---|---|---|---|---|---|
| I | I | H | I | I | I | I |

| | | | | | | |
|---|---|---|---|---|---|---|
| I | H | | H | I | H | H |
| H | I | I | H | I | I | H |
| H | H | | H | H | I | I |

A műveletek közül az implikáció némileg magyarázatra szorul: ez állítás B értékéről ha A igaz. Ha A hamis, akkor A→B igaz, függetlenül B értékétől. Megjegyezzük, hogy A és B között nem feltételezünk semmilyen ok-okozati kapcsolatot.

Az, hogy egy állítás következik-e a már ismert állítások halmazából, az ítéletkalkulusban igazságtábla segítségével ellenőrizhetjük. Ha adott a premisszák halmaza (P) és egy konklúzió (Q), akkor készíteni kell egy igazságtáblát a P→Q mondat ellenőrzésére, ahol P a premisszák konjunkciója. Ha a tábla minden sora igaz, akkor a P→Q mondat érvényes, ami azt jelenti, hogy Q következik P-ből. Ilyen eljárás mutat példát a következő táblázat, ahol a premisszák halmaza {P1vP2,¬P2}, a konklúzió pedig P1.

| **P1** | **P2** | **P1vP2** | **¬P2** | **(P1vP2)^¬P2** | **(P1vP2)^¬P2→P1** |
|---|---|---|---|---|---|
| I | I | I | H | H | I |
| I | H | I | I | I | I |
| H | I | I | H | H | I |
| H | H | H | I | H | I |

Az igazságtábla segítségével való következtetés teljesen mechanikussá tehető, hiszen nem kell mást tenni, mint minden lehetséges módon I és H értékeket rendelni a premisszák ítéletváltozóihoz és kiértékelni az ilyen módon kapott értelmezett mondatokat a műveleti jelek jelentése alapján. Ehhez tehát nem kell semmit tudni a mondatok jelentéséről, arról, hogy a világ mely tényeit írtuk le velük.

A gond csupán az, hogy az eljárás idő és helyigénye exponenciálisan nő az ítéletváltozók számával, azaz ha az igazolandó mondat n szimbólumot tartalmaz akkor fel kell sorolni az igazságtábla $2^n$ sorát.

#### *1.3.4.3 Induktív következtetés*

Az induktív következtetés egyedi esetekből, példákból jut általános érvényű következtetésekre, szemben a logikai rendszerek deduktív következtetéseivel, amely az általános ismeretekből indul ki. Ez a technika a gépi tanulás egyik korai eredményén alapul, ami a példák alapján való tanulás módszere. Az ismereteket egy véges halmaz tartalmazza és ezek tartalmából szabályokat generál a rendszer.

A tudásbázis általában valamilyen táblázatos formában tartalmazza a példákat. Egy algoritmus ezek alapján a példák alapján szabályokat generál, majd ezek bekerülnek a tudásbázisba. A következtető rendszer egy új problémánál a legjellemzőbb szempontok alapján keres hasonló példát és a példához tartozó megoldás lesz az eredmény. Hasonló példa hiányában nem ad megoldást a rendszer.

#### *1.3.4.4 Esetalapú következtetés*

Az esetalapú következtetésnél ahhoz, hogy megoldjunk egy esetet, hasonlítjuk azt a tudásbázisban lévő korábbi esetekhez. Az eredményül kapott esetek felhasználásával a rendszer javaslatot tesz a megoldásra. Abban az esetben, ha ezzel megoldódik a probléma, akkor az aktuális esetet is berakjuk a tudásbázisba, ha szükség van rá, akkor akár módosítva.

A tudásbázisból a hasonló esetek kiválasztása a következő módon történik:

- Meg kell határozni azokat a kulcsfontosságú tulajdonságokat, amelyek jellemzik az adott problémát.
- Ki kell keresni a tudásbázisból a legjobban hasonlító eseteket, valamilyen hasonlóságot kifejező értékkel együtt.
- Ki kell választani azt az esetet ami a legnagyobb hasonlóságot mutatja a megoldandóval.

Egy adott probléma megoldásakor az esetalapú következtető gép megkeresi azokat a korábban megoldott eseteket, amelyek nagyon hasonlítanak az aktuálishoz. Ha ez meg van,

akkor a következtető gép átülteti a megtalált eset megoldását a vizsgált esetre. Ha szükséges, akkor módosításokat is végez az eseten, annak függvényében, hogy milyen adatokban és milyen mértékben tér el a régi eset az újtól. Ha a probléma sikeresen megoldódott, akkor a ezt a módosított esetet letárolja a tudásbázisban, annak érdekében, hogy később ezt újra lehessen haszonsítani.

Legelőször a rendszer megpróbálja megérteni a problémát, vagyis összegyűjti azokat a jellemzőket, amelyek egyértelműen azonosíthatják a problémát és elkülöníthetik azt más problémaosztályoktól. Ezek a jellemzők indexként szolgálnak a tudásbázisban, melyek segítségével könnyen és gyorsan elő lehet keresni a hasonló eseteket. Az ún. hasonlító függvények megpróbálják megtalálni a leginkább hasonló esetet a tudásbázisban az előbb említett indexek alapján. Az ilyen módon megtalált esetekre a rendszer, mint lehetséges megoldásokra tekint. A gyakorlatban nagyon kicsi annak az esélye, hogy olyan esetet találjunk amelyik teljes mértékben illeszkedik, emiatt a rendszernek adaptálnia, azaz át kell dolgoznia a megoldást. Az ún. adaptáló szabályok azt vizsgálják meg, hogy az egyes attribútumok melyekben eltérés mutatkozik, mennyiben változtatják meg a megoldási módszert. Ha sikeresen megoldódik a probléma, az eset eltárolásra kerül, annak megoldásával együtt. Ez gyakorlatilag egy tanulási folyamat. Léteznek olyan rendszerek, amelyek a sikertelen megoldásokat is eltárolják.

### 1.3.5 Tudásbeszerzés

A tudásbeszerzés a tudásalapú rendszerek fejlesztésének egyik legfontosabb részfeladata. Azt a folyamatot jelenti, amelynek során különböző technikákat alkalmazva a tudásmérnök megszerzi a tárgyköri szakértőtől azokat az információkat, melyekből a rendszer alapját képező tudásbázist fel fogja építeni. Ennek a folyamatnak a megkönnyítésére számos eszközt fejlesztettek ki, ezek hivatottak megkönnyíteni a tudásmérnök dolgát. A tudásbeszerzés különböző technikákkal végezhető, melyek a következő problémákat próbálják meg megoldani:

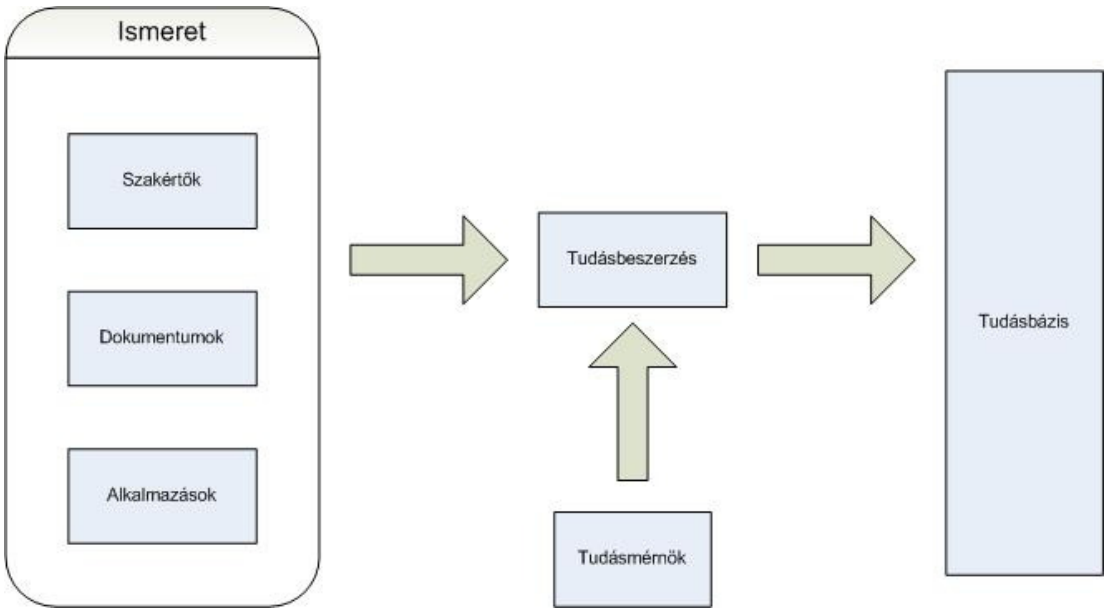
- Sok esetben a tudás a szakértők fejében van, nem pedig valamilyen kézzelfogható formában.
- A szakértők rengeteg információval rendelkeznek a szakterületükön.

- A passzív tudás megszerezése. Nagyon nehéz ezen tudást megragadni, mivel a szakértő nem minden esetben van tisztában azzal, hogy milyen ismereteket birtokol és használ.
- A szakértők nehezen érhetőek el, mert általában nagyon elfoglalt emberek.
- Nem létezik olyan szakértő, aki mindent tudna a szakterületéről.
- A tudásuk nem feltétlen örök érvényű.
- Előfordulhat az az eset, hogy ugyanazon szakterületen dolgozó szakértőknek a véleménye eltér az adott probléma megoldásáról.

A fentieket figyelembe véve, olyan tudásbeszerzési technikára van szükségünk, melyek:

- Rövid időre vonják el a szakértőket a munkájuktól.
- Lehetővé teszik, hogy a laikusok számára is érthető módon fogalmazzuk meg a tudást.
- Csak a rendszer szempontjából lényeges ismeretekre fókuszálnak.
- Lehetővé teszik a passzív tudás megragadását is.
- Lehetővé teszik, hogy a tudást több különböző szakértőtől is beszerezhessük.
- Lehetővé teszik a tudás ellenőrzését és bővítését a későbbiekben.

Gyakran előfordulnak olyan esetek, hogy a tudásmérnöknek szakértő nélkül kell dolgoznia, bár a megszerzett tudás forrása a legtöbb esetben a tárgyköri szakértő. Ilyen esetek akkor fordulnak elő, ha esetleg a szakértő nem elérhető, de olyan is előfordulhat, hogy a tudásmérnök a rendszert olyan egyéb ismeretekkel szeretné felruházni, melyek nincsenek az adott szakértő birtokában. Ekkor különböző szakirodalmakhoz fordulhat, majd azok elemzésével bővítheti a tudásbázist. Ennek van egy nagyon fontos feltétele, rendelkeznie kell olyan alapismeretekkel az adott szakterületen, hogy felismerje annak kulcsfontosságú fogalmait összefüggéseit.




***6. ábra: Tudásbeszerzés***

Sokféle módszert fejlesztettek ki a tudásbeszerzésre, ami a szakértő ismeretek széles skálájának köszönhető. Különböző technika alkalmazására van szükség, akkor ha az aktív tudásra vagy ha a passzívra vagyunk kíváncsiak. Ugyanígy különböző szakterületek ismereteit más-más módszerek ragadják meg a legkönnyebben.

#### *1.3.5.1 Kapcsolatháló*

A kapcsolatháló egy olyan gráf, melynek csomópontjaiban az adott szakterület fogalmai vannak. Ezen csomópontokat össze kell kötni nyilakkal, annak megfelelően, hogy van-e köztük valamilyen kapcsolat. A nyilakat fel kell címkézni, annak megfelelően, hogy milyen kapcsolatban áll egymással a két összekötött elem. Akkor alkalmazható nagy hatékonysággal ez a módszer, ha az alapismereteket szeretnénk összegyűjteni.

#### *1.3.5.2 Folyamatábra*

Olyan ábra vagy diagram, melynek segítségével az adott szakterületen fellelhető folyamatokat lehet megjeleníteni. Részei a folyamatban szereplő elemek, illetve a folyamatok

által előállított végtermékek is. Jól használható annak ábrázolására, hogy milyen módon kerülnek alkalmazásra az egyes ismeretek a gyakorlatban. Ez a módszer alkalmas lehet a passzív tudás feltérképezésére. Ehhez hasonló az állapot diagram is, mellyel a különböző folyamatokban résztvevő objektumok különböző állapotait, illetve az azokból elérhető újabb állapotokat ábrázoljuk.

#### *1.3.5.3 Mátrixalapú technikák*

A mátrixalapú technikák legtöbbször kétdimenziós táblázatok elkészítését jelentik, melyek a következő információk közötti viszonyokat tartalmazzák strukturált formában:

- Tulajdonság - Érték
- Problémák - Megoldás
- Hipotézisek - A diagnózishoz szükséges technikák
- Feladatok - Szükséges erőforrások

A mátrix elemei lehetnek szimbólumok, színek, számok vagy leggyakrabban szöveges információk. Az ilyen technikákat jellemzően a korábban megszerzett tudás ellenőrzésére, validálására és rendszerezésére használjuk, semmint újabb ismeretek beszerzésére. Emiatt a módszert csak más technikákkal kiegészítve használhatjuk.

#### *1.3.5.4 Interjú*

Ez az egyik legáltalánosabb módszer az ismeretek beszerzéséhez. A tudásmérnök kérdésekkel próbál meg információt szerezni az adott témakörben. A kérdésekre adott válaszokat a tudásmérnök megpróbálja mérnöki módon megragadni, illetve rendszerezni azokat. Három fő interjú típus van, melyek mindegyike fontos szerepet játszik egy tudásbeszerzési folyamatban.

Ezek a típusok a következők:

- **Strukturálatlan, spontán interjú**
  Nagy szabadságot enged meg mindkét félnek. Célja, hogy tisztázza a felhasználni kívánt ismeretek határait, illetve kiindulópontként szolgáljon a későbbi interjúk

számára.

- **Félig strukturált interjú**

  A tudásmérnök a megelőző interjúk alapján előre felkészül és olyan kérdéseket tesz fel a szakértőnek, melyek nagyban elősegítik a tudásbázis megalkotását. Ezen interjú típus lehetővé teszi, hogy miközben válaszokat ad a szakértő, a tudásmérnökben felmerülő újabb pontosítást igénylő kérdéseket is megválaszolja. Általában a kidolgozott kérdéseket előre elküldik a szakértőnek, hogy fel tudjun készülni azokra. Ez a típusú interjú az egyik legkedveltebb, mivel segíti, hogy a szakértő a lényegre fókuszáljon, és csak kulcsfontosságú információkat adjon a tudásmérnök részére.

- **Strukturált, kötött interjú**

  Semmiféle rugalmasságot nem enged meg a tudásmérnöknek. Előre megfogalmazott kérdéseket tesz fel a szakértőnek. A legtöbb esetben egy ilyen interjú inkább táblázatok kitöltésével és diagramok rajzolásával telik semmint beszélgetéssel.

Ezeknek interjú típusoknak a nagy előnyük, hogy a tudásbeszerzés bármely szakaszában jól használhatóak. Vagyis egészen a folyamat elejétől a végéig. Egyetlen nagy hátránya, hogy csak azon tudást képes megragadni, melyekkel a szakértő maga is tisztában van, vagyis a passzív tudás megszerzésére nem alkalmas. Olyan szakterületeken, ahol fontos szerepet játszhat a passzív tudás is, ott kiegészítő technikákat kell alkalmazni ezek megszerzésére.

# 2. Az üzleti szabály motorok

## *2.1 Mi az üzleti szabály*

Az üzleti szabályok leírják azokat a műveleteket, definíciókat és megszorításokat, amiket alkalmazva egy szervezet eltudja érni a céljait. Például egy üzleti szabály azt mondja, hogy "*a törzsvásárlóknak nem kell a hitelképességét vizsgálni*".

Azt meg kell jegyezni, hogy az üzleti szabályok különböznek a stratégia menedzsmenttől. Az üzleti szabályok egy stratégia implementációi. Ezek a szabályok mondják meg az üzletnek, hogy mit kell csinálnia, a stratégia pedig azt mondja meg, hogy hogyan és mire fókuszáljunk az eredmények optimalizálása érdekében. Nézzük egy kicsit más szemszögből: a stratégia biztosít egy magas szintű direktívákat arra, hogy az üzlet tudja, hogy mit kell csinálnia, az üzleti szabályok pedig azt mondják meg, hogy a stratégiát, hogyan kell lefordítani műveletekre.

