

TDK-dolgozat

Készítette:

Bodnár Tamás

Eötvös Loránd Tudományegyetem

Informatikai Kar, Információs Rendszerek Tanszék

Dinamikus algoritmusok készítése

Készítette:
Bodnár Tamás
MSc hallgató,
1.évfolyam

Témavezető:
Ács Zoltán
tanársegéd

A dolgozat a TÁMOP 4.2.1/B-09/1/KMR-2010-0003 számú pályázatának a támogatásával készült.

2012

Tartalomjegyzék

Feladat	3
Megoldás	5
Felhasznált technológiák.....	5
Rendszer felépítése.....	10
Kliens megvalósítása	13
Szerver oldali funkciók részletezése	23
Példa	43
Tesztek – eredmények.....	45
Összefoglalás	48
Referenciák	49
Melléklet	50

1 Feladat

1.1 Feladat célja

A kutatás célja egy *bővíthető algoritmus készletet* megvalósító prototípus megoldás készítése, illetve az eredmények részletes dokumentálása. A kialakított prototípusnak egy olyan felületet kell biztosítania, amely alkalmas új algoritmus logikák definiálására, létező logikák módosítására a rendszerben elérhető leprogramozott függvények illetve létező adatelérések alapján. A létrehozott metódusokat egy létező környezetbe be kell tudni illeszteni úgy, hogy közben a környezetet nem lehet leállítani.

1.2 Tulajdonságok leírása

A feladat két részből tevődik össze: szerver- illetve kliens-oldal. A kliens egy grafikus felület, amely lehetővé teszi a szerver oldalon lévő adatelérések, illetve az előre megírt eljárások bizonyos szintű összekombinálását. A kliens oldali alkalmazás tehát le tudja kérdezni a szerveren lévő eljárásokkal kapcsolatos információkat.

Az alkalmazásnak az összekattintott új logikát megvalósító kód előállítását és mentését a szervernél kell kezdeményeznie. A létrehozott algoritmus logikák futtatását a szerver végzi, a futás eredményét a kliens számára lekérdezhetővé kell tenni.

Az alkalmazással szemben támasztott elvárásokat az alábbi pontok foglalják össze:

- Támogassa a legelterjedtebb műveleteket:
 - **Matematikai alapl műveletek:** +, -, /, *, x^y .
 - **Logikai operátorok:** and, or, not.
 - **Összehasonlítási operátorok (numerikus adatra):** <, >, =, ≤, ≥, ≠
 - **Karakterlánc műveletek:** rész karakterlánc, egyszerű minta előfordulás, hossz függvény, összefűzés
 - **Ciklus** Két fajta ciklus lesz.
 - **for** Egy űrlapon keresztül a felhasználó meg tudja adni a kezdeti értéket, a növelés mértékét illetve a ciklus feltételt.
 - **foreach** Egy megadott gyűjtemény elemein történő végig haladást teszi lehetővé.
 - **Feltételes elágazás** A felhasználható adhaszon meg olyan lépés(ek)e)t, amelyek az általa megadott feltétel teljesülése esetén hajtódik végre.
 - **Dátum függvény** Például két dátum közötti napok száma.
- Szerkesztett algoritmusok verziókezelése. (verziószám)
 - Felhasználó láthassa az algoritmusok minden egyes verzióját, és választhasson a különféle verziók közül.

- Az algoritmusokhoz név és leírás rendelhető.
- Az egyes eljárásoknak legyen plusz egy kimenete a definiáltakon kívül, ami a hiba esemény esetén használható naplózási célokra.
- Az egyes algoritmus kimenetek többször felhasználhatóak legyenek.
- Változók használatának lehetősége.
- Az algoritmusoknak több bemenő illetve kimenő paraméter is megadható legyen, azaz összetett (primitív típusokból összeállított) típust is meg lehessen adni paraméterként.
- Szemantikai hibát nem kezel a prototípus! Ezért például a végtelen ciklusok elkerüléséről a felhasználónak kell gondoskodnia.
- Típus-ütközéseket nem ellenőriz.
- Az elkészített metódusok kívülről is meghívhatóak legyenek

1.3 Feladat nehézségei

A feladatnak több fontos része van, de talán a két legfontosabb rész a kód generálása, fordítása és a lefordított állomány betöltése a környezetbe futási időben.

1.3.1 Kódgenerálás

A cél az, hogy a grafikus felületen összeállított algoritmust valamilyen tárolható formába átalakítsuk, ami emellett azoknak a követelményeknek is megfelel, hogy könnyen mozgatható legyen, visszaállítható legyen belőle a grafikus felületen létrehozott metódus és a kódgenerálás is megoldható legyen a leírásból. Erre a célra a legmegfelelőbb egy XML vagy XML-ek használata, ami az összes elvárásnak eleget tesz.

Az előzetes terv az, hogy a felületen előállított metódusból egy olyan XML-t generálunk, ami minden megfelelő adatot tárol ahhoz, hogy ebből a felület is visszaállítható legyen és a kód generálásához is elegendő információval rendelkezik, mivel a kódgeneráláshoz egyedül ezt szeretnénk felhasználni.

A megoldás részletes tárgyalása a 2.4 pont alfejezeteiben található.

1.3.2 Lefordított állomány használata

A lefordított állományt úgy akarjuk használni, hogy közben a rendszert nem állíthatjuk le és nem indíthatjuk újra, ennek a megoldása pedig nem triviális. Azt is meg kell oldani, hogy a metódusokat esetleg távolról is futtathassuk.

A megoldás részletes tárgyalása szintén a 2.4 pont alfejezeteiben található.

2 Megoldás

A megoldás során több, már létező technológia fel lett használva például a verziókezeléshez, a fordításhoz és a már létező adatstruktúrák rendszerbe illesztéséhez. Először ezeknek a technológiáknak a bemutatására kerül sor, ahol a használatukról is szó esik. Ezután következik a konkrét megoldások bemutatása.

2.1 Felhasznált technológiák

Ezen fejezetben ismertetjük azokat a *Java* technológiákat, amelyeket a feladat megoldása során felhasználtunk. Minden alfejezet több részből tevődik össze, ahol a technológia rövid ismertetése mellett a néhány példakód is található.

2.1.1 Compiler API

Ez az *API* lehetővé teszi kódok futási időben történő fordítását. A most következő részben bemutatjuk az *API* lehetőségeit.

2.1.1.1 Alkalmazás

A *Java Compiler API* segítségével java forráskódból futási időben tudunk fordítani egy másik java forrást. A következő példakód bemutatja, hogyan is tehetjük meg ezt.

```
package test;

public class MyClass {
    public void myMethod() {
        System.out.println("My Method Called");
    }
}
```

1. kódrészlet: MyClass.java tartalma

A *MyClass* osztályt fogjuk lefordítani dinamikusán futásidőben a *SimpleCompilerTest* osztályból. Ehhez megadjuk a szükséges útvonalat, majd létrehozunk egy *JavaCompiler* típusú compiler objektumot. A *compiler run(InputStream inputStream, OutputStream outputStream, OutputStream errorStream, String arguments)* metódusával fordíthatjuk le a fájlt. Alapértelmezett használatban az első három paramétert *null*-ként adhatjuk meg, ekkor ezek megfelelnek a *System.in*, *System.out* és a *System.err*-nek. Ha a metódus visszatérési értéke 0, akkor sikeres volt a fordítás, egyébként hibás a fordítandó fájl. A *SimpleCompilerTest.java* lefuttatása után a *Compilation is successful* eredményt kapjuk.

```

package test;
import javax.tools.*;
public class SimpleCompileTest {
    public static void main(String[] args) {
        String fileToCompile = "test/MyClass.java";
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        int compilationResult = compiler.run(null, null, null,
fileToCompile);
        if(compilationResult == 0){
            System.out.println("Compilation is successful");
        }else{
            System.out.println("Compilation Failed");
        }
    }
}

```

2. kódrészlet: SimpleCompileTest.java tartalma

Ha hibásan adjuk meg a fordítandó állományt, például leghagyjuk a pontosvesszőt a *myMethod()* 5. sorában, akkor a *SimpleCompileTest.java* futtatása során ezt ki fogja jelezni.

```

test\MyClass.java:5: ';' expected
System.out.println("My Method Called") 1 error Compilation Failed

```

3. kódrészlet: Fordítási hiba

2.1.1.2 Compiler API felhasználása

A prototípus megvalósításánál a szerver oldalon a fordítási feladatokat ezzel az API-val oldottuk meg.

2.1.2 Content Repository API

A tartalomkezelő alkalmazások növekvő népszerűsége miatt nyilvánvalóvá vált, hogy szükséges egy szabványosított tartalomkezelő API létrehozása. A Java Content Repository API (JSR-170) célja egy ilyen interfész nyújtása. A Java Content Repository – amit a Java Community Process JSR-170 részeként fejlesztettek ki – célja, hogy megfeleljen az ipari igényeknek. A specifikáció egy egységesített API-t nyújt a *javax.jcr* névtér alatt, amely lehetővé teszi bármely specifikáció szabványnak megfelelő implementálását. Az API-szabványosítás nem az egyetlen funkció, amit a JCR letesz az asztalra. A JSR-170 nagy előnye, hogy nem kötődik semmilyen konkrét mögöttes architektúrához. A JSR-170 back-end adattárolás implementációja lehet például egy fájlrendszer, egy WebDAV Repository, egy XML-alapú rendszer, vagy akár egy

SQL adatbázis. Végül a JCR egy egyszerű interfészt nyújt a már meglévő Content Repository összetett funkcionalitásához, mint a verziókezelés, hozzáférés-szabályozás és keresés.

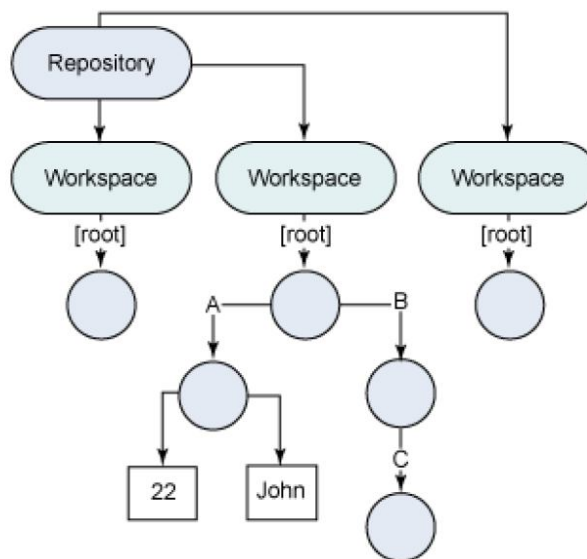
2.1.2.1 Áttekintés

A Repository modell egy egyszerű hierarchia, ami úgy néz ki, mint egy n-áris fa. Egy content repository-t tartalmaz egy vagy több munkaterülettel. Mindegyik munkaterület tartalmaz egy elemekből álló fát, egy elem lehet csomópont vagy tulajdonság. Egy csomópontnak nulla vagy több gyerek és nulla vagy több kapcsolódó tulajdonság tartozik, ahol a tényleges tartalom tárolódik.

Minden csomópontnak egyetlen elsődleges csomópont típusa van. Az elsődleges csomópont típusa határozza meg a csomópont jellemzőit, mint például a tulajdonságokat és a gyermek csomópontokat, amennyiben azok megengedettek. Az elsődleges csomópont típus mellett egy vagy több *mixin* típus is tartozhat egy csomópontnak. A *mixin* típus extra jellemzőket nyújt egy csomópontnak. A JCR implementáció három előre definiált *mixin* típust nyújt:

- *mix:versionable* - lehetővé teszi egy csomópontnak a verziókezelés támogatását,
- *mix:lockable* - lehetővé teszi egy csomópontnak a zárolási képességeket,
- *mix:referenceable* - egy automatikusan létrehozott *jcr:uuid* tulajdonságot nyújt, amely a csomópontnak egy egyedi, *referenceable* azonosítót ad.

Ezt a struktúrát az **1. ábra** szemlélteti. A körök reprezentálják a csomópontokat, a téglalapok a tulajdonságokat. Az A csomópont két tulajdonsággal rendelkezik, egy „John” string-gel és egy 22 integer-rel.



1. ÁBRA - A REPOSITORY MODELL TÖBB MUNKATERÜLETTEL

Előre definiált csomópont típusok

Minden `repository`-nak támogatnia kell az elsődleges csomópont típust, az `nt:base`-t. Ezen kívül számos más típust is támogathat:

- `nt:unstructured` – a legrugalmasabb csomópont típus. Megenged bármennyi gyerek csomópontot vagy tulajdonságot tetszőleges névvel.
- `nt:file` – reprezentálja a fájlokat. Egyetlen gyermek csomópontot követel meg, amit `jcr:content` – néven nevezünk.
- `nt:folder` – csomópont típus reprezentálja a mappákat.
- `nt:resource` – gyakran jelenti egy fájl tényleges tartalmát.
- `nt:version` – a szükséges csomópont típus a verziókövetés támogatásához.

Névterek

Egy hasznos, de gyakran figyelmen kívül hagyott eleme a `Repository` modellnek a névterek támogatása. A névtereknek köszönhetően elkerülhetőek az ütközések a különböző forrásokból származó elemek között. Névtereket *prefix*ként definiálhatunk, a `:` jellel elválasztva. Az eddigi leírásban már találkozhattunk névterekkel, ilyen a `jcr` a JCR belső tulajdonságokhoz, a `mix` a `mixin` típusokhoz és az `nt` a csomópont típusokhoz.

2.1.2.2 Content Repository API felhasználása

A prototípus ilyen **repository**-t használ a szerkesztett algoritmusok perzisztens tárolására. A végleges topológia leírása a *Rendszer felépítése* pontban szerepel részletesen.

2.1.3 Reflection API

A **Java Reflection API** lehetővé teszi az osztályok, *interface*-ek, mezők, metódusok futási idejű vizsgálatát anélkül, hogy a vizsgált elemeket fordítási időben rögzítenénk.

Az *API* része a `java.lang` csomagnak.

2.1.3.1 Áttekintés

A **Java Reflection API**-val könnyen, egyszerűen futási időben szerezhethetünk információt az osztályokról, mezőkről, metódusokról.

2.1.3.1.1 Class betöltése futási időben.

Ehhez a művelethez az osztálynak elérhetőnek kell lennie a futási idejű `classpath`-ról.

```
Class<?> c = Class.forName("java.lang.String");
```

4. kódrészlet: Az osztály betöltése név alapján

A `Class.forName` módszerrel név alapján tudunk betölteni egy osztályt, ahol a nevet egyszerű szöveggé kell megadni.

2.1.3.1.2 Osztály metódusainak lekérdezése

Egy adott osztályban definiált metódusokat a következő módon kérdezhetünk le.

```
Method[] methods = c.getDeclaredMethods();
```

5. kódrészlet: Az osztály metódusainak a lekérdezése, ahol `c` az osztály

, ahol `c` az osztály.

Ekkor a metódusról több adatot is megtudunk:

- Nevét
- Paramétereinek a számát és típusát
- A visszatérésének típusát
- A metódusra vonatkozó annotációkat és azok típusát
- A metódusra vonatkozó módosítókat
- ...

2.1.3.1.3 Osztály mezőinek lekérdezése

Egy adott osztályban definiált mezőket a következő módon kérdezhetünk le.

```
Field[] fields = c.getDeclaredFields();
```

6. kódrészlet: Az osztály mezőinek a lekérdezése, ahol `c` az osztály

, ahol `c` egy **Class** típusú objektum.

Ekkor az egyes mezőkről további adatok érhetők el:

- A neve
- A típusa
- Az annotációi
- A módosítói
- ...

2.1.3.1.4 Metóduş meghívása

Lehetőség van a dinamikusan betöltött osztályok metóduşainak futtatására.

```
Object[] par;  
Method m;  
...  
Object o = m.invoke(null, par); //null - statikus  
metóduş
```

7. kódrészlet: A metóduş meghívása

Egy metóduş futtatásához szükség van a bemeneti paramétereire, amiket egy **Object** tömbben adhatunk meg (**Object[] par**) és szükségünk van a metóduş referenciájára is. Ezután a metóduş **invoke** függvényével lehet a metóduş meghívni. Az **invoke** első paramétere az az objektum, amire a metóduş meghívjuk és **null**, ha a metóduş statikus. A második paramétere a metóduş bemeneti paramétereinek a tömbje. A visszatérési érték **Object** típusú és a metóduşhívás eredményét tartalmazza.

A szemléltetett példákön túl még rengeteg hasznos szolgáltatást nyújt az API, ezekről részletesen a dokumentációban lehet olvasni: <http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html>

2.1.3.2 Felhasználás, alkalmazás

A **Reflection** csomagot az alkalmazás során több helyen is felhasználtuk:

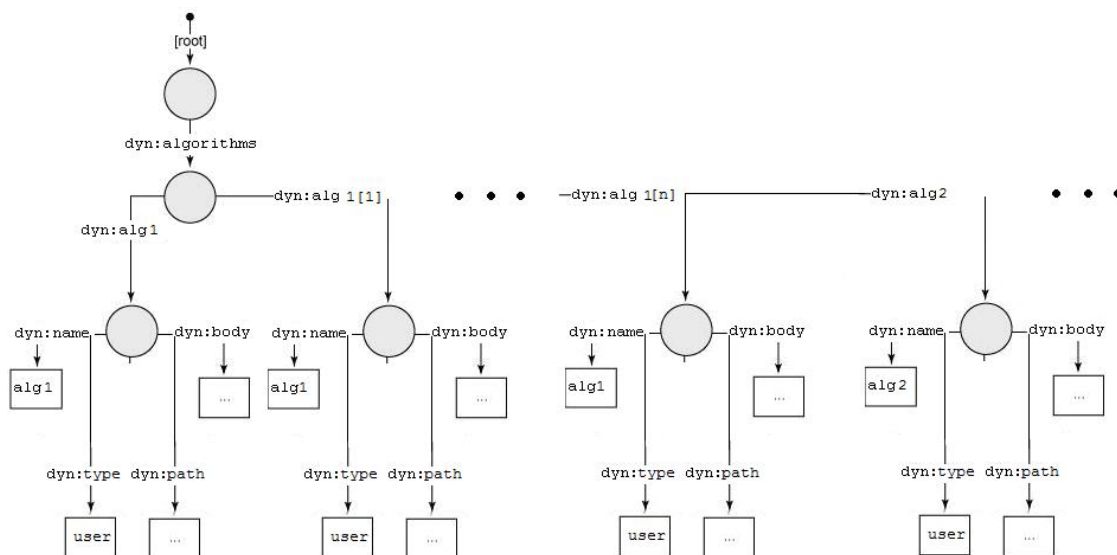
1. **Beépített elemek felolvasása:** a beépített elemeket speciális annotációval kell ellátni, melyek alapján a megfelelő alkalmazás képes az adott elemet reprezentáló XML fájl létrehozására, és betöltésére a repositoryba. Részletesen lásd a felolvasásnál.
2. **Metóduşok futtatása:** lehetőség van a felhasználók által létrehozott metóduşok lefordítására, futási idejű betöltésére és futtatására.

2.2 Rendszer felépítése

2.2.1 Prototípus működési elve

Logikailag a prototípus két részre bontható fel: kliens- illetve szerver-oldal. A két oldal közötti kommunikáció a Java EJB technológia felhasználásával valósul meg. A szerver oldali rész funkcionálisok alapján három fő részre bontható:

- **Compiler** → a Compiler feladata különböző – a prototípus által létrehozott – java forrásállományok lefordítása dinamikusan, futásidőben. A fordításhoz az 2.1.1. alfejezetben ismertetett Java Compiler API-t használtuk. Két feladatot lát el ez az egység. Az első feladata a beépített komplex típusok lefordítása, hogy azokat használhassa a felhasználó a saját algoritmusaiban a felhasználói felületen. A lefordított fájlok a szerveroldalon egy kitüntetett mappában fognak elhelyezkedni. A másik feladata a felhasználó által létrehozott algoritmusok lefordítása futásidőben. Ehhez meg kell kapnia a klienstől a lefordítandó algoritmus nevét és XML leíróját, ami alapján legenerálja a *java* állományt majd le is fordítja azt. A fordítás eredményéről tájékoztatja a klienst.
- **Repository** → a Repository feladata beépített függvények, komplex típusok, adatforrások, valamint a felhasználó által elkészített algoritmusok tárolása és verziókezelése. A 2.1.2. alfejezetben taglalt Java Content Repository API-t, konkrétan ennek az Apache Jackrabbit implementációját használtuk fel ezen feladat megvalósítására. A Repository felépítése a következő. A gyökércsúcs alatt a `dyn:algorithms` csúcs áll, amely összefogja az algoritmusokat. Alatta az egyes algoritmusok csúcsai találhatók `dyn:algoritmusnev` formában. Az azonos algoritmusok különböző verziói testvéreként szerepelnek. Egy algoritmus négy property-vel rendelkezik, ezek a következők: `dyn:name`, amely az algoritmus nevét tárolja, `dyn:body`, amely az algoritmus XML leíróját tárolja szöveggént, `dyn:type`, amely a csúcs típusát tárolja (*built-in*, *komplex*, *adatforrás*, vagy *felhasználó által készített algoritmus*), illetve a `dyn:path`, amely az elérési utat tartalmazza.



