TDK-dolgozat

Bognár Bálint Cserép Máté

Generikus szoftvermodell-vizualizációs keretrendszer

Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Szerzők:

Bognár Bálint
Programtervező Informatikus
V. évfolyam

Cserép Máté
Programtervező Informatikus
V. évfolyam

Témavezető:

Varga Virág
PhD hallgató

2012

Tartalomjegyzék

1. fejezet

Bevezetés

Napjaink szoftverrendszerei egyre nagyobbá válnak, amely egyben nagyfokú bonyolultságnövekedést is jelent. Fontossá vált e komplex rendszerek megértésének támogatása a szoftver hatékony karbantarthatóságának és továbbfejleszthetőségének érdekében. A szoftverelemzés egyik legdinamikusabban fejlődő területévé mára a forrásszövegek statikus elemzése vált. A hatékony felhasználáshoz az így nyert információkat különböző nézetekben, vetületekben kell megjeleníteni. Míg az ipari szoftverfejlesztés már több ilyen célú eszközzel állt elő, szabadon elérhető, nyílt forrású szoftvervizualizációs eszköz csak kevés akad, és ezek között még kevesebb a dinamikusan fejlesztett, naprakész program.

Dolgozatunk témája egy olyan nyílt forrású, absztrakt, nyelvfüggetlen szoftvermodell és erre épülő vizualizációs keretrendszer megalkotása és bemutatása, amely támogatja az említett célokra alkalmas szoftverek fejlesztését, azok számára alapul szolgálhat.

A programok forráskódjának szövegalapú elemzése hatékonyan nem kivitelezhető, ezért olyan reprezentációt érdemes választani, amelyben ezek a feladatok könnyebben megfogalmazhatóak. A problémára jellemző megoldás, hogy a forráskódot annak szintaktikus alapú hierarchikus szerkezetével – azaz az *absztrakt szintaxisfával* – ábrázolják.

Ennek megfelelően a programmegértés támogatása két részre bontható: a vizsgált program forráskódjának modellezésére, valamint ennek a modellnek a megjelenítésére. A dolgozatban bemutatott keretrendszer definiál egy szoftvermodellt, amely alkalmas tetszőleges programozási nyelv absztrakt szintaxisfájának leírására. Erre építünk egy eszköztárat, amely lehetőséget nyújt különböző vizualizációs alkalmazások hatékony készítésére.

A szoftvermodell az absztrakt szintaxisfák reprezentációján túl az azokon végezhető tetszőleges műveletekkel bővíthető. Ezek egy speciális osztálya például lehetővé teszi a szoftver szerkezetének különböző vetületekben történő kezelését egységes interfészen keresztül. Ilyen módon lehetőség adódik a forráskód eltérő hierarchikus szinteken történő vizsgálatára, mellyel nagyban elősegíthető a vizsgált program megértése.

A vizualizációt támogató réteg modell-nézet architektúrára építve lehetővé teszi a modellnek a megjelenítés számára történő tetszőleges transzformálását (szűrését, annotálását, átrendezését stb). A végfelhasználói program tervezésekor fontos a felhasználóbarát felület, melynek kialakítását keretrendszerünk is támogatja: például azonos modell egy időben több vizualizációs eszközzel való megjelenítése esetén ezek között automatikus szinkronizációt biztosít.

Dolgozatunk 2. fejezetében áttekintjük a szoftvervizualizációs eszközök történetét, a jelenleg a szakterületen elérhető programokat, keretrendszereket, szakirodalmat és eredményeket. A 3. fejezetben bemutatjuk, milyen követelmények és tervezési szempontok, megfontolások mentén fogtunk hozzá saját eszközünk megvalósításához, valamint összehasonlító elemzést végzünk a 2. fejezetben tárgyalt eszközökkel. A 4. fejezetben mélyebben, az implementáció egyes részleteire is kitérve kifejtjük keretrendszerünk szerkezetét és alkalmazási lehetőségeit. Zárásként a dolgozat 5. fejezetében összefoglaljuk és értékeljük az elért eredményeket, összegezzük a megszerzett tapasztalatokat és áttekintjük a továbbfejlesztési lehetőségeket.

2. fejezet

Szoftvervizualizációhoz kapcsolódó eszközök vizsgálata

Számos szoftvereszköz készült programok struktúrájának, különböző vetületeinek és metrikáinak megjelenítésére vagy a bonyolultság szemléltetésére. Az ilyen eszközök többségének deklarált célja a programmegértés támogatása, azaz a komplex szoftverrendszerek tanulásának, megismerésének segítése. Az ezen túlmutató célok között gyakoriak a tipikus programozási hibák felderítésének, illetve a szoftver életciklusa során végbemenő változások elemzésének támogatása.

Ez a fejezet ilyen jellegű szoftvereszközök bemutatását tűzi ki célul. Az első szakaszban a tárgyterület történetéről adunk rövid áttekintést. Ezután sorra vesszük azokat a jelentősebb szoftvervizualizációs megoldásokat, melyek legalább szakmai jellegű dokumentáció (cikkek, publikációk) szintjén az interneten szabadon elérhetőek, tehát belső szerkezetük vizsgálható[1]. Az eszközöket két csoportra bontva mutatjuk be: szoftvermodellező, illetve adatvizualizációs eszközök. A felosztást az indokolja, hogy bár sok, a témában készült munka egyaránt foglalkozik mindkét aspektussal, ezek közül az egyik általában túlsúlyban van. A szétválasztás tehát nem szigorú és inkább azon alapul, hogy a rendelkezésre álló források alapján melyik szempontot éreztük hangsúlyosabbnak. A fejezet végén áttekintést adunk a megvizsgált eszközökről és hivatkozunk közvetetten kapcsolódó, vagy a tárgyterülettel foglalkozó számára esetleg érdekes egyéb eredményekre.

[1]Ezeken kívül a szoftverpiac még számos megoldást kínál szoftvervizualizációra, amelyek nagy része azonban egyedi licencelés alapján, csak ipari szereplők által megfizethető áron kerül forgalomba, továbbá forráskódja és megvalósításának részletei ipari titkot képeznek.

2.1. Történeti áttekintés

A szoftverek méretének és komplexitásának robbanásszerű növekedésével létrejött szoftverkrízis következményeként a szoftverek forráskódjának áttekinthetősége és érthetősége jelentősen romlott. Abból a célból, hogy a szoftverfejlesztők számára a programok megértését segítsék, az 1980-as évek kezdetétől különböző módszereket, eljárásokat és eszközöket kezdtek el kidolgozni [1, 2].

A szoftvervizualizációs eszközök rendszerint az alábbi három cél közül egy vagy több megvalósítására készültek [2]:

- a kód megértésének segítése a programozó számára;
- a forráskód *debuggolásának*, azaz a hibák felderítésének és kijavításának megkönnyítése;
- az elemzett program teljesítményének monitorozása.

A korai szoftvervizualizáció az alkalmazott módszerek és a konkrétabb célok mentén gyorsan több ágra szakadt, elsődlegesen az alábbi szempontok szerint [1, 3]:

- A forráskód érthetőségének támogatása, melynek kezdeti változatai közé a különböző *prettyprint* megoldások[2] is tartoznak. Ezek a módszerek a szoftverfejlesztői világban mára lényegében nélkülözhetetlen eszközökké váltak.

- A programok működését és viselkedését elemző és animáló eszközök. Ezek között az egyik első volt a *Brown University*-n fejlesztett *BALSA* [2], amely algoritmusok dinamikus, futás idejű működésének tanulmányozására volt alkalmas. Közös jellemzőjük azonban ezeknek az eszközöknek, hogy az eredeti, elemzendő forráskódot ki kell egészíteni az állapotváltozások jelzésével az elemző alkalmazás részére (*code instrumentation*). Ezek a módszerek ezért elsősorban az oktatásban tudtak elterjedni, szoftveripari környezetben nem.
  A terület később kiterjedt a párhuzamos alkalmazások viselkedésének elemzésére is, az egyik ilyen korai program a *POLKA* volt [2].

- A kódszerkezet automatizált elemzése és átlátható megjelenítése a szoftvervizualizáció napjaink egyik legaktívabban fejlődő ága. Ezek az eszközök a forráskód *statikus elemezését* követően, annak megértését támogatandó, *áttekintő nézeteket* (például gráf vagy különböző diagram nézetek) biztosítanak.

[2] A *prettyprint* módszerek a forráskódot a behúzásokkal, tördeléssel és a szintaktikai elemek kiemelésével teszik könnyebben áttekinthetővé.[1]

Ide tartozik az adatszerkezetek absztrakt reprezentálása is, amely különösen az objektumközpontú programozási nyelvek esetén vált előnyössé. Fontos tényező, hogy a folyamat automatizált – programozói interakció nem szükséges –, így nagy, szoftveripari projektek esetén is könnyen alkalmazhatóvá válik a módszer.

A terület fejlődését és fejlesztését egyrészt azért övezi aktív figyelem napjainkban is, mert újabb és újabb módszerek és gyakorlatok látnak napvilágot mind a forráskód elemzését, mind a vizualizálását illetően. Emellett új igényeket is támasztanak a szakterülettel szemben [3], ilyen például egy szoftver különböző revízióinak statikus elemzése, összehasonlítása, majd az eredmények együttes vizualizációja, azaz a szoftver *evolúciójának* elemzése.

Dolgozatunk további részében az utolsóként említett, kódstruktúra-elemzéssel és -vizualizálással fogunk részletesen foglalkozni.

2.2. Szoftvermodellező eszközök

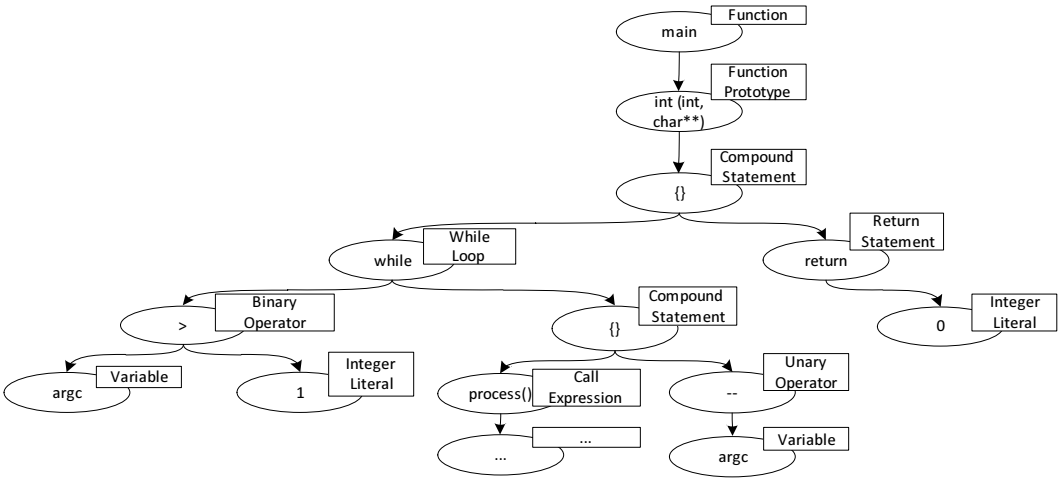
A szoftverelemzés és -megismerés első lépése a megfelelő szoftvermodell felépítése. A modell jellemzően az absztrakt szintaxisfára épül, mivel ez a forráskódból egyszerűen előállítható, azonban kezelése a közvetlen szöveges feldolgozásnál jelentősen egyszerűbb.

Az *absztrakt szintaxisfa* a forráskód szintaktikus elemeinek gráfreprezentációja. A gráf csúcsai a program szintaktikus elemei, élei az elemek tartalmazási vagy logikai kapcsolatai [4]. Egy forráskód-részletet és a hozzá tartozó szintaxisfát mutat be a 2.1. és a 2.2. ábra.

```
1 int main(int argc, char **argv) {
2   while (argc > 1) {
3     --argc;
4     process(argv[argc]);
5   }
6   return 0;
7 }
```

2.1. ábra. Forráskódrészlet

Bár több-kevesebb megjelenítési lehetőséggel is rendelkeznek, az alábbi eszközök fő célja szoftver forráskódjából absztrakt modell készítése.

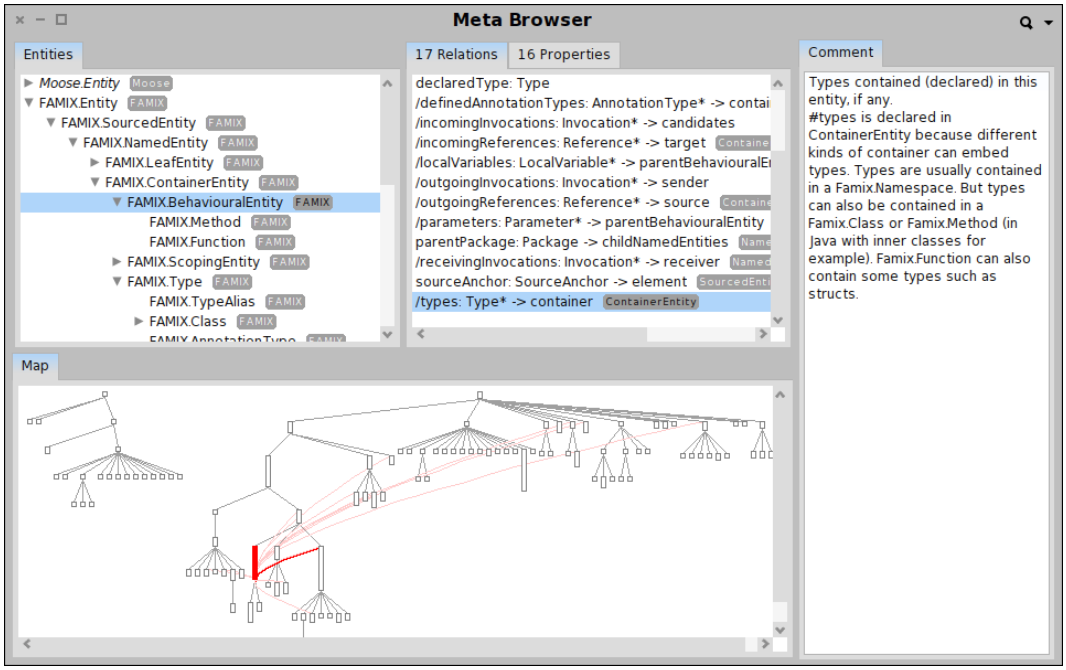


2.2. ábra. A forráskódrészlethez tartozó absztrakt szintaxisfa

2.2.1. A *MOOSE* keretrendszer

A *MOOSE* [5] elsősorban objektumorientált szoftverrendszerek kódjának hatékony elemzésére és refaktorálására készült, nyelvfüggetlen keretrendszer. Előtérbe helyezi a célrendszerekre jellemző nagy komplexitás hatékony kezelését.