Példák üzleti szabályokra:

- Árképzési politika.
- Piaci stratégiák.
- Humán erőforrás tevékenységek.

## *2.2 Mi az üzleti szabály motor*

Az üzleti szabály motor egy szoftver rendszer, ami segít menedzselni és automatizálni az üzleti szabályokat. Mint szoftver többek közt segít regisztrálni, osztályozni és menedzselni ezeket a szabályokat, és ellenőrizni, hogy konzisztensek-e.

Egy informatikai alkalmazásnál a szabályok gyakrabban változhatnak, mint alkalmazás kódja. Az üzleti szabály motorok olyan szoftver komponensek, melyek elkülönülnek az alkalmazás kódjától az üzleti szabályokat. Ez lehetővé teszi az üzletnek, hogy úgy módosítsák az üzleti szabályokat, hogy nincs szükség informatikusok beavatkozására,

tehát ezáltal az alkalmazás sokkal jobban tud alkalmazkodni a gyorsan változó üzleti igényekhez.

Az üzleti szabály motorok a szakértő rendszereknél megismert szabály alapú tudás ábrázolást használják, a következtetést pedig általában egy előre láncoló szabály alapú következtető gép végzi. Tehát mondhatjuk azt, hogy az üzleti szabály motorok egy fajta speciális szakértő rendszerek.

Általánosan jellemző, hogy a tények szabályokhoz való illesztését valamilyen hatékony algoritmus végzi, ilyenek például a `RETE`, a `LEAPS` és a `TREAT` algoritmusok. Nagyon fontos, hogy ezek az algoritmusok jó teljesítményt nyújtsanak, mert nagy mennyiségű szabály esetén komoly erőforrásokra lehet szükség az illesztés elvégzésére.

### 2.2.1 Miért és mikor használjunk egy üzleti szabály motort?

Gyakran feltesszük a következő kérdéseket:

- Mikor kell egy szabály motort használnunk?
- Milyen előnyei vannak egy szabály motornak, a kézzel programozott "if-then" szerkezetekkel szemben?

A következőkben ezekre és ehhez hasonló kérdésekre próbálok meg választ adni.

### 2.2.2 A szabály motorok előnyei

- **Deklaratív programozás**
  Ha egy szabály motort használunk, akkor azt kell megfogalmaznunk hogy mit szeretnénk, ahelyett, hogy azt mondanánk meg, hogy hogyan kell azt megcsinálni. Ennek az előnye akkor mutatkozik meg igazán, ha igen bonyolult problémákat akarunk megoldani, mert szabályokat sokkal könnyebb megfogalmazni, majd később azt visszaolvasva megérteni azt, mint a program kódokat.
- **Adat és logika különválasztása**

Az adatainkat objektumokban tároljuk, a logikát a szabályokban. Ez ugyan felborítja az objektum-orientáltságot, de nézőpont kérdése, hogy ez előny vagy hátrány. Nekünk most ez előny, mivel ilyen formán a szabályokat sokkal könnyebb karbantartani.

- **Sebesség és skálázhatóság**

  A `RETE`, `TREAT`, `LEAPS` algoritmusok és ezek leszármazottai nagyon hatékony mintaillesztést (szabályok illesztése a tényekhez) biztosítanak. Különösen hatékonyak, akkor ha az adathalmazunk egyáltalán nem, vagy csak nagyon kis mértékben változik.

- **A tudás centralizálása**

  Szabályok használatával létrehozunk egy tárházat a tudásnak, ezt nevezzük tudásbázisnak.

- **Magyarázó alrendszer**

  Egy szabály motor lehetőséget biztosít arra, hogy megnézhessük, hogy egy döntés miért is született, vagyis képes megmagyarázni a döntését.

- **Olvashatóbb és érthetőbbek a szabályok**

  A szabályok sokkal könnyebben olvashatóak és érthetőbbek mint a programkód. Ennek főleg, akkor van előnye, ha a szabályok kidolgozásában és karbantartásában részt vesznek nem technikai emberek is.

### 2.2.3 Mikor használjunk egy szabály motort?

Ha rövid választ szeretnénk, akkor a következőt mondhatjuk: "Amikor nincs kielégítő tradicionális programatikus megoldás a probléma megoldására." Ez a rövid válasz igényel némi magyarázatot. Az ok, amiért nem használható a tradicionális megközelítés:

- A problémát túl körülményes lenne a hagyományos úton megoldani. A probléma lehet, hogy nem túl komplex, de nincs más járható út a megoldásra.
- A probléma túl mutat egy hagyományos algoritmikus megoldáson.
- A logika túl gyakran változik. A logika maga lehet egyszerű (bár nem feltétlenül kell annak lennie), de nagyon gyakran változik.
- Ha az üzleti elemzők, szakemberek akik megalkotják a szabályokat nem technikai emberek.

Figyelembe kell vennünk azt, hogy az üzleti szabály motorok technológiája nem egy triviális

technológia, ezért a bevezetése számos csapdát rejthet magában!

Egy modern objektum-orientált (OO) alkalmazásban, amiben a szabály motort használjuk, az üzleti logika fontosabb részeit a szabály motor tartalmazza, ez ellentétben áll az objektum-orientáltság alapelvével, hogy az adatokat és a logikát egy objektum fogja össze. Ez persze nem jelenti azt, hogy ki kell dobnunk az OO praktikákat. Ha már tapasztaltunk olyat, hogy a kódunk rengeteg `if`, `else` és `switch` szerkezetet tartalmaz, és kezd a kód áttekinthetetlen lenni és a karbantartása is nehézkessé válik, akkor érdemes megfontolni a szabályok használatát. Vagy azokban az esetekben, amikor az adott probléma megoldására nincs kielégítő algoritmus vagy valamilyen minta, akkor szintén érdemes megfontolni a szabályok használatát.

A szabályainkat beágyazhatjuk az alkalmazásunkba, de természetesen szolgáltatásként is használhatjuk azokat. Sok esetben az állapottartó szabály komponensek működnek a leghatékonyabban, ha az alkalmazás elválaszthatatlan részét képezik a szabályok. Más esetekben, például szolgáltatások esetében az állapotmentes komponensek a leghatékonyabbak.

## *2.3 A Drools üzleti szabály motor*

### 2.3.1 Bevezetés

A Drools, egy teljes értékű üzleti szabály motor. Rengeteg komponenssel rendelkezik, amik egy részének használata teljesen opcionális. Ezek a főbb részei:

- **Drools üzleti szabály motor magja**
- **A szabályok menedzseléséhez egy Eclipse alapú felület**
  Ez egy letölthető Eclipse Plugin, elsősorban fejlesztők részére készült. Az alkalmazás fejlesztésénél és tesztelésénél vehetjük nagy hasznát.
- **Egy web alapú BRMS (Business Rule Management System)**
  Ez egy olyan eszköz mely segítségével az üzleti szabályainkat tudjuk menedzselni, elsősorban nem technikai emberek számára lehet hasznos.

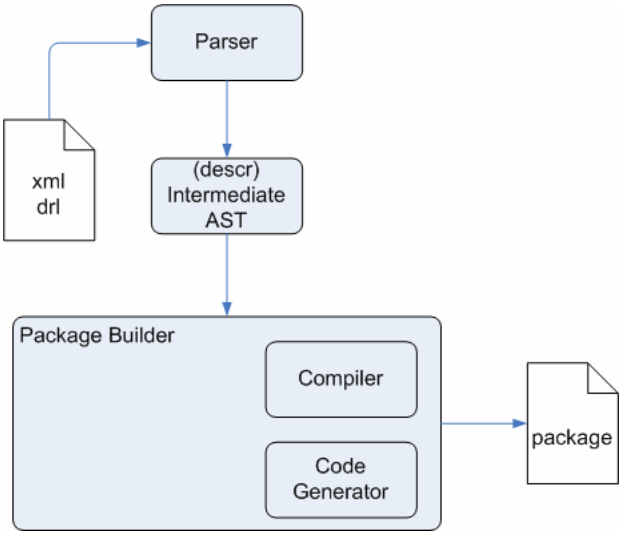
- **Döntési táblák (Decision tables) támogatása**

  A szabályok valamilyen táblázatos formában (például Excel segítségével) történő leírására szolgál. Ebből kerülnek automatikusan generálásra a szabályok.
- **A Java Rule Engine API (JSR94) támogatása**

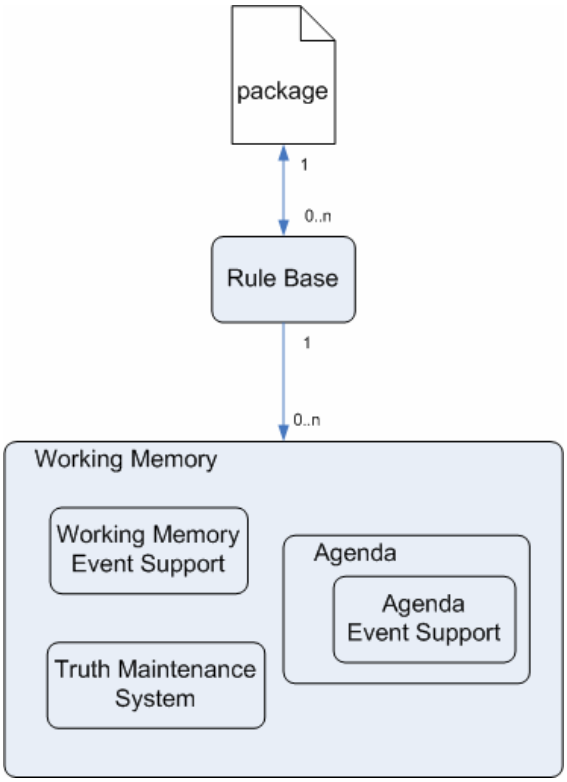
A dolgozatban a Drools magját (architektúráját, lehetőségeit) és a JSR94 API támogatását fogom ismertetni.

### 2.3.2 Az architektúra áttekintése

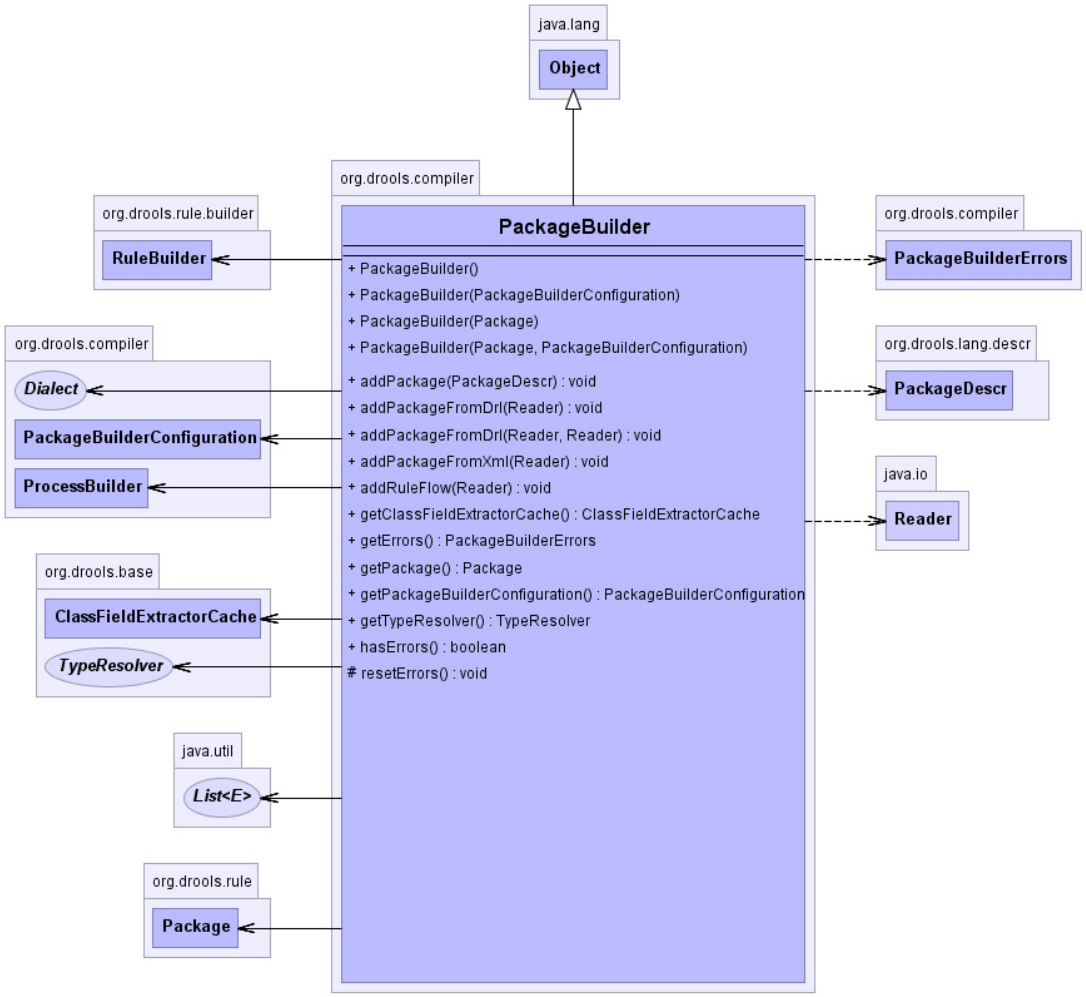
A Drools szabály motort két fontos részre lehet osztani: Azonosítás (Authoring) és Futásidő (Runtime). Az azonosítás folyamat magában foglalja a szabályok elkészítését, amiket DRL vagy XML fájlok tartalmaznak, és amiket egy Antlr 3 nyelvtan definiál. A szintaktikai elemző (parser) ellenőrzi, hogy ezek a szabályok helyesen formázottak-e és generál belőlük egy köztes formátumot a "descr" számára, ahol a "descr" jelenti az AST-t, ami a szabályokat leírja. Azután az AST-t (Abstract Syntax Tree) a "csomag építőnek" (Package Builder) adjuk át, ami csomagokat (Packages) készít belőle. A csomagok elkészítéséhez szükséges kódgenerálás és fordítás feladatát is magára vállalja a Package Builder. A csomag egy szerializált objektum, ami egy vagy több szabályból áll, önmagát is tartalmazhatja és telepíthető.



A szabálybázis (Rule Base) egy futásidejű komponens, ami egy vagy több csomagot tartalmaz. Csomag bármikor hozzáadható és eltávolítható a szabálybázisból. A szabálybázis bármikor példányosíthat egy vagy több munkamemóriát (Working Memory); ha másként nem konfiguráljuk, akkor gyenge referenciákat tart fenn. A munkamemória több komponensből épül fel, ezek a következők: Working Memory Event Support, Truth Maintenance System, Agenda és Agenda Event Support. Objektumok beszúrása aktiválások (Activations) létrehozását eredményezheti. A napirend (Agenda) felelős ezen aktiválások végrehajtásának ütemezéséért.