2. ábra A VERZIÓKEZELŐ RENDSZER TOPOLOGIÁJA

A Repository böngészésére a Jackrabbit Explorer[4] alkalmas. A Repository fizikai áthelyezése egyszerű, a szerveroldalon szereplő repository.xml fájlt és repository mappát kell átmásolni az új, Repository-t használni kívánó projektbe.

- o Loader – Egyik feladata, hogy az előállított algoritmusok beépülhessenek futási időben a rendszerbe. A másik feladata, hogy a szerver oldali elérhető algoritmusokat letárolja a futó repository-ba.

A kliens funkcióit tekintve szintén 3 részre bontható:

- o RepoConnector → Egyrészt a repository információi alapján feltölti a kliensalkalmazásnak a létező algoritmusokkal kapcsolatos információkat. Másrésztől szerkesztett algoritmusokat tud visszatölteni a repository-ból, illetve egy szerkesztett algoritmus mentését kezdeményezheti.
- o GraphViewer → Az algoritmusok gráf ábrázolásáért felelős modul.
- o FormController → Az űrlapok generálásáért felelős modul.

A működés legfontosabb elemei az alábbi felsorolásban pontokba szedve össze vannak foglalva. Minden egyes lépésnél zárójelben meg vannak jelölve azok a technológiák, amelyeket az implementáció során az adott lépéshez fel kellett használni.

1. Loader betölti a *built-in*(beépített, a rendszerben már meglévő) függvényeket a Repository-ba. (**Reflection API**)

2. A kliens alkalmazás az indulásakor lekéri a függvények leíróit a Repository-ból.
(**Content Repository API**)

4. **Kliens betölt egy algoritmust.** Ehhez az algoritmushoz tartozó XML leíróat használja fel, ami alapján felépíti a gráf-ábrázolást.

5. **Fordítás, mentés.** A kliens elküldi a szervernek az általa generált XML-t, amit a Compiler megkap, kigenerálja a kódot, előállítja a *java* és a *class* fájlokat a megfelelő helyen. Ha sikeres volt a fordítás, akkor a metódus bekerül a repository-ba, ami egy verziószámot ad neki. Ezt a verziószámot kapja vissza a kliens.

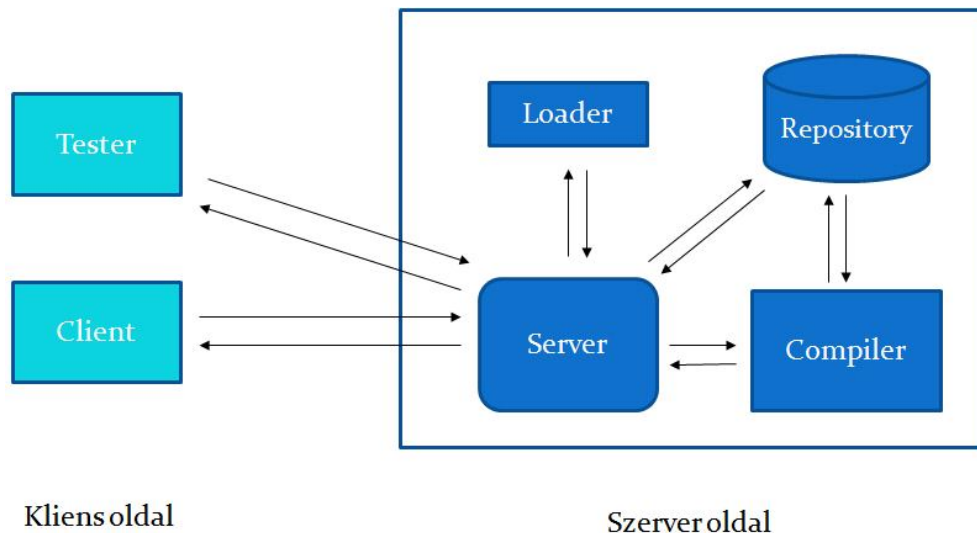
(**Compiler API, Content Repository API**)

6. **Futtatás.** Kliens elküldi a Repository-nak az algoritmus nevét, verzióját és a bemeneti paramétereket. A repository elküldi a *class* fájl nevét és a paramétereket a Loader-nek, ami

betölti a *class*-t és lefuttatja az algoritmust. Végül az eredményt a Repository visszaküldi a kliensnek.

(**Reflection API, Content Repository API**)

A prototípus működésének elvét a 3. ábra szemlélteti.



3. ábra A prototípus architektúrája

2.3 Kliens megvalósítása

A kliens feladata, hogy a felhasználó számára lehetővé tegye eljárások grafikus úton történő összeszerkesztését, mentését és tesztelését. A szerkesztéshez felhasználhat már előtte elmentett eljárásokat (felhasználói metódusok) vagy már előre elkészített metódusokat, úgynevezett beépített eljárásokat (beépített metódusok). Ezen kívül elérhetővé kell tenni számára különböző adatforrásokat, amin dolgozhat és tesztelhet.

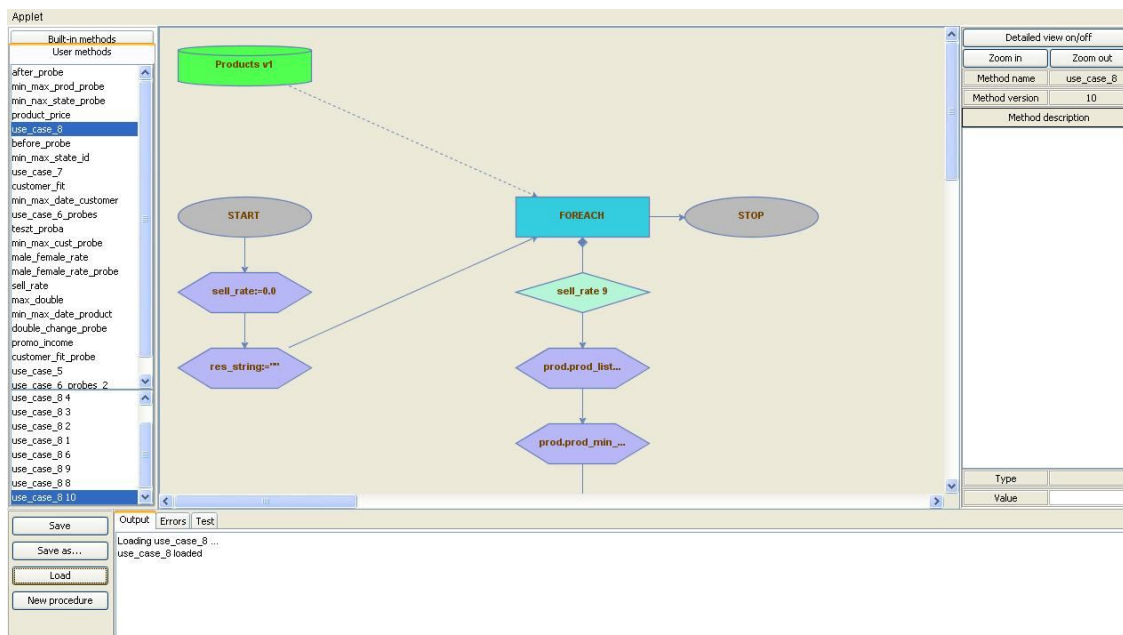
A kliens alapvetően 3 funkciót lát el:

1. Gráfszerkesztő funkció – a metódusok előállítására.
2. A gráf csúcsainak a tartalmát szerkesztő felület – űrlapokon keresztül.
3. Adat- és információcsere a szerverrel, ahol a szerkesztett metódusok tárolódnak és a tesztek futnak – ide tartozik a leírók generálása és felolvasása is.

A felhasználói felületet egy Applet alkotja, így a kliens akár egy weboldalba is beágyazható. Ekkor a következő paramétert lehet az Applet-nek beállítani, amivel a szerver adatait adhatjuk meg:

Paraméter neve	Alapértelmezett érték	Érték
java.naming.provider.url	localhost:3700	A szerver címe és a port száma, amire kapcsolódni kell

1. Táblázat Az applet-nek adható speciális paraméter



4. ábra A kliens működés közben

A kliens természetesen parancssorból is futtatható, mégpedig a következő módon:

```
java -jar FunctionBuilder.jar <server> <port>
```

8. kódrészlet: A kliens futtatása parancssorból

- <server> : a szerver címe
- <port> : az orb listener port, amire kapcsolódni kell

Pl. java -jar FunctionBuilder.jar dblab.inf.elte.hu 3700.

Az alapértelmezett beállítások:

- <server> : localhost
- <port> : 3700

2.3.1 A kliens funkciói

2.3.1.1 Támogatott típusok

- Alaptípusok: `integer`, `string`, `char`, `double`, `Date`, `boolean`
- Komplex típusok: az alaptípusokból összeállított összetett típusok

2.3.1.2 Metódus szerkesztése

A kliens legfőbb feladata, hogy a felhasználónak egy könnyen használható felületet nyújtson a metódusok szerkesztéséhez. A szerkesztéshez hozzátartozik a metódus struktúrájának a kialakítása és a struktúra feltöltése adatokkal (pl. értékadás beállítása).

A metódust egy gráfként lehet elkészíteni, ami többfajta csúcsból és többfajta élből áll össze. Egy csúcs általában egy utasítást jelöl. Az algoritmusban lévő utasítások végrehajtásának az irányát az élek határozzák meg.

A csúcsok típusai funkcióknak megfelelően:

- **Értékadás csúcs:** egy értékadást jelképező csúcs, ahol egy változónak tudunk egy egy kifejezést értékül adni
- **Switch csúcs:** az elágazást jelképező csúcs, amiből több feltételcsúcs húzható
- **Feltétel csúcs:** egy feltételt jelképező csúcs.
- **Default feltétel csúcs:** az elágazás utolsó, *else* ágát jelképező csúcs.
- **Adatforrás csúcs:** az adatforrást jelképező csúcs
- **For csúcs:** Egy általános for ciklust jelképező csúcs
- **Foreach csúcs:** egy iteráló ciklust jelképező csúcs. Egy adatforráscsúcsot kell hozzákötni, aminek az adatain végig lehet iterálni
- **Metódus vagy function csúcs:** egy meglévő metódus használatát lehetővé tevő csúcs
- **Log csúcs:** hiba- vagy *log*-üzenet megadására szolgáló utasítást jelképező csúcs.
- **Start csúcs:** a metódus kezdőcsúcsa, ahol be lehet állítani a használt változókat, a bemeneti változókat és új típusokat lehet létrehozni
- **Stop csúcs:** a metódus célcsúcsa, ahol be lehet állítani a metódus kimeneti paramétereit

Ezeknek a csúcsoknak a segítségével lehet felépíteni a metódusokat.

Az élek fajtái:

- a. Az *adatforráscsúcs*ból a *foreach* csúcsba húzott él, a felületen szaggatott éllel van jelölve
- b. A végrehajtásban a rákövetkezőt jelképező él, a felületen egy sima irányított él.
- c. A *switch* csúcsból egy *feltétel* csúcsba húzott él, a felületen egy sima irányítatlan él.
- d. A *ciklusmag*ból a *for* vagy *foreach* csúcsba húzott él, a felületen egy sima irányított él, aminek a vége egy rombusz.

Az élek fajtáit nem a felhasználónak kell beállítania, mert a rendszer ezt képes magától kikövetkeztetni. A grafikai megkülönböztetés a felhasználó számára egy segítséget jelent az áttekinthetőség szempontjából.

2.3.1.3 A metódus mentése

Miután egy metódust elkészítettünk vagy esetleg egy meglévő metódust módosítottunk, lehetőség van a mentésre. Egy új metódus esetén ekkor nevet és leírást kell neki adni. A kliens az elkészített eljárásból egy XML leírót készít, amit elküld a szervernek mentésre. Ha a mentés sikeres volt (sikerült a kódgenerálás is), akkor a felületen megjelenik a mentett eljárás. Ha már egy létező eljárást mentettünk, akkor egy új verziószámot kap a metódus, ha pedig újat, akkor az 1 verziószámot kapja.

A generált leíró megfelel a **2.4.2.3** pontban kifejtett sémának.

2.3.1.4 Egy régebben szerkesztett metódus betöltése

A kliens az összes elkészített metódus leíróját tárolja, így betöltéskor ezt az XML-t dolgozza fel és ez alapján hozza létre a felületen a gráfot, amit ezután szabadon lehet szerkeszteni. A leíró, amiből felépíti a felületet, megfelel a **2.4.2.3** pontban kifejtett sémának.

2.3.1.5 Tesztelés

A kliens lehetőséget ad az elmentett metódusok tesztelésére azzal a megkötéssel, hogy a tesztelendő metódus bemeneti- és kimeneti paraméterei is alaptípusúak.

A tesztelés indításakor meg kell adni a metódus bemeneti paramétereit, amire egy külön ablak áll rendelkezésre, ahol beépített típusellenőrző is van, tehát csak megfelelő típusú adatokat adhatunk meg.

Az adatok megadása után a teszt elindul, ami a szerveren fut le, majd az eredményt visszaküldi a kliensnek, ami megjeleníti azt.

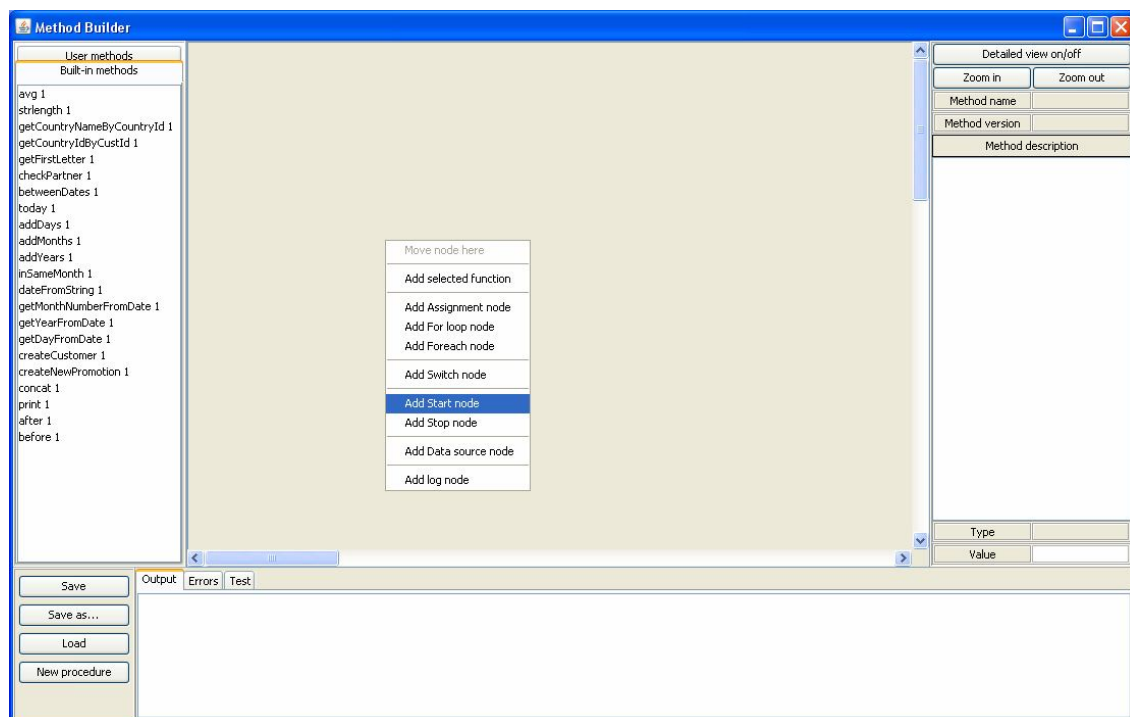
2.3.1.6 Egyéb funkciók

- **Részletes nézet:** Egy plusz funkció, ami arra szolgál, hogy a csúcsokban tárolt adatokat a felületen egyszerre megjelenítsük. Ekkor minden csúcshoz megjelennek a benne tárolt adatok egy táblázatban, ami a csúcs alatt helyezkedik el.
- **Zoomolás:** Az áttekinthetőség kedvéért zoomolási lehetőség is van, amivel a szerkesztett gráfra ráközelíthetünk, illetve eltávolodhatunk tőle.

2.3.2 Egy egyszerű példa megvalósítása a felületen

Példa feladat: Ha a bemenő paraméter tíznél kisebb, akkor szorozzuk meg kettővel, különben adjunk hozzá egyet. Tegyük fel, hogy 'x' a bemenő paraméter neve.

Minden metódus előállításánál az első feladat a **Start** csúcs hozzáadása. Csúcsokat a munkaterületre jobb egérgombbal kattintva tudunk létrehozni. Ekkor megjelenik egy felugró menü, ahol a megfelelő csúcsot kiválasztva hozzá tudjuk azt adni a metódushoz.

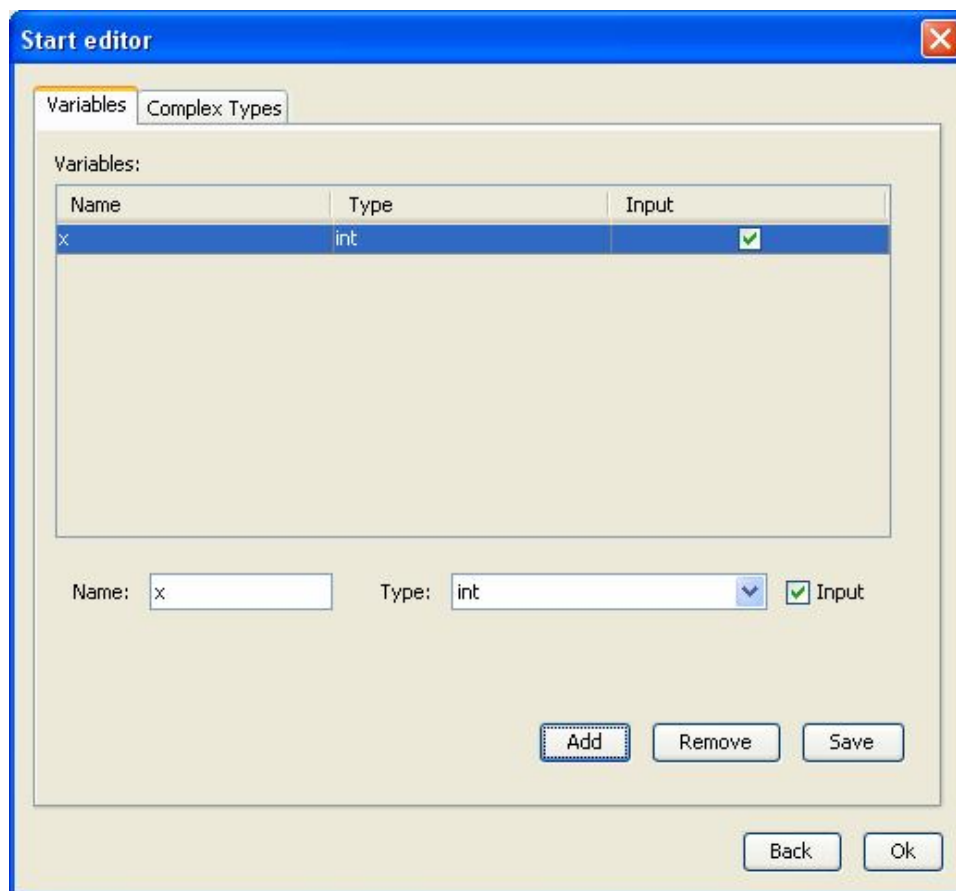


5. ábra Csúcs hozzáadása a munkaterületen

A **Start** csúcs hozzáadása után a bal egérgombbal duplán a csúcsra kattintva előjön a csúcsához tartozó űrlap, ahol az adatait tudjuk szerkeszteni. Ez minden csúcsra így működik.

A Start csúcs űrlapján változókat tudunk definiálni és új típusokat tudunk létrehozni.

Most vegyünk fel egy **x** nevű, egész típusú változót. A név mezőbe írjuk be, hogy '**x**', a **Type** legördülő menüből pedig válasszuk ki az '**int**'-et. A változót bemeneti változónak is beállíthatjuk az input négyzet kijelölésével. Miután ezzel megvagyunk, az Add gombra kattintva létrehozzuk a változót. Az **OK** gombbal elhagyhatjuk az ablakot.