A rendszer megvalósítása *SmallTalk* nyelven történt. A modell szigorúan objektumorientált fogalmakra (osztály, metódus, attribútum és leszármazás), illetve ezek kapcsolatára épül. Beépülők (*plugin*) segítségével új (nyelvfüggő) modellkomponensek vezethetők be.



2.3. ábra. MOOSE Meta Browser

A *MOOSE* keretrendszer kiértékelése néhány százezres nagyságrendű kódsort tartalmazó kódbázisokon történt. A vizsgált rendszerek méretéből adódó memóriaproblémákat (a futtató gép memóriája nem volt elég a teljes modell tárolásához) a modell adatbázisra való leképezésével oldották meg.

A *MOOSE*-ra épülő egyszerűbb megjelenítő alkalmazás a *MOOSE Meta Browser* (ld. a 2.3. ábrát), míg jóval összetettebb nézetek vizualizálására képes a *CodeCrawler* (ld. a 2.3.3. szakaszt).

2.2.2. *SolidFX*

A *SolidFX* és a hozzá kapcsolódó eszközkészlet [6, 7] a szoftvermodellezési, -elemzési és -vizualizációs feladatok széles skáláját támogatja. Fejlesztése kutatási projektként indult, de piaci termékcsaláddá nőtte ki magát. A *SolidFX* célja nagyméretű C++ kódbázisok hatékony elemzése és ezek megjelenítésének különböző fejlesztőkörnyezetekbe[3] való ergonomikus, felhasználóbarát integrálása. A *SolidSource* termékcsalád többi tagja még számos felhasználási lehetőséget támogat.[4]

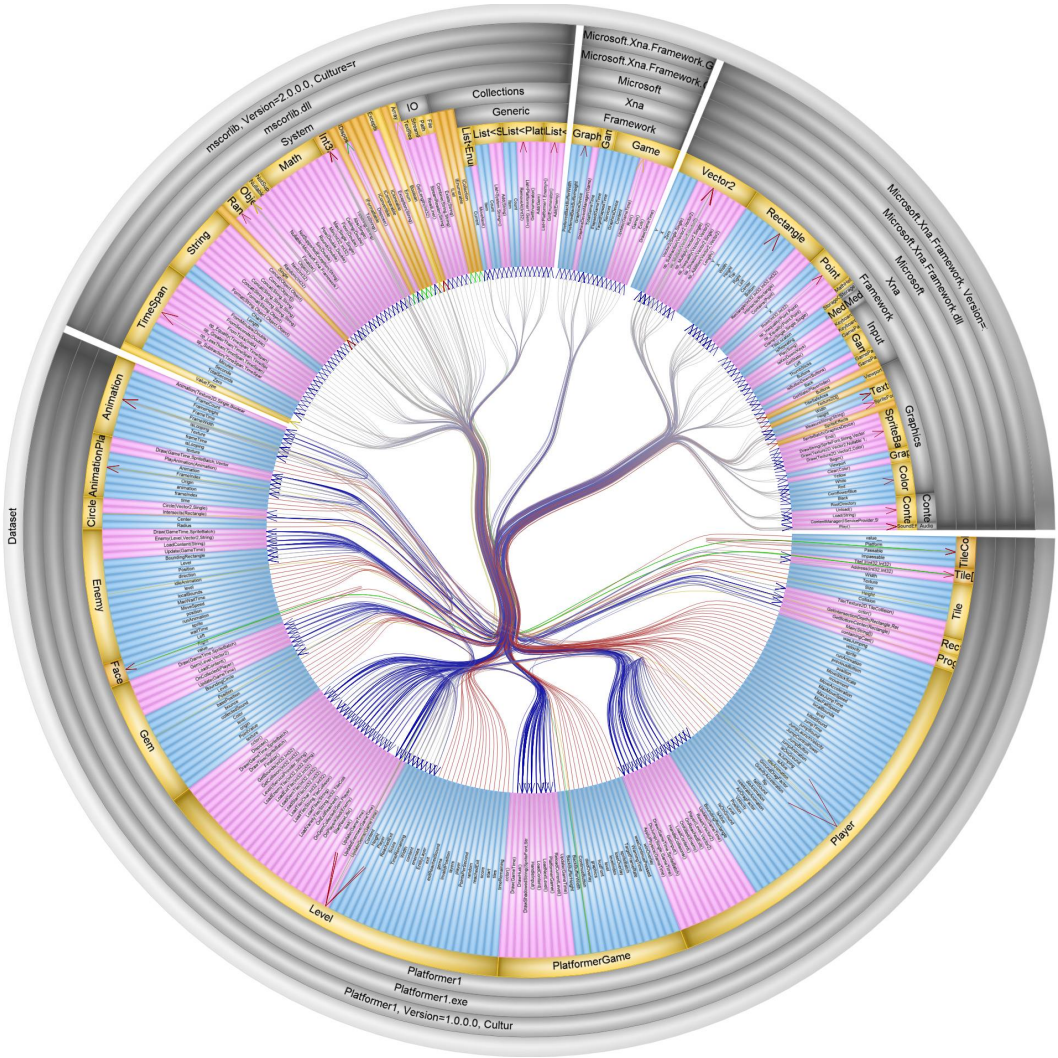
A *SolidFX* modellje az ún. tényadatbázison (*fact database*) alapul. Ez tartalmazza a forráskódból kinyert tényeket (C++ absztrakt szintaxisfát, pozícióinformációkat, illetve előfeldolgozó-direktívák helyét és kifejtését), melyeket saját bináris formátumban, fájlokra bontva tárol a lemezen.

A tényekhez való hozzáférés egységesen ún. *szelekciókon* keresztül történik, amelyek tényazonosítók rendezett sorozatai – ez a modell központi fogalma és interfésze a megjelenítési és egyéb eszközök felé. Szelekciókra alkalmazhatók lekérdezések (*query*), melyek újabb szelekciókat állítanak elő. A tények közötti navigáció is ezen a mechanizmuson keresztül valósul meg.

A *SolidFX*-re épülő megjelenítő alkalmazás a *SolidSX*. Egy játékprogram ilyen módon vizualizált, ún. *radial view* nézetét láthatjuk a a 2.4. ábrán.

[3] Az eszköz [6] szerint a *Microsoft Visual Studio* környezetbe beágyazható, de a fejlesztési tervek között szerepel az *Eclipse*-be való beépülés támogatása is.

[4] Részletes termékinformációk a cég weboldalán: `http://www.solidsourceit.com/`.



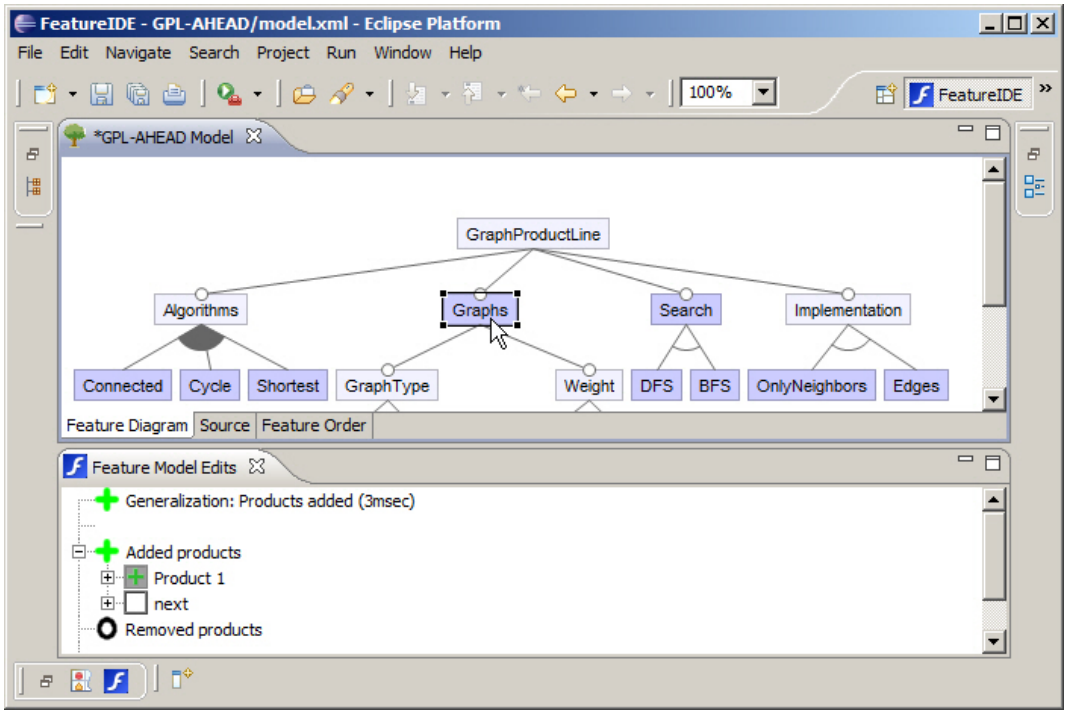
2.4. ábra. Egy *Microsoft .NET XNA Framework*-re épülő játékszoftver *SolidFX* alapú vizualizációja *SolidSX*-szel.

2.2.3. *FeatureIDE*

A *FeatureIDE* [8] egy BSc szakdolgozat részeként – Java nyelven – elkészített Eclipse-beépülő[5]: felülete a 2.5. ábrán látható. A dolgozat célja elsősorban a *feature-oriented programming* (szabad fordításban *funkció-orientált programozás*) támogatása. Ehhez kötődően a szerző követelményeket fogalmaz meg az alkalmazandó szoftvermodell és az ehhez kapcsolódó *IDE* (*Integrated Development Environment*, *integrált fejlesztőkörnyezet*) felé.

A *FeatureIDE* szoftvermodellje egy erősen leegyszerűsített, faszerkezetű (hierarchikus) modell. Csomópontjai kétféle típusúak lehetnek: belső (*inner*), illetve

[5]Az *Eclipse* egy kiterjeszthető, számos programozási nyelvet támogató integrált fejlesztői környezet. `http://eclipse.org`



2.5. ábra. FeatureIDE grafikus felülete

levél (*leaf*) csúcsok. A belső csúcsok az elemzett szoftverprojekt mappa- vagy csomaghierarchiáját, a levél csúcsok a forrásfájlokat reprezentálják. A levelek beágyazva tartalmazzák az adott forrásfájl szintaxisfáját. A modell minden csúcsa rendelkezik két szöveges (*string*) attribútummal: *név*vel és *típus*sal. Más attribútum nem rendelhető hozzájuk.

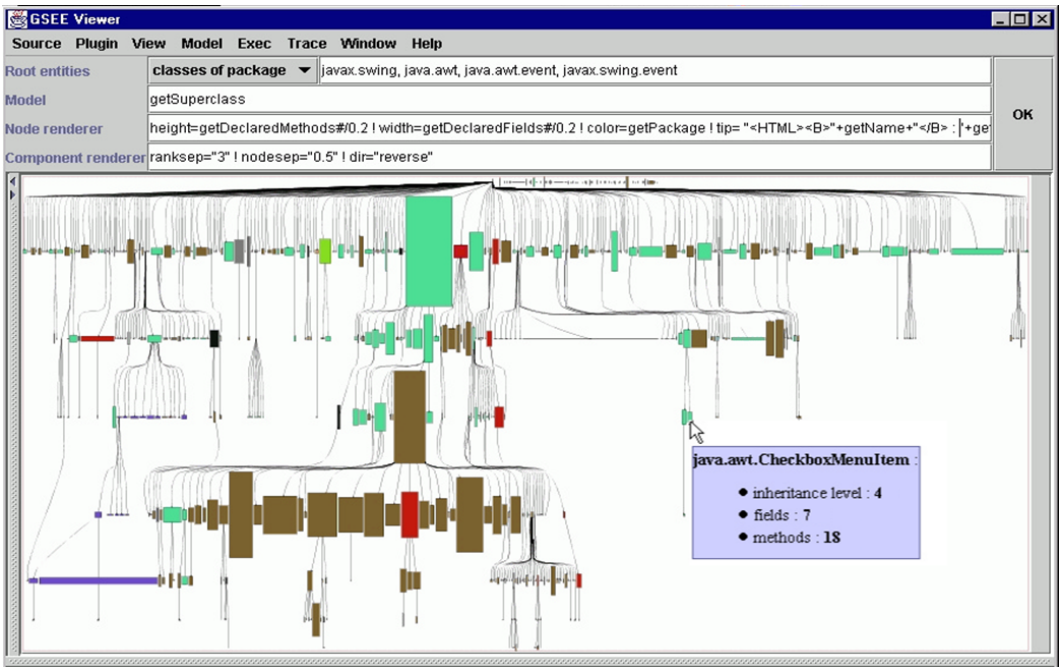
A modell kiértékelése kis méretű (kevesebb, mint 1000 kódsort tartalmazó) szoftvereken történt, így hatékonyságának és skálázhatóságának megítéléséhez nem áll rendelkezésre elegendő információ.

2.3. Adatmegjelenítő (vizualizációs) eszközök

A szoftvermegértés és -elemzés támogatásához a jó modell mellett annak megfelelő vizualizációjára is szükség van. Eben a szakaszbak olyan eszközöket mutatunk be, melyek egy többé-kevésbé absztrakt adatmodell adatainak (általában grafikus) megjelenítésére képesek különböző formákban.