### 2.3.3 Azonosítás (Authoring)




Négy osztályt használunk az azonosításra: a `DrlParser`, az `XmlParser`, a `ProcessBuilder` és a `PackageBuilder` osztályokat. A két szintaktikai elemző (parser) osztály készíti el a "descr" AST modelleket egy `Reader` példány segítségével. A `ProcessBuilder` osztály segítségével tudunk beolvasni szabály folyamokat (Rule Flow). A `PackageBuilder` osztály biztosít számunkra egy kényelmes API-t, tehát akár többnyire el is felejthetjük a többi osztályt. Ezt, három remek metódus biztosítja számunkra: az `addPackageFromDrl()`, az `addPackageFromXml()`, és az `addRuleFlow()` metódusok. Mind három metódusnak paraméterül kell adni egy-egy `Reader` példányt. A lenti példa megmutatja, hogy hogyan készítsünk el egy csomagot, ami DRL, XML és szabály folyamot is tartalmaz. Itt meg kell jegyeznünk, hogy minden hozzáadott csomag forrásának ugyanabban a névtérben

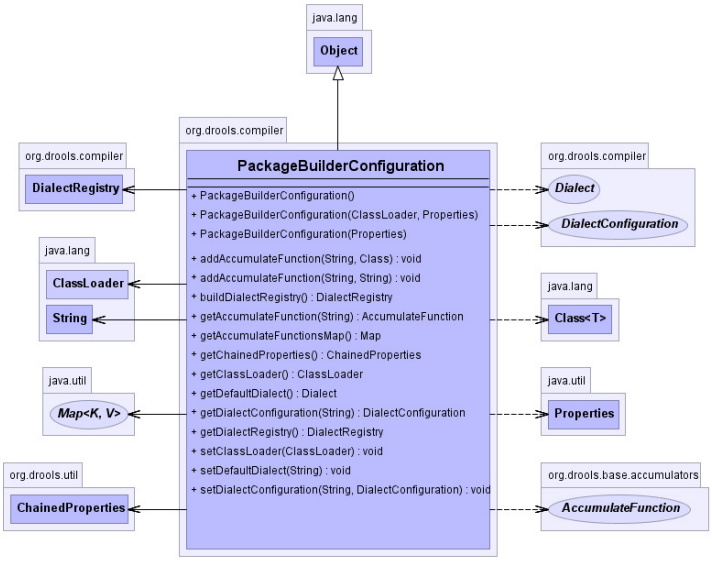
kell lennie amiben az aktuális `PackageBuilder` példány is van!

```
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl(new
InputStreamReader(getClass().getResourceAsStream("package1.drl")));
builder.addPackageFromXml(new
InputStreamReader(getClass().getResourceAsStream("package2.xml")));
builder.addRuleFlow(new
InputStreamReader(getClass().getResourceAsStream("ruleflow.frm")));
Package pkg = builder.getPackage();
```

Nagyon fontos, hogy mielőtt a `PackageBuilder`-t használni kezdenénk, mindig ellenőrizzük, hogy keletkezett-e valamilyen hiba. Ha a szabálybázishoz megpróbálunk hozzáadni egy hibás csomagot, akkor egy `InvalidRulePackage` kivétel dobódik, és csak egy `toString()` ekvivalens hibaüzenet fog a rendelkezésünkre bocsátani. A `PackageBuilder` sokkal részletesebb hibaüzenettel tud szolgálni.

```
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl(new
InputStreamReader(getClass().getResourceAsStream("package1.drl")));
PackageBuilderErrors errors = builder.getErrors();
```

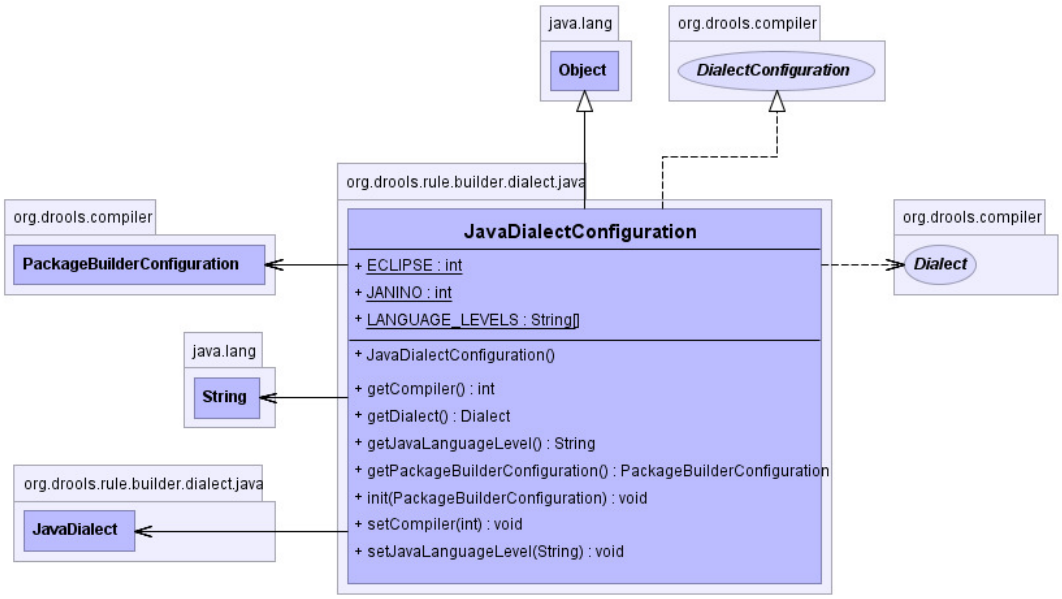
A `PackageBuilder` konfigurálható a `PackageBuilderConfiguration` osztály segítségével.



Ugyan a `PackageBuilder`-nek van alapértelmezett beállítása, de programatikusan felülírhatjuk ezt példányosításkor. A beállítások szívét a `ChainedProperties` osztály képzi, ami számos helyet végignéz `drools.packagebuilder.conf` fájlok után kutatva; amint talált egyet, hozzáadja a mester tulajdonság listához; ez biztosít egy fajta fontossági sorrendet. Fontossági sorrendben, ezek a helyek a következők: System Properties, felhasználó által definiált fájl a System Properties-ben, a felhasználó "home" könyvtára, az aktuális könyvtár, és számos META-INF hely. A drools-compiler.jar-nak az alapértelmezett beállításainka a META-INF könyvtárban kell lennie.

Jelenleg a `PackageBuilderConfiguration` kezeli az Accumulate és Dialect funkciók nyilvántartását és a fő ClassLoader-t.

A Drools cserélhető dialektus rendszerrel van felvértezve, ami lehetővé teszi más nyelveken történő kifejezések és blokkok fordítását és futtatását. Jelenleg két támogatott dialektus van, ezek a Java és az MVEL. Mindegyiknek meg van a saját `DialectConfiguration` implementációja.




A `JavaDialectConfiguration` lehetővé teszi, hogy támogatva legyen a fordító és

különböző nyelvi szintek. A `drools.dialect.java.compiler` tulajdonságot átállítva a packagebuilder.conf fájlban (amit a `ChainedProperties` példány majd megtalál) tudjuk az értékét szabályozni, vagy megtehetjük futásidőben, ahogy az lentebbi példában is látszik.

```
PackageBuilderConfiguration cfg = new PackageBuilderConfiguration();
JavaDialectConfiguration javaConf =
(JavaDialectConfiguration)cfg.getDialectConfiguration("java");
javaConf.setCompiler(JavaDialectConfiguration.JANINO);
```

Ha nincs a classpath-ban az Eclipse JDT Core, akkor mindenek előtt felül kell írni a fordító beállításait, hogy példányosítani tudjuk a `PackageBuilder`-t. Ezt két módon tehetjük meg, az egyik, hogy a `packagebuilder` tulajdonságait tartalmazó fájl szerkesztjük, amit a `ChainedProperties` osztály megtalál, vagy a másik, hogy programatikusan állítjuk be, ahogy a lentebbi példa is szemlélteti.

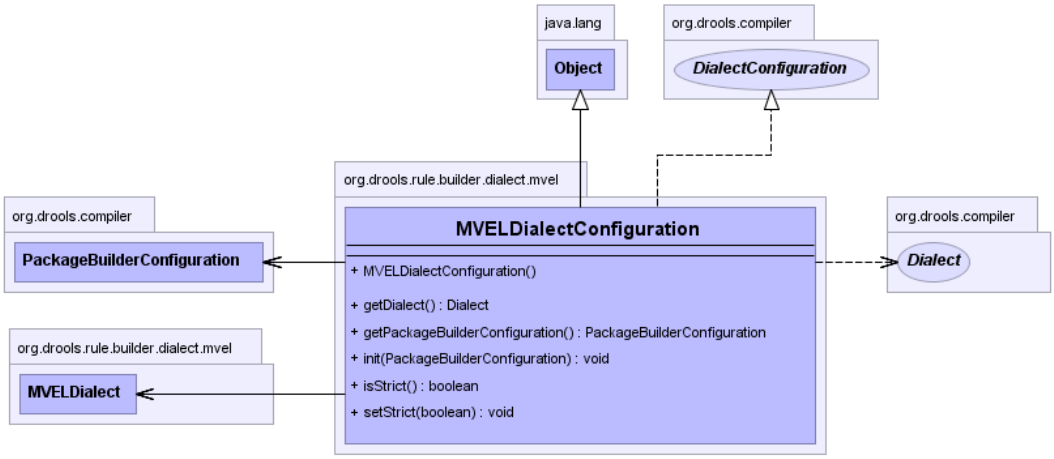
```
Properties properties = new Properties();
properties.setProperty("drools.dialect.java.compiler", "JANINO");
PackageBuilderConfiguration cfg = new
PakcageBuilderConfiguration(properties);
JavaDialectConfiguration javaConf =
(JavaDialectConfiguration)cfg.getDialectConfiguration("java");
assertEquals(JavaDialectConfiguration.JANINO, javaConf.getCompiler());
```

Jelenleg alternatív fordítónak a JANINO-t és az Eclipse JDT adhatjuk meg, különböző forrás szintekkel (1.4 vagy 1.5) és szülő ClassLoader-rel. Az alapértelmezett fordító az Eclipse JDT Core, 1.4-es forrás szinttel, a szülő ClassLoader pedig a `Thread.currentThread().getContextClassLoader()`.

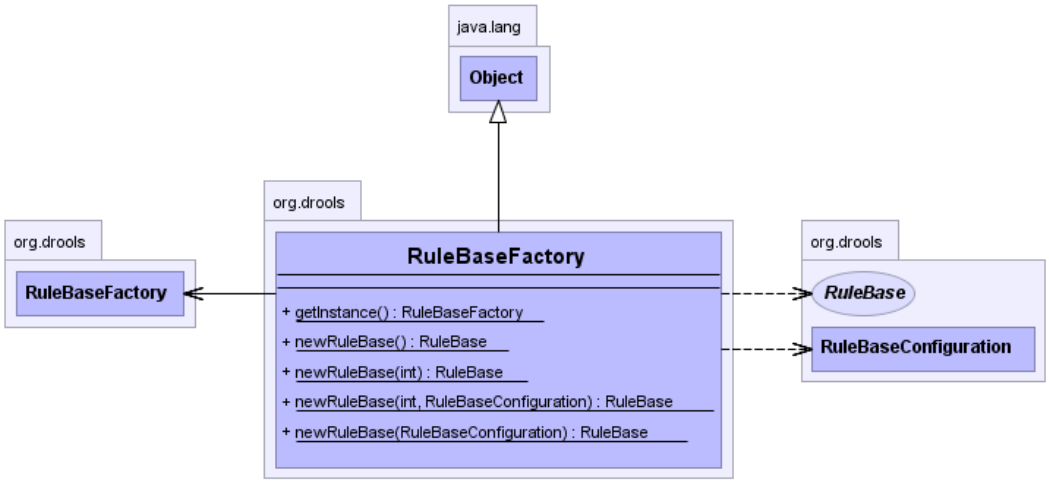
A következő példa megmutatja, hogy hogyan állítsuk be a JANINO fordítót programatikusan.

```
PackageBuilderConfiguration conf = new PackageBuilderConfiguration();
conf.setCompiler(PackageBuilderConfiguration.JANINO);
PackageBuilder builder = new PackageBuilder(conf);
```

Az `MVELDialectConfiguration` sokkal egyszerűbb. Csak annyit tudunk állítani, hogy a szigorú mód be legyen-e kapcsolva vagy sem; alapértelmezettként be van kapcsolva. Ez annyit jelent, hogy minden metódus hívásnak típus biztosnak (type-safe) kell lennie, akár következtetéskor, akár explicit híváskor.



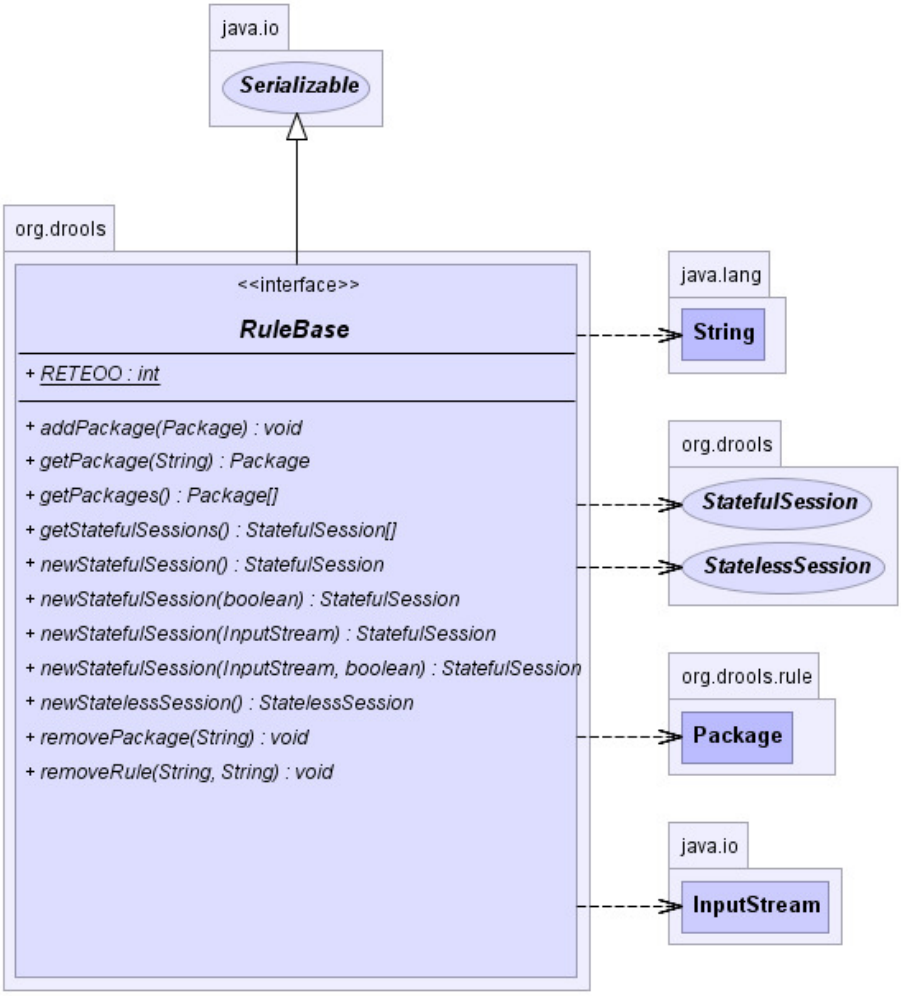

### 2.3.4 Szabálybázis (Rule Base)




A `RuleBase` osztályt a `RuleBaseFactory` segítségével tudjuk példányosítani, alapértelmezett esetben egy ReteOO szabálybázist ad vissza. Csomagokat hozzáadni az `addPackage()` metódus segítségével tudunk. Bármilyen névtérben elhelyezett és több

ugyanazon névtérben lévő csomagot is hozzáadhatunk a szabálybázishoz.

```
RuleBase ruleBase = RuleBaseFactory.newRuleBase();
ruleBase.addPackage(pkg);
```




A szabálybázis egy vagy több, szabályokból álló csomagokat tartalmaz, használatra készen. Ez annyit jelent, hogy a tartalmuk ellenőrizve lett és le lett fordítva. A szabálybázis szerializálható, tehát használható JNDI vagy más hasonló szolgáltatással. Egy szabálybázis az első használatkor le lesz generálva és gyorsítótárazásra kerül. Ez azért előnyes, mert a folyamatos újra generálása nagyon költséges művelet.

A szabálybázis példányai szál-biztosak (thread-safe), tehát egy példány megoszthatunk több szál között is az alkalmazásunkban. A leggyakoribb művelet amit egy szabálybázissal csinálunk, az az új szabály-session létrehozása, ami lehet állapottartó (stateful) vagy állapotmentes (stateless).

A szabálybázis referenciákat tárol minden belőle származtatott állapottartó session-ről, tehát ha egy szabály megváltozik (akár hozzáadunk, akár eltávolítjuk), akkor azt azonnal frissíthetjük a legújabbal, anélkül, hogy szükségtelenül újraindítanánk a session-t. Határozhatunk úgyis, hogy a szabálybázis ne tartson fenn referenciákat, de ilyenkor jobb, ha tisztában vagyunk azzal, hogy változások esetén a session-ök nem kerülnek frissítésre. Állapotmentes session-ökhöz a referenciák nem kerülnek tárolásra.

```
ruleBase.newStatefulSession(); // tárol referenciákat
ruleBase.newStatefulSession(false); // nem tárol referenciákat
```

Csomagok bármikor hozzáadhatók és eltávolíthatók a szabálybázisból. Minden változás ki lesz terjesztve a létező állapottartó session-ökre; de ebben az esetben ne felejtsük el meghívni a `fireAllRules()` metódust, hogy a végrehajtásra jelölt szabályok végrehajtódjanak.