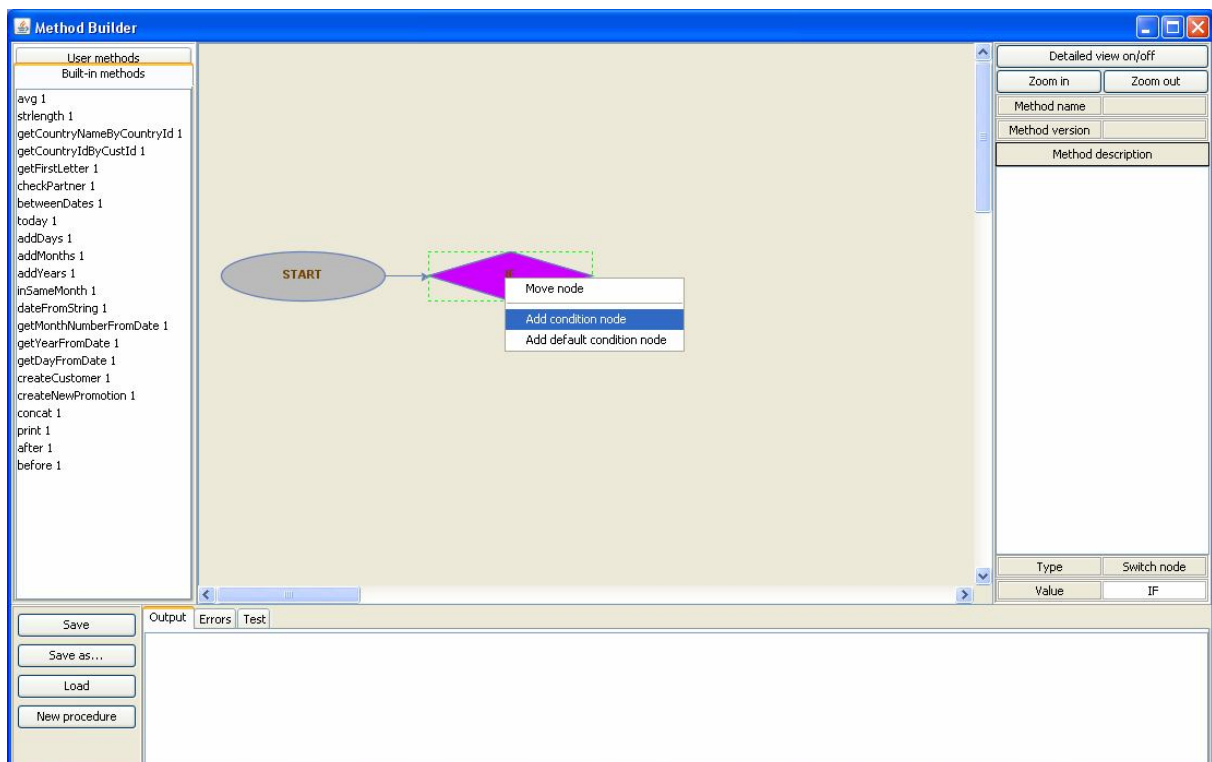


6. ábra Változó hozzáadása

A következő lépés egy elágazás létrehozása, ahol megvizsgáljuk, hogy az **x** értéke kisebb-e, mint 10. Ehhez hozzá kell adni egy **Switch** csúcsot, ami a **Start** csúcsához hasonlóan lehetséges. A **Start** csúcsból húzzunk egy élt a létrehozott **Switch** csúcsba, hogy a végrehajtás erre haladjon. Egy csúcsból úgy tudunk élt húzni egy másikba, hogy a csúcsra kattintunk a bal egérgombbal és lenyomva tartva az egérgombot az egeret a másik csúcs közepére húzzuk, majd pedig felengedjük az egérgombot.

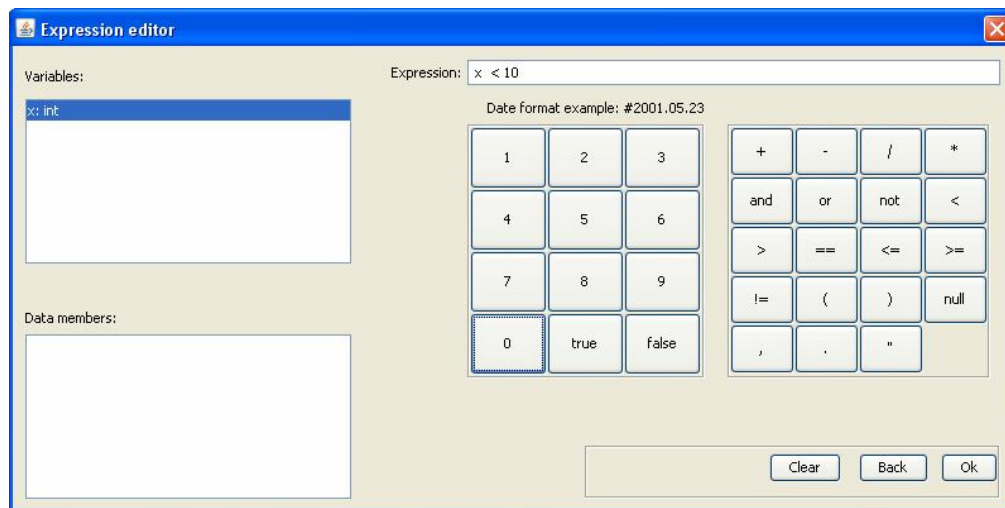
A **Switch** csúcsból feltételcsúcsokat ágaztathatunk ki. Ehhez a csúcsra kell kattintani jobb egérgombbal. Ekkor megjelenik egy felugró menü, ahol az **Add condition node** menüpontot választva egy feltételcsúcsot ad hozzá, az **Add default condition node** menüponttal pedig egy olyan feltétel ágat, ami akkor fut le, ha egyik feltételcsúcs sem tudott lefutni.

Adjunk hozzá egy feltételcsúcsot és egy default feltétel csúcsot is. A feltétel csúcsra duplán kattintva hozzuk elő a feltételcsúcs űrlapját, ahol a feltételt lehet szerkeszteni. A feltétel mellett lévő **Edit** gombra kattintva egy szerkesztő-ablak jelenik meg, ami kifejezések szerkesztésére való (**Expression Editor**).



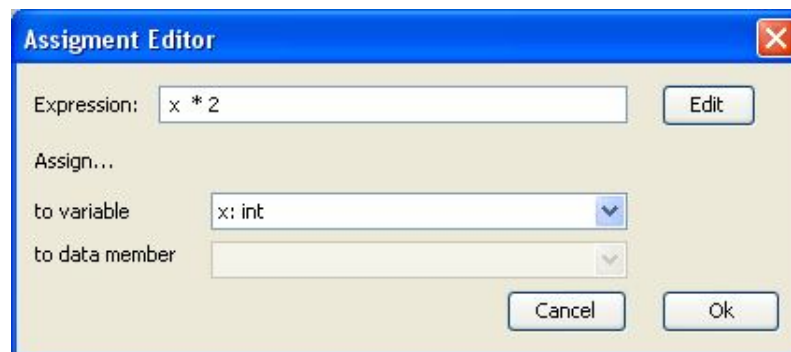
7. ábra Feltétel csúcs hozzáadása

Ezen a felületen bonyolultabb kifejezéseket tudunk összeszerkeszteni úgy, hogy közben nem szükséges a billentyűzetet használni. Szerkesszük össze az $x < 10$ kifejezést, majd kattintsunk az **Ok** gombra, amivel visszajutunk a feltétel csúcs űrlapjára, ahol ugyanígy az **Ok** gombbal elfogadjuk a szerkesztett kifejezést.



8. ábra A kifejezés-szerkesztő ablak

Ezután mindkét feltétel csúcs – az egyik default feltétel csúcs – alá adjunk hozzá egy-egy értékadás csúcsot. Ezt az ismert módon tehetjük meg, a felugró menüben az **Add assignment node** értéket kiválasztva. Az **x < 10** feltételt tartalmazó csúcs alatt lévő értékadás csúcs űrlapján ugyanúgy szerkeszthetjük meg a kifejezést, mint a feltétel csúcsén. Itt szerkesszük meg az **x * 2** kifejezést. Azt, hogy melyik változónak szeretnénk értéket adni, egy legördülő menü segítségével választhatjuk ki. Ott válasszuk ki az **x** változót. A végén az **Ok** gombbal visszatérhetünk a munkaterületre.

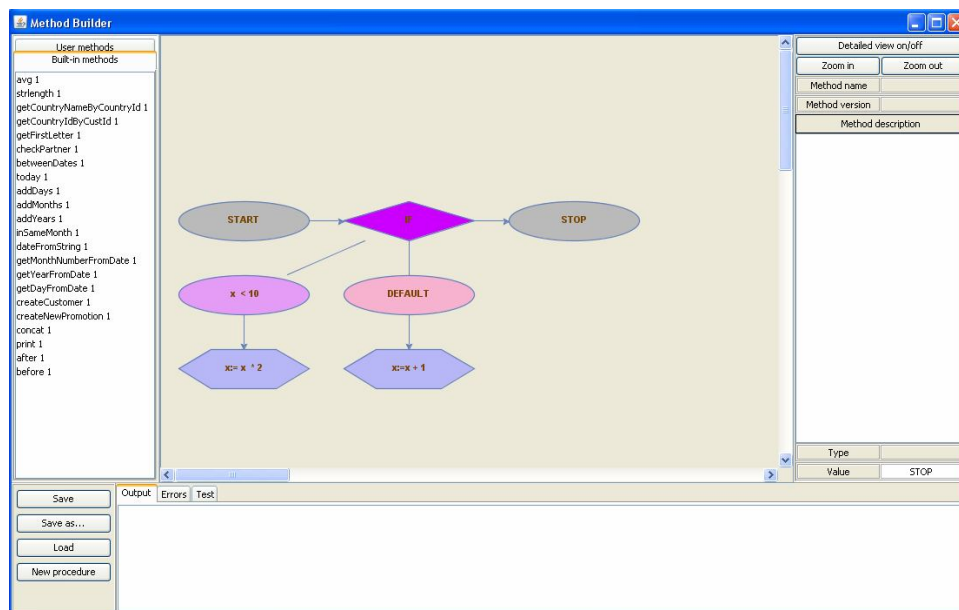


9. ábra Az értékadás-szerkesztő űrlap

A másik értékadásnál ugyanígy eljárva állítsuk be az **x + 1** kifejezést, amit ugyanígy az **x** változónak adunk értékül. Ezután húzzunk az értékadás csúcsokba egy-egy élt a megfelelő feltételcsúcsokból.

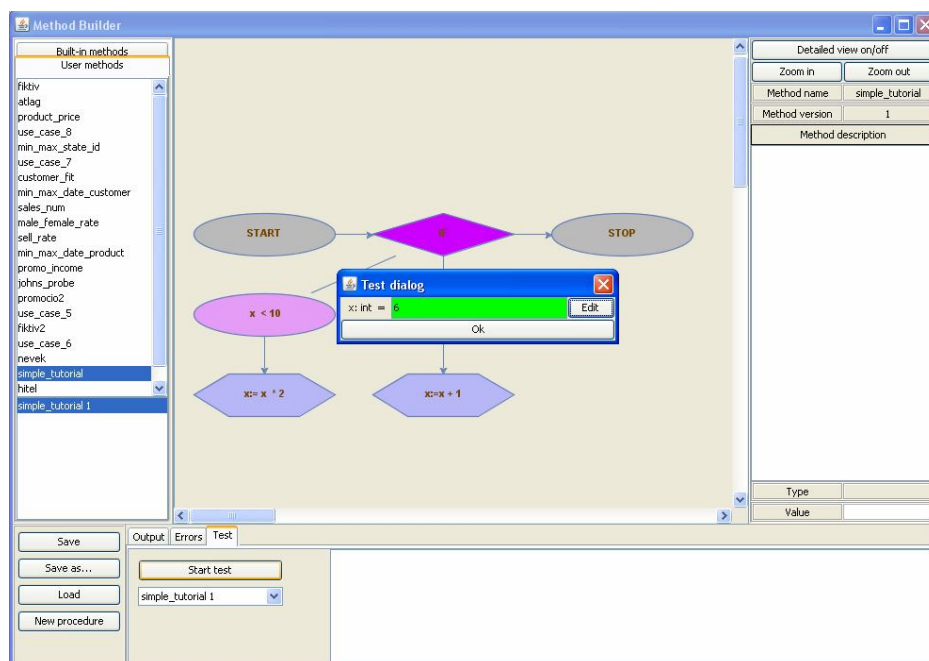
Végül illesszünk be egy **Stop** csúcsot és a **Switch** csúcsból húzzunk bele egy élt. Ekkor már csak a kimenő paramétereket kell beállítani, amit a **Stop** csúcs űrlapján tehetünk meg. Itt egy új paraméter hozzáadásához meg kell adni egy paraméternevet, majd pedig a legördülő menüből ki kell választani azt a változót, aminek az értékét a paraméter a végén

felveszi. A beállított paramétert az Add gombbal lehet ténylegesen hozzáadni a metódushoz. Az elkészült metódus a 10. ábrán látható.



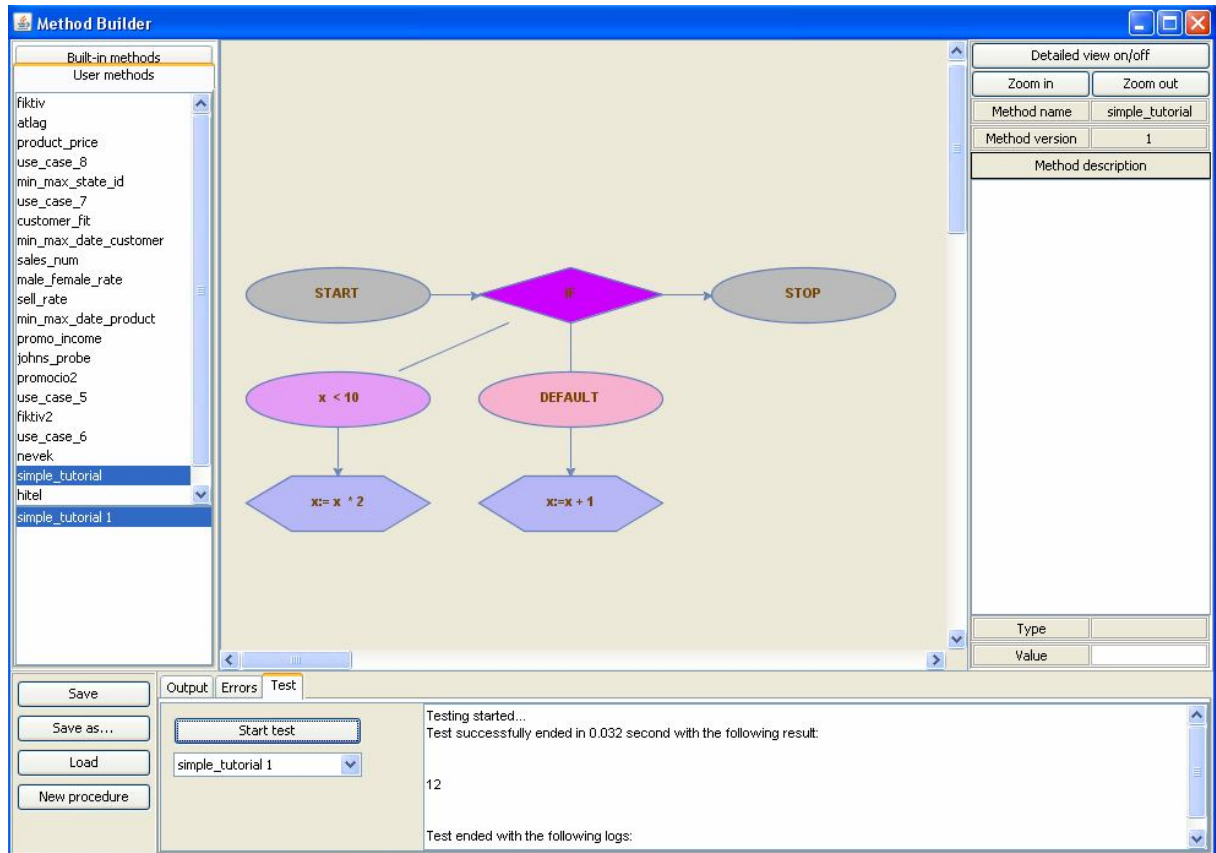
10. ábra Az elkészített metódus

Az elkészített metódusok teszteléséhez a felület alján lévő panel **Test** fülére kell kattintani, amire láthatóvá válik a teszt panel. Itt a legördülő menüből ki kell választani a tesztelni kívánt metódust (figyelni kell a verziószámra is), majd pedig a **Start test** gombbal tudjuk elindítani a tesztet. Ekkor megjelenik a bemenő paraméterek bevitelére szolgáló ablak, ahol meg kell adni a kívánt adatokat, majd az **Ok** gombra kattintva elindul a teszt.



11. ábra Tesztelés bemeneti adatainak a megadása

A teszt végeztével a tesztpanelen lévő szövegdobozban megjelenik a teszt eredménye.



12. ábra A teszt eredménye

Szerver oldali funkciók részletezése

A szerver három komponensből áll össze:

- **Repository** – Metódusok és adatstruktúrák leíróinak a tárolására
- **Compiler** – A metódusok leíróiból java kód generálása és annak lefordítása
- **Loader** – Külső adatstruktúrák, metódusok leíróinak a betöltése a rendszerbe

Ezek a komponensek különböző XML sémákat használnak a feladatuk végrehajtására.

2.3.3 Repository modell

A Repository feladata a metódusok és adatstruktúrák leíróinak a tárolása. Ehhez különböző XML sémákat hoztunk létre, amiket ebben az alfejezetben foglalunk össze. Ahogyan a Rendszer felépítésénél már volt róla szó, a tárolást és verziókezelést egy kész rendszer látja el, a **Java Content Repository**.

2.3.3.1 Komplex típus

A komplex típusok és struktúrák leírására szolgáló séma részleteit az alábbi két táblázat foglalja össze.

2.3.3.1.1 Elemek

A komplex típusokat leíró sémában a következő elemek szerepelnek.

Elem	Leírás
Class	Beépített típus esetén a típushoz tartozó osztály nevét tárolja. Felhasználói típusoknál nem kötelező kiírni, ekkor a generate.types.<name> az alapértelmezett, ahol a <name> a típus neve.
Properties	A típus mezőit fogja össze.
Property	A típus egy mezőjének a leírása.

2. táblázat A komplex típusokat leíró séma elemei

A **beépített típus** az a típus, amit a rendszerbe kívülről olvasunk be, a **felhasználói típus** pedig az, amit a rendszeren belül, a felületen hozunk létre.

2.3.3.1.2 Attribútumok

A komplex típusokat leíró sémában a következő attribútumok szerepelnek.

Attribútum	Típus	Leírás
Complex		
Name	Xsd:string	A típus neve, ami megjelenik a felhasználói felületen

Attribútum	Típus	Leírás
Property		
Name	Xsd:string	A mező neve.
Type	Xsd:string	A mező típusa. (A típushoz tartozó osztály.)
Getter	Xsd:string	Amennyiben az adattaghoz nem a default getter metódus (<code><mező típusa> get<Mező neve>()</code>) tartozik, akkor itt adhatjuk meg.
Setter	Xsd:string	Amennyiben az adattaghoz nem a default setter metódus (<code>void set<Mező neve>(<mező típusa> value)</code>) tartozik, akkor itt adhatjuk meg.

3. táblázat A komplex típusokat leíró séma elemei

2.3.3.2 Datasource

A **Datasource** az adatbázis-elérésből származó adatok leírására alkalmas struktúra. A leírójának a részleteit az alábbi táblázatok tartalmazzák.

A **Datasource** nem más, mint egy speciális metódus, így a leírónak a metódus helyét – melyik osztályban van -, nevét és típusát kell tárolnia. A **Datasource** metódusról bővebben a 2.4.1.8 pontban lehet olvasni.

2.3.3.2.1 Elemek

A leíró sémájának az elemei a következő táblázatban találhatók.

Elem	Leírás
Class	A datasource metódust tartalmazó osztály neve.
Method	A datasource metódus neve.
Type	A datasource metódus visszatérési kollekciójába tartozó elemek típusa. (A típus neve kell és nem az osztálya.)

4. táblázat A Datasource metódust leíró séma elemei

2.3.3.2.2 Attribútumok

A leíró sémájában található attribútumok a következő táblázatban vannak összefoglalva.

Attribútum	Típus	Leírás
Datasource		
Name	Xsd:string	A datasource neve, ami megjelenik a felhasználói felületen

Attribútum	Típus	Leírás
Method		
Static	Xsd:string	Megmondja, hogy a datasource metódus statikus-e. Ennek a metódushíváskor lesz jelentősége. Az attribútum értéke rögzített, csak a jelenléte fontos.

5. táblázat A Datasource metódust leíró sémában található attribútumok

2.3.3.3 Metódusok - Method

Az eljárások és algoritmusok leírására szolgáló XML séma részleteit az alábbi táblázatok tartalmazzák. Az **id** attribútum minden elem esetén egy a fájlban belül globálisan egyértelmű azonosítót jelent. A típusa `xsd:int`. Egy metódus leírója mindig tartalmazza a nevét, valamint a ki- és bemeneti paramétereit. Ezen kívül, ha a metódust a felületen szerkesztettük össze, akkor a metódus építőelemét is tartalmazza.