2.3.1. *G*^{SEE}

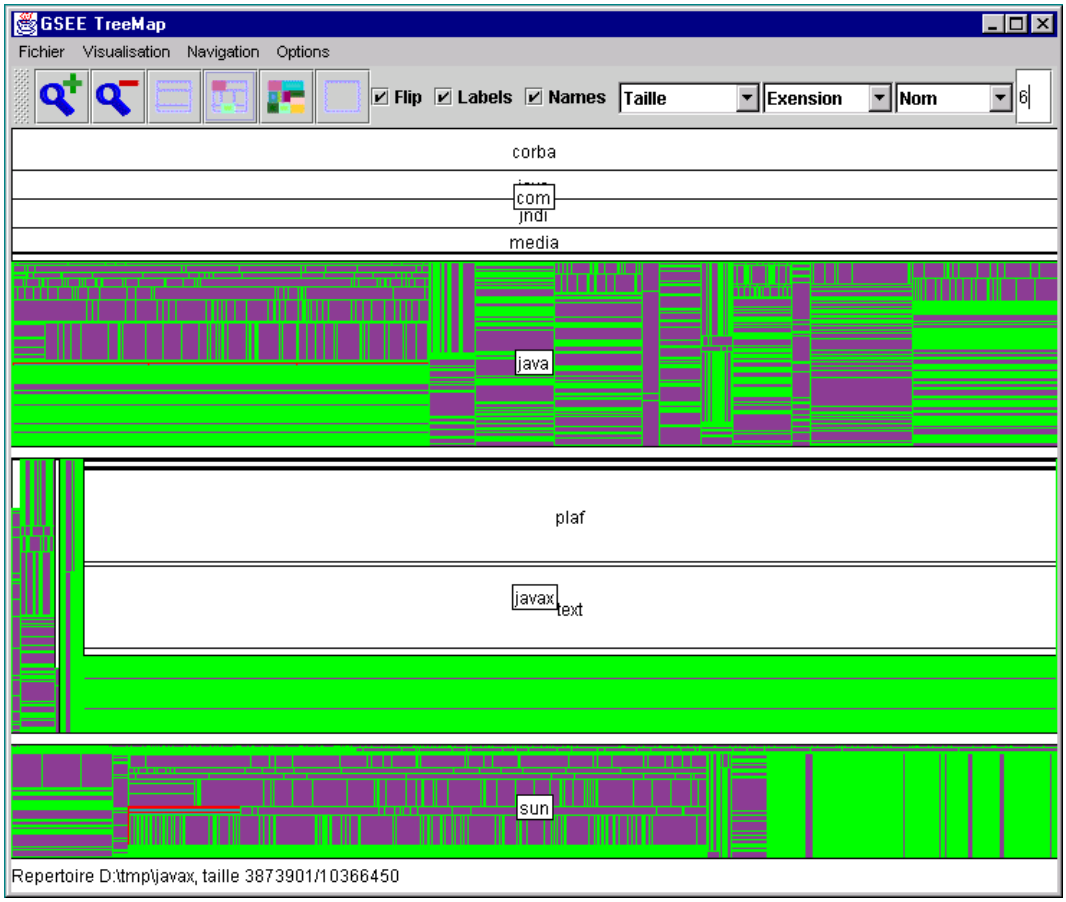
A *G*^{SEE} (*Generic Software Exploration Environment*) [9] egy Java alapokon készült objektumorientált szoftvermodell-vizualizációs keretrendszer, melynek elsődleges célja, hogy a felhasználó a rendelkezésre álló szoftvermodellhez a lehető legkevesebb új kód megírásával készíthessen új grafikus nézeteket. A megjelenítésre példa a 2.6. és a 2.7. ábrákon látható.



2.6. ábra. *G*^{SEE} Viewer

A rendszer alapjául szolgáló modell nyelvfüggetlen: tetszőleges, strukturált adathalmaz reprezentálására alkalmas. Alapeleme és elsődleges kiterjesztési pontja az ún. *Successor* interfész. Az interfész egyetlen metódusa a `Collection getSuccs(Object o)`, mely előállítja egy tetszőleges modellbeli objektum rákövetkezőinek gyűjteményét. Különböző operátorok (unió, kompozíció, tranzitív lezárt stb.) segítségével az interfész egyes megvalósításai kombinálhatók, ezáltal egyszerű elemekből építhetők fel bonyolultabb struktúrákat kezelő modellek.

A keretrendszer részét képezik a *Successor* interfész különböző, alapértelmezett megvalósításai is, melyekkel elérhetőek többek közt egyszerű ASCII szöveges fájlok, relációs és objektumorientált adatbázisok, adatbázis-lekérdezések eredményei vagy memóriában tárolt adatszerkezetek (szótárak, hasítótáblák). A *G*^{SEE} érdekessége, hogy a megjelenítő komponensek leírása is a *Successor* interfész megvalósításával, illetve megvalósítások kombinálásával történik.



2.7. ábra. *G*^{SEE} TreeMap

A *G*^{SEE} definiál egy egyszerű, funkcionális paradigmán alapuló nyelvet – és ehhez értelmezőt – is, amelyben a kompozíciós műveletek segítségével leírhatók különböző paraméterezett *Successor*-implementációk, illetve ezek alkalmazhatók adatforrásokra, majd az eredményül kapott modell megjeleníthető.

2.3.2. *InfoVis* eszközkészlet

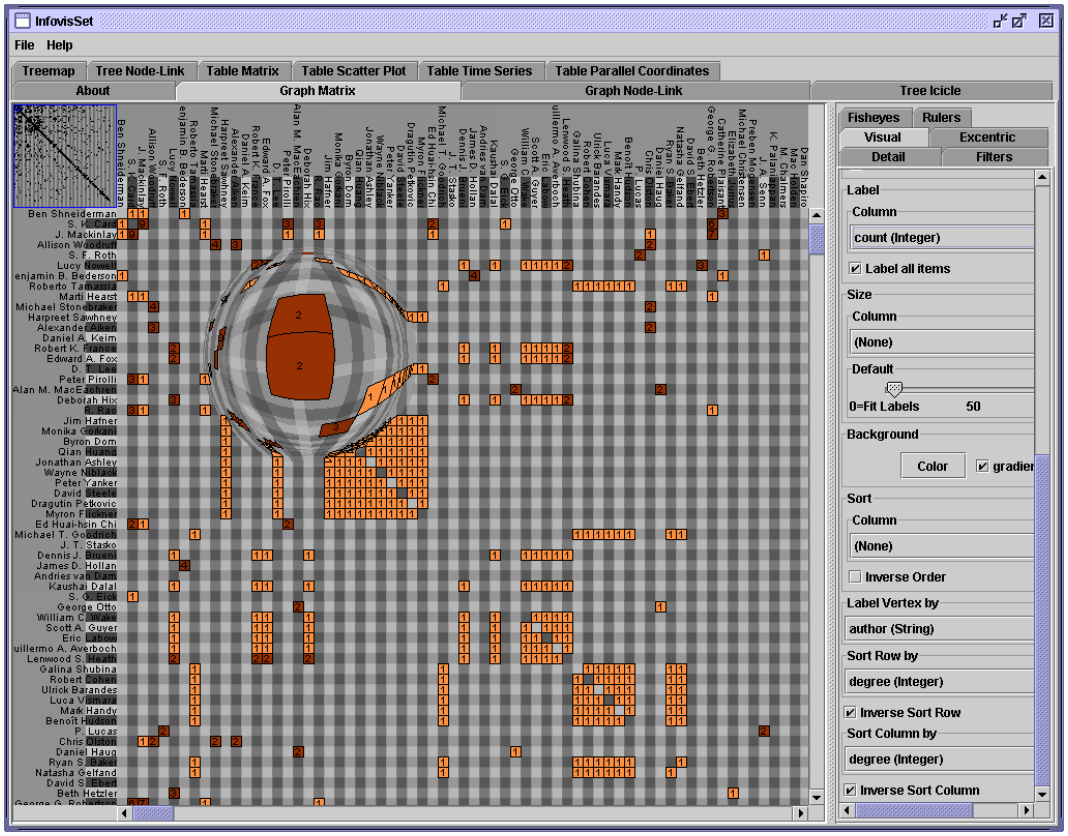
Az *InfoVis* eszközkészlet [10] a *G*^{SEE}-hez hasonlóan Java nyelven készült. Táblázatszerű adathalmazok, valamint fák és általános gráfok megjelenítését teszi lehetővé különböző formákban a Swing[6] segítségével, *OpenGL*[7] támogatással.

Az *InfoVis* adatmodelljének alapja az ún. *táblázat*. Az táblázat oszlopainak típusa uniform, azaz egy oszlop minden sorában azonos típusú adatok találhatók.

[6]Java nyelvű ablakozó keretrendszer [11].

[7]Az *Open Graphics Library* vagy röviden *OpenGL* egységes programozási felület 2D és 3D grafikus feladatok végrehajtására. Jellemzően grafikus processzorok programozására használják. `http://www.opengl.org`

Különböző beépített megvalósítások elérhetők az eszközkészlet részeként, például speciálisan gráfokhoz (ahol a táblázat a csúcsok szomszédsági mátrixát tárolja), illetve fákhoz. A gráf szomszédossági mátrixának megjelenítésére példaként ld. a 2.8. ábrát.



2.8. ábra. InfoVis Graph Matrix

Az eszköz nagy hangsúlyt fektet a különböző nézetekre. Ezek a megjelenítés egyes jellemzőit (szín, címke, méret, pozíció-korlátozások, a felhasználó által aktuálisan kiválasztott elemek stb.) saját oszlopokban tárolják, és mind ezeknek az attribútumoknak, mind az alapul szolgáló adatmodell adatainak változásáról értesülnek, és automatikusan újrarajzolják a szükséges területeket. A nézetek biztosítanak a megjelenítés beállításainak megváltoztatásához interaktív ablakfelületet is.

2.3.3. *CodeCrawler*

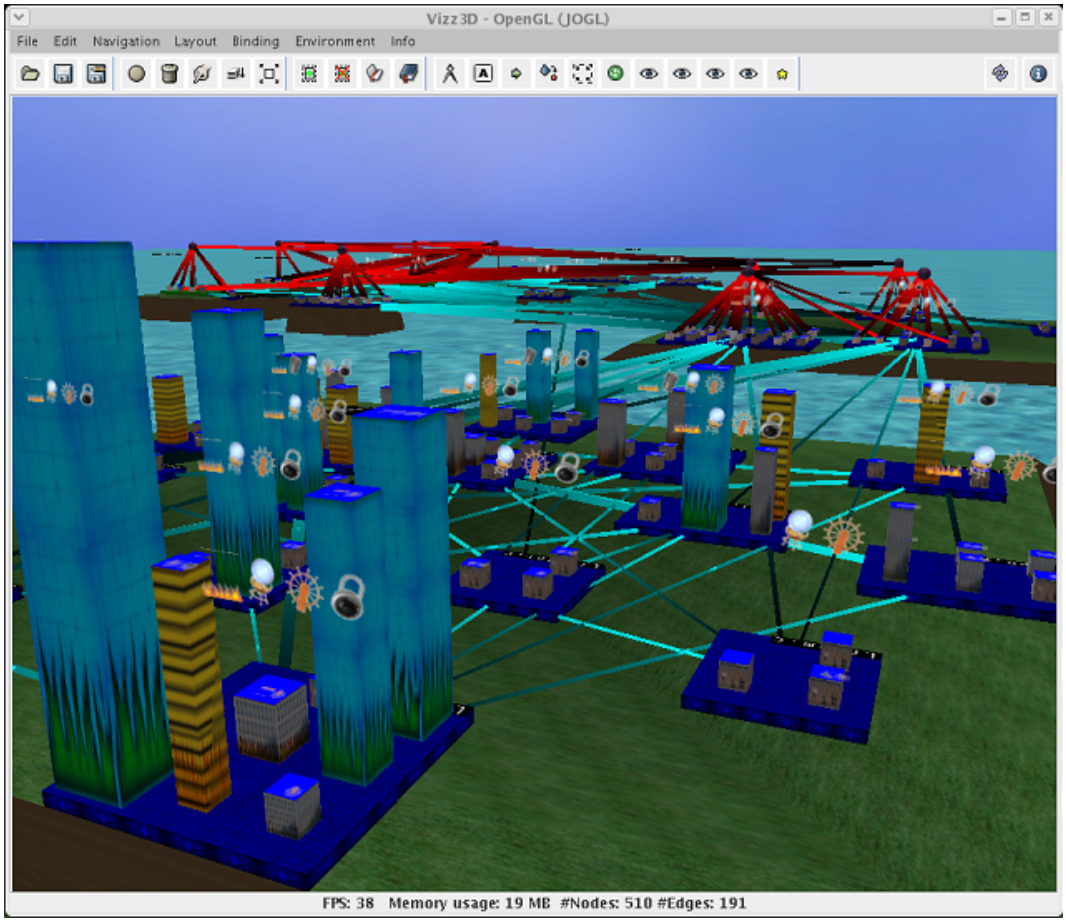
A *CodeCrawler* [12] a *MOOSE*-ra (2.2.1. szakasz) épülő megjelenítő eszköz, mely számos nézettel rendelkezik: ezek nagy része ún. *polimetrikus nézet*, azaz a grafikus elemek különböző paramétereiben (pl. méret, szín) jelennek meg szoft-

vermetrikák. Ezekkel nyomon követhető a vizsgált programrendszer szerkezete, a kódelemek kapcsolatrendszere, illetve a kód evolúciója.

A *CodeCrawler*-ról sem technikai információ, sem forráskód nem fellelhető, ezért a megvalósítás részleteit nem tudtuk megvizsgálni.[8]

2.3.4. C/C++ kód architekturális vizualizációja

A [13] cikkben bemutatott eszköznek szerzői nem adtak fantázianevet. Célja speciálisan C és C++ nyelű kódok architektúrájának megjelenítése a programmegértés támogatására. A megvalósítás Java-ban és C++-ban készült, a *Babel* eszköz [14] segítségével kapcsolva össze a különböző nyelveken készült kódokat.



2.9. ábra. *Vizz3D* grafikus motor *város metafora* nézete

[8] *CodeCrawler* néven elérhető a világhálón egy Apache Tomcat alkalmazásszerverre épülő eszköz, mely az internetes keresőmotorokhoz hasonló keresést tesz lehetővé szoftverek kódbázisán – ez azonban nem kapcsolódik sem az itt tárgyalt *CodeCrawler* alkalmazáshoz, sem dolgozatunk témájához.

Az eszköz a forráskód feldolgozására a *Rose* fordító-infrastruktúrát [15] (illetve annak *ROSEVA* nevű kiterjesztését) használja fel, míg a megjelenítést a *Vizz3D* [16] grafikus motor végzi.

A felépített modell C/C++-specifikus fogalmak mentén definiált csomópontokból és azok között futó élekből áll. A csomópontok, valamint az élek két előre definiált (*label*, *type*) szöveges címkével, illetve további tetszőleges attribútumokkal láthatóak el. A modell érdekessége, hogy a kapcsolatok szintjén redundáns információkat is tárol (pl. forrásfájlok közötti hívási függőségre külön élt tárol, ami a két fájlban található függvények között futó hívási élek aggregációja). A felhasználás gyakorlati korlátja a modell entitásainak száma: a rendszer legfeljebb 1 millió csomóponttal, 50 forrásfájllal, illetve 10 ezer kódsorral képes hatékonyan dolgozni.

A megjelenítés számára [13] követelményeket is megfogalmaz, amelyeket a dolgozat későbbi részében elemzünk. A megjelenítés elsősorban a *város metafora* mentén történik (a program entitásait épületek, azokból álló városok stb. reprezentálják) 3 dimenziós térben, melyben a felhasználó szabadon barangolhat, közelíthet vagy távolíthat (ld. a 2.9. ábrát).