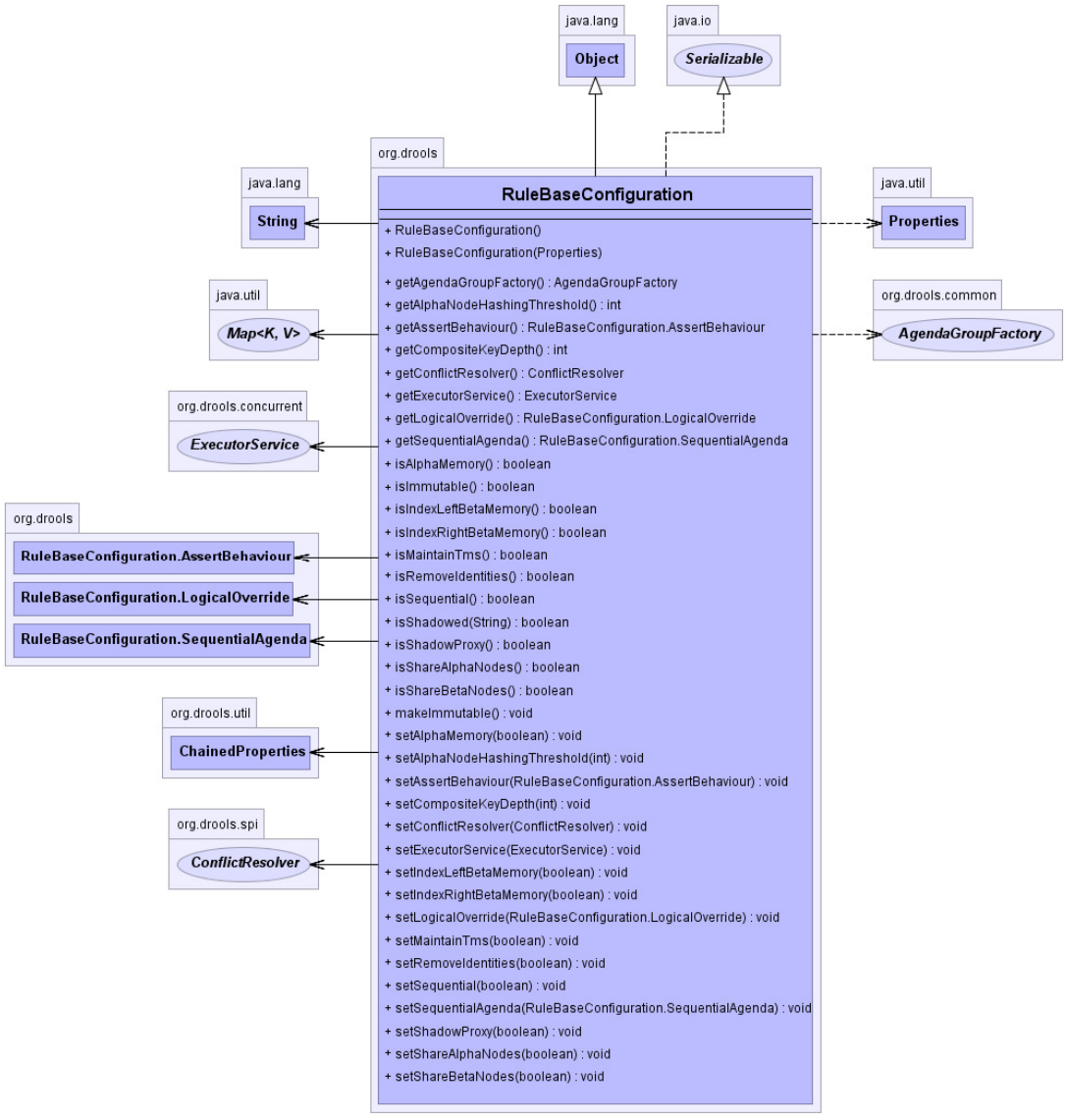
```
ruleBase.addPackage(pkg); // egy csomag hozzáadása
ruleBase.removePackage("org.com.sample"); // csomag eltávolítása (az
összes részével együtt), névtér alapján
ruleBase.removeRule("org.com.sample", "my rule"); // egy szabály
eltávolítása az adott névtérből
```

Rendelkezésünkre áll ugyan egy olyan metódus, amivel a szabályokat egyenként tudjuk eltávolítani, de olyan nincs amivel egyenként tudnánk hozzáadni. Ha mégis ezt szeretnénk elérni, akkor hozzunk létre egy új csomagot, amiben csak egy szabály van.

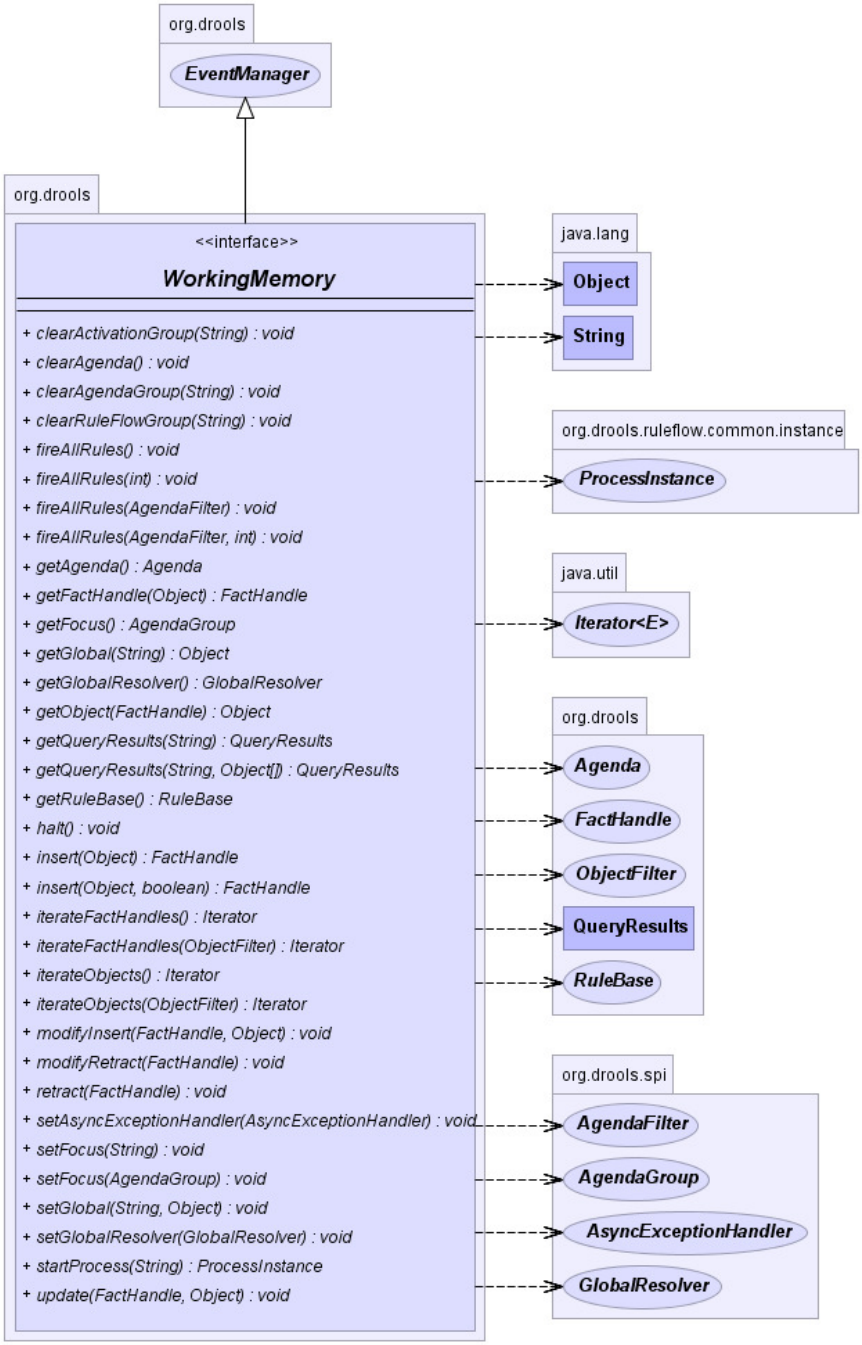
A `RuleBaseConfigurator` osztály használatos arra, hogy meghatározzuk a szabálybázis viselkedését. A `RuleBaseConfigurator` megváltoztathatatlan lesz miután hozzáadtuk a szabálybázishoz. Majdnem minden, a szabály motort érintő optimalizálás ki és

bekapcsolható, és a végrehajtás működése is szintén beállítható. Például a felhasználó számára fontos lehet, a beillesztés módja (azonosság vagy egyenlőség).

```
RuleBaseConfiguration conf = new RuleBaseConfiguration();
conf.setAssertBehaviour(AssertBehaviour.IDENTITY);
conf.setRemoveIdentities(true);
RuleBase ruleBase = RuleBaseFactory.newRuleBase(conf);
```



### 2.3.5 Munkamemória és állapottartó/állapotmentes session-ök




A munkamemória (`WorkingMemory`) az összes beillesztett objektum referenciáját tárolja, mindaddig amíg vissza nem vonjuk azt. A munkamemória mindig állapottartó, lehet rövid- és hosszú életű.

#### 2.3.5.1 Tények (Facts)

A tények az alkalmazás objektumai (tipikusan JavaBean-ek), ezeket illesztjük/dobjuk be a munkamemóriába. Egy tény lehet bármely Java objektum, melyet a szabályok el tudnak érni. A szabály motor nem klónozza a tényeket, hanem a rájuk mutató referenciákat tárolja. Például a `String`, vagy egyéb olyan osztályok amik nem rendelkeznek "getter-ekkel" és "setter-ekkel", nem lehetnek érvényes tények. Tehát csak olyan objektumok lehetnek érvényes tények, melyek teljesítik a JavaBean sztenderd előírásait, vagyis egy objektummal csak a "getter-ein" és "setter-ein" keresztül lehet kommunikálni.

#### 2.3.5.2 Beillesztés (Insertion)

A beillesztés az a művelet, amivel jelezzük a munkamemóriának a tényeket. Például: `WorkingMemory.insert(someObject)`. Amikor beillesztünk egy tényt, a szabály motor megkeresi az illeszkedő szabályokat, stb. Tehát ez azt jelenti, hogy az összes munka a beillesztés alatt történik; bár azt hozzá kell tenni, hogy egyetlen szabály sem kerül végrehajtásra mindaddig, amíg meg nem hívjuk a `fireAllRules()` metódust. Addig nem szabad meghívni a `fireAllRules()` metódust, amíg be nem illesztettünk minden tényt. Az egy gyakori félreértés a felhasználók körében, hogy a munka akkor történik, amikor meghívjuk a `fireAllRules()` metódust!

Amikor beillesztünk egy objektumot, akkor visszakapunk egy `FactHandle`-t. A `FactHandle` egy "jelkép", amivel a beillesztett objektumunkat reprezentáljuk a munkamemórián belül. Ezen keresztül léphetünk interakcióba a munkamemóriával, ha az objektumunkat módosítani vagy vissza akarjuk vonni.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = session.insert(stilton);
```

Ahogy említettem korábban a szabálybázis résznél, a munkamemória két módban tud

működni: egyenlőség (equality) és azonosság (identity). Az azonosság az alapértelmezett.

Az azonosság azt jelenti, hogy ha új példányt adunk a munkamemóriához, akkor új `FactHandle`-t kapunk vissza. Vagyis ha olyan referenciát illesztünk a munkamemóriába ami eddig még nem szerepelt benne, akkor beilleszti azt, egyébként figyelmen kívül hagyja és a korábban beillesztettet adja vissza.

Az egyenlőség az jelenti, hogy a beillesztés csak akkor tér vissza új `FactHandle`-lel, ha nem talál egyenlő (equal) objektumot.

#### *2.3.5.3 Visszavonás (Retraction)*

A visszavonás az a művelet, amikor megmondjuk a munkamemóriának, hogy az adott tény többet nem vesz rész a feldolgozásban. Vagyis a visszavont tényt többet nem illesztjük a szabályokhoz, és azon szabályokat amelyeket aktivált a tény érvénytelenítjük. Meg kell jegyeznünk, hogy lehetnek olyan szabályok melyek egy szabály "nem létezésén" alapulnak, abban az esetben, ha visszavonunk egy szabályt, akkor ezen szabályok aktivizálódhatnak (lásd `not` és `exist` kulcs szavakat). A visszavonást minden esetben egy `FactHandle` objektumon (amit a beillesztéskor kaptunk vissza) keresztül tudjuk megtenni.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = session.insert(stilton);
...
session.retract(stiltonHandle);
```

#### *2.3.5.4 Frissítés (Update)*

A szabály motort minden esetben értesíteni kell a módosított tényekről, hogy azokat újra fel tudja dolgozni. A módosítás belül egy visszavonás és beillesztés.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = workingMemory.insert(stilton);
...
stilton.setPrice(100);
workingMemory.update(stiltonHandle, stilton);
```

#### *2.3.5.5 Globálisok (Globals)*

A globálisok névvel ellátott objektumok, amiket áttudunk adni a szabály motornak, anélkül, hogy be kéne őket illesztenünk. Ezeket gyakran statikus információk tárolására használjuk vagy ezek szolgáltatások is lehetnek, amiket a szabályok jobb oldalán használhatunk. Használhatjuk a globálisokat arra is, hogy a szabály motor objektumokat helyezzen el benne. Ha egy globálist egy szabály bal oldalán használunk, győződjünk meg arról, hogy az megváltozhatatlan. Mielőtt használnánk egy globálist, deklarálnunk kell azt.

```
global java.util.List list
```

Ha ez a deklaráció meg van, akkor lehetőségünk van, hogy bármely session `setGlobal()` metódusát meghívva, hozzáadjuk a globálisunkat.