2.3.3.3.1 Elemek

A metódus leírójának a sémájának az elemeit az alábbi táblázat foglalja össze.

Elem	Leírás
Description	A metódushoz rövid leírást rendelhetünk, mely megjelenik a felhasználói felületen.
Inputs	A metódus formális input paramétereit tartalmazza.
Outputs	A metódus formális output paramétereit tartalmazza.
Var	Egy változó leírása.
BuiltIn	Beépített függvények esetén jelenik meg, a függvény használatához szükséges információkat tartalmazza.
Class	Beépített függvények esetén a tartalmazó osztály nevét adja meg.
Body	A felhasználó által létrehozott metódus elemeinek a reprezentációja.
Defsize	A szerkesztőfelületen megjelenő elemek mérete.
Sources	A datasource csúcsokat fogja össze.
Source	Egy datasource csúcs leírása
Statements	A szerkesztőfelületen létrehozott nem datasource csúcsok leírása.
Start	A start csúcs.
Coord	Egy adott csúcs koordinátáit tartalmazza. Ez alapján lesz elhelyezve a csúcs a szerkesztőfelületen.
Variables	A start csúcson belül helyezkedik el. A metódus során használt, metódus szintjén globális változókat fogja össze.
Stop	Stop csúcs.
Output	A stop vagy a function csúcson belül helyezkedik el. A metódus formális outputjához rendelünk egy változót, ez lesz a paraméter aktuális értéke. Jelenleg a megfeleltetés pozíció alapján történik. (Azaz az outputs elemekben található változók sorrendje megegyezik a stop elemekben található output elemek sorrendjével.)
Assignment	Értékkadás csúcs.

Elem	Leírás
Expression	Egy kifejezés leírására szolgál. A felhasználói felületen egy külön szerkesztő ablakban adhatók meg a kifejezések.
Foreach	A datasource-ok bejárását segítő ciklus.
loopVar	Egy ciklus ciklusváltozóját írja le.
Statement	Egy utasításblokk kezdete. Elhelyezhető a for, foreach, case, default elemeken belül.
For	For ciklus leírására
startExpr	Az Expression elemhez hasonlóan működik, speciális a for elemben a kezdeti kifejezés megadásához.
Inc	A for ciklus lépésköze.
StopCond	Az Expression elemhez hasonlóan működik, speciális a for elemben a leállási feltétel megadásához.
Switch	Feltételes elágazás leírásához.
Case	A switch elemen belül egy feltétel megadása. Ha a feltétel teljesül, akkor a statement elemmel folytatjuk a feldolgozást, különben a következő case ágra ugrunk (, esetleg a default ra, ha nincs több case).
Default	A switch elágazás default ágát adhatjuk meg. Ha egyik feltétel sem teljesül, akkor ugrunk ide.
Log	Speciális üzenetek megadására szolgál. A log csúcsok üzenetei egy listába kerülnek, mely egy speciális kimeneti változóként továbbadódik a hívónak.
Message	A log elemhez tartozó üzenet.
Function	Függvényhívás leírására. Mind beépített mind a felhasználó által létrehozott függvény hívható-
Input	A function elemen belül a hívott függvény formális input listájához rendelünk aktuális paramétereket. (A hozzárendelés pozíció alapján történik, azaz a hívott függvény inputs elemében található var elemek és a function elem input elemeinek sorrendjének meg kell egyezniük.)
Steps	A csúcsok egymásutániségát leíró elemek összefogására.
Step	Megadja, hogy az adott csúcs után melyik csúcs következzen. Lásd még a statement elemet.

6. táblázat A metódusokat leíró séma elemei

2.3.3.3.2 Attribútumok

Az **id** attribútum minden elem esetén egy, a fájlban belül globálisan egyértelmű azonosítót jelent. A típusa xsd:int.

Attribútum	Típus	Leírás
Method		
Name	Xsd:string	A metódus neve.
Var		
Name	Xsd:string	A változó neve.
Type	Xsd:string	A változó típusa. (Komplex típusnál a típus neve és nem az osztálya.)

Attribútum	Típus	Leírás
Var		
complex	Xsd:string	Komplex változók esetén kell megadni. Az értéke rögzített, csak a jelenlét a fontos.
Foreach	Xsd:string	Csak a start elem variables elemei között van jelentőség. Jelzi, hogy az adott változó egy foreach ciklus ciklusváltozója. A kódgenerálás szempontjából fontos.
builtin		
static	Xsd:string	Megmondja, hogy a datasource metódus statikus-e. Ennek a metódushíváskor lesz jelentősége. Az attribútum értéke rögzített, csak a jelenléte fontos.
Defsize		
Height	Xsd:int	A csúcsok magassága.
Width	Xsd:int	A csúcsok szélessége.
Coord		
X	Xsd:int	X koordináta
Y	Xsd:int	Y koordináta
Source		
Name	Xsd:string	Az adatforrás neve
Version	Xsd:int	Az adatforrás verziója.
Output		
Name	Xsd:string	A formális input paraméter neve.
Var	Xsd:string	A változó neve.
Complex	Xsd:string	Komplex típusú változó esetén jelezni kell, hogy komplex. (ld Var -nál)
Member	Xsd:string	Amennyiben egy komplex típusú változó valamelyik mezőjére hivatkozunk, akkor azt itt adjuk meg.
Assignment		
Variable	Xsd:string	A változó neve, melyhez értéket akarunk rendelni.
Member	Xsd:string	Komplex típusú változó esetén lehet adattag is.
Foreach		
sourceRef	Xsd:int	A datasource azonosítója, melyet a ciklussal bejárunk.
Statement		
Ref	Xsd:int	Egy utasításblokk első utasítására hivatkozik. Az utasításhoz tartozó csúcs azonosítóját kell megadni.
Function		
Name	Xsd:string	A meghívandó függvény neve.
Version	Xsd:int	A függvény verziószáma.
builtin	Xsd:string	Jelenléte jelzi, hogy beépített függvény hívásáról van szó. Az attribútum értéke fix.
Input		
Az attribútumok megegyeznek az output elem attribútumaival		
Step		
from	Xsd:int	Melyik csúcs után
to	Xsd:int	Melyik csúcs következik.

7. táblázat A metódusokat leíró séma attribútumai

2.3.4 Kódgenerálás - Compiler

A kódgenerálás során az elemek XML reprezentációjából futtatható *Java* fájl készül.

Az **EJBReflection** projektben található **codegenerator.CodeGenerator** osztály felel a kódgenerálásért.

- Komplex típusok fordítása:

```
org.w3c.dom.Document xml;  
...  
codegenerator.CodeGenerator.generateComplex(xml)
```

- Metódusok fordítása:

```
org.w3c.dom.Document xml;  
int version  
...  
(new codegenerator.CodeGenerator()).generate(xml,  
version)
```

2.3.4.1 Komplex típusok

A felhasználó által létrehozott komplex típusokat reprezentáló XML-ekből a következők alapján generálódik *Java* osztály:

```
Package generate.types;  
Public class <class.name> implements java.io.Serializable {  
    //A mező  
    private <properties.property[0].type> <properties.property[0].name>;  
    //Setter  
    public void  
    set<properties.property[0].Name>(<properties.property[0].type>  
    <properties.property[0].name>) {  
        this.<properties.property[0].name> = <properties.property[0].name>;  
    }  
    //Getter  
    Public <properties.property[0].type>  
    get<properties.property[0].Name>() {  
        Return <properties.property[0].name>;  
    }  
  
    //A többi mező feldolgozása  
}
```

9. kódrészlet: Komplex típusokból generált kód

A komplex típusból egy, a típus nevével megegyező nevű *szerializálható* osztály generálódik, aminek az adattagjai a típus adattagjai, ezen kívül pedig minden adattaghoz generálódik egy *getter* és egy *setter* metódus is.

A kiemelt elemek az XML-ből vannak olvasva:

- `<Elem>'<attribútum>` alapján;
- `['<sorszám>']`, az azonos elemek megkülönböztetésére szolgál.

Amennyiben valamelyik property elemnél meg van adva a *getter* vagy a *setter* attribútum, úgy a megfelelő metódus neve módosul.

Példa:

```
<complex name="Komplex">
  <properties>
    <property name="value" type="int"/>
    <property name="name" type="String"/>
  </properties>
</complex>
```

10. kódrészlet: a Komplex nevű típust leíró xml

```
package generate.types;

public class Komplex implements java.io.Serializable {
    private java.lang.Integer value;
    public void setValue(java.lang.Integer value){
        this.value=value;
    }
    public java.lang.Integer getValue(){
        return value;
    }

    private java.lang.String name;
    public void setName(java.lang.String name){
        this.name=name;
    }
    public java.lang.String getName(){
        return name;
    }
}
```

11. kódrészlet: a Komplex nevű típushoz generált kód

2.3.4.2 Method

A felhasználó által létrehozott metódusokat reprezentáló XML-ekből a következő szabályok alapján generálódik *Java* kód:

1. Az osztály fejléce:

```
Package generate;  
Public class <method.name>v<version>{  
    Public static Object[] <method.name>(  
        <inputs.var[0].type> <inputs.var[0].name>,  
        //...  
    ) throws java.lang.Exception{  
        Java.util.List<String> _log = new java.util.ArrayList<String>();  
        Java.util.DateFormat _dateformat = new  
        java.text.SimpleDateFormat("yyyy.mm.dd");  
        //..
```

12. kódrészlet: a generált osztály fejléce

A félkövér elemek az XML-ből vannak olvasva.

A **<version>** elem pedig a **repository**-ből.

Az osztály neve a következő: *<metódusnév>v<verzió>*

A generált osztályba egyetlen statikus metódus generálódik, aminek a neve a *metódusnév*, a paraméterei pedig a megfelelő bemeneti paraméterek. A statikus metódus visszatérési értéke egy **Object[]**, ebbe kerül bele a metódus végeredménye. A végeredmény utolsó eleme mindig a hibaüzenetek listája.

Minden esetben szükséges a hiba- és log-üzenetek tárolására egy szöveg lista (**_log**), valamint a dátumok kezelésére egy dátumformátum (**_dateformat**), így ezeket is a metódus elején hozzuk létre.

2. **start** csúcsban található változók deklarálása, inicializálása

A változókat a metódus elején deklarálja, alapvetően a következő kód generálódik minden változóhoz:

```
<var.type> <var.name>;
```

13. kódrészlet: Változó deklaráció

Speciális esetek:

- Ha a **complex** attribútum jelen van, akkor a típus a komplex típushoz tartozó osztály lesz.
- Ha a **foreach** attribútum jelen van, akkor a változó ciklusváltozó, ezért valamely **foreach** ciklusra nézve lokális lesz, tehát nem itt kell deklarálni.
- Néhány alaptípushoz kezdőérték tartozik:
 - **Java.lang.Integer**: 0
 - **Java.lang.Double**: 0.0
 - **Java.lang.Boolean**: false

3. A **start** csúcsot követő elem feldolgozása

A soron következő elemet a megfelelő **step tag**-ból tudjuk kiolvasni. (Amelyiknek a **from** attribútumának az értéke megegyezik a **start** csúcs azonosítójával. A **to** attribútum értéke tartalmazza a következő csúcs azonosítóját)

4. Az egyes elemek feldolgozása

A táblázat bal oldalán az XML részlet, a jobb oldalon pedig a belőle generált *Java* kód található.

Xml	Java kód
Assignment	
<pre><assignment id="6" variable="counter"> <coord x="1" y="2"/> <expression>0</expression> </assignment></pre>	<pre>counter = 0;</pre>
Amennyiben meg van adva a member attribútum, úgy az adott változóra a memberhez tartozó setter metódus hívódik meg.	
Customer.xml <pre><complex name="Customer"> <class>utils.Customer</class> <properties> <property name="age" type="int"/> </properties> </complex></pre> Method.xml <pre>... <assignment id="6" variable="cust" member="age"> <coord x="1" y="2"/> <expression>0</expression> </assignment> ...</pre>	<pre>cust.setAge(0);</pre>

Xml	Java kód
Foreach	
Sale.xml <complex name="Sale"> <class>utils.Sale</class> <properties> <property name="xy" type="int"/> </properties> </complex> Sales.xml: <datasource name="Sales"> <class>utils.Sale</class> <method static="yes">getAll</method> <type>Sale</type> </datasource> Method.xml: ... <source id="5" name="Sales" version="1"/> ... <foreach id="7" sourceRef="5"> <coord x="2" y="2"/> <loopVar>row</loopVar> <statement ref="12"/> </foreach> ...	for(utils.Sale row: utils.Sale.getAll()){ //12-es csúccsal kezdődő blokk feldolgozása } Amennyiben a Sales.xml fájlban a method elemnél nem szerepelne a static attribútum, akkor létre kellene hozni egy példányt, és arra meghívni a metódust: for(utils.Sale row: (new utils.Sale()).getAll()){ //... }
For	
<for id="13"> <coord x="2" y="1"/> <loopVar>i</loopVar> <startExpr> min </startExpr> <inc>1</inc> <stopCond>i <= max </stopCond> <statement ref="25"/> </for>	Integer i = 0; Integer min = 0; Integer max = 0; ... for(i = min; i <= max; i = i + 1){ //25-ös csúccsal kezdődő blokk feldolgozása }
Switch	
<switch id="8"> <coord x="2" y="3"/> <case id="9"> <condition>tmp == country</condition> <statement ref="14"/> </case> <case id="9"> <condition>tmp == country2</condition> <statement ref="15"/> </case> <default> <statement ref="16"/> </default> </switch>	if (tmp == country){ //14-es csúccsal kezdődő blokk feldolgozása }else if (tmp == country2){ //15-ös csúccsal kezdődő blokk feldolgozása }else{ //16-os csúccsal kezdődő blokk feldolgozása } A default elem megadása nem kötelező, ekkor az utolsó else ág nem generálódik.
Log	
<log id="17"> <coord x="2" y="6"/> <message>Vélhetően rögzítési hiba.</message> </log>	_log.add("Vélhetően rögzítési hiba");

Xml	Java kód
Function	
Beépített függvény hívása	
GetFirstLetter.xml <pre> <method name="getFirstLetter"> <inputs> <var type="java.lang.String"/> </inputs> <outputs> <var name="result" type="char"/> </outputs> <builtin static="yes"> <class>utils.Utils</class> </builtin> </method> </pre> Method.xml <pre> ... <function builtin="yes" id="12" name="getFirstLetter" version="1"> <coord x="2" y="4"/> <input complex="complex" member="first_name" name="arg0" var="row"/> <output name="result" var="letter"/> </function> ... </pre>	<pre> Java.lang.Character letter; ... letter = utils.Util.getFirstLetter(row.getFirst_name()); </pre> Megjegyzés: A row egy komplex változó, a first_name mezőjéhez jelen esetben a getFirst_name() getter metódus tartozik. Ha az output oldalon állna, akkor a setter metódus hívódna meg. Ha a getFirstLetter.xml-ben nem szerepelne a builtin elemnél a static="yes" attribútum, akkor példányosítani kéne az osztályt: <pre> letter = (new util.Util()).getFirstLetter(row.getFirst_name()); </pre>
Felhasználói függvény hívása	
Novel.xml <pre> <method name="novel"> <description/> <inputs> <var name="x" type="java.lang.Integer"/> </inputs> <outputs> <var name="result" type="java.lang.Integer"/> </outputs> ... </method> </pre> Method.xml <pre> ... <function id="3" name="novel" version="1"> <coord x="0" y="2"/> <input name="x" var="inp"/> <output name="result" var="inp"/> </function> ... </pre>	<pre> { Object [] os = generate.novelv1.novel(inp); inp= (java.lang.Integer)os[0]; _log.addAll((java.util.List<java.lang.String>)os[1]); } </pre> Megjegyzés: A novel egy felhasználói függvény, ezért statikus, és Object [] a visszatérési értéke. A tömbből kell kiválogatni és cast-olni a megfelelő elemeket. A tömb utolsó eleme mindig a log üzeneteket tartalmazó lista.

8. táblázat A metódust leíró xml elemeiből történő java kód-generálás

5. A metódus visszatérési értékének és az osztály láblécének generálása

```
{
    Object[] _result = new Object[<formális output-ok száma + 1>];
    _result[<i>] = <stop.output.var[i>;
    // ...
    _result[<last>] = _log;
}
} //metódus
} //osztály
```

14. kódrészlet: Visszatérési érték és lezárás

A metódus visszatérési értéke egy `Object[]` (`_result`), amibe a metódusban meghatározott sorrendben kerülnek bele a metódus kimeneti paramétereinek az értékei. Ezen felül az utolsó eleme mindig a hibaüzenetek listája.

2.3.5 Expression parser

Az XML leíróban lévő kifejezéseket át kell alakítani, hogy szabályos *Java* kód keletkezzen belőlük, mivel a leíróban olyan formában vannak benne, ahogyan a kliens felületén szerepelnek.

Az átalakítás során feltételezzük, hogy a kifejezés elemei *space* karakterrel el vannak választva egymástól.

2.3.5.1 Logikai műveletek

A logikai műveletek az olvashatóság miatt teljesen ki vannak írva, ám a *Java*-ban külön operátorok tartoznak hozzájuk:

- And - &&
- Or - ||
- Not - !

2.3.5.2 Dátum

A szöveges formában bevitt dátumot a `java.util.Date` osztály példányává kell konvertálni. Ezért a dátumot '`#yyyy.mm.dd`' formátumban kell megadni, ahol

- yyyy – az évszám 4 karakter hosszan
- mm – a hónap 2 karakter hosszan

- dd – a nap 2 karakter hosszan

Példák:

- #1998.02.01
- #2012.03.10

2.3.5.3 Egyenlőség vizsgálat

A felületen az értékek összehasonlítását a `==` operátor jelzi. Azonban *Java*-ban a `==` operátor a sekély összehasonlítást jelenti, vagyis objektumok esetén a pointerek egyezőségét vizsgálja. Ez a működés számunkra nem jó, ezért, amennyiben a `==` operátor valamely oldalán egy objektum szerepel, akkor le kell cserélni a mély összehasonlítást végző `equals` műveletre.

Példák:

Előtte	Utána								
Konstansok									
<code>2 == 3</code>	<code>2 == 3</code>								
<code>"alma" == "x"</code>	<code>"alma".equals("x")</code>								
Egyszerű változók									
Az összes egyszerű, azaz nem komplex típusú változó objektum típusú, ezért ilyen esetben mindig szükséges az átalakítás. Tegyük fel, hogy a felhasználói felületen a start csúcsban az alábbiakat állítottuk be:									
Variables: <table> <thead> <tr> <th>Name</th><th>Type</th></tr> </thead> <tbody> <tr> <td>x</td><td>int</td></tr> <tr> <td>y</td><td>int</td></tr> <tr> <td>a</td><td>string</td></tr> </tbody> </table>		Name	Type	x	int	y	int	a	string
Name	Type								
x	int								
y	int								
a	string								
Előtte	Utána								
Egyszerű változók									
<code>x == 5</code>	<code>x.equals(5)</code>								
<code>5 == y</code>	<code>y.equals(5)</code>								
<code>x == y</code>	<code>x.equals(y)</code>								
<code>"alma" == s</code>	<code>"alma".equals(s)</code>								
<code>s == "alma"</code>	<code>"s.equals("alma")</code>								
Komplex változók									
Komplex változók összehasonlítása esetén minden esetben szükséges a csere.									
Komplex változók adattagjai									
Amennyiben egy komplex változónak valamely adattagja primitív típusú, és azt egy szintén primitív típusú értékkel hasonlítjuk össze, úgy nem szükséges a csere. Minden más esetben csere történik.									

9. táblázat Példák az egyenlőségjelek átalakítására

2.4 Felolvasás - Loader

A felolvasáson azt a tevékenységet kell érteni, amikor valamilyen *Java* kódból előállítjuk a kódhoz tartozó XML reprezentációt, melyre az alkalmazásnak szüksége van.

A felolvasásért a **reflection** projektben található **reflection.Reflection** osztály a felelős.

Fontos, hogy az osztály hívásakor:

- a felolvasni kívánt osztály,
- annak függőségei,
- és a következő jar fájlok:
 - **appserv-rt.jar**
 - **gf-client.jar**
 - **javaee.jar**
 - **jndi-properties.jar**

a **classpath**-ban legyenek.

Ekkor a hívás az alábbiak szerint történhet:

```
reflecion.jar <className> <classpath>: process a class file
reflecion.jar -dir <dir> [<classpath>]: process dir folder, which
contains class files
reflecion.jar -jar <jar> [<classpath>]: process a compressed file, which
contains class files
```

15. kódrészlet: A felolvasás hívása parancssorból

2.4.1 Fogalmak

2.4.1.1 Alaptípus

Az alkalmazás az alábbi típusokat kezeli alaptípusként:

int, double, char, boolean, java.lang.Integer, java.lang.Double,
java.lang.Character, java.lang.Boolean, java.util.Data,
java.lang.String.