2.4. További eszközök

Az eddig említetteken túl számos egyéb szoftvermodell-, illetve adatmegjelenítő eszköz lelhető fel az interneten, amelyeket a részletesebb elemzés igénye nélkül említésre érdemesnek gondoltunk. Ezeket gyűjtöttük össze itt[9].

A *PCVIA* projekt A *PCVIA* projekt (*Program Comprehension by Visual Inspection and Animation*) [17] olyan eszközök kifejlesztésére irányult, melyek segítségével algoritmusok (függvények törzse, illetve függvényhívási gráfok) statikus, illetve dinamikus tulajdonságai vizualizálhatók. A dinamikus tulajdonságok felderítéséhez az eszköz módosítja a forráskódot (*code instrumentation*). Nagy méretű objektumorientált szoftverrendszerek magas absztrakciós szinten való elemzését nem is célozza. Érdekessége az algoritmusok elemzéséhez bevezetett ún. *pattern tree* (mintafa) reprezentáció: ennek az elemei a működés szempontjából lényeges minták, például *értékadás*, *kiírás*, *függvényhívás* stb.

[9]Egyes, itt szereplő eszközök programkódok szerkezetének megjelenítésére nehezen vagy egyáltalán nem alkalmazhatóak, ezeket valamilyen érdekességük miatt mutatjuk be mégis röviden.

X-Ray Az *X-Ray* [18] egy BSc szakdolgozat keretében készített Eclipse-beépülő, mely Java-kódok statikus nézeteinek (rendszerkomplexitás, osztály- és csomagfüggőségek) megjelenítését teszi lehetővé. Az eszközt 2008 óta nem fejlesztik, a hozzá kapcsolódó dolgozat nem elérhető.

Javac AST Az eszköz [19] a *NetBeans* fejlesztői környezetet bővíti a Java fordító (`javac`) által generált szintaxisfa grafikus megjelenítési lehetőségével. A megoldás jól modularizált. A fordító által generált szintaxisfából kiindulva egyszerű (egyforma csomópontokból és élekből álló) fát épít, majd ezt jeleníti meg a forráskód mellett. A forráskód és a fa között a kiválasztás szinkronizált, azaz ha a felhasználó (pl. kattintással) kiválaszt egy elemet a szintaxisfában, akkor a forráskódban kijelölésre kerül az adott elemhez tartozó szakasz.

A *Rose* fordító A *Rose* fordító-infrastruktúra [15] a fordítás során felépített absztrakt szintaxisfából képes *DOT* [20] gráfleíró formátumú állományt generálni. A *DOT* formátumból kép generálható a *graphviz* [21] eszköz segítségével.

IBM: ManyEyes Az *IBM* egy kísérleti projektje a *ManyEyes* vizualizációs webszolgáltatás [22], mely rengeteg formában képes megjeleníteni táblázatba rendezett adatokat. Ingyenes regisztrációt követően saját adatokat tölthetünk fel, ezen túlmenően akár új vizualizációkat is készíthetünk.

The Visual Code Navigator Az *Eindhoveni Műszaki Egyetem* által fejlesztett *VCN* [23] (teljes nevén *Visual Code Navigator*) a CVS verziókövető rendszer adatmodelljére épülő szoftvervizualizációs alkalmazás. Fejlesztői a hangsúlyt elsősorban az ún. *evolúciós nézetre* helyezték, mely a forrásfájlok revíziói közötti különbségek kódstruktúrális elemzésére és megjelenítésére szolgál. Érdekes funkciója a programnak a szimbólum nézet, amely a forrásfájlok fordításából előálló tárgykódok megjelenítését szolgálja *treemap*[10] formájában az eltárolt szimbólumok alapján.

C++ programok evolúciójának elemzése A *Structural Analysis and Visualization of C++ Code Evolution using Syntax Trees* címmel megjelent cikk [25] a C/C++

[10] A *treemap* faszerkezetű adatok olyan megjelenítési formája, melyben a fa csomópontjait egymásba ágyazott téglalapok reprezentálják. A beágyazott téglalapok mérete szoftverek esetén sokszor valamilyen metrikával (például a csomópont által reprezentált forráskód-részlet sorainak számával) arányos. [24]

programok verzióinak a kódstruktúra alapú összevetésére vonatkozó módszertant tárgyalja a népszerű verziókövető rendszerekben megszokott sor vagy bájt alapú összehasonlítás helyett. A cikk alkalmazási példákat is bemutat gráf, illetve *tree-map* alapú vizualizációval, de konkrét szoftvertermék nem kapcsolódik hozzá. Az alkalmazott módszer elsősorban a kód kis és közepes méretű eltérései esetén használható.

AST-Vis A *Visualizing the Haskell AST* [26] címmel megjelent cikk és a hozzá tartozó, bő száz soros *AST-Vis* program egy igen egyszerű eszköz *Haskell* programok absztrakt szintaxisfájának szövegszerű, indentált megjelenítésére.

Esprima A *Esprima* [27] elsődlegesen *ECMAScript* [28] szabvány[11] szerinti *parser JavaScript* nyelven implementálva. A fő felhasználási területe a szintén *JavaScript* nyelven írt alkalmazások elemezése. A program a kódelemzés eredményét egyszerűen *JavaScript* objektumok szintaxisfájaként bocsátja rendelkezésre, melyhez csak nagyon kezdetleges, szövegszerű nézeteket kínál. Ezt felhasználva azonban már valós környezetben hasznosítható alkalmazások (például dokumentációs eszközök, Eclipse-plugin) is épülnek a programra.

2.5. Összefoglalás

A fejezetben áttekintettük a szoftvermodellezés és -vizualizáció történetét, valamint megvizsgáltunk számos szoftvereszközt, amelyek ilyen céllal készültek. A vizsgált eszközök között közös jellemzőket és eltéréseket egyaránt felfedezhetünk.

A megvalósítás nyelve a legtöbb részletesebben elemzett eszköz esetében a *Java* volt. Ez alól kivétel a *MOOSE* (SmallTalk) és a *SolidFX* (C++). A 2.3.4. pontban tárgyalt szoftver *Babel* környezetben, a Java mellett C++ használatával készült.

Az eszközök által készített, illetve alapul vett modellek között jelentős különbségeket találunk. Ezek közül nyelvfüggetlen szoftverreprezentációra a hierarchikus szerkezetű[12], de paradigmafüggetlen modellek a legalkalmasabbak: ilyennel a fentiek közül a G^{SEE} rendelkezik. A speciális nyelvi konstrukciókhoz igazított vagy az objektumorientált fogalmakra épülő modellek (a 2.2.1., 2.2.2. és 2.3.4. szakaszokban tárgyaltak) túl sok megkötést jelenthetnek, vagy a modell eredeti szemantikájától eltérő használatot igényelhetnek más jellegű (például funkcionális) programozá-

[11] ISO/IEC 16262:2011

[12] Az absztrakt szintaxisfa hierarchikus (fa) szerkezetű.

si nyelvek modellezése esetén; míg a hierarchia ábrázolására csak közvetett módon képes (pl. a 2.3.2. pontban bemutatott táblázatos) forma túl általánosnak mondható.

Hasonlóan széles szórást mutat a bemutatott kódmegértést támogató eszközöknek a kezelt probléma méretére vonatkozó skálázhatósága is. Míg egyes alkalmazások (köztük pl. a *MOOSE* és a *SolidFX*) nagyobb mennyiségű forráskódon is tesztelten helyesen működnek, addig a több hasonló eszköz (mint pl. a *FeatureIDE*) csak kis vagy közepes méretű szoftvereken került kipróbálásra, vagy akár már elméletileg is szűk korlátai vannak a kezelhető probléma maximális méretének, ahogyan azt láthattuk a 2.3.4. szakaszban bemutatott eszköz esetében.

A vizualizációs alkalmazások között megjelennek egy- és többnézetes változatok is. A 3. fejezetben érvelünk majd az utóbbi megoldás mellett, kiemelve ugyanazon programrész több nézete közötti szinkronizáció fontosságát.

A kódstruktúra vizualizációja során természetes módon (a szöveges forma mellett) a faszerkezetű megjelenítések a legkézenfekvőbbek, mivel a forrásszövegek szerkezete általában ilyen jellegű. Ezen kívül több alkalmazásban találunk *treemap* nézetet, illetve függőségi gráfokat különböző elrendezésekben. A különböző metrikák grafikus megjelenítése jellemzően a szerkezeti vizualizáció elemeibe komponálva jelenik meg.

Az alábbi, 2.1., 2.2. és 2.3. táblázatokban összegezzük az elemzett eszközök vizsgálata során nyert tapasztalatainkat és észrevételeinket. Az összefoglalóból látható, hogy a jelenleg rendelkezésre álló eszközök a szoftvervizualizáció hatékony támogatásának igényeit csak speciális esetekben, részlegesen, vagy komoly kompromisszumok árán képesek kielégíteni.

Mivel a 2.3.4. szakaszban bemutatott eszköznek nem adtak nevet, ezért a továbbiakban *ArchViz*-nek hívjuk. A *Structural Analysis and Visualization of C++ Code Evolution Using Syntax Trees* címmel megjelent cikk [25] elsősorban módszertant mutat be alkalmazási példákkal. Mivel konkrét szoftvertermék nem kapcsolódik hozzá, ezért nem szerepeltetjük itt.

Szoftvermodellező eszközök	
Eszköz neve	*Tapasztalataink és észrevételeink*
MOOSE	Beépülők segítségével kényelmesen kiterjeszthető, nagy méretű projektek elemzésére is alkalmas eszköz. Hátránya, hogy modellje szigorúan objektumorientált fogalmakra épít, illetve hogy a megvalósítására használt *SmallTalk* nyelv napjainkban háttérbe szorult. Vizualizációs alkalmazásokban történő valós felhasználásáról kevés az információ (ld. 2.3.3).
SolidFX	Professzionális eszköz magas szintű vizualizációs lehetőségekkel. Korlátja, hogy sem a szoftver, sem annak forrása nem használható fel szabadon, végfelhasználói termékként pedig csak a *Windows* platformot támogatja.
FeatureIDE	A termék egy Eclipse-plugin, ezért erős megszorítások jelentkeznek a platformfüggetlenség és a nyelvfüggetlenség tekintetében. Az alkalmazás modelljének kiértékelése csupán kis méretű szoftvereken történt, így hatékonysága és skálázhatósága nem tisztázott egy ipari méretű szoftver esetén.

2.1. táblázat. A megvizsgált szoftvermodellező eszközök erősségeinek és hátrányainak tömör összevetése

Adatmegjelenítő (vizualizációs) eszközök	
Eszköz neve	*Tapasztalataink és észrevételeink*
G^{SEE}	A *Successor* interfész általánossága nagy rugalmasságot biztosít a megjeleníthető modellek terén. Skálázódásáról, illetve gyakorlati felhasználásáról nincs információ.
InfoVis	Az alkalmazott adatmodell (*táblázat*) szoftvermodellek megjelenítésére kevésbé alkalmas, mivel hierarchikus szerkezetek vagy gráfok leírása ilyen környezetben nehézkes és nem intuitív. A megjelenítő komponensek grafikus képességei látványosak.

Eszköz neve	*Tapasztalataink és észrevételeink*
ArchViz	Amellett, hogy a C és C++ nyelvekre specifikus, tehát csak ilyen nyelvű forráskódok feldolgozására alkalmas, a kezelhető kódméretre vonatkozó korlátai bizonytalanná teszik ipari szoftvertermékek elemzésére való képességét.

2.2. táblázat. A megvizsgált adatmegjelenítő eszközök erősségeinek és hátrányainak tömör összevetése

További eszközök	
Eszköz neve	*Tapasztalataink és észrevételeink*
PCVIA	A *PCVIA* projekt célja alapvetően eltér a dolgozat témájától. Az eszközt érdekes, minta-alapú szoftverreprezentációja miatt mutattuk be röviden.
X-Ray	Az alkalmazás Eclipse-beépülőként került megvalósításra, Java forráskódok statikus, rendszerszintű nézeteinek megjelenítésére képes. A programot régóta nem fejlesztik.
Javac AST	Az eszköz a NetBeans integrált fejlesztői környezet beépülőjeként került megvalósításra. Előnye az ugyanazon modellt megjelenítő nézetek szinkronizációja. Hátránya platform- és környezetspecifikussága mellett az erős nyelvfüggősége is, hiszen csak Java programok elemzését és megjelenítését teszi lehetővé.
Rose fordító	A *Rose* érdekessége, hogy az önálló megjelenítést támogató komponenssel egyáltalán nem rendelkezik. Helyette az absztrakt szintaxisfát *DOT* formátummá alakítja, ami további eszközzel – pl. a *graphviz*-zel – jeleníthető meg.
IBM: ManyEyes	Ez a kísérleti webalkalmazás elsődlegesen a vizualizáció kiterjedt támogatására helyezte a hangsúlyt. Dolgoztunk témájához csak érintőlegesen kapcsolható, ugyanis bemenete specifikus módon táblázatba rendezett adatok.

Eszköz neve	*Tapasztalataink és észrevételeink*
Visual Code Navigator	A *VCN* koncepcionálisan különbözik a többi vizsgált eszköztől, ugyanis a hangsúlyt elsődlegesen nem a kódmegértésre, hanem a szoftverevolúció megértésének támogatására helyezi a modell és a vizualizáció oldaláról egyaránt. Érdekes megjelenítő nézeteivel szemben hátrányát kepezik erős függőségi megszorításai: kizárólag C/C++ kód elemezésére alkalmas CVS verziókövető rendszeren.
AST-Vis	Ez az alkalmazás nem rendelkezik valós szoftvervizualizációs támogatással, ugyanis Haskell programok elemzésén és szintaxisfájának felépítésén túlmenő magasabb, a megjelenítést segítő interfészt nem kínál.
Esprima	Az eszköz speciálisan az *ECMAScript* szabvány nyelveihez készült kódelemző program. Vizualizációs nézetekkel történő kiterjesztése nehézkes, mert a modellt csak alacsony szintű reprezentációban bocsátja rendelkezésre.

2.3. táblázat. A további megvizsgált vizualizációs alkalmazás erősségeinek és hátrányainak tömör összevetése

3. fejezet

Elemzés és tervezés

Mint azt a 2.5. szakaszban láttuk, hiányként jelentkezik a szoftvervizualizációs eszközök fejlesztésének támogatására egy nyílt forrású, általános, mégis hatékony keretrendszer. A dolgozat témájában foglalt szoftverkomponens ezt a hiányosságot igyekszik pótolni.