```
List list = new ArrayList();
session.setGlobal("list", list);
```

Ha egy szabály kiértékelése közben olyan globálist talál, amit nem adtunk hozzá, akkor a rendszer dob egy `NullPointerException`-t.

#### *2.3.5.6 A tulajdonság figyelő interfész*

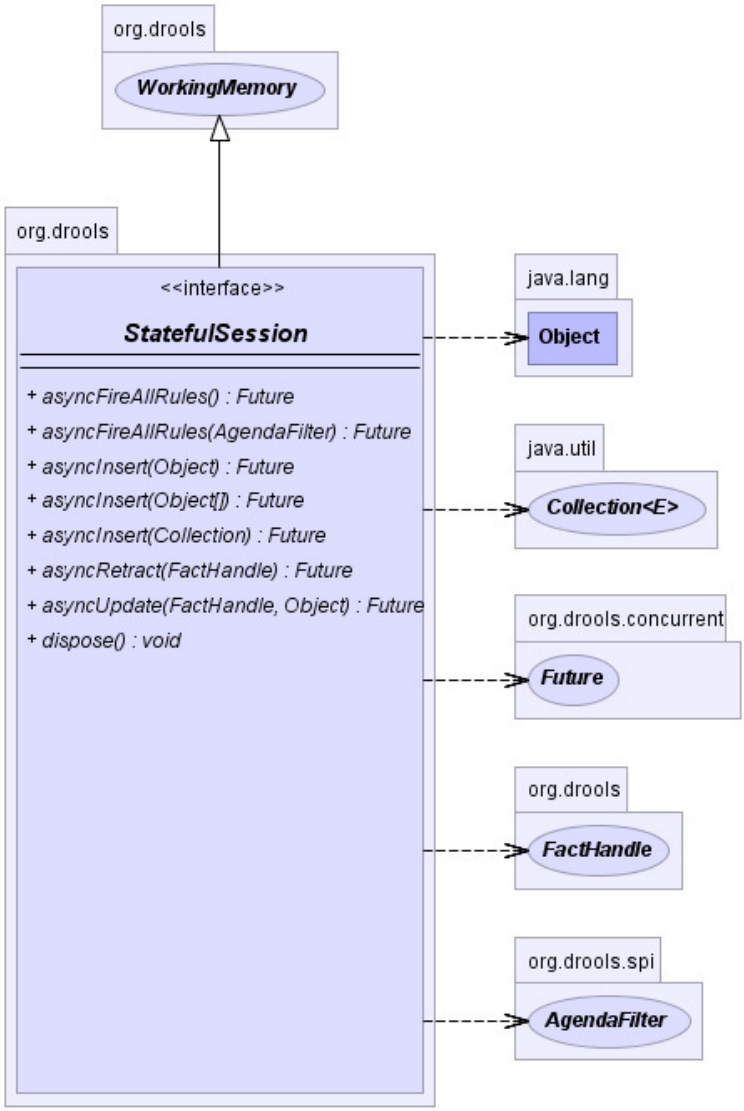
Ha az objektumok, amiket használunk azok "szabványos" JavaBean-nek, akkor lehetőségünk van egy tulajdonság figyelőt hozzáadni, és megmondani a szabály motornak, hogy az adott Bean rendelkezik ezzel a tulajdonság figyelővel vagyis dinamikus. Ez azért jó, mert nekünk nem kell azzal foglalkozni, hogy minden alkalommal értesítsük a szabály motort, hogy ha egy adott tény megváltozott.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = workingMemory.insert(stilton, true); // Ezzel
mondjuk, meg, hogy egy dinamikus tényről van szó.
```

Ahhoz, hogy egy Bean-t dinamikussá tegyünk, hozzá kell adni egy adattagot, a `PropertyChangeSupport`-ot és a hozzá való add és remove metódusokat. Ezen kívül még az egyik "setter"-nek értesíteni kell a `PropertyChangeSupport` adattagot a változtatásról.

```
private final PropertyChangeSupport changes = new
PropertyChangeSupport(this);
...
public void addPropertyChangeListener(final PropertyChangeListener l) {
 this.changes.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(final PropertyChangeListener l)
{
 this.changes.removePropertyChangeListener(l);
}
...
public void setState(final String newState) {
 String oldState = this.state;
 this.state = newState;
 this.changes.firePropertyChange("state", oldState, newState);
}
```

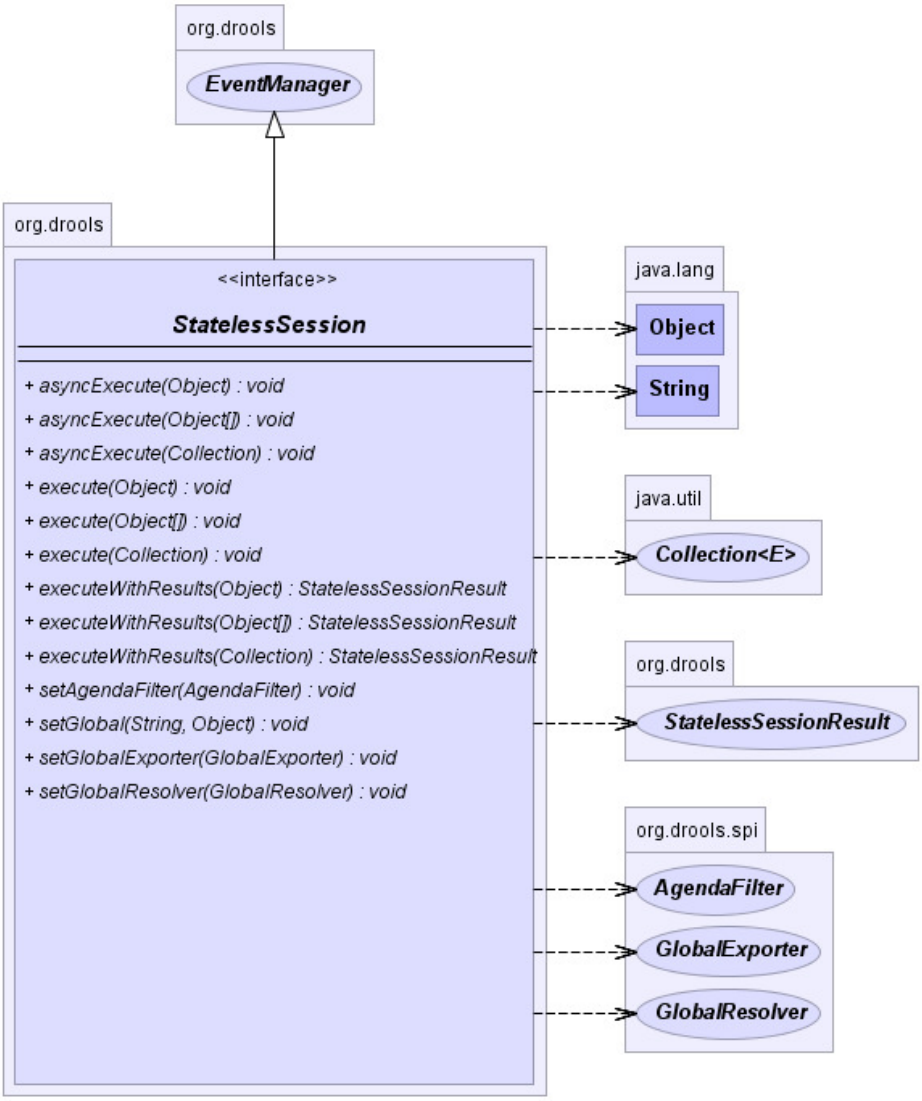
### 2.3.6 Állapottartó session (Stateful session)




Az állapottartó session egy olyan munkamemóriát reprezentál, ami a hívások között megtartja az állapotát. A `StatefulSession` interfész a `WorkingMemory` interfész kiterjesztése. Ha megfigyeljük, akkor hozzáad async metódusokat és a `dispose()` metódust. A `RuleBase` tárolja a `StatefulSession`-re vonatkozó referenciát, ezért tudjuk majd frissíteni, ha új szabályokat adunk hozzá. A `dispose()` metódust meg kell hívnunk, hogy a megfelelő referenciát elengedje a `RuleBase`. Ha nem tesszük ezt meg, akkor a memória el kezdhet "szivárogni" (memory leak)!

```
StatefulSession session = ruleBase.newStatefulSession();
```

### 2.3.7 Állapotmentes session (Stateless session)




A `StatelessSession` körülfogja a `WorkingMemory`-t, ahelyett, hogy kiterjesztené azt, mivel a célja, hogy döntés szolgáltatásokat valósítson meg.

```
StatelessSession session = ruleBase.newStatelessSession();
session.execute(new Cheese("cheddar"));
```

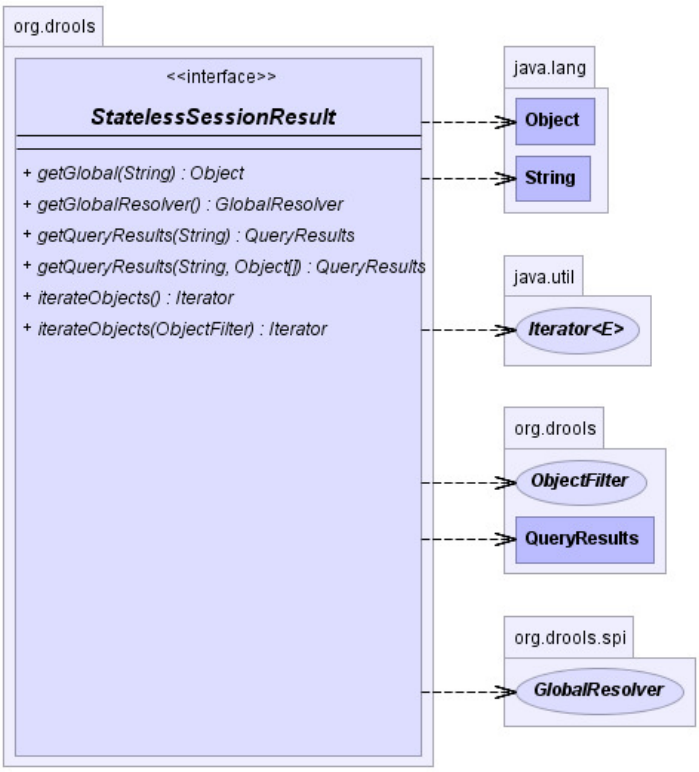
Az API redukálva lett, az adott problémakörre, ezáltal egyszerűbb lett, ami segít a döntési szolgáltatások megvalósításában. A `RuleBase` nem tárol `StatelessSession`-re vonatkozó referenciát, ezért nem kell használnunk a `dispose()` metódust. A `StatelessSession`-nek egy metódusa van, az `execute()`, ennek három verziója létezik, az egyik egy `Object`-et vár, a második egy `Object` tömböt, a harmadik pedig egy `Collection`-t, tehát nincs `insert()` vagy `fireAllRules()` metódus. Az `execute()` metódus végig iterál a paraméterként kapott objektumon, és a végén meghívja a `fireAllRules()` metódust, majd ezután befejeződik a session. Ha szükség van arra, hogy valamilyen értékkel térjen vissza, akkor használjuk az `executeWithResults()` metódust, ami visszatér egy `StatelessSessionResult`-tal. Mivel nem mindig van szükségünk a visszatérési értékre, ezért ennek a metódusnak a használata opcionális.

A `setAgendaFilter()`, a `setGlobal()` és a `setGlobalResolver()` metódusok által beállított értékek globálisak lesznek, vagyis az összes session-re vonatkozni fognak, vagyis minden `execute()` híváskor érvényesek lesznek például a `setAgendFilter()` metódus által beállított értékek.

Az állapotmentes session-ök jelenleg nem támogatják a `PropertyChangeListener`-eket.

Az `execute()` metódus async verziója támogatott, azonban ne felejtsük el módosítani az `ExecutorService` implementációját, hogy megfeleljen olyan speciális követelményeknek, ami például egy JEE környezetben található.

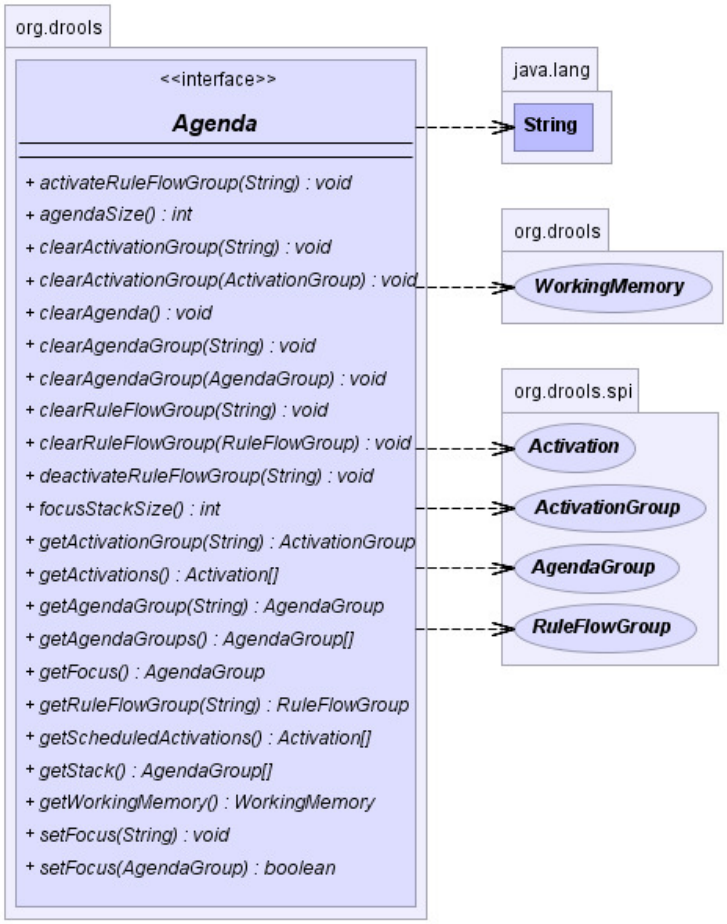
Az állapotmentes session-ök támogatják a szekvenciális módot (sequential mode). Ez egy speciális mód, ami arra van optimalizálva, hogy kevesebb memóriát használjon és gyorsabb legyen a végrehajtás.




Az `executeWithResults()` metódus minimális felületet biztosít arra, hogy megvizsgáljuk a session adatait. A beillesztett objektumokon végig tudunk iterálni, tudunk különböző lekérdezéseket indítani és le tudjuk kérdezni a globális beállításokat. Ha azonban a `StatelessSessionResult`-ot szerializáljuk, akkor elveszítjük a munkamemória és a szabálybázis referenciáit, tehát nem tudunk majd lekérdezéseket indítani. Viszont a globális beállításokat le tudjuk kérdezni, és a beillesztet objektumon végig tudunk iterálni. Ahhoz, hogy a globális beállításokat megtudjuk nézni, exportálnunk kell azokat; az exportálási stratégiát a `setGlobalExporter()` metódus segítségével tudjuk beállítani. Két implementáció áll a felhasználó rendelkezésére, az egyik a `CopyIdentifiersGlobalExporter`, a másik a `ReferenceOriginalGlobalExporter`. Természetesen a felhasználónak lehetősége van, hogy implementálja a saját megoldását.

```
StatelessSession session = ruleBase.newStatelessSession();
session.setGlobalExporter(new CopyIdentifiersGlobalExporter(new String[]{
"list" }));
StatelessSessionResult result = session.executeWithResults(new
Cheese("stilton"));
List list = (List)result.getGlobal("list");
```

### 2.3.8 Napirend (Agenda)




A napirend egy RETE jellemző. A munkamemória műveletek alatt szabályok válhatnak alkalmassá a végrehajtásra. Egyetlen munkamemória művelet alatt, akár több is. Ha egy szabály alkalmassá válik a végrehajtásra, akkor létrejön egy aktivizálás, ami hivatkozik a szabályra és a kiváltó tényekre, majd elhelyezésre kerül a napirendben. A napirend határozza meg a szabályok végrehajtási sorrendjét, a konfliktus feloldó stratégia alapján.

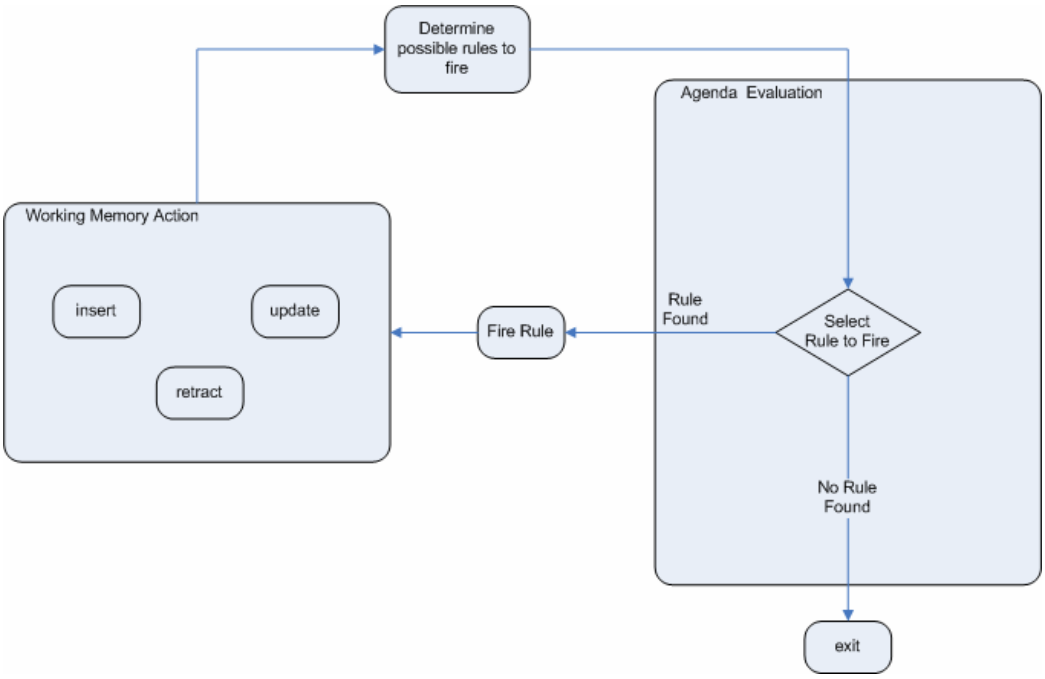
A motor két fázisú módban működik, amik a következők:

- **Munkamemória műveletek**

  Ez az a fázis, ahol a legtöbb munka történik. Ez lehet a következtetés, vagy akár maga a fő Java programunk folyamatai is. Amint befejeződött a következtetés a Java programunk meghívja a `fireAllRules()` metódust a motor átkapcsol a napirend kiértékelés fázisba.