2.4.1.2 Ismert típus

Ismert típusok a következők:

- Minden alaptípus ismert típus.

- Azok a típusok, melyeket az **EntityMarker** annotációval megjelölt osztályokból olvasunk fel.

2.4.1.3 Annotációk

A felolvasandó beépített elemek megjelölésére az **EJBClient** project **annotation** csomagjában található annotációk használhatók. Mindegyik annotáció **Runtime** idejű, vagyis a fordítás során megmaradnak.

A beépített elemek felolvasása a **Java reflection** eszközével történik, vagyis az osztályokat be kell olvasni a *Java Virtuális Géphe* az elemzéshez.

2.4.1.4 MethodMarker

Az annotáció a beépített függvények megjelölésére szolgál. Csak olyan metódusok jelölhetők meg a segítségével, melyek teljesítik az alábbi feltételeket:

- A metódusnak publikusnak kell lennie.
- A metódus nem lehet absztrakt.
- Az input paramétereknek *ismert típusúnak* kell lenniük.
- A visszatérési érték *ismert típusú* vagy *void* lehet.
- A befoglaló osztály publikus és nem absztrakt.
- Ha a metódus nem statikus, akkor a befoglaló osztályban lennie kell egy publikus paraméter nélküli konstruktornak. (Hiszen ilyenkor a hívás előtt példányosítani kell az osztályt.)

2.4.1.5 Parameter

A **MethodMarker** osztállyal annotált metódusok paramétereit jelölhetjük meg az annotációval.

A felhasználói felületen megjelenő nevet rendelhetjük a paraméterhez a *value* megadásával.

Ha ezt nem tesszük meg, akkor a paraméter alapértelmezett neve

arg<sorszám>

alakú lesz, ahol a **<sorszám>** a metódus paraméterlistájában elfoglalt pozíciót jelenti.

2.4.1.6 EntityMarker

Az annotációval azokat az osztályokat jelölhetjük meg, melyekből beépített típust szeretnénk létrehozni.

A típushoz a felhasználói felületen megjelenő nevet rendelhetünk a `value` megadásával. Ha ezt nem tesszük meg, akkor az osztály (egyszerű) neve lesz a típus neve.

Figyelni kell, hogy a név globálisan egyértelmű legyen az alkalmazásban!

A megjelölhető osztályokra vonatkozó megszorítások:

- Az osztálynak publikusnak kell lennie.
- Az osztály nem lehet absztrakt.
- Az osztályban kell lennie egy paraméter nélküli publikus konstruktornak.

2.4.1.7 FieldMarker

Az **EntityMarker**-rel annotált osztályok adattagjait jelölhetjük meg.

A **FieldMarker**-rel annotált mezők lesznek az osztályból képzett komplex típus adattagjai.

A mezőhöz tartozó *getter* metódus nevét adhatjuk meg a *getter* taggal, valamint a mezőhöz tartozó *setter* metódus nevét a *setter* taggal.

Ha ezt nem tesszük meg, akkor a *Java* névkonvenciója szerinti metódusok lesznek hozzárendelve a mezőhöz:

- `'get' + <mező neve nagy kezdőbetűvel>`
- `'set' + <mező neve nagy kezdőbetűvel>`

A megjelölhető mezőkre vonatkozó megszorítások:

- A mező csak alaptípusú lehet.
- A megadott *getter*-re:
 - A visszatérési érték típusa megegyezik a mező típusával (vagy annak leszármazottja is lehet)
 - Paraméter nélküli.
- A megadott *setter*-re:
 - A visszatérési érték `void`
 - Pontosan egy paramétere van
 - A paraméter típusa megegyezik a mező típusával (vagy annak őse)

2.4.1.8 DataSourceMarker

Úgynevezett **datasource**-ok annotálására szolgál az osztály.

A felületen megjelenő nevet a *name* taggal adhatjuk meg, a **DataSource** elemeinek a típusát pedig az *entity* taggal.

A **datasource**-ok olyan metódusok, melyek megfelelnek a következő feltételeknek:

- A metódus publikus.
- A metódus nem absztrakt.
- A metódus paraméter nélküli.
- A visszatérési érték egy olyan *kollekció*, mely bejárható a
 - **for(type e : getAll())** szerkezettel.
- A *kollekció* elemei olyan ismert típusok lehetnek, melyek nem alaptípusok. Vagyis az *entity* tagban megadott osztálynak fel kell lennie annotálva az **EntityMarker** annotációval.
- Ha a metódus nem statikus, akkor a befoglaló osztályban lennie kell egy publikus paraméter nélküli konstruktornak. (Hiszen ilyenkor a hívás előtt példányosítani kell az osztályt.)

2.4.2 Beépített metódus felolvasása

A következő táblázatban át lehet tekinteni, hogy milyen leíró generálódik a felannotált metódusokból. Bal oldalon a példa eljárások találhatóak, a jobb oldalon pedig a belőlük generált leírók.

Java kód	Xml
Egyszerű metódus felolvasása.	
<pre>Package utils; Public class Utils{ @MethodMarker public char getFirstLetter(String word) { return word.charAt(0); } }</pre>	<pre><method name="getFirstLetter"> <inputs> <var name="arg0" type="java.lang.String"/> </inputs> <outputs> <var name="result" type="char"/> </outputs> <builtIn> <class>utils.Utils</class> </builtIn> </method></pre>
Statikus metódus felolvasása	
<pre>@MethodMarker public static char getFirstLetter(String word) { return word.charAt(0); }</pre>	<pre><method name="getFirstLetter"> <inputs> <var name="arg0" type="java.lang.String"/> </inputs> <outputs> <var name="result" type="char"/> </outputs> <builtIn static="yes"> <class>utils.Utils</class> </builtIn> </method></pre>

Java kód	Xml
Többparaméteres metódus felolvasása	
<pre>import java.util.Date; ... @MethodMarker public Date addYears(Date date, int years){ Calendar cal = Calendar.getInstance(); cal.setTime(date); cal.add(Calendar.YEAR, years); return cal.getTime(); }</pre>	<pre><method name="addYears"> <inputs> <var name="arg0" type="java.util.Date"/> <var name="arg1" type="int"/> </inputs> <outputs> <var name="result" type="java.util.Date"/> </outputs> <builtIn> <class>utils.Utils</class> </builtIn> </method></pre>
Paraméter elnevezése	
<pre>Package reflectiontest; Public class ReflectionTest{ @MethodMarker public static void sout(@Parameter("out") String s){ System.out.println(s); } }</pre>	<pre><method name="sout"> <inputs> <var name="out" type="java.lang.String"/> </inputs> <outputs/> <builtIn static="yes"> <class>reflectiontest.ReflectionTest</class> </builtIn> </method></pre>

10. táblázat Metódusok felolvasása, a generált leírók

2.4.3 Komplex típus felolvasása

A következő táblázatban át lehet tekinteni, hogy milyen leíró generálódik a felannotált komplex típusokból. Kétféle eset lehetséges. Az egyik, amikor az adattagok *getter* és *setter* metódusai illeszkednek a *get<adattag-név>* és *set<adattag-név>* mintára. A másik, amikor ettől eltérő a nevük, ekkor az annotációban ezt külön meg kell jelölni a **getter** és a **setter** paraméterekkel.

Bal oldalon a példa típusok találhatóak, a jobb oldalon pedig a belőlük generált leírók.

Java kód	Xml
Egyszerű felolvasás	
<pre>package reflectiontest; ... @EntityMarker("Ent") public class Entity { @FieldMarker private int x; public void setX(int x){ System.out.println("setX: " + x); } public int getX(){ return x; } }</pre>	<pre><complex name="Ent"> <class>reflectiontest.Entity</class> <properties> <property name="x" type="int"/> </properties> </complex></pre>

Java kód	Xml
Getter, Setter módosítás	
<pre>package reflectiontest; ... @EntityMarker("Ent") public class Entity { @FieldMarker(getter="gx", setter="sx") private int x; public void sx(int x){ System.out.println("setX: " + x); } public int gx(){ return x; } }</pre>	<pre><complex name="Ent"> <class>reflectiontest.Entity</class> <properties> <property name="x" type="int" getter="gx" setter="sx" /> </properties> </complex></pre>

11. táblázat A komplex típusok felolvasása, a generált leírók

2.4.4 Datasource felolvasása

A **DatasourceMarker**-rel megannotált metódusokból a következő táblázatban látható módon generálódik leíró.

Java kód	Xml
Egyszerű felolvasás	
Customer.java <pre>@EntityMarker public class Customer { ... }</pre> Utils.java <pre>... @DataSourceMarker(entity=Customer.class,name="Customers") public static List<Customer> getAll(){ ...} ...</pre>	Customers.xml <pre><datasource name="Customers"> <class>utils.Utils</class> <method static="yes">getAll</method> <type>Customer</type> </datasource></pre> Customer.xml <pre><complex name="Customer"> ... </complex></pre>

12. táblázat A Datasource metódusok felolvasása, a generált leírók

2.5 A generált metódusok futtatása

A futtatásnak az a problémája, hogy a lefordított állományt képesnek kell lenni azonnal, futási időben újraindítás nélkül lefuttatni. Erre a megoldást a **Java Reflection** csomag nyújtja.

Mivel a generált állományok felépítése olyan, hogy minden megszerkesztett eljárást egy külön osztályba generál, amiben egyetlen egy, meghatározott névvel rendelkező metódus van, ezért a név alapján a **Reflection** csomag segítségével ezt az eljárást le tudjuk

futtatni, mégpedig az **invoke** metódus meghívásával. Az osztálynak és egyetlen metódusának a kikeresésére is a **Reflection** csomag használható.

Arra, hogy a futási időben lefordított osztályok bekerüljenek a **Java classpath**-ba – ami azért szükséges, hogy utána a **Reflection** csomag metódusait használni lehessen rajtuk –, a **ClassLoaderUtil** **addFileToClassPath** metódusát használjuk. A **ClassLoaderUtil** a **Jodd** nevű ingyenes *Java* csomagban található.

Ez kódban a következő módon érhető el:

par: a metódus paramétereinek a tömbje

name: a metódus neve

version: a metódus verziója

result: a hívás eredménye

```
ClassLoaderUtil.addFileToClassPath(PROJECT_HOME);  
Class clazz = Class.forName("generate." + name + "v" +  
version.replace(".", "_"));  
  
Method[] ms = clazz.getDeclaredMethods();  
Method m = ms[0];  
  
result = m.invoke(null, par);
```

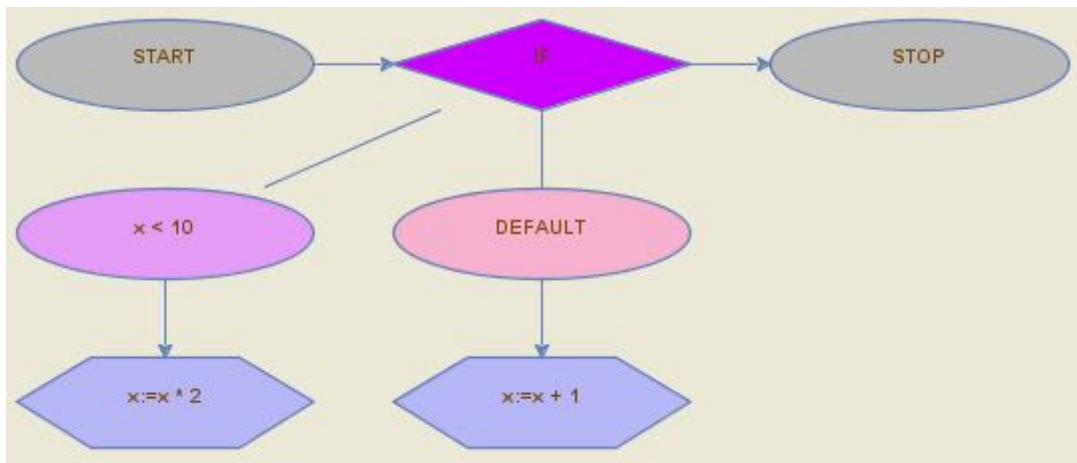
16. kódrészlet A futtatás kódja

Először betöltjük az osztályt a nevének a felhasználásával, majd pedig lekérjük az egyetlen metódusát, ami statikus is. Legvégül az **invoke** segítségével meghívjuk a metódust.

3 Példa

A rövid példában a 2.3.2 pontban elkészített metódusból generált leíró és *Java* kódot mutatom be.

Példa feladat: Ha a bemenő paraméter tíznél kisebb, akkor szorozzuk meg kettővel, különben adjunk hozzá egyet. Tegyük fel, hogy 'x' a bemenő paraméter neve.



13. ábra A felületen elkészített metódus

Generált XML:

```
<method name="pelda_kod">
  <description/>
  <inputs>
    <var name="x" type="java.lang.Integer"/>
  </inputs>
  <outputs>
    <var name="result" type="java.lang.Integer"/>
  </outputs>
  <body>
    <defsize height="45" width="150"/>
    <sources/>
    <statements>
      <start id="2">
        <coord x="0" y="1"/>
        <variables/>
      </start>
      <stop id="4">
        <coord x="2" y="1"/>
        <output name="result" var="x"/>
      </stop>
      <switch id="3">
        <coord x="1" y="1"/>
        <case id="5">
          <condition> x < 10</condition>
          <statement ref="7"/>
        </case>
        <default id="6">
          <statement ref="8"/>
        </default>
      </switch>
    </statements>
  </body>
</method>
```

```

        </default>
    </switch>
    <assignment id="7" variable="x">
        <coord x="0" y="3"/>
        <expression> x * 2</expression>
    </assignment>
    <assignment id="8" variable="x">
        <coord x="1" y="3"/>
        <expression> x + 1</expression>
    </assignment>
</statements>
<steps>
    <step from="2" to="3"/>
    <step from="3" to="4"/>
</steps>
</body>
</method>

```

17. kódrészlet: A 13. ábrán látható metódusból generált XML leíró

Generált Java kód:

```

package generate;
public class pelda_kodv1{
    public static Object[] pelda_kod(java.lang.Integer x) throws
java.lang.Exception{
        java.util.List<String> _log = new java.util.ArrayList<String>();
        java.text.DateFormat _dateFormat = new
java.text.SimpleDateFormat("yyyy.mm.dd");
        if(x < 10 ){
            x=x * 2 ;
        }
        else{
            x=x + 1 ;
        }
        Object[] _result = new Object[2];
        _result[0]=x;
        _result[1]=_log;
        return _result;
    }
}

```

18. kódrészlet: A 17. kódrészletben látható XML leíróból generált java kód

4 Tesztelés – eredmények

A rendszer teszteléséhez az **Oracle Sales** sémáját használtuk fel, amit egy külön adatbázisszerverről ér el az a szerver, amin a tesztek futnak. A **Sales** sémában vásárlók(**Customers**), termékek(**Products**), promóciók(**Promotions**), országok(**Countries**) és eladások(**Sales**) adatai találhatóak meg.

A teszteken a generált algoritmusok hatékonyságát az adathozzáférések aránya szerint vizsgáltuk. Az előre várt eredmény az, hogy az adathozzáférések aránya nem változik a feldolgozott adatmennyiség növekedésével sem. Ha esetleg nőne az arány, akkor az azt jelentené, hogy minél több adattal dolgozunk, annál hatékonyabb a feldolgozás, ha pedig csökkenne, akkor annál rosszabb lenne a hatékonyság.

A megadott futási időeredményekben nincs benne az az idő, ami az adatok eléréséhez szükséges, hanem csak a feldolgozási idő.

Tesztfeladatok:

- a) Egy megadott ország ügyfelein menjen végig, és ha férfi, akkor legyen a keresztnéve módosítva Béla-ra, ha nő, akkor a marital_status legyen módosítva hajadon-ra. A végén adja vissza ki, hogy hány esetben volt a módosítás előtti keresztnév B kezdetű.

Feladat számításigénye: kicsi – elég egyszer végigiterálni az adatokon

Felhasznált tábla	Adatsorok száma
Customers	55.500
Adathozzáférések száma: 55.500	
Futási idő: 0,047 mp	
Hatékonyság: ~1.180.851 adathozzáférés / mp	

13. Táblázat Az a, tesztfeladat statisztikája

- b) A termékeknek olyan árat szeretnénk adni, amely figyelembe veszi az eladási adatokat. Az emelés mértéke legyen a havi átlag eladás és a legjobb havi eladás hányadosa. Ezen számított érték alapján frissítsük az árakat (**PROD_LIST_PRICE**, **PROD_MIN_PRICE**).

Feladat számításigénye: nagy – minden termékhez többször végig kell iterálni az eladásokon és ebből kell kiszámolni az eredményeket, ezért nagyon számításigényes a feladat. A megoldás három eljárásból áll össze.

Felhasznált tábla	Adatsorok száma
Products	72
Sales	918.843
Adathozzáférések száma: ~ 727.723.656	
Futási idő: 624,516 mp	
Hatékonyság: ~1.165.260 adathozzáférés / mp	

12. Táblázat A b, tesztfeladat statisztikája

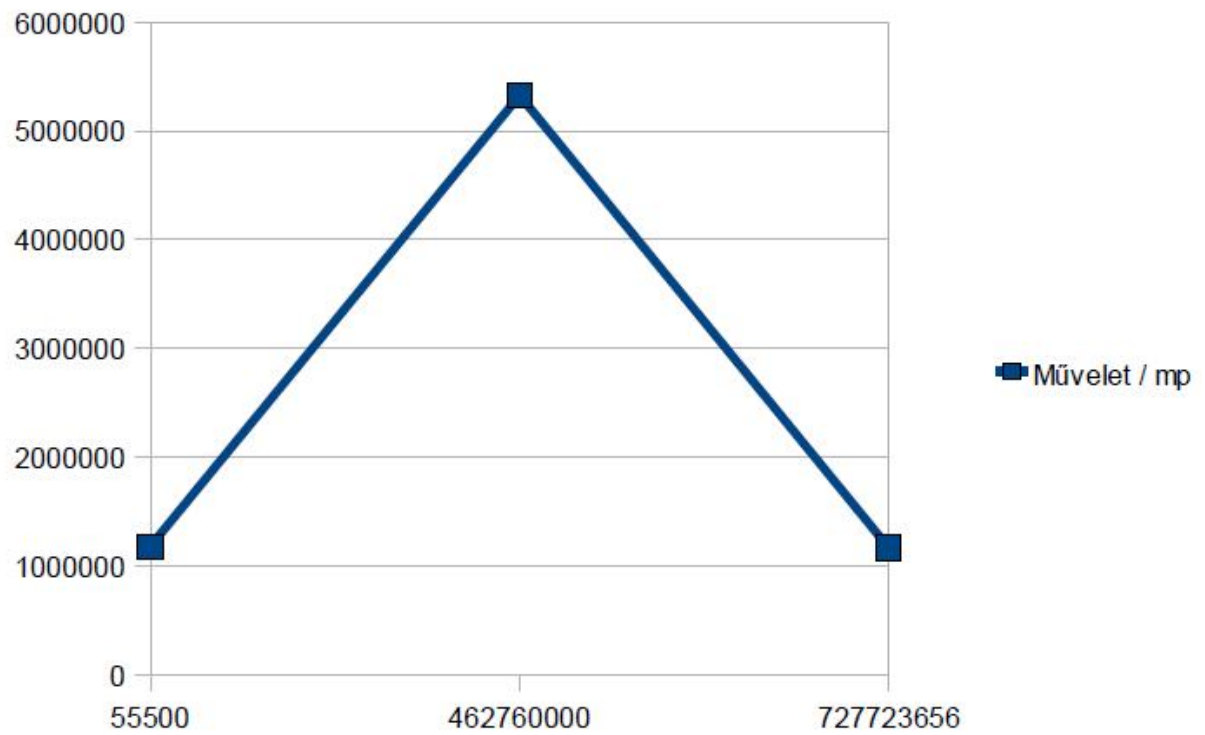
- c) A franciaországi promóciók hatékonyságát szeretnénk megvizsgálni. A hatékonyság mérőszáma legyen a teljes bevétel és a promóció költségének hányadosa. Ezt követően a legeredményesebbnek talált promócióból indítsunk egy újat a holnapi naptól.

Feladat számításigénye: nagy – minden promócióhoz végig kell iterálni az eladásokon, valamint néhányszor a termékeken és ebből kell kiszámolni az eredményeket, ezért számításigényes a feladat. A megoldás három eljárásból áll össze.