Ebben a fejezetben először elemezzük azokat a követelményeket, amelyek általában egy szoftvervizualizációs eszköztől elvárhatók, majd lefektetjük konkrét elvárásainkat a dolgozat keretében fejlesztett keretrendszerrel szemben. Ezután bemutatjuk azokat a tervezési megfontolásokat, amelyek mentén a kitűzött célokat megvalósítottuk, illetve átfogó architekturális képet adunk a keretrendszerről.

3.1. Követelmények elemzése

A minőségi szoftverfejlesztés szellemében első lépésként a programmal szemben támasztott követelményeket kell lefektetni. Elvárások adódhatnak a végfelhasználói program konkrét funkcióiból, illetve megállapíthatunk olyan általános követelményeket is, amelyek a szoftvermodellezés és -vizualizáció területén minden, valós környezetben használni kívánt alkalmazástól elvárhatóak. Az előbbi szempont elemzése a keretrendszerünkre építő programozó feladata, így itt az utóbbi nézőpont szerinti követelményeket fogjuk bemutatni, két csoportra osztva őket aszerint, hogy az elvárások az eszköz fejlesztői vagy felhasználói oldaláról állnak-e fenn [2]. Ahol szükséges, példákkal egészítjük ki a magyarázatot.

Fontos megjegyezni, hogy legtöbbször (mint a mi esetünkben is) nem csak a fejlesztői, hanem a felhasználói oldal is programozókat takar ezeknél az eszközöknél.

A rendszerrel szemben támasztott fejlesztői követelmények

Nyelvfüggetlenség A megjelenítéshez alkalmazott modell legyen általános, ne épüljön speciálisan egy programozási nyelvre vagy paradigmára.

Skálázhatóság A vizualizáló programnak kellő mértékben skálázhatónak kell lennie a kezelt probléma méretének tekintetében. A kódstruktúra elemzésének esetében ez vonatkozik például arra, hogy a megjelenítő komponensnek különböző részletességű áttekintő nézeteket kell nyújtania az elemzett alkalmazásról.

Kiterjeszthetőség A program modellező és megjelenítő rétegének egyaránt kellően flexibilisnek kell lennie ahhoz, hogy a későbbiekben új ötletekkel – például új adatokkal vagy nézetekkel – bővíteni lehessen.

Hordozhatóság Az elemző eszközökkel szemben gyakran kívánalom a platformfüggetlenség, amire már a tervezéskor célszerű hangsúlyt fektetni, mind a program működési elvének kidolgozásakor, mind a felhasznált szoftvereszközök (programozási nyelv és környezet) kiválasztásakor.

A rendszerrel szemben támasztott felhasználói követelmények

Minimális behatás Az elemzett alkalmazás forráskódját a lehető legkevésbé kelljen megváltoztatni annak érdekében, hogy vizualizálható legyen. Ez olykor - például az algoritmusok működését animáló rendszerek esetén - elkerülhetetlen, de a kódstruktúrát statikusan elemző programok rendszerint nem igényelnek ilyen behatást, és ezzel teljes mértékben teljesítik ezt a követelményt.

Minimális beavatkozás Az elemző program automatikusan el tudja készíteni az egyes nézeteket, a lehető legkevesebb – lehetőleg nulla – beavatkozást várja el a felhasználói oldal programozóitól. Ha ez a követelmény nem teljesül, az egyrészt fárasztó a felhasználó számára, másrészt a hibák elsődleges forrása lehet.

Valós problémák kezelése A különböző szoftverelemző és -vizualizáló eszközök összevetésekor lényegi szempont, hogy milyen jellegű és méretű alkalmazásokat tudnak feldolgozni. Egy termék szoftveripari felhasználásához elhagyhatatlan elvárás, hogy általános és nagy méretű programokat is kezelni tudjon az elemző eszköz. A fejlesztés első fázisaiban ez még gyakran nem teljesül, de a későbbi verzióknak mindenképpen teljesíteniük kell ezt a követelményt.

Megjelenített információk A felhasználói interfész szempontjából egy szoftvervizualizációs eszköz akkor jó, ha a megfelelő információkat a megfelelő módon jeleníti meg. Az ehhez szükséges mechanizmus természetesen függ a konkrét alkalmazás specifikumaitól, és a felhasználó szemszögéből is szubjektív fogalom, de általános elveket megfogalmazhatunk:

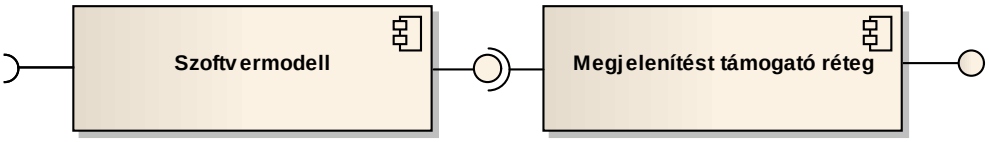
- Az eszköz nézetei a kitűzött cél szempontjából fontos és hangsúlyos aspektusait jelenítsék meg az elemzett programnak, amelyek nem csak a felhasználó addigi ismereteit erősítik meg, hanem elősegítik, hogy új információkhoz jusson az elemzett alkalmazásról, megismerhesse azt. A gyakran terjedelmes méretű forráskód hatékony és áttekinthető megjelenítése az adott céloknak megfelelő speciális nézetekkel valósítható meg [3, 29].
- Az elemző alkalmazás vezérlése és nézeteinek megjelenítése legyen a felhasználó által kellő mértékben testreszabható, hogy minél inkább saját igényeinek megfelelően használhassa.
- A megjelenítés adjon lehetőséget a vizuális elemek interaktív szűrésére, ezáltal a megjelenő információmennyiség szűkítésére; illetve egyes kiválasztott információelemek részletezésére (*semantic zooming*) [13].
- A vizualizáción belül történő navigáció (mozgás, közelítés/távolítás) során segíteni kell a vizuális kontextus megőrzését [13].
- Hasznos, ha azonos modellek vagy részletek különböző nézetekben egy időben megjeleníthetőek, mivel ez segít az adott komponens különböző aspektusainak megértésében, az azok közötti logikai kapcsolat felismerésében. Ebben az esetben fontos, hogy a nézeteket lehessen szinkronizálni (például az egyik nézet kiválasztott elemére fókuszálhasson a felhasználó a párhuzamos nézet(ek)ben).
- Az elemző eszköz felülete legyen igényes, a dekoráción túl információközlő módon is.

3.2. A keretrendszer architektúrája

A 3.1. szakaszban lefektetett követelmények mentén a dolgozat témáját képező keretrendszert a modell-nézet architektúra[1] alapján terveztük meg. Ennek lényege az adatok kezelésének és a felhasználó számára történő megjelenítésének éles elhatárolása, ahol a modell és a nézetek között kizárólag a modellek felé irányuló kapcsolat állhat fenn. A választott architektúra a keretrendszer számára modularizációt és széles kiterjeszthetőséget nyújt.

A nézetek az egyedi felhasználás konkrét igényeinek függvényében változnak, ezért ezek megvalósítása a konkrét vizualizációs alkalmazás fejlesztőjének feladata. A dolgozatban bemutatott keretrendszer a modell komponensre fókuszál, ebben a szakaszban ezt fogjuk részletesen elemezni.

A modell komponenst további két rétegre osztottuk: a *szoftvermodell* rétegére és egy, a *megjelenítést támogató* rétegre (ld. az UML[2] komponensdiagramot a 3.1. ábrán.).



3.1. ábra. A generikus keretrendszer komponens diagramja

A 3.1. ábrán látható **szoftvermodell** az absztrakt szintaxisfa (ld. a 2.2.) általánosítása olyan módon, hogy a gráf csúcsai közti kapcsolatrendszer tetszőlegesen definiálható. Ilyen módon lehetőség adódik a forráskód eltérő hierarchikus szinteken történő vizsgálatára[3]. A gráf csúcsairól a specifikáció szintjén nem tételezünk fel semmit, azonos típusúnak tekintjük őket. Szoftvermodellünkben a továbbiakban a csúcsokat *csomópontoknak*, az közöttük futó élek rendszerét *relációnak* nevezzük.

[1] A modell-nézet egy ismert tervezési minta, a modell-nézet-vezérlő – elterjedt rövidítéssel *MVC* – minta egyszerűsítése [30].

[2] Az UML [31] egy szabványos, általános célú modellező nyelv, melynek segítségével grafikus és szöveges modelleket készíthetünk egy szoftver statikus és dinamikus szerkezetéről, viselkedéséről stb.

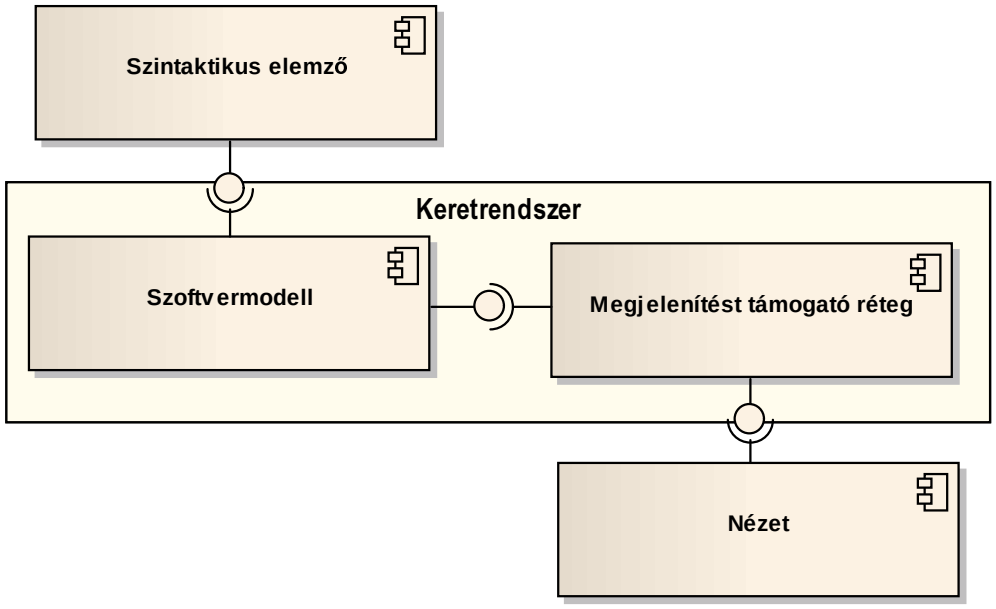
[3] Például az összes szintaktikus elemből felépített, teljes *absztrakt szintaxisfa*, csak a függvények közötti hívási kapcsolatok, esetleg egy objektumorientált nyelven készült programban az osztályok közötti leszármazási hierarchia is reprezentálható így.

A csomópontokról információt ún. *szolgáltatások* segítségével nyerhetünk. A szolgáltatás olyan osztály, amelynek metódusait egy vagy több csomóponton értelmezzük. Ezek a specifikáció szintjén absztraktak, definiálásuk a szoftvermodell konkrét implementációjára tartozik. A keretrendszer rendelkezik alapértelmezett szolgáltatás-interfészekkel (ld. később ezen szakasz szolgáltatásokat kifejtő részében), melyeken túl a felhasználó – saját igényei szerint – újakat is bevezethet.

A fent leírt szoftvermodell az elemzett szoftver szerkezetének alacsonyabb szintű, generikus leírására alkalmas, azonban a megjelenítéshez szükséges összetettebb információk előállítására már nem. Ezért bevezettük egy, a **megjelenítést támogató réteget**, amely a csomópontok, a relációk és a szolgáltatások összefogása mentén két funkciót valósít meg:

- egységes, magas szintű felületet biztosít a nézet komponens(ek) felé, melybe meglévő vagy új szolgáltatások is dinamikusan integrálhatók,
- lehetővé teszi a modell adatainak szűrését, transzformálását, illetve annotálását a megjelenítés számára.

A fent leírt módon a keretrendszerünk tetszőleges programozási nyelven készült forráskód szintaktikus elemezésének alacsony szintű kimenetét könnyen megjeleníthető, magas szintű formára alakítja (ld. a 3.2. ábrát).

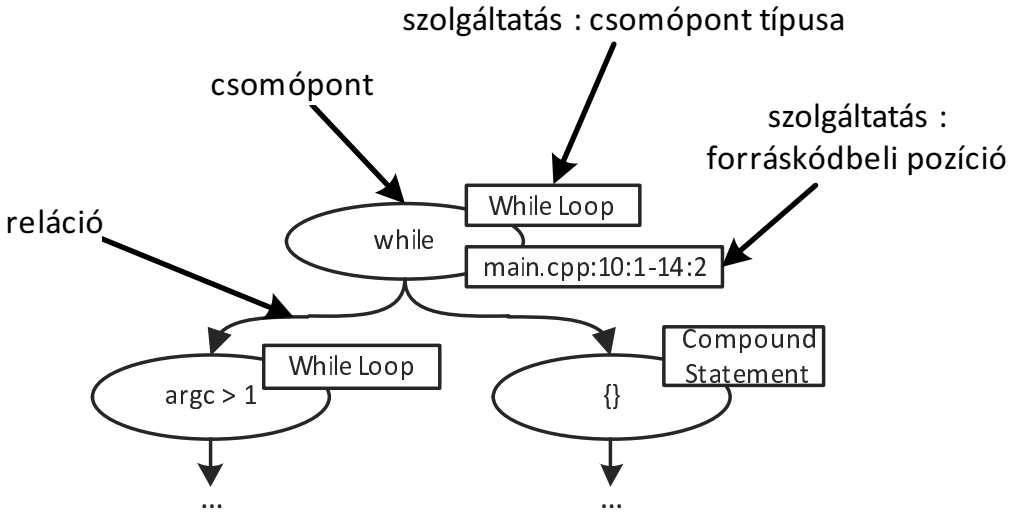


3.2. ábra. Az architektúra komponens diagramja a külső kapcsolatok ábrázolásával

3.3. A forráskód modellje

A keretrendszer szerkezetének általános áttekintése után ebben a szakaszban részletesen bemutatjuk a vizualizált szoftver modellezésére használt fogalmakat. A 3.3. ábrán egy C nyelvű forráskód szintaxisfájának kis részlete látható, melyen nyilak mutatják az egyes fogalmak szemléletes jelentését:

- a **csomópontot**, mely a fa egy csúcspontja (pl. egy *while* ciklus);
- a **relációt**, mely a gráfban futó éleknek felel meg (pl. a ciklus gyerekei egy logikai kifejezés és egy blokk-utasítás);
- különböző **szolgáltatásokat**, melyek a gráf csúcsainak címkéi, azokról szerezhető különböző információk (pl. a ciklus utasítás a *main.cpp* állomány 10. sorában található).