- **Napirend kiértékelés**

  Megpróbál egy szabályt kiválasztani végrehajtásra, ha az adott szabályt nem találja, akkor kilép, egyébként megpróbálja végrehajtani azt. Majd visszavált az előző fázisra.




A fent felvázolt folyamat addig tart, amíg a napirend ki nem ürül. Ha ez bekövetkezik, akkor a vezérlés visszakerül a hívó alkalmazásra. Azt azonban jó, ha tudjuk, hogy a munkamemória fázisban egyetlen szabály sem hajtódik végre, csak végrehajtásra lesznek jelölve.

#### *2.3.8.1 Konfliktus feloldás (Conflict resolution)*

A konfliktus feloldás egy fontos művelet, mivel a napirendben több szabály is lehet.

Tehát valamilyen módon el kell dönteni, hogy a szabályok milyen sorrendben aktivizálódhatnak. Azért is fontos ez, mert a szabályok egymásra is hathatnak. Például: "A" szabály aktivizálása "B" szabály eltávolítását jelenti a napirendből.

Az alapértelmezett konfliktus feloldás a Drools-ban: a priorizálás (salience) és a LIFO (Last In, First Out).

A priorizálásnál a felhasználónak lehetősége van egy értéket rendelni a szabályokhoz, minél nagyobb ez a szám, annál előrébb kerül a végrehajtáskor.
LIFO esetén a munkamemória művelet számláló értéke kerül hozzárendelésre az adott szabályhoz. Ha több szabály is keletkezik ugyanabban az műveletben, akkor mindegyikhez ugyanaz a szám rendelődik - ezek végrehajtási sorrendje nem definiált.

A szabályok megalkotásánál nem jó ötlet figyelembe venni azt, hogy mi lesz a végrehajtási sorrend. Próbáljunk meg úgy dolgozni, hogy nem "aggódunk" a végrehajtási sorrend miatt.

Ha egyedi konfliktus feloldó stratégiát akarunk megadni, akkor a `RuleBaseConfiguration` osztály `setConflictResolver()` metódusát vagy `drools.conflictResolver` tulajdonságot használjuk.

#### *2.3.8.2 Napirend csoportok (Agenda groups)*

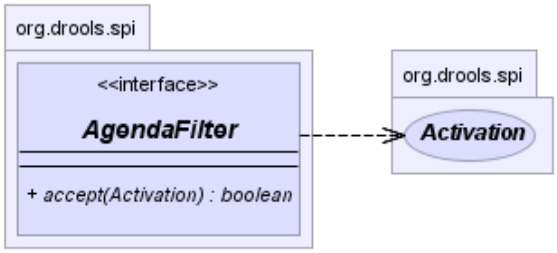
A napirend csoportok arra szolgálnak, hogy valamilyen módon csoportosítani tudjuk a végrehajtásra jelölt szabályainkat. Egy időben egyszerre csak egy csoport lehet aktív, ami azt jelenti, hogy az ebben a csoportban lévő szabályok hatásai érvényesülhetnek. Lehetnek olyan szabályaink melyek "auto fókuszosak", ez annyit jelent, hogy amennyiben az adott szabály igazzá válik, a hozzátartozó napirend csoport aktivizálódik.

A napirend csoportok nagyon jól jönnek, ha valamilyen "folyamot" akarunk létrehozni a csoportosított szabályok között. A csoportok közötti tudunk váltogatni. Ezt megtehetjük a szabály motorból vagy API hívással. Ha a szabályainknak szükséges a több fázisú vagy

szekvenciális feldolgozás, akkor érdemes megfontolni a napirend csoportok használatát.

Minden alkalommal, amikor a `setFocus()` metódus meghívódik, az aktuális napirend csoport egy verem tetejére kerül; amint az aktív csoport kiürül, az eltávolításra kerül a verem tetejéről. Ezután a következő csoport, ami a verem tetején van, kerül feldolgozásra. Egy napirend csoport a veremben több helyen is előfordulhat. Létezik egy alapértelmezett csoport, amit "MAIN"-nek hívnak. Ide kerül minden olyan szabály ami nem tartozik egyetlen csoporthoz sem. Természetesen ez az első csoport a veremben, és ez lesz legelőször aktív.

#### *2.3.8.3 Napirend szűrők (Agenda filters)*




A szűrők opcionális implementációi az `AgendaFilter` interfésznek. Arra használatosak ezek a szűrők, hogy a végrehajtásra jelölt szabályok közül egyeseket engedélyezzen vagy tiltson. Hogy mik kerülnek tiltásra illetve engedélyezésre az implementáció függő.

A Drools a következő implementációkat biztosítja a felhasználó részére:

- `RuleNameEndWithAgendaFilter`
- `RuleNameEqualsAgendaFilter`
- `RuleNameStartsWithAgendaFilter`
- `RuleNameMatchesAgendaFilter`

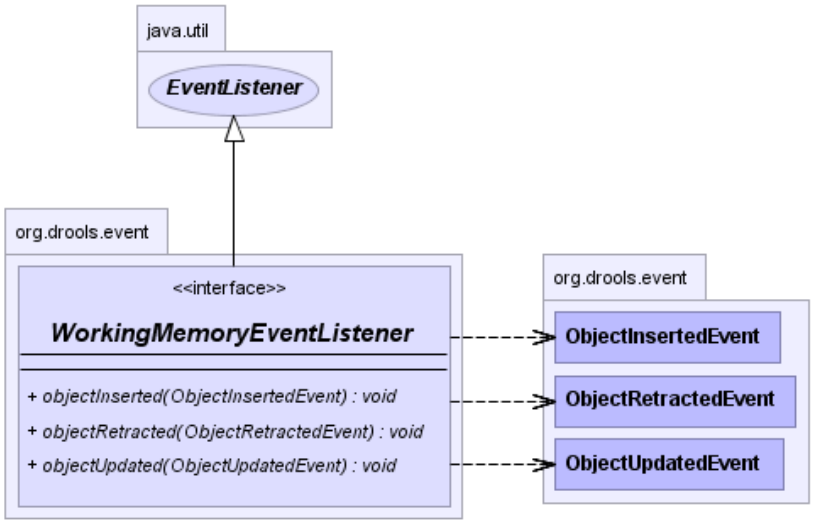
Ha használni szeretnénk egy szűrőt, akkor azt a `fireAllRules()` metódus híváskor tehetjük meg.

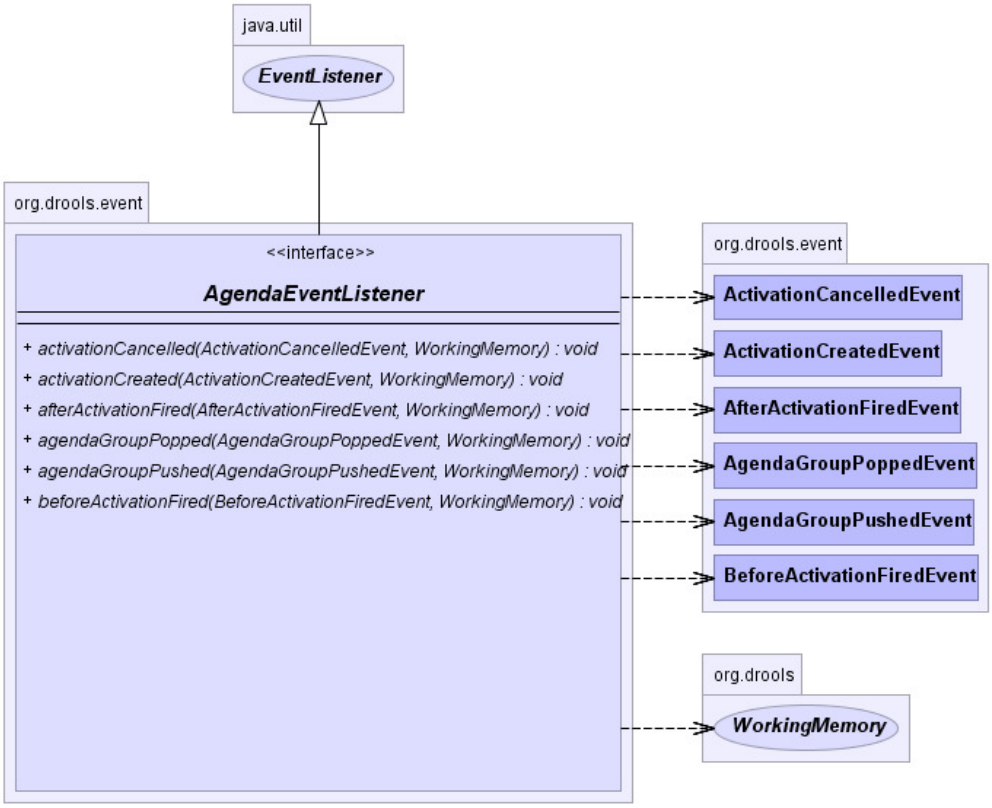
```
workingMemory.fireAllRules(new RuleNameEndsWithAgendaFilter("Test"));
```

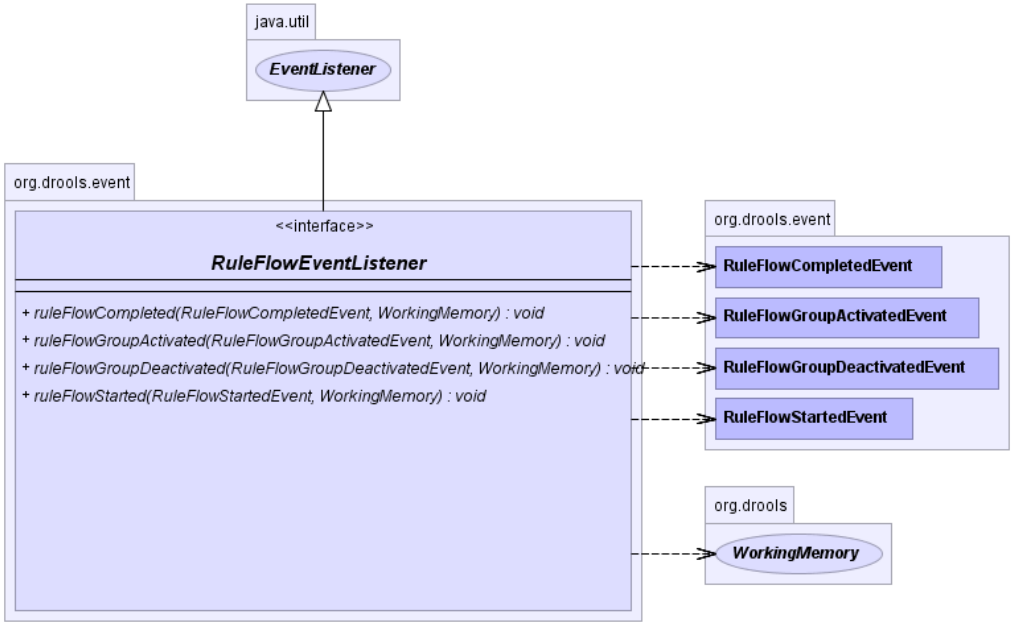
### 2.3.9 Esemény modell (Event model)

Az `org.drools.event` csomagban lévő interfészek és osztályok olyan lehetőségeket biztosítanak nekünk, hogy értesüljünk bizonyos tevékenységek végrehajtásáról. Ilyenek lehetnék például egy szabály végrehajtása, objektum beillesztése a munkamemóriába, stb. Ez lehetővé teszi nekünk, hogy például a naplózó részt külön válasszuk az alkalmazás kódjától.

Három különböző eseményfigyelő áll a rendelkezésünkre: `WorkingMemoryEventListener` (a munkamemória beli eseményekért felelős), `AgendaEventListener` (a napirend eseményeiért felelős), `RuleFlowEventListener` (a szabály-folyam eseményekért felelős).

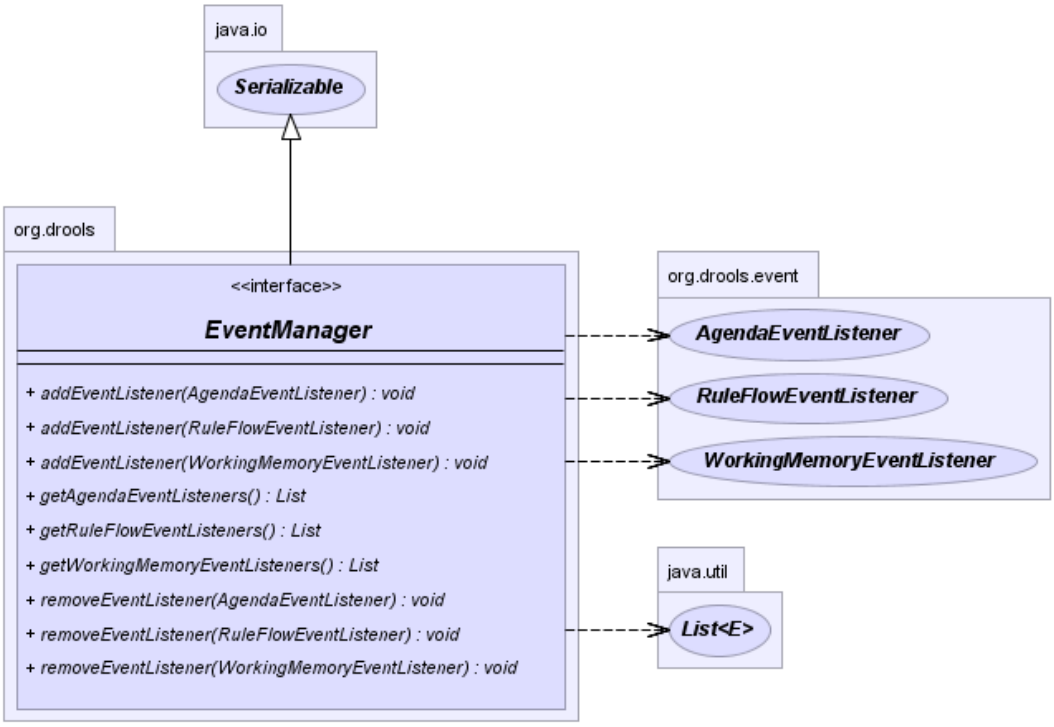





Mind az állapottartó és mind az állapotmentes session implementálja az `EventManager`

interfészt, ami lehetővé teszi, hogy eseményfigyelőket adjunk hozzájuk.




Minden eseményfigyelőnek meg van az alapértelmezett implementációja, ami ugyan minden metódust megvalósít, de azok nem csinálnak semmit. Ezek gyakorlatilag adapter osztályok (`DefaultAgendaEventListener`, `DefaultWorkingMemoryEventListener`, `DefaultRuleFlowEventListener`). Ez kényelmes, mert csak azokat metódusokat kell implementálnunk, melyekre szükségünk van.

A lenti példa azt mutatja, meg, hogy hogyan terjesszük ki és adjuk hozzá a session-höz a `DefaultAgendaEventListener` adapter osztályt.

```
session.addEventListener(new DefaultEventListener() {
 public void afterActivationFired(AfterActivationFiredEvent event) {
  super.afterActivationFired(event);
  System.out.println(event);
 }
});
```

A Drools biztosít számunkra további három osztályt (`DebugWorkingMemoryEventListener`,

`DebugAgendaEventListener`, `DebugRuleFlowEventListener`), amivel a hibakeresést könnyíti meg.

```
session.addEventListener(new DebugWorkingMemoryEventListener());
```

## 2.3.10 A Java szabály motor API-ja (Java Rule Engine API)

### *2.3.10.1 Bevezetés*

A Drools biztosít számunkra egy Java szabály motor API implementációt (JSR94). A JSR94-es ajánlás egy egységes interfészt definiál a szabály motorok szolgáltatásainak elérésére, de magáról a szabályokat leíró nyelvről nem mond semmit.