Felhasznált tábla	Adatsorok száma
Promotions	503
Sales	918.843
Products	72
Adathozzáférések száma: ~ 462.760.000	
Futási idő: 86,828 mp	
Hatékonyság: ~ 5.329.617 adathozzáférés / mp	

13. Táblázat A c, tesztfeladat statisztikája

Ahogy az adatokból és az 1. diagramon is láthatjuk, a hatékonyság az első és utolsó esetben szinte azonos, de középen van egy nagy hatékonyság-emelkedés. Ez annak is tulajdonítható, hogy a második esetben sokkal kevesebb mellékszámítást kell végeznie az algoritmusnak, mint a maradék kettőben, így hatékonyabb az adathozzáférés. Lényegében megállapítható, hogy a hatékonyság nem változik az adatmennyiség növekedésével.



1. diagram A mérések eredménye, ahol az x tengelyen az adathozzáférések száma, az y tengelyen pedig az egy másodperc alatti adathozzáférések aránya látható

5 Összefoglalás

Egy olyan rendszer készült el, aminek a segítségével grafikus úton tudunk egy metódust előállítani, ezt el tudjuk menteni, vissza tudjuk tölteni és tesztelni is tudjuk. A rendszer fontos tulajdonsága, hogy a metódusokat futási időben lefordítja és azonnal használható állapotba állítja, így például nem kell újraindítani az alkalmazást ahhoz, hogy tesztelni tudjunk. Egy másik jó funkciója, hogy egy létező és futó környezetből az adatstruktúrákat be tudjuk illeszteni a rendszerbe, így már létező típusokkal tudunk dolgozni. Sőt, a futó környezetből is meg tudjuk hívni az előállított algoritmusokat, így lehetőség van arra, hogy futási időben megváltoztassunk egy használt eljárást anélkül, hogy a környezetet le kellene állítani és újra kellene indítani.

A rendszer előnyeit tehát akkor lehet teljesen kihasználni, ha van mellette egy olyan környezet, amibe úgymond beépülve kiegészíti azt.

A rendszer azon komponense, ami a metódusok futási időben történő lefordításáért, majd pedig a futtatásért felelős, külön is nagyon hasznos és akár más rendszerek esetében is felhasználható lenne.

A grafikus felületen összeállított metódusokból generált kódok hatékonysága természetesen általában nem éri el a programozók által ugyanarra a problémára írt kódok szintjét, de mivel csökkentett eszközkészlet áll rendelkezésre az algoritmusok megalkotásakor és a hatékonyság nagyban függ a metódust a felületen összeszerkesztő személytől is, ezért a hatékonyságcsökkenés elfogadható mértékű.

Úgy gondolom, hogy egy hasznos rendszer készült el, amit tovább lehetne fejleszteni mind funkcionalitásában, mind pedig a felhasználói felület terén, de a prototípus az alap funkciók tesztelésére, esetleg üzleti alkalmazására teljes mértékben alkalmas.

Referenciák

- [1] Compiler API - <http://www.javabeat.net/articles/73-the-java-60-compiler-api-1.html>
- [2] Content Repository API - <http://www.ibm.com/developerworks/java/library/j-jcr/>
- [3] Java Reflection API -
<http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html>
- [4] Jackrabbit Explorer - <http://www.jcr-explorer.org/>
- [5] Content Repository Node dokumentáció - <http://www.day.com/maven/jsr170/javadocs/jcr-1.0/javax/jcr/Node.html>

Melléklet

Tutoriálok és példa alkalmazások a Java Reflection API-hoz:

<http://www.roseindia.net/java/reflect/>

Rendszerkövetelmény

Glassfish 3.1-es alkalmazáserver

Java 1.6+ futtató környezet (Java Virtual Machine)

A fejlesztés a Netbeans 7.1-es verziójú keretrendszerben történt.

Szerver beállítása

A prototípus szervertől része egy **Glassfish 3.1-es** alkalmazáserveren fut. A szervert kezelnie kell a Repository-t, valamint a Loader-t, ehhez a következő lépésekre van szükség.

- Első lépésként a `<glassfish-install-dir>/glassfish/domains/<your-domain>/lib` mappába be kell másolni a **jcr-2.0.jar**, a **jodd-3.3.2.jar**, illetve a **jodd-wot-3.3.2.jar** fájlokat, majd újraindítani a Glassfish-t, ahol a `<glassfish-install-dir>` az a mappa, ahova a Glassfish fel van telepítve és a `<your-domain>` az aktuális domain.
- Második lépésként fel kell telepíteni a **jackrabbit-jca-2.4.0.rar** fájlt. Ezt az Admin Console-on belül az Applications/Deploy menüponton keresztül tehetjük meg.
- Harmadik lépésként létre kell hozni egy új Connector Connection Pool-t (Resources/Connector Connection Pools), és Resource Adapterként felhasználni az előbb telepített rar fájlt, majd a következő lapon meg kell adni két tulajdonságot:

ConfigFile: `<glassfish-install-dir>/glassfish/domains/<your-domain>/repository.xml`

HomeDir: `<glassfish-install-dir>/glassfish/domains/<your-domain>/repository`

- Ezután be kell másolnunk a `repository.xml` fájlt a `<glassfish-install-dir>/glassfish/domains/<your-domain>/` mappába. A Repository esetleges költöztetésekor ezt a két fájlt kell átmásolni.
- Következő lépésként létre kell hozni egy új Connector Resource-t (Resources/Connector Resources), kiválasztani az előbb létrehozott pool-t, és megadni a JNDI névnek a következőt: `jcr/repo`.

A szervert be kell állítani, azt az utat, ahova a generált Java kódok kerülnek. Ehhez a szervert be kell állítani egy új JVM beállítást:

Ezt az **admin console** elindítása után a **Configurations -> server-config -> JVM Settings** pontban lehet megtenni. Itt az **Add JVM Option** gombra kattintva beállíthatjuk az új beállítást, ami a következőképpen néz ki:

-Ddinalg.home=< PATH/TO/GENERATION/FOLDER/ >

Emellett létre kell hozni az elérési út alá egy

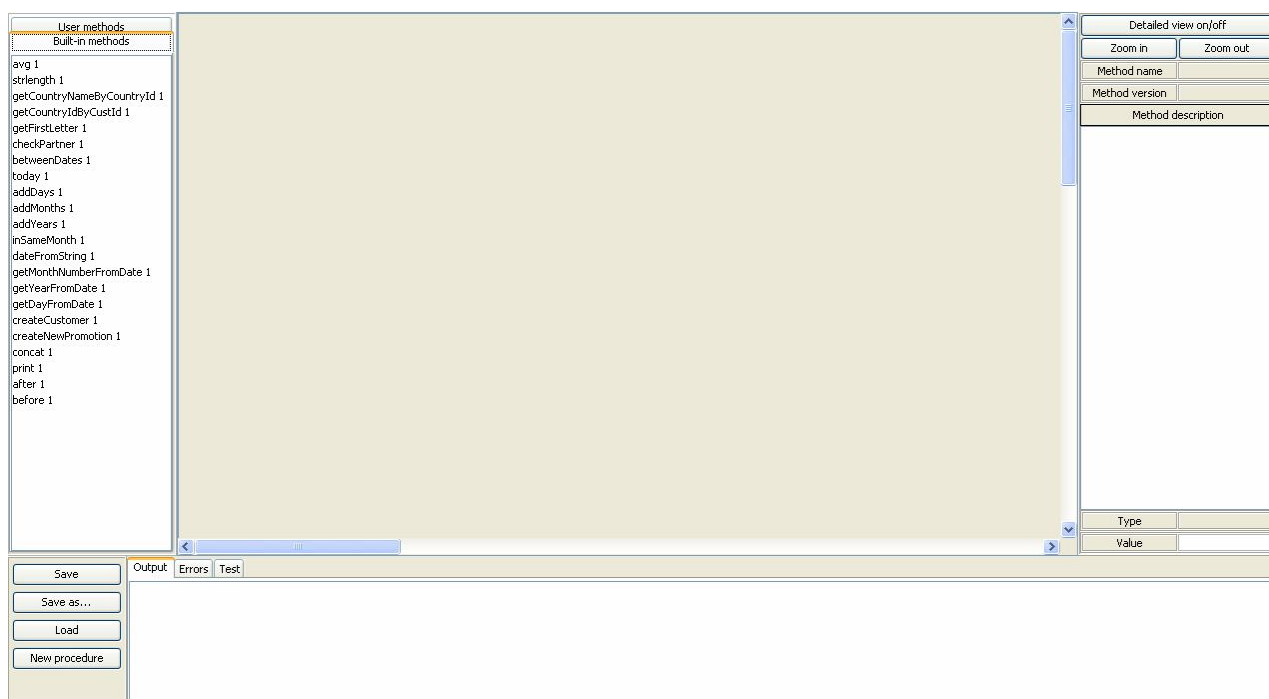
- **generate** és egy
- **generate/types** mappát,

melyekbe a generált Java kódok kerülnek a szerver futtatása során.

Felhasználói kézikönyv

A felhasználó munkájának megkönnyítésére összeállítunk egy felhasználói kézikönyvet, ami az alkalmazás használatát képekkel is illusztrálja. Ennek során az alkalmazás összes lehetőségét bemutatjuk és leírjuk a különböző korlátokat is.

A felület:

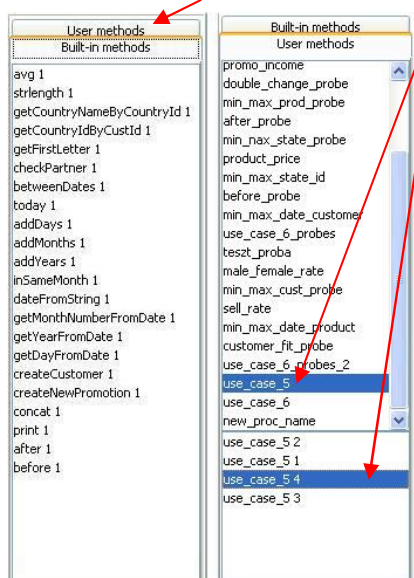


2. ábra A felület

A felület felosztása:

A bal oldalon a már elkészített metódusok (User methods) és a beépített metódusok (Built-in methods) listája található. Ezek között a fülök segítségével lehet váltogatni. Az elkészített metódusoknak több verziója is lehet ugyanazzal a névvel.

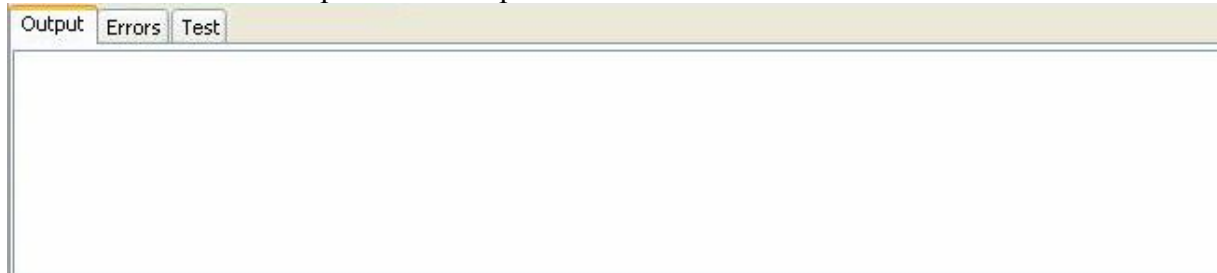
Ezek a verziók az alsó dobozban jelennek meg (lásd 2. ábra).



A listák alatt az utasítógombok találhatók. A **Save** gombbal a szerkesztett metódust lehet menteni, a **Save as...** gombbal pedig egy új névvel lehet elmenteni a szerkesztett eljárást. A **Load** gombbal a mentett metódusok közül kiválasztott metódust lehet betölteni a munkaasztalra, míg a **New procedure** gombra kattintva egy új metódust kezdetünk el szerkeszteni.



A felhasználó számára az értesítések a felület alsó részére érkeznek, ahol három fülön három panel található. Az Output fülön található panel az általános információkat tartalmazza, míg az Error fülön a hibaüzenetek jelennek meg. A test fülön az elkészített és sikeresen elmentett metódusokat lehet tesztelni. Fontos, hogy tesztelés közben nem lehet menteni vagy új tesztet elindítani. Az is lényeges, hogy tesztelni csak olyan metódust lehet, aminek nincs összetett típusú bemenő paramétere.



Detailed view on/off	
Zoom in	Zoom out
Method name	
Method version	
Method description	
Type	
Value	

A panel jobb oldalán találhatók a szerkesztett metódus tulajdonságait megjelenítő mezők. Ezek fölött a **Detailed view on/off** gombbal a részletes nézetet le be- és kikapcsolni. A **Zoom in** gombbal rá lehet közelíteni, míg a **Zoom out** gombbal ki lehet távolítani a munkaterületről

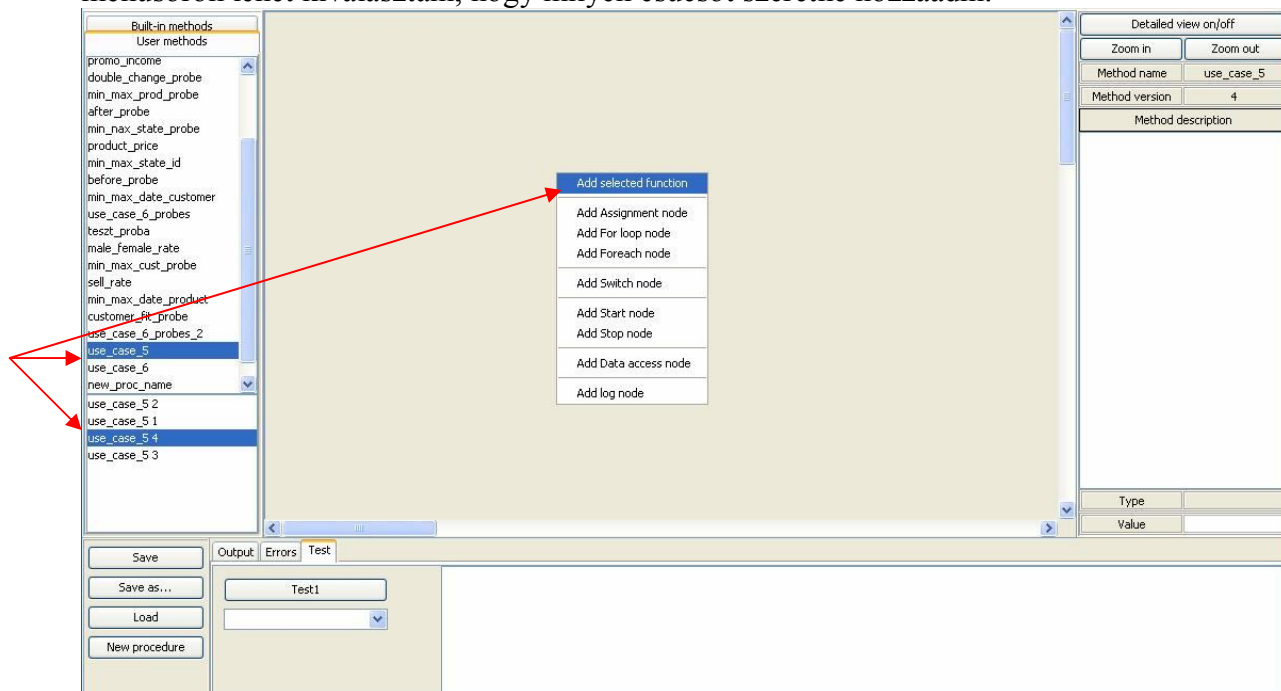
Betöltés

A kijelölt metódus betöltése a **Load** gombra kattintva működik.

The image shows the main workspace of the software. On the left is a 'Built-in methods' list with 'use_case_5' selected. The central workspace displays a flowchart for 'Promotions v1' with nodes like 'START', 'FOREACH', 'createNewPromo...', 'STOP', and various decision diamonds. On the right is a 'Detailed view on/off' panel showing 'Method name: use_case_5' and 'Method version: 4'. At the bottom left, a red arrow points to the 'Load' button in the 'Save' section.

Új csúcs hozzáadása

Új csúcsot felvenni a munkaterületre jobb egérgombbal kattintva lehet. Ekkor egy felugró menüsoron lehet kiválasztani, hogy milyen csúcsot szeretne hozzáadni.

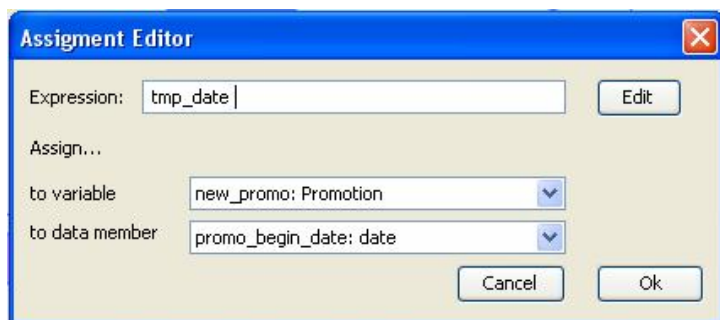
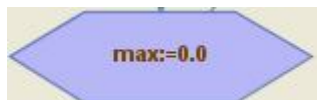


Már elkészített metódust is fel lehet használni a munkaterületen. Ehhez ki kell választani a bal oldali listákon a használni kívánt metódust (felhasználói metódus esetén a verziót is), majd pedig a jobb egérgombra felugró menüből az Add selected function menüpontot kell kiválasztani.

Csúcstípusok és a szerkesztők

A csúcsokhoz tartozó szerkesztőfelületek a csúcsra való dupla bal egérgomb-kattintással jönnek elő.

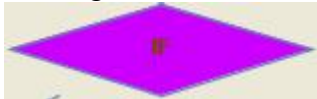
1. **Értékkadás csúcs:** egy változónak vagy adattagjának értékkadás. A megadott kifejezés értéke lesz a változónak vagy adattagjának az új értéke.



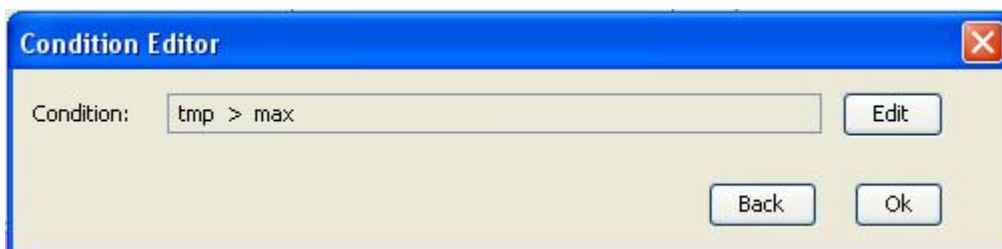
A **to variable** címke mellett lévő legördülő menüből lehet kiválasztani, hogy melyik változónak legyen az új értéke a szerkesztett kifejezés. A **to data member** címke mellett lévő legördülő menüben pedig a változó adattagját lehet kiválasztani, ha annak akarunk értéket adni. Ez maradhat üresen is.

Az értékadó kifejezést az Edit gombra kattintva a kifejezésszerkesztőben lehet megszerkeszteni.

2. **Switch/If csúcs:** Egy elágazást jelentő csúcs. Ennél a csúcsnál a végrehajtás elágazhat a belőle kiinduló feltételcsúcsok irányába.



3. **Feltétel csúcs:** Olyan feltételt kell megadni, aminek az igaz volta esetén ebből a csúcsból folytatódik tovább a végrehajtás.



A feltételt az Edit gombra kattintva megjelenő kifejezésszerkesztővel lehet megszerkeszteni.

4. **Default feltétel csúcs:** ha az elágazás csúcsához kapcsolt egyik feltételcsúcs feltétele sem igaz, akkor ezzel a csúccsal folytatódik a végrehajtás.



5. **For csúcs:** Egy feltétel teljesüléséhez kötött ciklus csúcsa. A ciklusmag (az a csúcissorozat, amelyet minden körben végre kell hajtani) első csúcsából élt kell húzni a For csúcsba. A for csúcsba először mindig a ciklusmagból kell élt behúzni és csak utána a többi csúcsból



A loop variable címke mellett lévő legördülő menüben a ciklusváltozót lehet kiválasztani. A start expression-t a mellette lévő Edit gombra kattintva feljövő kifejezésszerkesztőben lehet megszerkeszteni, mint ahogy a Stop condition-t is hasonlóan. Az incrementation-t beírva, vagy a fel/le nyílra kattintva lehet megváltoztatni. Ide csak egész számok írhatók be (pozitív vagy negatív, de általában ne legyen nulla – ez a stop condition kifejezéstől függ).

A for működése

Mikor a végrehajtás a for csúchoz érkezik, a ciklusváltozó felveszi a kezdeti kifejezés (Start expression) értékét. Ezután addig hajtja végre a hozzá kötött ciklusmagot, amíg a megállási feltétel(Stop condition) igaz. Minden kör végén növeli a ciklusváltozó értékét az Incrementation-ben megadott értékkel.

A képen látható For tehát azt csinálja, hogy kezdetben az i értéke 0 lesz, majd amíg i értéke kisebb, mint 10, addig végre hajtja a ciklusmagot. Minden körben növeli az i-t 1-el, így összesen 10-szer fogja végrehajtani a ciklusmagot.