3.3. ábra. A szoftvermodell elemeinek szemléltetése absztrakt szintaxisfán

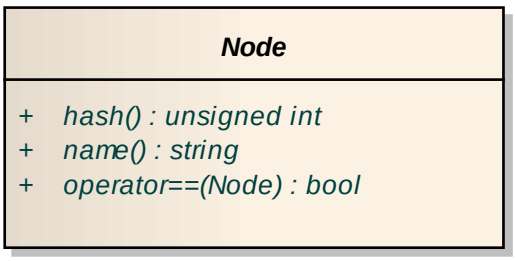
A csomópont

A *csomópont* a gráfként tekintett szoftvermodell egy *csúcsa* vagy *pontja*. Keretrendszerünk szempontjából egy absztrakt osztály, melynek megvalósítása felé az alábbi követelményeket támasztjuk (ld. a 3.4. ábrát).

- Legyen megállapítható két csomópontról, hogy azok azonosak-e (**egyenlőségvizsgálat**). Ez a megközelítés lehetővé teszi, hogy a modellt megvalósító programozó tetszőleges típusú azonosítókkal dolgozhasson, általánosabban: tetszőleges ekvivalenciarelációt definiálhasson a saját modelljének megvalósításában.

- A csomópontoknak a megjelenítést támogató rétegben való hatékony tárolásához és kereshetőségéhez a csomópontra egy **„hasító" metódus**[4] implementációjára is szükség van. Erről a metódusról a kertrendszer azon túl, hogy azonos csomópontra mindig azonos értéket szolgáltasson, nem tesz semmilyen – például statisztikai tulajdonságra vagy egyediségre vonatkozó – megkötést.[5]

- A csomópontnak rendelkeznie kell továbbá egy egyszerű szöveges attribútummal (*névvel*). Ennek sem kell feltétlenül egyedinek lennie, elsődleges célja, hogy alapértelmezett megjelenítést biztosítson az entitás számára akkor is, ha nem érhető el hozzá más információ.



3.4. ábra. Csomópont

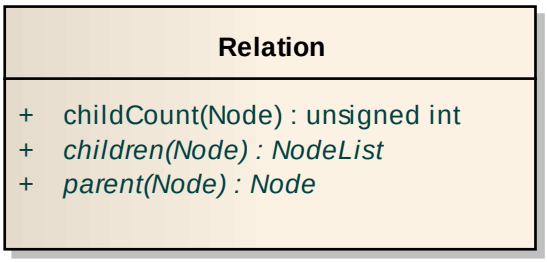
Reláció

A csomópontok alapvető hierarchiája faszerkezetű, ezt a kapcsolatrendszert egy ún. *reláció* biztosítja. Egy relációmegvalósítás két metódust tartalmaz, melyek a paraméterként kapott csomópont *szülőjét*, illetve *gyerekeinek rendezett listáját* állítják elő (ld. a 3.5. ábrát).

Egy végfelhasználói program, de akár egyetlen vizualizációs komponens is használhat egy időben több különböző relációt a megjelenítéshez. Relációk alacsonyabb szinten is kombinálhatóak, hiszen a reláció-interfész egy új konkretizálása felhasználhat már meglévő megvalósításokat. Így előállíthatják például két reláció metszetét, unióját vagy konkatenációját.

[4] A hasító függvény egy olyan algoritmus, amely – változó méretű – nagyobb adatszerkezetekhez fix méretű kulcsértékeket rendel determinisztikus módon.

[5] A megjelenítést támogató réteg hatékonysága szempontjából szerencsés, ha a függvény értékei kellően szóródnak, azaz a csomópontok többségére különböző értéket adnak.



3.5. ábra. Reláció

Megjegyzés. *Hasznos lehet például egy objektumorientált nyelven megírt forráskód megjelenítése az alábbi különböző kapcsolatrendszerek mentén: (a) az egyes kifejezések, utasítások közötti kapcsolat (b) az osztályok közötti öröklődési hierarchia (c) metódusok egymásra való hivatkozási (hívási) kapcsolatai. A felsorolt relációk mind különböző absztrakciós szinteken nyújtanak információt a vizsgált szoftverről.*

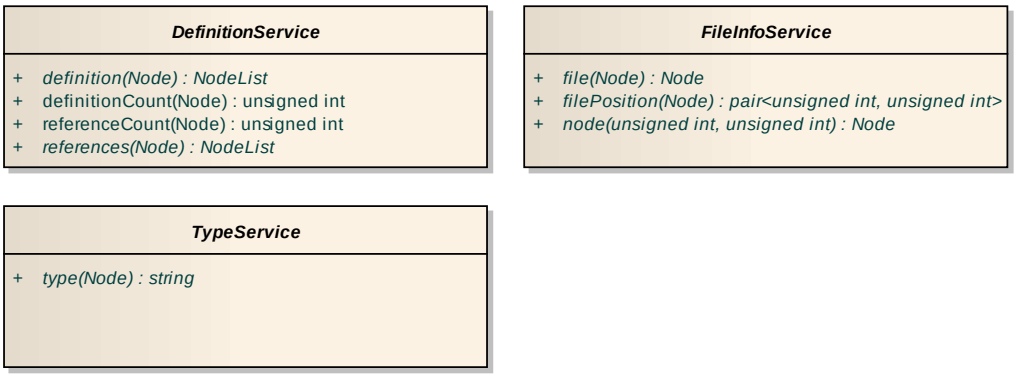
Szolgáltatások

Csomópontokról információ *szolgáltatásokon* keresztül nyerhető. A szolgáltatások olyan objektumok, melyek publikus metódusai paraméterként egy vagy több csomópontot várnak, és más csomópontokat, vagy tetszőleges egyéb adatot állítanak elő. A szolgáltatások nem rendelkeznek közös ősosztállyal, tetszőleges típusú objektum felhasználható szolgáltatásként, mely teljesíti a fenti feltételt. Ez nagyfokú flexibilitást biztosít a programozó számára.

A keretrendszerben a felhasználó kényelme érdekében definiáltunk három olyan interfészt, amelyeknek a konkrét szoftvermodellre vonatkozó megvalósítására egy szoftvervizualizációs alkalmazásnak jellemzően szüksége lehet[6] (ld. a 3.6. ábrát).

Definíció A *definíció* szolgáltatás a *reláció* kiterjesztése. A definíció olyan speciális, a szoftvermodellezésben előforduló relációfajta, amelyben „szülő” és „gyermekek” helyett jellemzően „definíciót” vagy „deklarációt”, illetve „hivatkozásokat” mondunk. A definíció szolgáltatás ezért a szoftvermodell logikájának világosabbá tételéhez átnevezi a reláció attribútumait és metódusait a fenti módon. A reláció kiterjesztéseként pedig a faszerkezet helyett általános gráfszerkezetet nyújt, ugyanis egy csomóponthoz több deklaráció is tartozhat.

[6]Ezek egyben példaként is szolgálnak arra, mit értünk szolgáltatás alatt.



3.6. ábra. Beépített szolgáltatások

Megjegyzés. *A programozási nyelvekben* dekaráció *egy név (például változó-, típus- vagy függvénynév) bevezetése, azaz annak kifejezése, hogy a programban létezik ilyen entitás. A* definíció *olyan deklaráció, mely egyben tárhelyet foglal (egy változó számára), vagy meghatározza az entitás működését (például egy függvény törzsét). Hivatkozás alatt azt értjük, mikor a program valamely részén egy korában bevezetett entitást a nevével (vagy más módon) azonosítva felhasználunk.*

Típus A típus olyan egyszerű szolgáltatás, mely a csomópont típusát karakterláncként írja le. Egy vagy akár több megvalósítása is hasznos lehet, ha a csomópont *név* attribútuma nem elegendő a megfelelő felcímkézésre.

Fájlinformációk Mivel a szoftverek általában forrásfájlok sokaságából épülnek fel, ezek (szintaxiskiemeléssel, navigációs és egyéb szolgáltatásokkal támogatott) szöveges megjelenítése természetes módon adódik. A fájlinformációs szolgáltatásinterfész ehhez nyújt segítséget az alábbi funkciókkal:

- adott csomóponthoz az őt tartalmazó forrásfájl csomópontjának hozzárendelése (ha van ilyen),
- a csomópontnak megfelelő forrásszöveg fájlon belüli pozíciójának megadása,
- forrásfájlon belüli pozícióhoz a tartalmazó csomópont megadása.

Megjegyzés. *Forrásfájlbeli pozíció csomóponthoz rendelése nem mindig egyértelmű. Például az* `int counter = 0;` *programsor első pozíciója a teljes változódefiníciónak és az* `int` *típusnévnek egyaránt része. Ilyen esetekben valamilyen heurisztikát érdemes használni a hozzárendeléshez. Jó módszer lehet például a pozícióhoz tartozó csomópontok közül azt választani, amelyik a szintaxisfa hierarchiájában legmélyebben található.*

3.4. A vizualizáció támogatása

Az előző szakaszban bemutatott absztrakt modell alkalmas szoftverek szerkezetének logikai leírására, illetve számítógépes kezelésére és tárolására. Közvetlenül ebből kiindulva azonban nehézkes lenne a kódmegértést támogató megjelenítések létrehozása, valamint sok kód megírását követelné a felhasználótól. Keretrendszerünk megjelenítést támogató rétege ezt a terhet veszi le a programozó válláról: lehetővé teszi a modell szűrését, transzformálását, továbbá megjelenítésspecifikus információk, attribútumok hozzáadását és a különböző nézetek közötti szinkronizációt[7].

3.4.1. Alapmodell

Ahhoz, hogy a fentebb leírt funkcionalitást elérjük, egy modell-osztályt (a *NodeModel* a 3.7. ábrán) vezettünk be, mely a szoftvermodell egy vetületét reprezentálja egy gyökérelem (*Node*) és egy reláció aggregációjával (ld. a 3.7. ábrát). A modell-osztály ilyen módon egységbe zárja (*enkapszulálja*) az alacsonyabb szinten megvalósított modell-fogalmakat, miközben egy magasabb szintű interfészt nyújt.

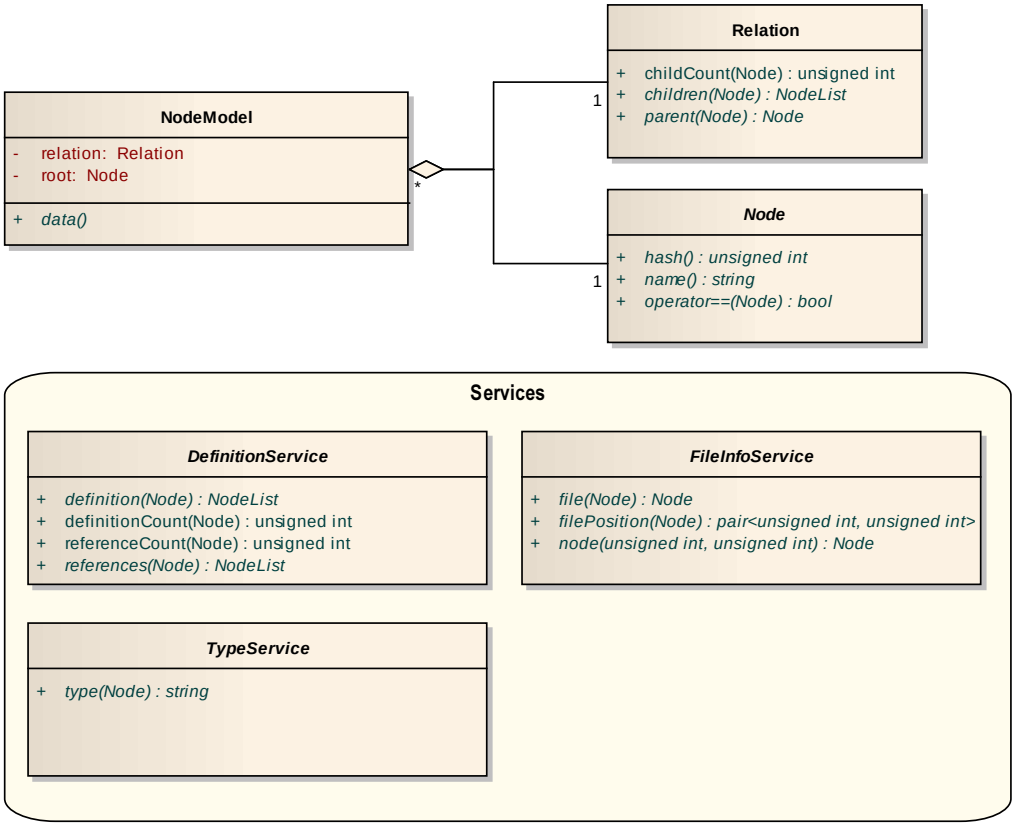
Az ábrán látható az is, hogy a szolgáltatások nem az alapmodellbe épülnek be. A modellnek a szolgáltatások segítségével vagy más módon történő igény szerinti bővítésére és transzformálására a 3.4.2. pontban bemutatott proxy-modellek által van lehetőség.

3.4.2. Proxy-modellek

A réteg alapvető fogalma a *helyettesítő* vagy *proxy-modell*[8]. A proxy-modellek egy tárgymodellhez kapcsolódnak, amellyel teljesen azonos publikus felülettel rendelkeznek. Ezért az tárgymodell a program tetszőleges pontján helyettesíthető a proxy-modellel, a modellre épülő kódok módosítása nélkül. A proxy-modellek így – a felsőbb rétegek számára észrevétlenül – tetszőlegesen módosíthatják a tárgymodelljük működését. A proxy-modellek láncolhatóak, azaz helyettes modell tárgya lehet maga is helyettes, így a módosítások hatása kombinálható.

[7] Ld. a követelményeket a 3.1. szakaszban.

[8] A *helyettes* vagy *proxy* egy tervezési minta, részletes leírása megtalálható például [30] könyvben.