A következő lehetőségeket biztosítja számunkra:

- Szabályok regisztrálása.
- Szabályok elemzése.
- Szabályok metaadatainak vizsgálata.
- Szabályok végrehajtása.
- Eredmények kinyerése.
- Az eredmények szűrése.

Amikről nem mond semmit a JSR94:

- A szabály motorról magáról.
- A szabályok végrehajtási sorrendjéről.
- A szabályokat leíró nyelvről.
- A JEE telepítési mechanizmusról.

#### 2.3.10.2 A JSR94 architektúrája

Az API két fő részből áll:

- **A szabály adminisztrátor API:**

  Ez az API a `javax.rules.admin` csomagban van definiálva. Az itt található osztályok segítségével tudjuk betölteni a szabályokat és a hozzájuk rendelt műveleteket is, mint végrehajtási halmazokat. Egy szabály végrehajtási halmaz, szabályok gyűjteménye. Szabályokat többféle külső forráson keresztül is betölthetjük. A csomag biztosít számunkra metódusokat, hogy kezeljük a szabályokat, például regisztráljuk őket. Továbbá itt van lehetőségünk a jogosultságok beállítására, hogy szabályozzuk a hozzáférést.

- **A futásidejű kliens API:**

  Az API a `javax.rules` csomagban található. Ez a csomag biztosítja azokat a lehetőségeket számunkra, hogy futtassuk a szabályokat és kinyerjük az eredményeket. Csak azok a szabályok lesznek elérhetőek itt, melyeket az adminisztrációs API segítségével regisztráltunk. Ez teszi lehetővé a kliensek számára, hogy szert tegyenek egy session-re, és ezen belül szabályokat hajtsanak végre.

#### 2.3.10.3 Hogyan használjuk

A JSR94 használatához két dolog szükséges, az egyik az adminisztratív API, amivel felépíthetjük és regisztrálhatjuk a szabály végrehajtási halmazokat (`RuleExecutionSet`), a másik a futásidejű eszközök, amivel szabályozhatjuk a session-ök működését.

#### 2.3.10.4 Szabály végrehajtási halmazok építése és regisztrálása

A `RuleServiceProviderManager` osztály kezeli a `RuleServiceProvider` példányok regisztrálását és kinyerését. A Drools `RuleServiceProvider` implementációja egy statikus blokk segítségével automatikusan regisztrálódik, amikor az osztály betöltésre kerül a `Class.forName()` metódust használva.

```
// RuleServiceProviderImpl a "http://drools.org/" néven kerül
regisztrálásra
Class.forName("org.drools.jsr94.rules.RuleServiceProviderImpl");
// A RuleServiceProvider kinyerése
RuleServiceProvider ruleServiceProvider =
RuleServiceProviderManager.getRuleServiceProvider("http://drools.org/");
```

A `RuleServiceProvider` osztály biztosít hozzáférést a RuleRuntime és RuleAdministration API-khoz. A RuleAdministration API egy adminisztratív API, aminek segítségével kezelhetjük, regisztrálhatjuk a szabály végrehajtási halmazainkat, majd ezeket a `RuleRuntime` osztályon keresztül kaphatjuk vissza.

Mielőtt regisztrálni tudnánk egy szabály végrehajtási halmazt, létre kell hoznunk azt; a `RuleAdministrator` osztály biztosít egy factory metódust, ami vissza ad egy üres `LocalRuleExecutionSetProvider`-t vagy egy `RuleExecutionSetProvider`-t. A `LocalRuleExecutionSetProvider`-t arra kell használnunk, hogy helyi, nem szerializálható forrásokból töltsünk be szabály végrehajtási halmazokat. A `RuleExecutionSetProvider`-t tudjuk használni a szerializálható forrásokon keresztül történő betöltésekre. A `ruleAdministrator.getLocalRuleExecutionSetProvider(null);` és a `ruleAdministrator.getRuleExecutionSetProvider(null);` metódusokat kell használnunk, és mindkettőnek null-t kell adni paraméternek.

```
// A RuleAdministration "kinyerése".
RuleAdministration ruleAdministrator =
ruleServiceProvider.getRuleAdministrator();
LocalRuleExecutionSetProvider ruleExecutionSetProvider =
ruleAdministrator.getLocalRuleExecutionSetProvider(null);

// Egy Reader létrehozása egy drl fájlhoz.
URL drlUrl = new URL("http://mydomain.org/sources/myrules.drl");
Reader drlReader = new InputStreamReader(drlUrl.openStream());

// Egy szabály végrehajtási halmaz létrehozása drl fájlból.
RuleExecutionSet ruleExecutionSet =
ruleExecutionSetProvider.createRuleExecutionSet(drlReader, null);
```

Ahogy a példában is látszik a `ruleExecutionSetProvider.createRuleExecutionSet(reader, null);` metódusnak második paraméterként null-t adtunk át. Ezzel a második paraméterrel (kulcs-érték párokat

kell átadni) tudjuk konfigurálni a bejövő forrást. Ha `null` ez az érték, akkor drl-ként (ez az alapértelmezés) kerül feldolgozásra a forrás. Kulcsként átadhatjuk a "source"-t, amihez a következő értékek tartozhatnak: "drl" vagy "xml". Értelemszerűen a megfelelő formátumot definiáljuk ezzel, vagyis a forrás ennek megfelelően kerül feldolgozásra. A másik megengedett kulcs a "dsl", ennek az érték lehet egy `Reader` vagy egy `String` (a dsl tartalma).

```
// A RuleAdministration "kinyerése".
RuleAdministration ruleAdministrator =
ruleServiceProvider.getRuleAdministrator();
LocalRuleExecutionSetProvider ruleExecutionSetProvider =
ruleAdministrator.getLocalRuleExecutionSetProvider(null);

// Egy Reader létrehozása a drl fájlhoz.
URL drlUrl = new URL("http://mydomain.org/sources/myrules.drl");
Reader drlReader = new InputStreamReader(drlUrl.openStream());

// Egy Reader létrehozása a dsl fájlhoz és tulajdonságok elhelyezése egy
map-ben.
URL dslUrl = new URL("http://mydomain.org/sources/myrules.dsl");
Reader dslReader = new InputStreamReader(dslUrl.openStream());
Map properties = new HashMap();
properties.put("source", "drl");
properties.put("dsl", "dslReader");

// Egy szabály végrehajtási halmaz létrehozása drl és dsl fájlból.
RuleExecutionSet ruleExecutionSet =
ruleExecutionSetProvider.createRuleExecutionSet(reader, properties);
```

Amikor regisztrálunk egy szabály végrehajtási halmazt, meg kell adnunk a nevét, amit a kinyerésénél fogunk használni. Lehetőségünk van átadni egy tulajdonság map-et, de ez jelenleg nem használja a Drools, ezért adjunk át egy `null`-t.

```
// A RuleAdministrator segítségével egy RuleExecutionSet regisztrálása.
String uri = ruleExecutionSet.getName();
ruleAdministrator.registerRuleExecutionSet(uri, ruleExecutionSet, null);
```

#### *2.3.10.5 Állapottartó és állapotmentes session-ök használata*

A `RuleRuntime` osztályt a `RuleServiceProvider`-től kaphatjuk meg, hogy létrehozzunk állapottartó vagy állapotmentes session-öket.

```
RuleRuntime ruleRuntime = ruleServiceProvider.getRuleRuntime();
```

Egy szabály session létrehozásánál két, a RuleRuntime osztályban található konstanst tudunk felhasználni. Ezek a `RuleRuntime.STATEFUL_SESSION_TYPE` és a `RuleRuntime.STATELESS_SESSION_TYPE`. Ezeket a `createRuleSession()` metódusnak tudjuk átadni paraméterül, ezzel meghatározva, hogy milyen session-t kívánunk létrehozni. Ezenkívül még két másik paramétert kell átadnunk, az egyik URI, a másik pedig egy tulajdonság map (ez lehet `null` is).

```
(StatefulRuleSession)session = ruleRuntime.createRuleSession(uri, null,
RuleRuntime.STATEFUL_SESSION_TYPE);
session.addObject(new PurchaseOrder("lots of cheese"));
session.executeRules();
```

A `StatelessRuleSession`-nek rendkívül egyszerű API-ja van, egyetlen metódust tudunk hívni, ez pedig az `executeRules(List list)`, ahol paraméterként egy objektum listát tudunk átadni. Az eredményt ennek a metódusnak a visszatérési értékeként kapjuk meg.

```
(StatelessRuleSession)session = ruleRuntim.createRuleSession(uri, null,
RuleRuntime.STATELESS_SESSION_TYPE);
List list = new ArrayList();
list.add(new PurchaseOrder("even more cheese"));

List results = new ArrayList();
results = session.executeRules(list);
```

#### *2.3.10.6 Globálisok*

A JSR94-gyel lehetőségünk van arra, hogy használjunk globálisokat. Mégpedig úgy, hogy a `RuleSession` factory metódus tulajdonság map paraméterét használjuk. A globálisokat definiálnunk kell először a drl vagy xml fájlunkban, egyébként egy kivétel fog dobódni. A tulajdonság map-ben a kulcsnak mindig meg kell egyezni a drl vagy xml fájlban deklarált azonosítóval, az értéknek pedig magának az objektumnak kell lennie, amit használni

kívánunk.

```
java.util.List globalList = new java.util.ArrayList();
java.util.Map map = new java.util.HashMap();
map.put("list", globalList);

StatelessRuleSession srs =
(StatelessRuleSession)runtime.createRuleSession("SistersRules", map,
RuleRuntime.STATELESS_SESSION_TYPE);
...
// Az executeRules() meghívása utána így tudjuk visszakapni a listánkat.
List list = (java.util.List)map.get("list");
```

```
package SistersRules;
import org.drools.jsr94.rules.Person;
global java.util.List list

rule FindSisters
 when
  $person1 : Person($name1 : name)
  $person2 : Person($name2 : name)
  eval($person1.hasSister($person2))
 then
  list.add($person1.getName() + " and " + $person2.getName() +" are
sisters");
  assert($person1.getName() + " and " + $person2.getName() +" are
sisters");
end
```

## 2.3.11 A szabály nyelv

### *2.3.11.1 Áttekintés*

A Drools 4.0-ás verziója rendelkezik egy natív szabály nyelvvel, ami szöveges, de nem XML alapú. Ez egy nagyon egyszerű formátum, kiterjesztések révén támogatja a speciális területek nyelveit, vagyis lehetővé teszik a kiterjesztések, hogy átfordítsuk ezeket a Drools natív szabály nyelvére. A szabály nyelvet egy Antlr 3 nyelvtan írja le (DRL.g).

Terjedelmi okok miatt itt nem kerülnek ismertetésre a Drools kulcsszavainak használata.

### 2.3.11.2 A szabály fájl

A szabályokat tartalmazó fájlok tipikusan .drl kiterjesztéssel rendelkeznek. Egy ilyen szabály fájl tartalmazhat szabályokat (rules), lekérdezéseket (queries), függvényeket (functions), ezenkívül import-okat és globálisok deklarációját is. Lehetőségünk van arra, hogy a szabályainkat szétszórjuk több fájlba, de ebben az esetben javasolt .rule kiterjesztés használata, de nem kötelező. Ha sok szabályunk van, akkor ez segíthet a kezelésükben.

Az általános felépítése egy szabály fájlnak a következő:

```
package package-name

imports

globals

functions

queries

rules
```

Hogy az elemeket milyen sorrendben használjuk, nem fontos. Az egyedüli megkötés, hogy a csomag név legyen az első bejegyzés a fájlban, ha meg van adva. A többi elem opcionális, tehát csak azokat használjuk melyekre szükségünk van.

### 2.3.11.3 Hogyan is néz ki egy szabály

```
rule "name"
 attributes
 when
  LHS
 then
  RHS
end
```

Ilyen egyszerűen néz ki egy szabály. Természetesen a névnél az idézőjeleket nem kötelező kirakni, mint ahogy az új sorok sem azok. Az attribútumok egyszerű utalások arra nézve, hogy hogyan kell a szabálynak viselkednie. Az LHS (Left Hand Side, a szabály bal

oldala) a szabály feltétel része. Az RHS (Right Hand Side, a szabály jobb oldala) olyan dialektus specifikus kódokat tartalmaz, amely végrehajtódik.

Fontos megjegyeznünk, hogy a "white space"-ek nem bírnak jelentőséggel a szabályoknál vagyis több sorosak is lehetnek.

#### *2.3.11.4 A Drools kulcsszavai*

Van néhány olyan szó a szabály nyelvben, melyek kulcsszavak. Erősen javasolt, hogy kerüljük el ezen szavak használatát, amikor elnevezzük az objektumainkat, tulajdonságainkat, metódusainkat és egyéb elemeket a szövegben. Csak a nyelvtan által meghatározott helyeken használjuk őket. Ugyan a szintaktikai elemző elég okos ahhoz, hogy felismerje, ha egy kulcsszót rossz helyen használunk, de inkább kerüljük el ezt.

A listában található kulcsszavak használata azonosítóban tilos:

- `rule`
- `query`
- `when`
- `then`
- `end`
- `null`
- `and`
- `or`
- `not`
- `exists`
- `collect`
- `accumulate`
- `from`
- `forall`
- `true`
- `false`
- `eval`

A következő kulcsszavak használatát azonosítókban, ha lehetséges próbáljuk meg elkerülni.

Ugyan a szintaktikai elemző ezeket általában jól kezeli:

- `package`
- `function`
- `global`
- `import`
- `template`
- `attributes`
- `enabled`
- `salience`
- `duration`
- `init`
- `action`
- `reverse`
- `result`
- `contains`
- `excludes`
- `memberOf`
- `matches`
- `in`
- `date-effective`
- `date-expires`
- `no-loop`
- `auto-focus`
- `activation-group`
- `agenda-group`
- `dialect`
- `rule-flow-group`

Természetesen arra lehetőség van, hogy a felsorolt kulcsszavak valamely másik szó részeként szerepeljenek azonosítókban, például: notSomething().

### 2.3.12 Egy Drools "Hello World" példa alkalmazás

Ahhoz, hogy a példákat letudjuk futtatni, szükségünk lesz az Eclipse fejlesztő

környezetre, a Drools Eclipse plugin-jére és az esetleges függőségek telepítésére, amit a plugin előír.

A következő lépeseket kell elvégezni a példák használatához:

- Csomagoljuk ki a drools-example.zip fájlt.
- Ezt importáljuk be egy üres Eclipse workspace-be.

#### *2.3.12.1 Hello World alkalmazás*

A Hello World példa egy egyszerű példa a szabályok használatára, mind MVEL és Java dialektusokat használva.