6. **Foreach csúcs:** A hozzá kötött adatforrás adatain iterál végig és minden adatsorra végrehajtja a ciklusmagot. Az adatsor értékét a ciklusváltozó vszi fel.



A ciklusváltozó nevét lehet megváltoztatni a szövegmezőben.

A Foreach működése

A foreach a hozzá kötött adatforrás minden során végigmegy, a sort beteszi a ciklusváltozóba, majd pedig végrehajtja a ciklusmagot. A ciklusmag végrehajtása után lép a következő sorra.

7. **Start csúcs:** A start csúcsban lehet beállítani a metódusban használt változókat és új komplex típusokat is itt lehet felvenni.



Name	Type	Input
best_promo	Promotion	☐
max	double	☐
tmp	double	☐
promo	Promotion	☐
income_tmp	double	☐
new_promo	Promotion	☐
tmp_date	date	☐
daysNum	int	☐
i	int	☐

Name: Type: ☐ Input

Add Remove Save

Back Ok

Új változó hozzáadása: Miután a Name (név) mező kitöltésre került és a változó típusa is ki lett választva, az **Add** gombbal lehet a változót hozzáadni. A táblázatban kijelölt változót a **Remove** gombbal lehet eltávolítani, illetve a kijelölt változó tulajdonságait meg is lehet változtatni. Ehhez a név és típusmezőket, valamint az Input jelölőnégyzetet lehet használni. A változtatás csak a **Save** gombra kettintva lép életbe. Az input jelölőnégyzet azt jelzi, hogy egy változó bemeneti változó vagy sem. Ha a négyzet be van jelölve, akkor bemeneti változó. Ez azt jelenti, hogy a szerkesztett metódus bemeneti paramétere lesz az adott változó.

Komplex típusok

A start csúcs Complex Types fülén keresztül az összetett típusok érhetők el.

Start editor

Variables Complex Types

Type Name: Customer Add new complex type

Complex types

Customer

Parameters:

Name	Type
credit_limit	int
country_id	int
first_name	string
last_name	string
year of birth	int

Name: Type: int

Add Remove Save

Back Ok

A létező összetett típusok a Complex types címke alatt lévő legördülő menüből érhetők el. Ezeket utólag már nem lehet megváltoztatni.

Új típus hozzáadásához először az adattagjait kell megadni, majd pedig a Type name címke melletti szövegdobozba be kell írni a nevét és az **Add new complex type** gombra kell kattintani.

Az adattagokat egyenként, a változókhoz hasonlóan kell hozzáadni. Az adattagokat változtatni csak addig lehet, amíg nem adtuk hozzá az összetett típust a többihez. Az adattagok típusa csak az alaptípusok közül kerülhet ki.

8. **Stop csúcs:** A stop csúcsban lehet beállítani a kimeneti paramétereket és a paraméternek értéket adni, amit a végén fel fog venni.



Stop editor

Output variables:

Name	Var
new_promo	new_promo.promo_name: string

Name: Variable: Data member:

Add Remove Save Back Ok

Új kimeneti paramétert felvenni a név beírásával és a neki értéket adó változó(esetleg annak egy adattagja) beállításával lehet. A nevet a Name: címke mellett található szövegdobozba kell beírni, a változót pedig a Variable címke után lévő legördülő menüből lehet kiválasztani. Ha egy összetett típusú változó lett kiválasztva, akkor a Data member címke mellett lévő legördülő menü is használható lesz. Innen a változó adattagját lehet kiválasztani, de üresen is lehet hagyni. Miután a megfelelő dolgok ki lettek választva az Add gombra kattintva új kimeneti paramétert lehet hozzáadni. A táblázatban kijelölt paramétert a Remove gombbal lehet eltávolítani, illetve a kijelölt paraméter adatait hasonlóan lehet megváltoztatni, mint a Start csúcsban a változókét.

9. **Log csúcs:** a működés során a végeredménybe kerülő hiba vagy figyelmeztető szöveget lehet beszúrni és szerkeszteni.

Message

Log form

Message:

Back OK

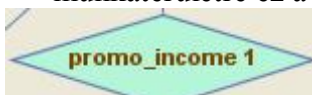
A szövegmezőbe lehet beírni a kívánt üzenet szövegét.

10. **Adatforrás csúcs:** az adatforrásokat jelképező csúcs. Ezekből nyerhető ki adat a futás során.



A szerkesztőfelületen a legördülő menüben lehet kiválasztani a kívánt adatforrást.

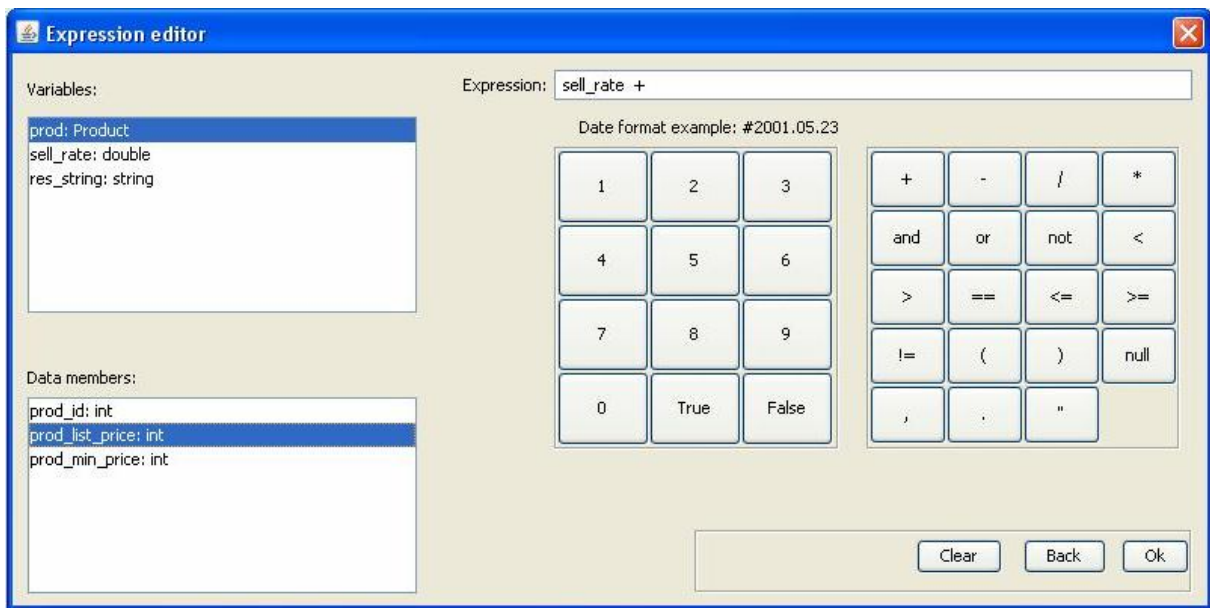
11. **Metódus csúcs:** már elkészített vagy beépített metódus beillesztésére a munkaterületre ez a csúcs használatos.



Minden paraméterhez be lehet állítani azt a változó és esetleg a változó adattagját, amit a paraméterhez hozzárendelünk. Ezeket a legördülő menükből lehet kiválasztani.

A kifejezések szerkesztése

Az alkalmazásban több helyen lehet kifejezéseket szerkeszteni. Erre a célra egy külön szerkesztőfelület áll rendelkezésre: a kifejezésszerkesztő.



A szerkesztett kifejezés az ablak tetején található. A kifejezéseket össze lehet kattintgatni vagy pedig be lehet gépelni.

A kifejezésben használható változók a bal oldalon a Variables címke alatt találhatók. Az bal egérgombra duplán kattintva a változó beillesztődik a szerkesztett kifejezésbe. A változók alatt egy másik dobozban a kijelölt változó adattagjai találhatóak, amik ugyanilyen dupla kattintásos módszerrel illeszthetők be a szerkesztett kifejezésbe. A maradék két dobozban, az ablak közepén és bal oldalán, az aritmtikai és logikai műveletek, illetve konstansok találhatóak (a dupla egyenlőségjel - == - az egyenlőségvizsgálat, a != pedig ennek az ellenkezője, ami akkor igaz, ha a két oldal nem egyenlő). Ezekre a gombokra elég egyszer kattintani ahhoz, hogy bekerüljenek a szerkesztett kifejezésbe.

A szövegeket két „macskaköröm” között kell beírni, míg ha csak egyetlen karakterről van szó, akkor azt aposztrófok közé kell tenni. FONTOS, hogy minden részt egy üres hely válasszon el a kifejezésen belül. Ha a felületen a kattintásos módszerrel visszük be az adatokat, akkor ezt az alkalmazás automatikusan berakja, viszont a szövegek beírásakor erre figyelni kell.

A dátumok megadására külön minta van, ezt pedig a következő mintán lehet szemléltetni:

#2001.05.16

Ez 2001 május 16-t jelöli.

A nem egész, hanem valós számokban tizedesvessző helyett pontot kell tenni.

A kifejezést törölni a Clear gombbal lehet, elfogadni pedig az Ok gombbal. A Back gombra kattintva kilép, de nem mentődik el a változtatás.

Fontos, hogy a program nem végez helyességvizsgálatot, erre a felhasználónak kell figyelnie.

Éltípusok

Az élék behúzása a bal egérgomb csúcson történő lenyomásával, majd pedig nyomva tartva a célsúcra való húzásával történik. A célsúcra elengedve az egérgombot az él beillesztődik.

1. Rákövetkezés él: A végrehajtás sorrendjét meghatározó él.



2. Adatforrásból foreach csúcsba húzott él: A foreach ciklus feladata, hogy végigiterál annak az adatforrásnak az adatain, amit hozzákötünk. Ez az él jelöli ezt a fajta kapcsolatot.



3. A Switch csúcsból egy feltételcsúcsba húzott él: Az elágazáscsúcshoz több feltételcsúcsot lehet kapcsolni. Ez az él jelöli, hogy egy feltételcsúcs az adott elágazáscsúcshoz tartozik.



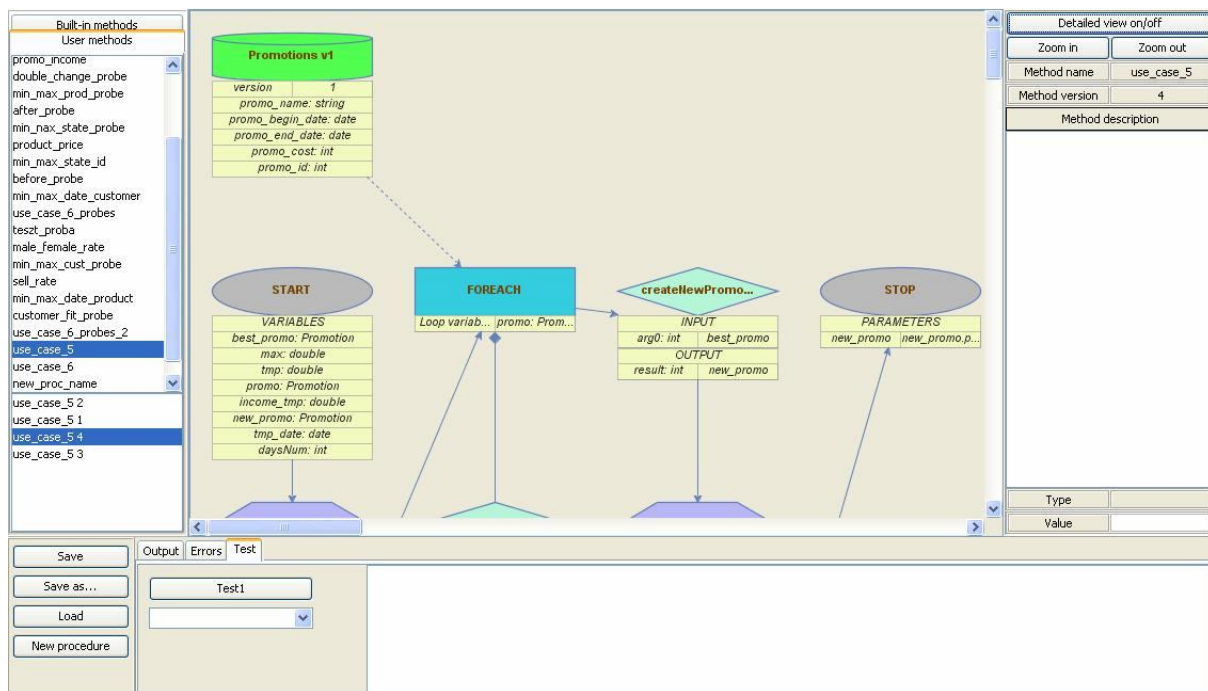
4. Ciklusmagból for vagy foreach csúcsba húzott él: A for és foreach csúcsoknak szüksége van egy ciklusmagra (azok az utasítások, amiket minden körben végrehajt). Ennek a ciklusmagnak az első utasítását kell a for vagy foreach csúcsához kötni.



Az éleket a program magától beállítja. Arra kell figyelni, hogy minden csúcsba megfelelő számú élt húzzunk be, illetve a For és Foreach csúcsokba először mindig a ciklusmag első csúcsából húzzuk be az élt és csak utána a többi. Hibázás esetén törölni kell a hibás élt és utána újra lehet próbálkozni.

Részletes nézet

A részletes nézetet a Detailed view on/off gombra kattintva lehet ki- és bekapcsolni. A részletes nézetben minden csúcs alatt megjelenik a hozzá tartozó információk listája. Ez csúcstípusonként változik. Ezeket az adatokat csak a szerkesztőfelületen lehet változtatni, a részletes nézet celláiban nem.



Kliens használata – részletes tutorial

Az alfejezet egy használati eset részletes, képekkel illusztrált bemutatását tartalmazza. A használati esetekben az Oracle SH sémáját használtuk fel. A használati eset a következő: Az algoritmus egy megadott ország ügyfelein menjen végig, és ha férfi, akkor legyen a keresztnéve módosítva Bélára, ha nő, akkor a marital_status legyen módosítva hajadonra. A végén adja vissza, hogy hány esetben volt a módosítás előtti keresztnév B kezdetű. Első lépésként létre kell hozni egy start csúcsot a megfelelő változókkal. A country változó lesz az input változónk, ezzel adjuk meg, hogy melyik ország ügyfeleit vizsgáljuk majd. A tmp változó fogja tárolni az aktuálisan vizsgált ügyfél nemzetiségét. A counter változó tárolja a módosítás előtti B kezdőbetűjű keresztnévvel rendelkezők számát. A képen látható row változó tárolja majd az adatbázisból kapott Customer típusú aktuálisan vizsgált vásárlót, ez később fog begenerálódni, most nem kell kézzel létrehozni. Végül a letter változóban tároljuk el az aktuálisan vizsgált vásárló keresztnévének első betűjét.

Start editor [X]

Variables | Complex Types

Variables:

Name	Type	Input
country	string	☒
tmp	string	☐
counter	int	☐
row	Customer	☐
letter	char	☐

Name: Type: ☐ Input

Add Remove Save

Back Ok

Második lépésként létre kell hoznunk egy értékadást az Add Assignment node menüponttal, ahol beállítjuk a counter értékét nullára. A counter változót a to variable legördülő listából választhatjuk ki. Kössük össze a start csúcsot az új csúccsal.

Assignment Editor [X]

Expression: Edit

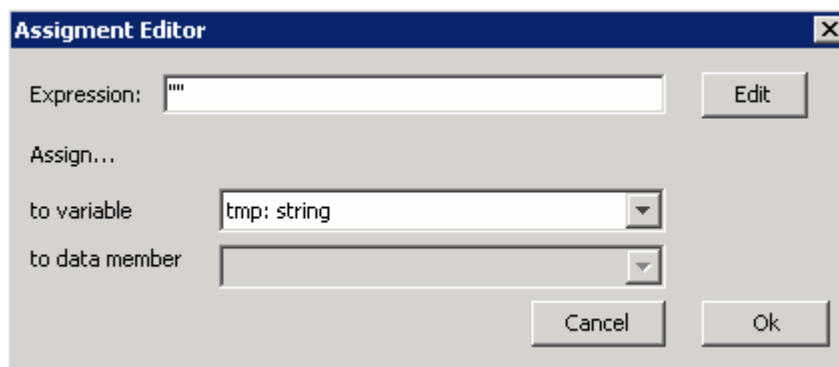
Assign...

to variable

to data member

Cancel Ok

Harmadik lépésként meg kell ismételni az előző lépést, csak most a tmp változót inicializáljuk az üres string értékre. Kössük össze az előbb létrehozott csúcsot az új csúccsal.



Assignment Editor

Expression:

Edit

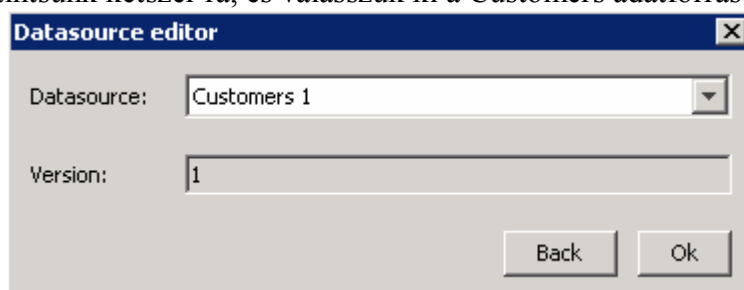
Assign...

to variable:

to data member:

Cancel Ok

Negyedik lépésként hozzá kell adnunk egy foreach csúcsot a munkaterülethez az Add Foreach node menüponttal. Ennek segítségével tudunk végigmenni a vásárlókon. Ahhoz, hogy tudjuk, milyen adatforrás elemein kell végigmenni, szükségünk van egy új adatforrásra, amit az Add Data acces node menüpont segítségével hozhatunk létre. Miután létrehoztuk, kattintsunk kétszer rá, és válasszuk ki a Customers adatforrást.



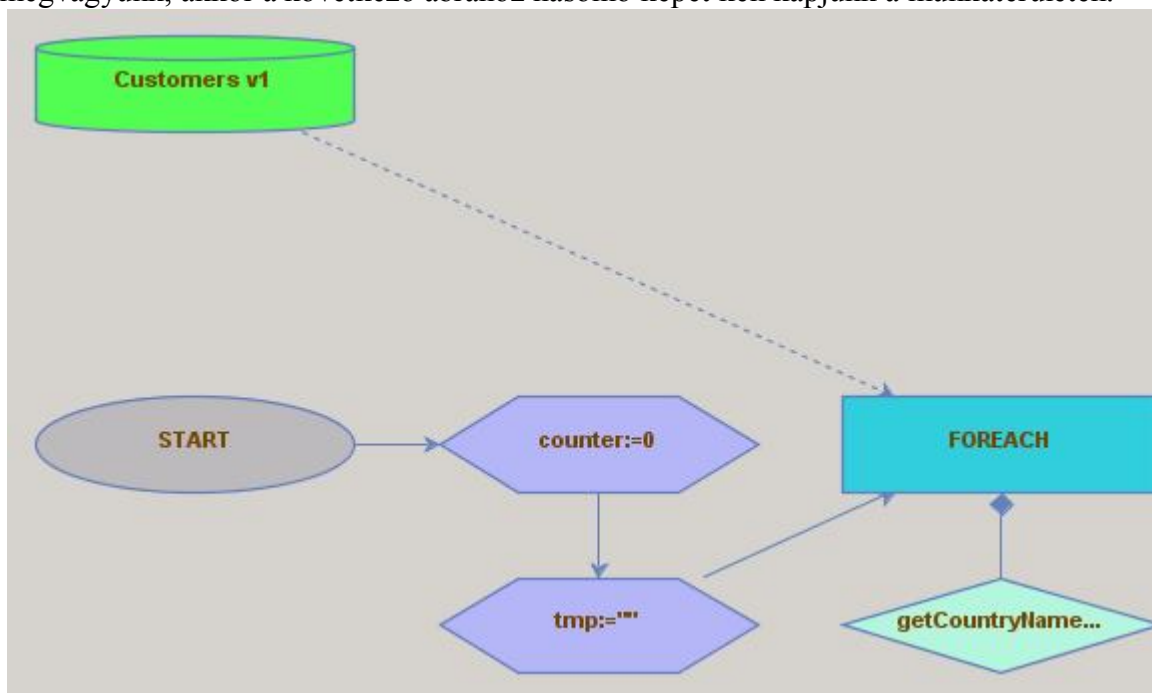
Datasource editor

Datasource:

Version:

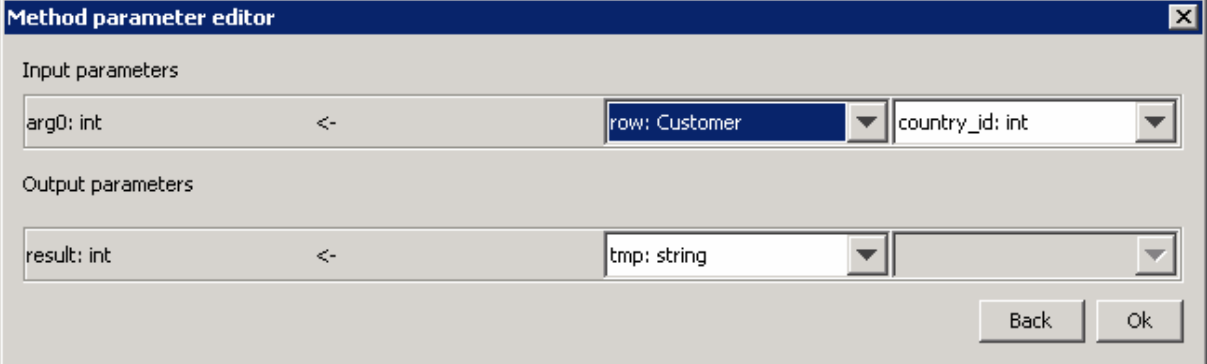
Back Ok

Ezután létre kell hoznunk a ciklusmagot. Először azt szeretnénk megtudni, hogy az aktuális vevő milyen országból származik. Ehhez ki kell választani a Built-in methods fülön a getCountryNameByCountryId beépített függvényt, és a munkaterületre beszúrni az Add selected function menüponttal. Ezt követően először kössük össze az előbb hozzáadott csúcsot a foreach csúccsal, majd az adatforrás csúcsot a foreach csúccsal. Ha ezzel megvagyunk, akkor a következő ábrához hasonló képet kell kapjunk a munkaterületen.