3.7. ábra. A modell és a szolgáltatások kapcsolata

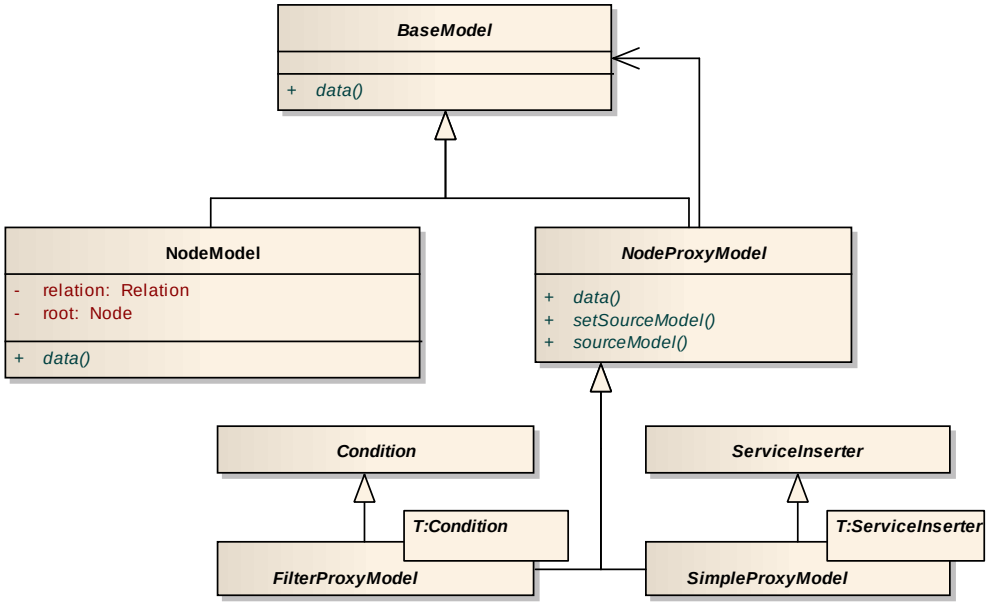
A 3.4.1. pontban bevezetett alapmodell és a proxy-modellek egységes interfészét a *BaseModel* közös absztrakt ősosztállyal biztosítjuk. Ezek kapcsolatrendszere a 3.8. ábrán látható.

Jól definiált proxy-modellek biztosításával a fent leírt módon a szűrési, illetve transzformációs feladatok jelentősen leegyszerűsíthetők. A keretrendszer proxy-modelljei számára a *NodeProxyModel* szolgál bázisosztályként (ld. a 3.8. ábrán). Ebből a keretrendszer részeként az alábbi „kényelmi” proxy-kat származtattuk:

SimpleProxyModel Ez a sablon osztály biztosítja új adatok hozzáadását a modellhez, melyek lehetnek szolgáltatás által nyújtott vagy egyéb (pl. a megjelenítést leíró) információk. Használatához egy egyszerű *ServiceInserter* interfészt kell megvalósítani (akár magában a szolgáltatásban), amellyel a *SimpleProxyModel* sablont példányosítva olyan proxy-modellt kapunk, mely bázismodelljét kibővíti a *ServiceInserter* által meghatározott szolgáltatásokkal.

FilterProxyModel Az előző sablon osztályhoz hasonlóan működő proxy sab-

lon paraméterként egy csomóponton értelmezett predikátumot igényel (*Condition*), mellyel példányosítva az előálló helyettes alapmodelljéből csak a predikátumnak megfelelő elemeket prezentálja a megjelenítés felé, míg a többit elhagyja.



3.8. ábra. A megjelenítést támogató réteg

A *NodeProxyModel*-ből közvetlen származtatással tetszőleges egyéb, bonyolult transzformációs feladat is megoldható.

Megjegyzés. *A keretrendszer – a* NodeProxyModel *leszármazottjaként – a 3.3. pontban tárgyalt alapértelmezett szolgáltatás-interfészek számára biztosítja a megfelelő proxy-modelleket is, melyek segítségével egy ilyen szolgáltatás a hozzá tartozó proxy-modell egyszerű példányosításával, a* ServiceInserter *interfész megvalósítása nélkül felhasználható.*

3.4.3. Szinkronizáció a megjelenítések között

A nézetek közötti szinkronizáció kérdése nem oldható meg a modell szintjén, hiszen a modell-nézet architektúrának megfelelően a modell réteg semmilyen információval nem rendelkezik az őt megjelenítő komponensekről. Mivel mégis támogatást kívántunk nyújtani ehhez a lényeges funkcióhoz, a megjelenítést támogató réteg egy egyszerű interfészt kínál a felhasználónak, melyet a megjelenítő komponensben megvalósítva automatikussá tehető a szinkronizáció két vagy több nézet között.

A *szinkronizációs interfész* három egyszerű metódust definiál:

- aktuálisan kiválasztott (fókuszban lévő) csomópont lekérdezése;
- adott csomópont kiválasztása (fókuszba hozása);
- az ún. *szinkronizációs modell* hozzárendelése a nézethez.

A szinkronizációs modell a *közvetítő* (*mediator*) [30] tervezési mintát követi. Több nézethez ugyanazon szinkronizációs modellt kapcsolva, ha az egyik nézet értesíti azt a fókusz megváltozásáról, akkor a közvetítő az eseményt továbbítja az összes, hozzá kapcsolt nézet felé. Így a szinkronizációhoz nincs szükség a nézetek páronkénti összekötésére, amely a felesleges kódolás mellett például ciklikus értesítéseket vagy más problémákat vonhatna maga után.

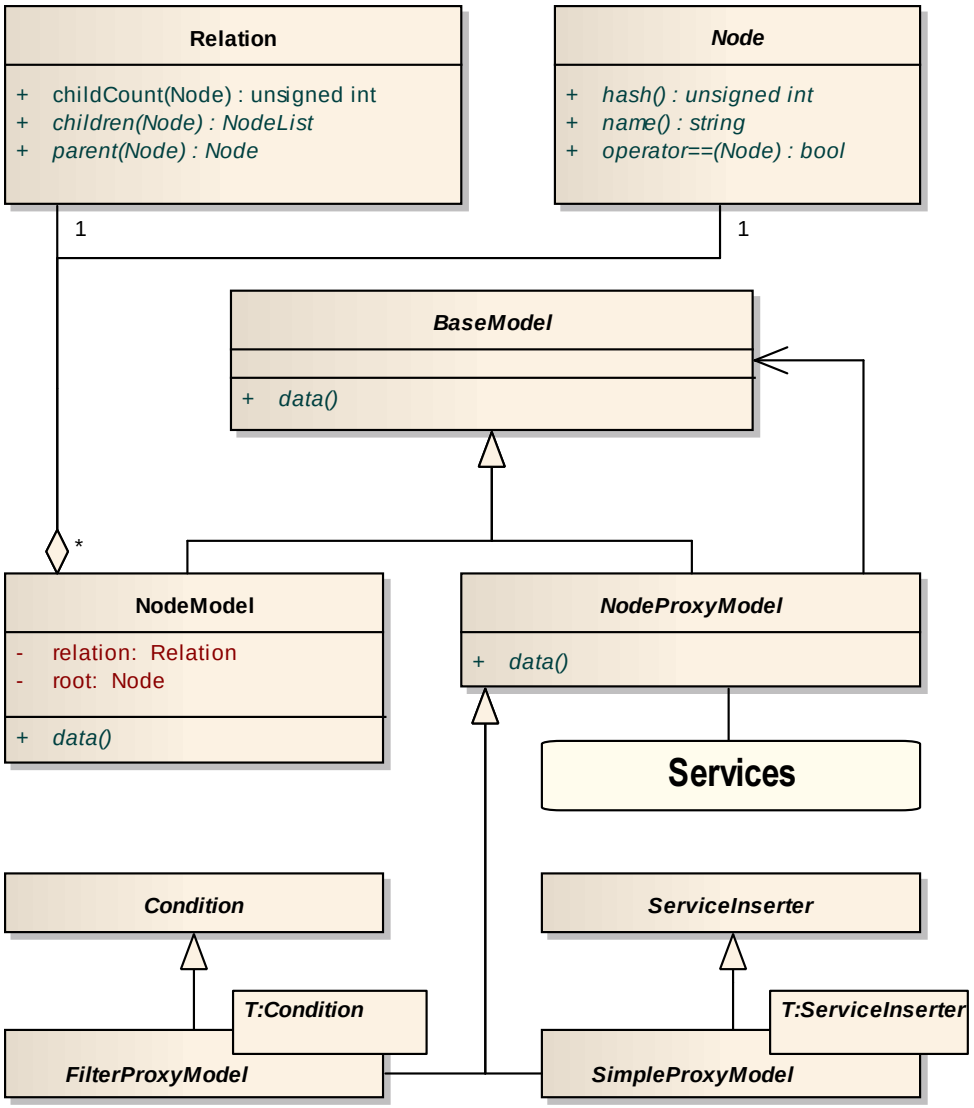
3.5. A keretrendszer áttekintése

A keretrendszer tervének a fejezetben bemutatott részletes elemzése alapján kijelenthetjük, hogy az a a 3.1. szakaszban megfogalmazott követelményeknek – illetve azok közül az ezen a szinten alkalmazhatóaknak – maradéktalanul eleget tesz.

A keretrendszer **nyelvfüggetlen**, mivel a kidolgozott szoftvermodell (3.3.) a hozzá kapcsolt elemző eszköztől függően tetszőleges hierarchikus szerkezet kezelésére képes, azaz különböző (nem csak imperatív, hanem funkcionális, logikai vagy más paradigmákra épülő) programozási nyelvek szintaktikus modelljét képes kezelni egységes formában. A **skálázhatóságot** a modellen definiálható különféle relációk, illetve a megjelenítés szintjén a proxy-modellek (3.4.2.) transzformációs képességei teszik lehetővé. A **kiterjeszthetőséget** elsősorban a tetszőlegesen definiálható szolgáltatások (továbbá ezeknek a megjelenítéssel való egyszerű összekapcsolhatósága) biztosítja. A **hordozhatóság** követelménye ezen a szinten azáltal teljesül, hogy tervek tetszőleges, az objektumorientált programfejlesztést támogató nyelven könnyen, minimális módosítás mellett megvalósíthatók, mivel nem építenek egyetlen nyelv speciális eszközeire sem.

A keretrendszer nem kíván meg semmilyen behatást az elemzett forráskódra, így a felhasználói követelmények közül a **minimális behatást** is kielégíti. A tervezés során bevezetett gyakran használt, alapértelmezett szolgáltatások (3.3.), továbbá a megjelenítés támogatásának oldaláról a kényelmi proxy-modell leszármazottak (3.4.2.) a szükséges felhasználói **beavatkozást** csökkentik.

A tervezés összefoglalásaként egy áttekintő diagramot adunk a teljes keretrendszer elvi felépítéséről a 3.9 ábrán, mely kapcsolataikkal együtt mutatja be a komponensek egyes elemeit. Az ábrán jól látható a relációkat (*Relation*) és a csomópontokat (*Node*) egységes interfész mögé rejtő alapmodell (*NodeModel*), továbbá a szolgáltatásoknak a megjelenítéshez való csatlakozási pontja a proxy-modelleken (*NodeProxyModel* és leszármazottai) keresztül.



3.9. ábra. A teljes keretrendszer közös modellje

4. fejezet

Megvalósítás és alkalmazás

Egy szoftver tervezése első közelítésben a megvalósítás konkrét eszközeitől függetlenül történik. Miután – az előzetes tervek, illetve a fejlesztés környezete és peremfeltételei alapján – meghatároztuk a felhasználni kívánt eszközöket, a meglévő tervek finomításával, esetleg kisebb módosításával készülhet el az implementáció részletes terve.

Ez a fejezet bemutatja a dolgozatban tárgyalt kertrendszer megvalósításához választott programozási nyelvet és környezetet, indokolva ezeket a döntéseket. Ezután áttekintjük, hogy az implementáció során milyen konvenciókat követtünk. Bemutatjuk, hogyan használtuk fel a választott környezet nyújtotta lehetőségeket, illetve azt, hogy mennyiben módosítottunk a korábbi terveken a programozási nyelvhez és környezethez való igazodás érdekében. Végül áttekintjük rendszerünk kiterjesztési pontjait és felhasználási lehetőségeit, illetve bemutatjuk munkánk valós alkalmazását egy, az Eötvös Loránd Tudományegyetem Informatikai Karán folyó kutatási projektben.

4.1. A technológia kiválasztása

Ahhoz, hogy keretrendszerünket megvalósíthassuk, először ki kellett választanunk az e célra legalkalmasabb programozási nyelvet, illetve környezetet. Ennek során az alábbi igényeket és feltételeket vettük figyelembe.

- A **platformfüggetlenség** a keretrendszerrel szemben támasztott elvárások között szerepel (ld. a 3.1. szakaszt), így csak olyan nyelv jöhetett szóba, amelyre operációs rendszerek széles körén érhető el naprakész fordító.

- A valós **felhasználhatóság** érdekében olyan nyelvet és környezetet szerettük volna használni, mely közismert, alkalmazása elterjedt és könnyen csatolható más – akár különböző nyelveken írt – szoftverekhez.

- Ahhoz, hogy munkánk végeredményét **nyílt forráskódú** szoftverként bárki hasznosíthassa, a felhasznált eszközök licencelésének ezt lehetővé kellett tennie.

- Mivel komponensünk célja az, hogy **grafikus alkalmazásokban** kerüljön felhasználásra, igyekeztünk olyan környezetet választani számára, amely magas szintű grafikai lehetőségekkel támogatja ilyen programok készítését, ezzel megkönnyítve bonyolultabb nézetek létrehozását is.

- Ipari méretű szoftverek elemzése jelentős számítási kapacitást igényelhet. Ezért lényeges szempont volt, hogy a program futási környezete a lehető **legkevesebb erőforrást** használja fel.

- A **hatékonyság és biztonság** érdekében szem előtt tartottuk, hogy a választott programozási környezetben minél több, a munkánk során felhasználható adatszerkezet és architekturális elem már rendelkezésre álljon. Ez egyrészt megkönnyíti az implementációt, másrészt hibalehetőségek széles körét zárja ki alkalmazásunkból a nagyobb programozói közösség által ellenőrzött és optimalizált kódok felhasználásával.

A fenti szempontok alapján még mindig több alternatíva közül választhattunk. A 2. fejezetben megvizsgált szoftverek alapján természetesen adódott a *Java* nyelv alkalmazásának lehetősége. Platformfüggetlensége és elterjedtsége mellett előnyére szólt, hogy számos kiegészítő grafikus könyvtár érhető el hozzá. Ellenérv volt a virtuális gép feletti futásból adódó többlet erőforrásigény. Szintén megfontolásra került a *C#* nyelv és a *Microsoft .NET* környezet használata magas szintű fejlettsége és kiterjedtsége miatt. Azonban hordozhatósága sajnos korlátozott, elsősorban a *Microsoft Windows* platformokon alkalmazott[1].