Ebben a példában megmutatom, hogy hogyan építsünk fel szabály bázisokat és session-öket, hogyan adjunk házzá audit log-ot, és hibakereső kimenetet. A `PackageBuilder` osztályt használjuk arra, hogy a drl fájlból csomagokat (Package) készítsünk, amit aztán át tudunk adni a szabály bázisnak (RuleBase). Az `addPackageFromDrl()` metódusnak át kell adni egy `Reader`-t paraméterként, amin keresztül be tudja olvasni a drl fájlt. Ha minden drl fájlt hozzáadtunk, akkor `PackageBuilder`-t meg kell kérdeznünk, hogy volt-e valamilyen hiba. Azért `PackageBuilder`-t és nem a `RuleBase`-t, mert az előbbi sokkal részletesebb információval tud szolgálni, ha valami hiba történt. Ha nincs egyetlen hiba sem, akkor elkérhetjük a csomagunkat a `PackageBuilder`-től és hozzáadhatjuk a szabály bázisunkhoz.

```
// A forrás beolvasása.
Reader source = new
InputStreamReader(HelloWorldExample.class.getResourceAsStream("HelloWorld
.drl"));
PackageBuilder builder = new PackageBuilder();
// Egy lépésben történik meg a szintaktikai elemzés és a fordítás.
builder.addPackageFromDrl(source);
// Ellenőrizzük, hogy történt-e hiba.
if (builder.hasErrors()) {
    System.out.println(builder.getErrors().toString());
    throw new RuntimeException("Unable to compile \"HelloWorld.drl\".");
}
// Elkérjük a lefordított csomagot, ami szerializálható.
Package pkg = builder.getPackage();
// A csomag hozzáadása a szabály bázishoz.
RuleBase ruleBase = RuleBaseFactory.newRuleBase();
ruleBase.addPackage(pkg);
StatefulSession session = ruleBase.newStatefulSession();
```

A Drools-nak van egy esemény modellje, ami világossá teszi, hogy mi történik a belsejében. Két alapértelmezett ellenőrző listener áll a rendelkezésünkre, ezek a `DebugAgendaEventListener` és a `DebugWorkingMemoryEventListener`, amik a ellenörző információkat írnak ki a konzol error kimenetére. Listener-ek hozzáadása egy nagyon egyszerű művelet, láthatunk erre itt egy példát. A `WorkingMemoryFileLogger` egy olyan kimenetet állít elő, melyet később egy erre specializált eszközzel megtudunk tekinteni. Ez egy specializált implementáció, ami napirend és munkamemória listener-ein alapul. Ha a motor befejezte a végrehajtást, akkor meg kell hívnunk a `writeToDisk()` metódust.

```
// Beállítjuk az ellenőrző listener-eket.
session.addEventListener(new DebugAgendaEventListener());
session.addEventListener(new DebugWorkingMemoryEventListener());
// Beállítjuk az audit loggolást.
WorkingMemoryFileLogger logger = new WorkingMemoryFileLogger(session);
logger.setFileName("log/helloworld");
```

A Hello World példában a `Message` osztály két mezőt tartalmaz: az üzenetet, ami egy `String` és a státuszt, ami egy `int`. A státusznak két értéke lehet, ezeket a `HELLO` és a `GOODBYE` konstansok definiálják.

```
public static class Message {
 public static final int HELLO = 0;
 public static final int GOODBYE = 1;
 private String message;
 private int status;
...
}
```

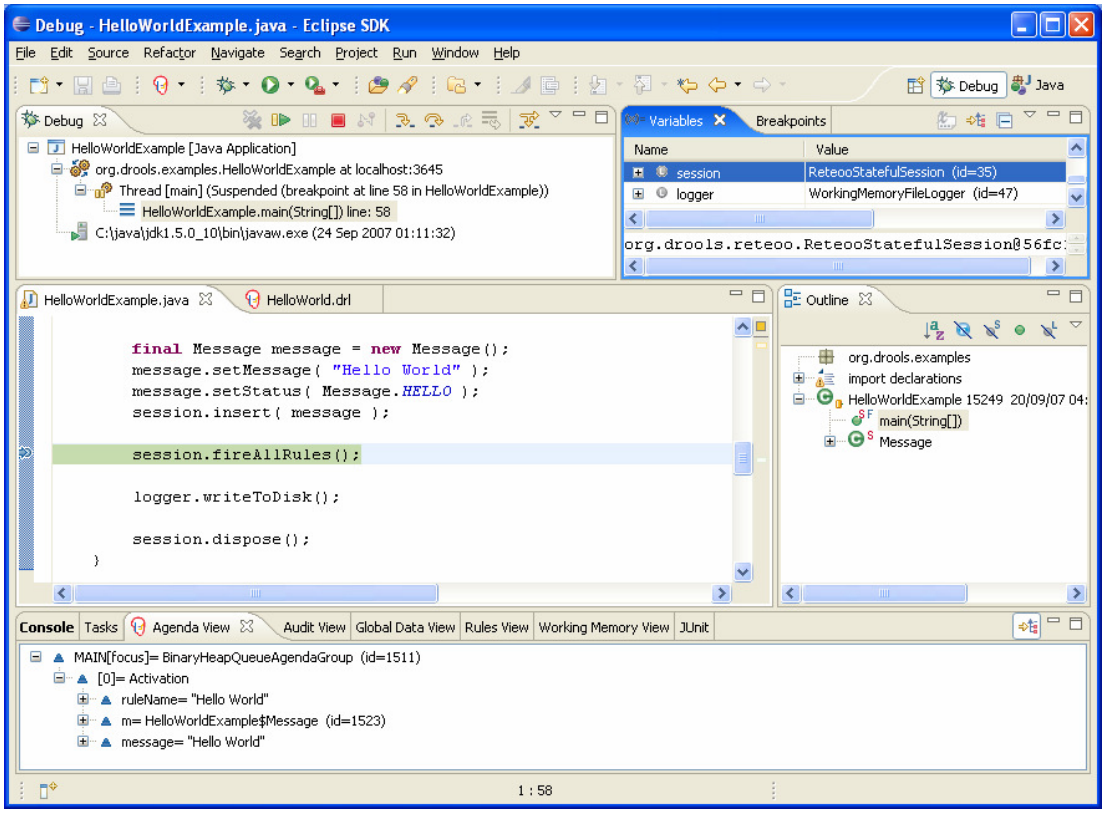
Egy egyszerű Message objektumot hozunk létre, amiben az üzenet a "Hello World" lesz és a státusz a `HELLO`. Ezt adjuk át a motornak, ami után meghívjuk a `fireAllRules()` metódust. Emlékeztetőül a kiértékelő műveletek a beillesztéskor történnek. Amikor a `fireAllRules()` metódust meghívjuk a motor már tudja, hogy mely szabályok lesznek igazak, és végrehajtásra készek.

```
Message message = new Message();
message.setMessage("Hello World");
message.setStatus(Message.HELLO);
session.insert(message);
session.fireAllRules();
logger.writeToDisk();
session.dispose();
```

Ahhoz, hogy az Eclipse-ben le tudjuk futtatni a Hello World példát, a következőket kell tennünk:

- Nyissuk meg a `org.drools.examples.HelloWorldExample` osztályt.
- Majd nyomjunk jobb egér gombot az osztályon és válasszuk ki a "Run as..." menüponton belül a "Java application"-t.

Váltsunk át Debug perspektívába és adjunk töréspontot (breakpoint) a `fireAllRules()` metódushoz. Ha így futtatjuk a példát, akkor a Variables fülnél láthatjuk, hogy már a beillesztés után minden mintaillesztési feladat elvégzésre került.




A példa kimenet a `System.out`-ra megy, az ellenőrző kimenet pedig a `System.err`-re.

```
Hello World
Goodbyte cruel world
```

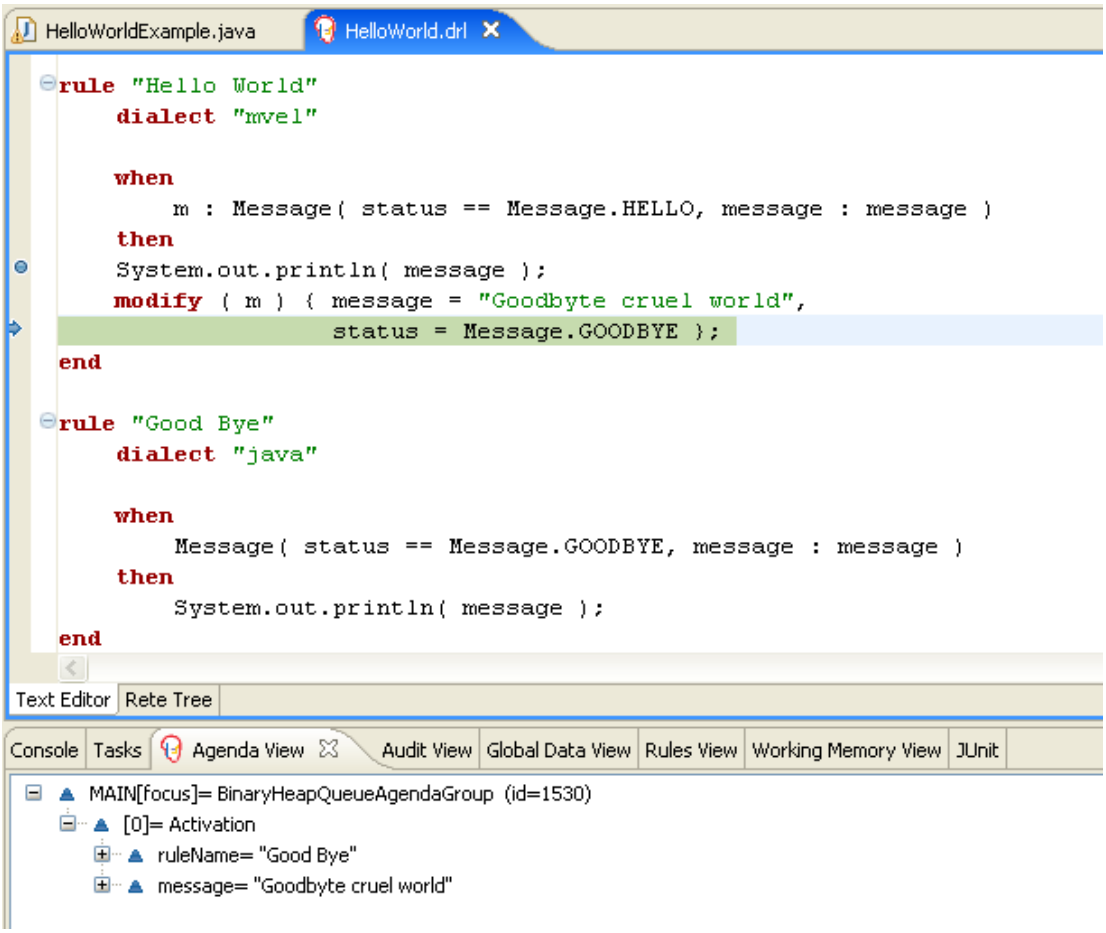
```
==>[ActivationCreated(0): rule=Hello World;

tuple=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96]]
[ObjectInserted:
handle=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96];

object=org.drools.examples.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;

tuple=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96]]
==>[ActivationCreated(4): rule=Good Bye;

tuple=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96]]
[ObjectUpdated:
handle=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96];

old_object=org.drools.examples.HelloWorldExample$Message@17cec96;

new_object=org.drools.examples.HelloWorldExample$Message@17cec96]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;

tuple=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96]]
[AfterActivationFired(4): rule=Good Bye]
```

Az **LHS** (when, ha) része a szabálynak minden olyan esetben aktiválódik, amikor egy `Message` objektumot hozzáadunk a munkamemóriához, aminek a státusza `Message.HELLO`. A "message" a message attribútumhoz rendelődik, az "m" pedig magához az aktuálisan illesztett objektumhoz.

Az **RHS** (then, akkor) részén a szabálynak MVEL dialektust használunk, ezt a szabály attribútumok részénél deklarálnunk kell. Miután kiírtuk az üzenet tartalmát a konzolra, a szabály megváltoztatja az üzenet és a státusz attribútumokat. Hogy ezt megtegyük, az MVEL modify kulcsszavát használjuk. A motort automatikusan értesíti a változásokról a modify blokk végén.

```
rule "Hello World"
  dialect "mvel"
 when
  m : Message( status == Message.HELLO, message : message )
 then
  System.out.println( message );
  modify ( m ) { message = "Goodbyte cruel world",
  status = Message.GOODBYE };
end
```

A drl fájlban is létre tudunk hozni töréspontokat, hogy például megtudjuk nézni, hogy a modify hívása alatt mi is történik. Részletes információval az "Agenda view" fül szolgál. Fontos itt megjegyeznünk, hogy ezúttal a programunkat úgy kell futtatni, hogy "Debug As" "Drools application", és nem "Java application".

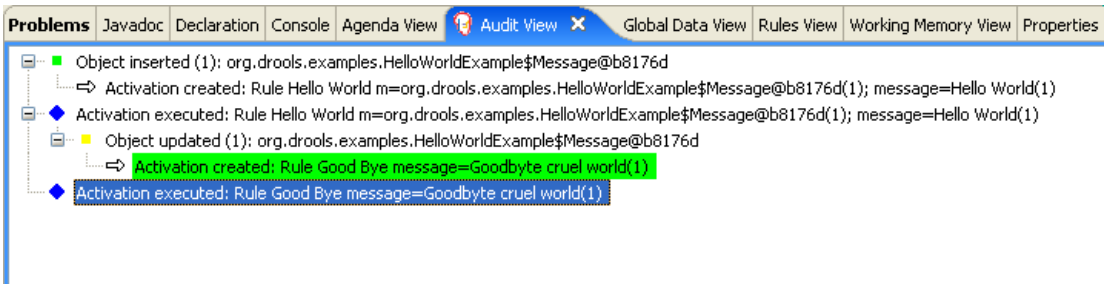



A "Good Bye" nevű szabályunk Java dialektust használ, és nagyon hasonlít a "Hello World" nevű szabályra, de csak olyan tényeknél válik igazzá, melyeknek a státusza

`Message.GOODBYE`. Az alapértelmezett konzolra írja ki az üzenetet.

```
rule "Good Bye"
  dialect "java"
 when
  Message(status == Message.GOODBYE, message : message)
 then
  System.out.println(message);
end
```

A példa elején létrehoztunk egy `WorkingMemoryFileLogger` példányt. A `logger.writeToDisk()` metódust hívtuk meg, hogy elkészítsünk egy audit log fájlt. Ezt az "Audit View" fül alatt tudjuk megtekinteni. Az audit log-on keresztül tudjuk végig követni az egész folyamatot, és a végrehajtások sorrendjét. A fülön láthatjuk, hogy mikor illesztettünk be egy objektumot, ami létrehozott egy aktiválást a "Hello World" szabály által, majd végrehajtásra került ez az aktiválás, ami a `Message` objektum frissíti, ami a "Good Bye" szabály aktiválását eredményezi, ami aztán végrehajtásra kerül.



# Összefoglalás

A használt üzleti alkalmazásoknak fel kell készülniük a piac dinamikus, olykor kiszámíthatatlan változásainak követésére. Példának említeném a banki szférában használt alkalmazásokat, melyek között olyanok vannak, mint az adós minősítés vagy a hitelezés, melyeknél rendszeresen változó peremfeltételeket kell kezelniük. Fontosnak tartom tehát, hogy az ilyen alkalmazásokat már a tervezésnél el kezdünk felkészíteni az ilyen környezetben való megfeleléshez. Ebből következik, hogy a legjobb megoldás egy üzleti szabály motor bevezetése lehet.

Az üzleti szabály motor egy olyan alkalmazás vagy szoftver komponens, amely segít abban, hogy az üzleti igényeket megvalósító logikát kiszervezzük az alkalmazásunkból, így lehetővé válik, hogy az alkalmazást vezérlő szabályokat úgy módosítsuk, hogy magához az alkalmazásunkhoz nem nyúlunk. Költség oldalról rengeteg időt és pénzt tudunk megtakarítani a szabály motorok használatával, mert (a dolgozatban ismertetett Drools is) lehetőséget biztosítanak arra, hogy informatikusok beavatkozása nélkül adjunk hozzá vagy éppen távolítsunk el szabályokat a meglévő rendszerünkből. Így meg tudjuk spórolni a fejlesztés és tesztelés hosszadalmas és költséges folyamatát, melyek egy alkalmazás módosításával járnak.

# Irodalomjegyzék

JBoss Drools
http://www.jboss.org/drools/documentation.html
http://downloads.jboss.com/drools/docs/4.0.4.17825.GA/html_single/index.html

JSR 94: Java Rule Engine API
http://jcp.org/en/jsr/detail?id=94

Dr. Bognár Katalin: Ismeretábrázolás és következtetés OO rendszerekben, mobiDIÁK könyvtár, 2004
http://iam035.inf.unideb.hu/mobidiak/filedownload.mobi?fid=261

P-H. Speel, A. Th. Schreiber, W. van Joolingen, G. van Heijst, G.J. Beijer: Conceptual Modelling for Knowledge-Based Systems
http://www.cs.vu.nl/~guus/papers/Speel01a.pdf

Aszalós László: Mesterséges intelligencia közgazdászoknak, mobiDIÁK könyvtár, 2005
http://www.inf.unideb.hu/~aszalos/diak/mik/mik.pdf

Dr. Dobrowiecki Tadeusz, Mészáros Tamás – A mesterséges intelligencia új területei: intelligens ágensek
http://home.mit.bme.hu/~meszaros/me/pubs/agensjegyzet_1999.pdf

Ovidiu S. Noran: The Evolution of Expert Systems
http://www.cit.gu.edu.au/~noran/Docs/ES-Evolution.pdf

Nyékyné G. Judit: Java 2 útikalauz programozóknak 1.3, ELTE TTK Hallgatói alapítvány

Han, J. & Kamber, M.: Adatbányászat, Panem 2004