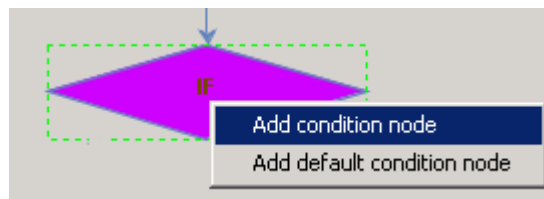
Miután eljutottunk ideig, fel kell paramétereznünk a getCountryNameByCountryId függvényt. Ez a függvény bemenő paraméterként megkapja egy ország id-ját, és visszaadja eredményként az ország nevét. Kattintsunk kétszer a csúcsra. Ezután egy űrlapon

megadhatjuk a paramétereket. Input paraméterként válasszuk ki a legördülő listából a row változót (ez tartalmazza az éppen vizsgált vásárlót), és a mellette lévő legördülő listából válasszuk ki a country_id változót. Ez fogja megadni, hogy a vásárló melyik országhoz tartozik. Az output paraméterek részéből pedig válasszuk ki a tmp változót, ez fogja tárolni a vevő országának nevét.



The image shows a 'Method parameter editor' dialog box. It has two sections: 'Input parameters' and 'Output parameters'. In the 'Input parameters' section, there are two dropdown menus. The first is labeled 'arg0: int' and has a value of 'row: Customer'. The second is labeled 'country_id: int'. In the 'Output parameters' section, there is one dropdown menu labeled 'result: int' with a value of 'tmp: string'. At the bottom right, there are 'Back' and 'Ok' buttons.

Ezek után el kell döntünk, hogy a vásárló az általunk megadott országhoz tartozik-e. Ha nem, akkor nincs további teendők vele, kérjük a következő vásárlót. Ehhez egy elágazás csúcs szükséges, amelyet az Add Switch node menüpont segítségével hozhatunk létre. Kössük össze a getCountryNameByCountryId csúcsot a most létrehozott elágazás csúccsal, majd kattintsunk jobb klikkel az elágazás csúcsra, és válasszuk az Add condition node menüpontot.



Ennek segítségével hozhatjuk létre az elágazás feltételét. Jelen esetben arra lesz szükség, hogy az általunk megadott ország megegyezzen a vásárló országával. Ezt a következőképp ellenőrizhetjük le. Kattintsunk kétszer az imént hozzáadott feltétel csúcsra, majd a Condition szövegdoboz melletti Edit gombra. Ekkor előugrik a kifejezés-szerkesztő ablak. Itt a Variables listából kiválaszthatjuk a használni kívánt változókat. Először kattintsunk kétszer a country változóra, majd a gombok közül válasszuk ki az == jelet, végül kattintsunk kétszer a tmp változóra ismét a Variables listából. Végül okézzuk le a kifejezés-szerkesztőt és a feltétel csúcs szerkesztőjét is. Így megkaptuk a feltétel ellenőrzését.

Expression editor

Variables:

```
country: string
tmp: string
counter: int
row: Customer
letter: char
```

Expression: `country == tmp`

Date format example: #2001.05.23

1	2	3
4	5	6
7	8	9
0	True	False

+	-	/	*
and	or	not	<
>	==	<=	>=
!=	(	)	null
,	.	"	

Data members:

Clear Back Ok

Ha az aktuális vásárló országa megegyezik az általunk vizsgált országgal, akkor további lépésekre van szükség. Először is elmentjük a keresztnévnek első betűjét a `getFirstLetter` beépített függvény segítségével. Ezt a függvény nevének a Built-in methods listából való kiválasztásával és a munkaterületen való Add selected function menüpontra való kattintással tehetjük meg. Ezt követően összekötjük az előbb beállított feltétel csúcsot a most létrehozott csúccsal, majd felparaméterezzük a függvényt. Ehhez kattintsunk kétszer a `getFirstLetter` csúcsra. A függvény egy szöveg első betűjét fogja visszaadni. Ehhez a `getCountryNameById` csúcsnál bemutatottakhoz hasonlóan kétszer kell rá kattintani, és input paraméternek megadni a `row` változó `first_name` mezőjét (ez tartalmazza a vásárló keresztnévét), míg output változónak kiválasztani a `letter` változót.

Method parameter editor

Input parameters

arg0: int	<-	row: Customer	first_name: string
-----------	----	---------------	--------------------

Output parameters

result: int	<-	letter: char	
-------------	----	--------------	--

Back Ok

Ezt követően egy újabb elágazásra lesz szükség, amely eldönti, hogy a vásárló férfi-e vagy nő. A korábbiakban leírtak alapján hozzuk létre az elágazás csúcsot, és adjuk hozzá a feltétel csúcsot. A feltételt állítsuk át úgy, hogy a `row` változó `gender` adattagjára nézzük meg, hogy egyenlő-e az M karakterrel (az adatbázisban az F jelenti a nőt, az M a férfit).

Expression editor

Variables:

country: string
tmp: string
counter: int
row: Customer
letter: char

Expression: `row.gender == 'M'`

Date format example: #2001.05.23

1	2	3
4	5	6
7	8	9
0	True	False

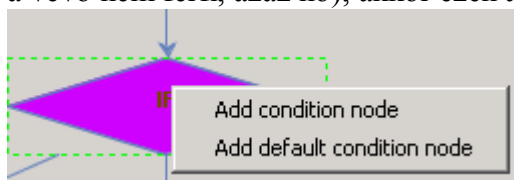
+	-	/	*
and	or	not	<
>	==	<=	>=
!=	(	)	null
,	.	"	

Data members:

credit_limit: int
country_id: int
first_name: string
last_name: string
year_of_birth: int
gender: char
marital_status: string

Clear Back Ok

Miután ezzel megvagyunk, kattintsunk jobb klikkel az elágazás csúcsra, és kattintsunk az Add default condition node menüpontra. Ezzel érjük el azt, hogy ha a feltétel hamis (tehát a vevő nem férfi, azaz nő), akkor ezen az ágon folytathassuk az algoritmust.



Először a női ágot kezeljük le. Hozzunk létre egy értékadás csúcsot a korábbiakban megismertek alapján, és kössük össze a default csúccsal, majd állítsuk be rajta a row változó marital_status adattagjának értékét „single”-re.

Assignment Editor

Expression: `"single"` Edit

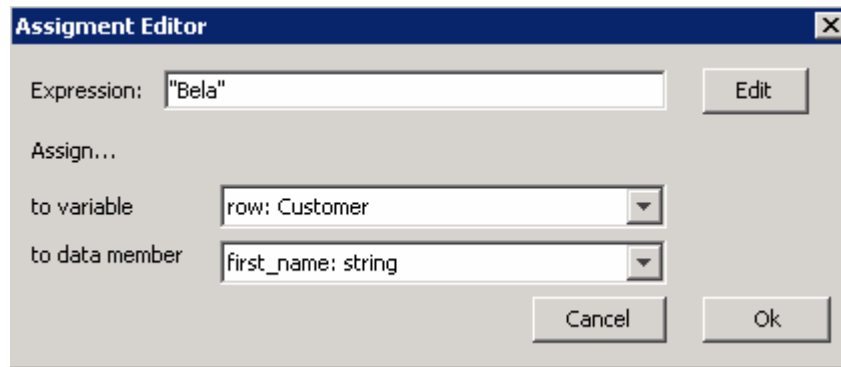
Assign...

to variable: row: Customer

to data member: marital_status: string

Cancel Ok

Ha idáig eljutottunk, akkor elértük azt, hogy az általunk megadott országba tartozó női vásárlók családi állapotát átállítottuk egyedülállóra. Következik a férfiak lekezelése. Az elágazás feltételcsúcsa alá (amellyel azt vizsgáljuk, hogy a vásárló férfi-e, azaz `row.gender == 'M'`) hozzunk létre egy értékadás csúcsot, és állítsuk be a row változó first_name adattagjának értékét Bélára. Kössük össze a feltételcsúcsot a most létrehozott értékadás csúccsal.



Assignment Editor

Expression: Edit

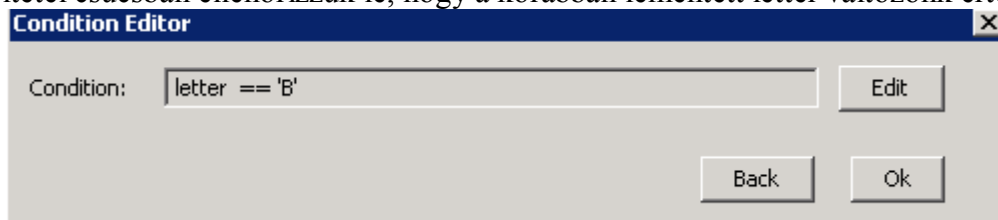
Assign...

to variable: ▼

to data member: ▼

Cancel Ok

Ezzel a férfiak átnevezését is megoldottuk. Most már csak arra van szükség, hogy megtudjuk, az átnevezés előtt B betűvel kezdődött-e a vásárló keresztnéve, és ha igen, akkor megnöveljük a számlálót. Ehhez ismét egy elágazásra van szükség. Hozzuk ezt létre, és a feltétel csúcsban ellenőrizzük le, hogy a korábban lementett letter változónk értéke B.

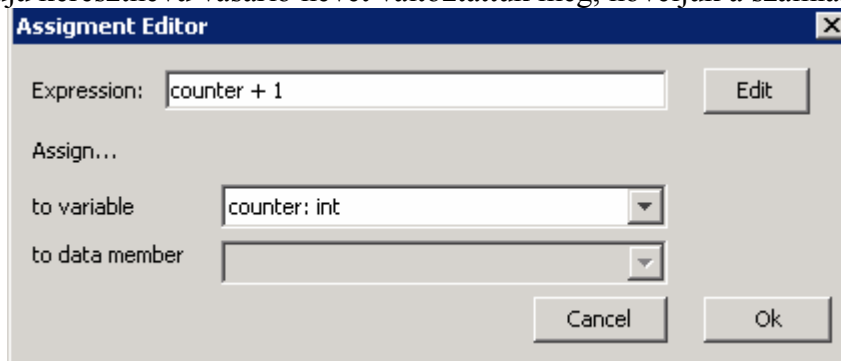


Condition Editor

Condition: Edit

Back Ok

Default ágra jelen pillanatban nincs szükség. Ezt követően hozzunk létre egy értékadás csúcsot, amelyen növeljük a számlálót, és kössük össze a feltétel csúccsal, így amennyiben B kezdőbetűjű keresztnévű vásárló nevét változtattuk meg, növeljük a számlálót egyel.



Assignment Editor

Expression: Edit

Assign...

to variable: ▼

to data member: ▼

Cancel Ok

Ezzel az algoritmusunk majdnem kész, már csak egy stop csúcs hozzáadására van szükség, amelyet az Add Stop node menüpont segítségével hozhatunk létre. Kattintsunk rá kétszer, és hozzuk létre a res nevű változót, amihez válasszuk ki a Variable legördülő listából a counter változót. Így beállítottuk, hogy az algoritmus a változtatások előtti B keresztnévű férfi vásárlók számát adja eredményül. Végző lépésként kössük össze a foreach csúcsot a stop csúccsal, és mentjük el az algoritmust.

Stop editor [X]

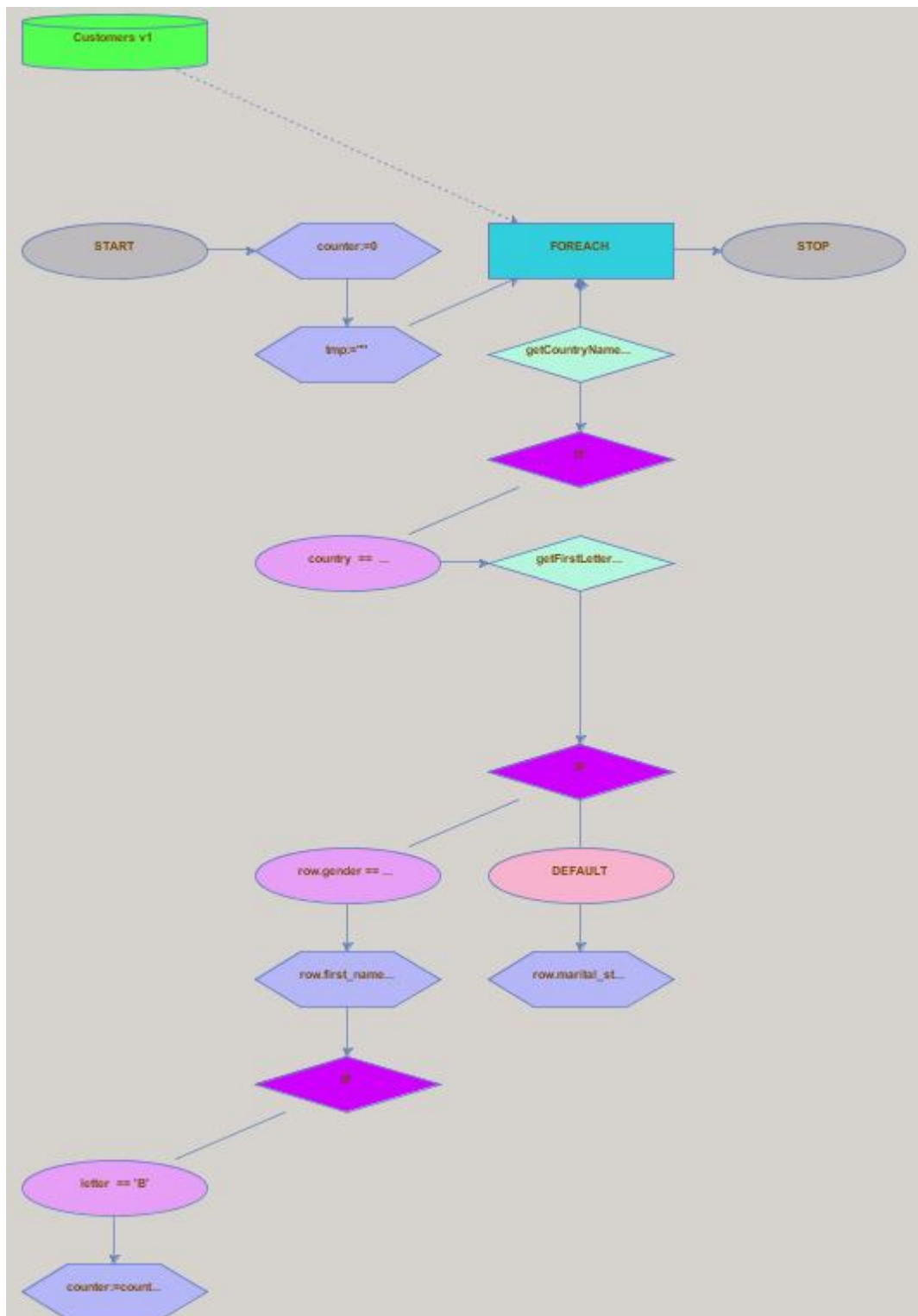
Output variables:

Name	Var
res	counter: int

Name: Variable:

Data member

Az elkészített algoritmus a következőképp néz ki:

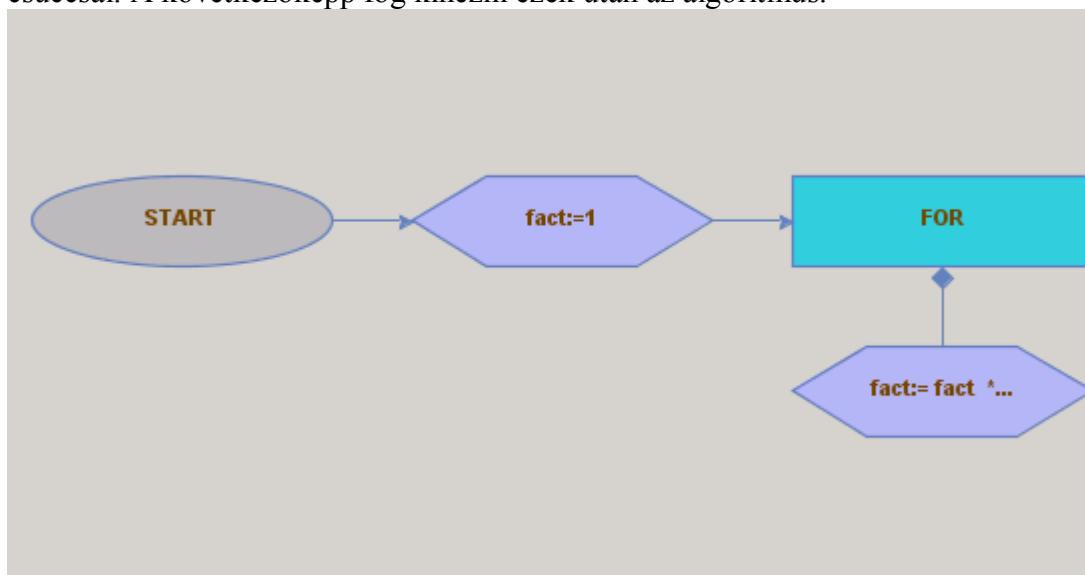


A szerkesztő még támogat egy csúcsot, amely használatára az előző használati esetben nem volt szükség, a for csúcsot, ezért ennek demonstrálására készítettem egy faktoriális számoló algoritmust.

Először létrehozunk egy start csúcsot, amelynek input paramétereként létrehozunk egy n változót (ez lesz az a szám, aminek a faktoriálisára vagyunk kíváncsiak) és egy fact változót, amire a számoláshoz lesz szükség. Ezt követően egy értékadás csúcsban a fact értékét beállítjuk egyre. Ezután jöhet az eddig nem használt for csúcs létrehozása. Ezt a munkaterületen az Add For loop node menüpont segítségével hozható létre. Miután ez megtörtént, kattintsunk is rá kétszer. Ekkor bejön egy űrlap, amivel beállíthatjuk a for

csúcsot paramétereit. A loop variable lesz a ciklusváltozó. A Start expression menüpontban beállíthatjuk a ciklusváltozó kezdő kifejezést (legyen ez most 2), majd az Incrementation menüpontban megadhatjuk a lépésközt (jelen esetben maradjon 1). Végül a Stop conditionban megadhatjuk a leállási feltételt, itt állítsuk most be az $i \leq n$ feltételt. Ez a csúcs magyarul azt jelenti, hogy 2-ről indulva addig növeljük a ciklusváltozó értékét, amíg az egyenlő nem lesz az n-nel, tehát mindig ezzel a számmal fogjuk megszorozni a faktoriális aktuális értékét.

Ha ezzel megvagyunk, hozzunk létre egy értékadás csúcsot, és adjuk meg neki, hogy a fact változó értéke legyen $\text{fact} * i$. Ez azt jelenti, hogy a ciklusváltozó aktuális értékével megszorozzuk a faktoriális értékét (tehát kezdetben a fact értéke 1, az i értéke kettő, tehát $\text{fact} = 1 * 2$, majd $\text{fact} = 2$, $i=3$, $\text{fact} = 2 * 3$, és így tovább). Ha ezzel megvagyunk, először kössük össze ezt az értékadás csúcsot a for csúccsal, majd a $\text{fact} = 1$ értékadást a for csúccsal. A következőképp fog kinézni ezek után az algoritmus.



Miután ezzel megvagyunk, már csak egy stop csúcs létrehozására van szükség, ahol létrehozuk a res változót, és hozzákötjük a fact változó értékét, aminek köszönhetően az algoritmus visszaadja n faktoriálisát.