Választásunk végül a *C++* nyelvre esett, mivel elterjedt, közismert, támogatja a tervezésnél felhasznált objektumorientált fogalmakat, futtató környezete nem terheli jelentősen az erőforrásokat, valamit rendelkezik egy sokrétűen alkalmazható

[1] A *Microsoft .NET* keretrendszer ugyan a *Mono* projektek eredményeképp más platformokra is elérhető, azonban ezeken a számunkra kiemelten fontos grafikus könyvtárak implementációja nem teljes.

szabványos könyvtárral. A megvalósítás megkönnyítésére a szabványos könyvtáron kívül felhasználtuk még a Qt alkalmazás-keretrendszert[2]. A Qt egy *C++* alapú, nyílt forrású, szabadon felhasználható[3], közösségi fejlesztésű, grafikus alkalmazások készítését támogató könyvtár. Fontos előnye, hogy külön alrendszere támogatja a tervek között is megjelenő *helyettes* tervezési mintát, valamint a modell-nézet architektúrát, beleértve a nézetek szinkronizációját is. További előnyei közé sorolható, hogy a grafikus elemek kirajzolása számára igény szerint *OpenGL* – és ezen keresztül hardveres – támogatást képes biztosítani.

4.2. Fejlesztési konvenciók

Egy program implementációjának megkezdése előtt célszerű lefektetni néhány alapvető irányelvet, illetve konvenciót, amely mentén a megvalósítás történik. Ez a programkód egységességét támogatja mind logikailag, mind formailag, ami megkönnyíti a későbbi karbantartást, a fejlesztés új résztvevői számára a gyorsabb megértést. Helyes konvenciókkal bizonyos típushibák elkövetésének lehetősége is csökkenthető.

A feladat megvalósítása során alkalmazott konvencióink meghatározásánál a *Google* ajánlásából[4] indultunk ki. Ebben a szakaszban bemutatjuk ennek az általunk is alkalmazott, fontosabb elemeit, illetve azokat a pontokat, ahol eltértünk tőle vagy kibővítettük azt.

Konzisztencia A konzisztencia szigorú értelemben nem tekinthető kódolási konvenciónak. Ennek ellenére mind a *Google C++ Style Guide*, mind ez a dolgozat igyekszik hangsúlyozni, hogy a forráskódok jó olvashatóságának egyik alapfeltétele az *egységes stílus*.

Névterek A teljes megvalósítást egy saját névtérbe (`gsmvf`) helyeztük. Nem használunk beágyazott névtereket a keretrendszer egyes részeire.

Angol nyelv használata A típusok, metódusok, változók stb. elnevezése angol nyelven történt. A nyelvválasztást az indokolja, hogy mivel a programot szé-

[2] `http://qt-project.org`

[3] A Qt keretrendszer nem üzleti célú alkalmazások készítésére GPL licenc szerint szabadon felhasználható. Bővebb licencelési információk: `https://qt-project.org/products/licensing`

[4] *Google C++ Style Guide*: `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`

les körű felhasználásra szánjuk, a magyar nyelvet nem ismerő programozók számára is szeretnénk megkönnyíteni a rendszerrel való munkát.

Elnevezések A program entitásainak elnevezése az ún. *CamelCase* forma szerint történt. Ezen belül:

- A típusok neve nagybetűvel kezdődik (pl. `SomeType`).
- A változók neve kisbetűvel kezdődik (pl. `someLocalVar`). A megkülönböztetés érdekében egy osztály nem publikus attribútumainak neve alulvonással (_) végződik (pl. `someAttribute_`).
- A forrásfájlok nevei csak kisbetűket, valamint szóhatároknál alulvonást (_) vagy kötőjelet (-) tartalmazhatnak.

Rövid függvények Törekedtünk arra, hogy a függvények definíciója minél rövidebb legyen, működésük pedig minél lényegretörőbb.

Referencia szerinti paraméterátadás Ha egy függvény referencia szerint kap paramétert, azt mindig `const` minősítővel láttuk el a véletlen módosítások elkerülése érdekében.

C++11 használata Munkánk során igyekeztük kiaknázni a C++ 2011-ben elfogadott szabványa[5] által bevezetett nyelvi és könyvtári újításokat. Mivel a fordítók egyelőre csak részlegesen implementálják ezt a szabványt, csak azokat a lehetőségeket használtuk, melyek a *GCC*[6] 4.6.3 verziójában támogatottak.

Intelligens mutatók Ahol csak lehetett, a C++ szabványos könyvtárának intelligens mutatóit (*smart pointer*) alkalmaztuk, ezzel csökkentve a memóriaszivárgás és az érvénytelen mutatók (*dangling pointer*) okozta hibák kockázatát[7].

Mutató típusok elnevezése A *Google* ajánlásának bővítéseként ha `SomeTpye` típusra intelligens mutató mutathat, akkor `std::shared_ptr<SomeType>` számára (lehetőség szerint a `SomeType` definícióját tartalmazó fejállományban) a `SomeTypePtr` formájú álnevet vezettük be.

[5] ISO/IEC 14882:2011

[6] *GNU Compiler Collection*: `http://gcc.gnu.org`

[7] A Qt keretrendszer sajátos memóriakezelése miatt ez nem mindenhol volt lehetséges, ám itt a problémák elkerülésében a Qt memóriakezelési mechanizmusa segítségünkre volt. A Qt memóriakezeléséről bővebben: `http://qt-project.org/doc/qt-4.8/objecttrees.html`

Adatszerkezetek A Qt tároló adatszerkezeteivel szemben az egységesség érdekében előnyben részesítettük a C++ szabványos konténereinek használatát.

Megjegyzések A dokumentációs (pl. függvények, osztályok működését leíró) megjegyzések a *Doxygen* automatikus dokumentációs eszköz[8] szabályai szerint íródtak. A függvények belsejébe szúrt, magyarázó kommentekkel a *Doxygen* nem foglalkozik. Ezekre stílusbeli megkötést nem alkalmaztunk.

4.3. A modell réteg megvalósítása

A modell réteg megvalósítása absztrakt *C++* osztályok segítségével történt. A *C++* nyelv sajátosságait figyelembe véve ezek tisztán virtuális metódusokat tartalmaznak. A 3.4. és 3.6. ábrákon bemutatott terveket lényegében a specifikációtól való eltérés nélkül tudtuk leképezni a választott programozási nyelvre.

Megjegyzés. *Egyes metódusokhoz, például a relációban a gyerekek számának lekérdezéséhez biztosítunk alapértelmezett implementációt, amelyet a felhasználó lehetőségei és igényei szerint hatékonyabb megvalósítással felüldefiniálhat.*

Egy konkrét szoftvermodell csomópont-típusa a fentiek alapján a *Node* absztrakt osztály leszármazottja, mely felüldefiniálja annak tisztán virtuális műveleteit. A csomópontok metódusok paramétereiben, illetve visszatérési értékeiben intelligens mutatókként (`std::shared_ptr<Node>`) jelennek meg, ilyen módon kihasználható a dinamikus kötés lehetősége, azaz keretrendszerünk a *Node* absztrakt osztály tetszőleges megvalósításával működhet.

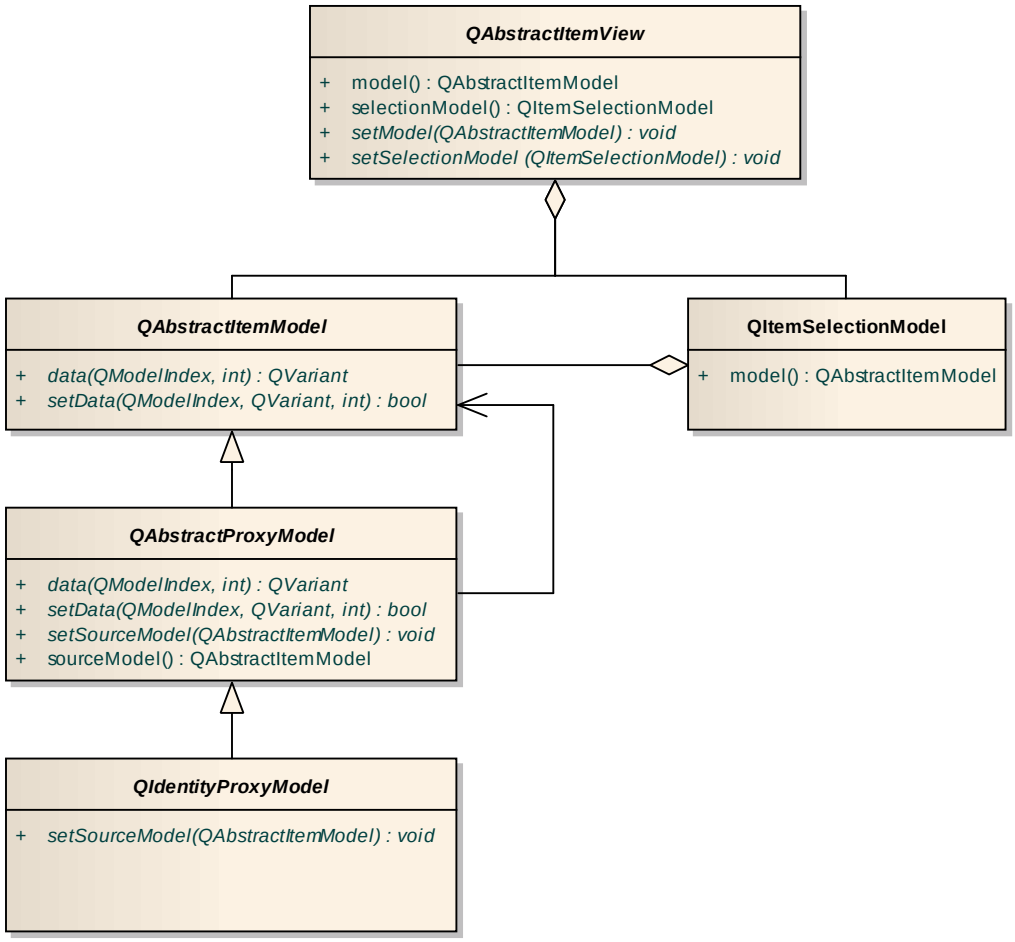
A *reláció* és a szolgáltatások realizációja során is hasonló logikát követtünk. Azon metódusok esetén, melyek csomópontok gyűjteményével térnek vissza (pl. a reláció gyerekeket előállító metódusa), a megvalósításban a C++ szabványos könyvtárának `std::vector`[9] konténer osztályát alkalmaztuk.

4.4. A megjelenítés támogatásának megvalósítása

A megjelenítési réteg megvalósításánál felhasználtuk azt, hogy Qt rendelkezik egy absztrakt modell-nézet alrendszerrel (ld. a 4.1. ábrát), melynek szerkezete lényegében igazodik a 3.4. pontban részletezett tervekhez.

[8] `http://doxygen.org`

[9] Az `std::vector` egy szekvenciális konténer, mely a tömb típushoz hasonló felületet nyújt.



4.1. ábra. A Qt modell-nézet alrendszere

A Qt modell-nézet alrendszere

A Qt modell-nézet alrendszerének alapjai a *QAbstractItemModel* (valamint az ebből származó *QAbstractProxyModel*) és *QAbstractItemView* absztrakt osztályok. Előbbi két osztály a helyettes tervezési mintát[10] követi. A harmadik (nézet) osztály leszármazottainak hasznos tulajdonsága, hogy tetszőleges modell-leszármazottat képesek megjeleníteni. Az alrendszer egyik érdekes lehetősége, hogy beépítve nyújtja a 3.4.3. pontban tárgyalt *szinkronizációs interfész* funkcionalitását, melyhez a *szinkronizációs modellt QItemSelectionModel* néven biztosítja.

[10]Ld. a 3.4.2. pontot.

Feladatok

A fentiek alapján a 3.8. ábrán látható *BaseModel* osztály megfelelője az implementációban a *QAbstractItemModel* lett. Feladataink az alábbiak voltak annak elérése érdekében, hogy a Qt erejét kihasználhassuk saját keretrendszerünkben:

- absztrakt szoftvermodellünket le kellett képezni a *QAbstractItemModel* osztály felületének megfelelően, illetve
- speciális, kényelmi proxy-modelljeinket a *QAbstractProxyModel*-ből kiindulva kellett létrehoznunk.

4.4.1. Leképezés a megvalósítás környezetének fogalmaira

A *QAbstractItemModel* osztály ún. *QModelIndex* leíró struktúrák segítségével azonosítja az általa reprezentált modell elemeit. Egy *QModelIndex* struktúra tárol egy sor-, illetve oszlopsorszámot (ezek rendre azt jelentik, hogy az entitás szülőjének hányadik gyermeke nullától indexelve, illetve hogy táblázatos megjelenítés esetén hányadik oszlopban kell szerepelnie), továbbá egy mutatót az őt kibocsátó modellre, valamint egy előjel nélküli egész típusú azonosító értéket, melyet a modell megvalósítója kedve szerint állíthat be.

Az egyes modellelemekről információ a `QVariant data (const QModelIndex &index, int role)` függvény segítségével kérhető.

- A függvény első paramétere egy *QModelIndex*, mely a modell azon entitását azonosítja, amire a lekérdezés vonatkozik.
- A második paraméter egy egész típusú érték, mely a kért adat jellegét azonosítja. A Qt terminológiájában ez a *szerep* (*role*) nevet viseli.
- A visszatérési érték *QVariant* típusú, mely egy bővíthető, ellenőrzött unió vagy *variáns rekord* típusnak tekinthető.

Megjegyzés. *A szerepek arra szolgálnak, hogy a* QModelIndex *által azonosított entitásról ne független metódusokon, hanem egy egységes mechanizmuson keresztül lehessen beszerezni különböző információkat. A szerepeket gyakran felsorolási típusok (*enum*) értékeivel azonosítják egyszerű egész literálok helyett a kód olvashatósága érdekében. A Qt a* $[0, 31]$ *intervallumba eső értékeket fenntartja előre definiált szerepek számára, melyeket a* Qt::ItemDataRole *felsorolási típusban definiál.*

Példák szerepekre (A Qt névtérben található nevek a Qt::ItemDataRole *típusban megadott, előre definiált értékek):*

Qt::DisplayRole *Az adott entitás helyén megjelenő érték.*

Qt::TooltipRole *Az egeret az adott elem fölé mozgatva szövegbuborékban megjelenő segítség vagy leírás.*

Qt::ForegroundRole *A rajzolásnál használt előtérszín (pl. szövegszín).*

TypeRole *Példa felhasználó által bevezetett szerepre. Egy ehhez hasonló szerep azonosíthatja például az adott entitás típusát leíró sztringet. Fontos, hogy a* TypeRole *egész értékének 31-nél nagyobbnak kell lennie.*

Keretrendszerünk tervének leképezését a fenti fogalmakra a 4.1. táblázat foglalja össze.

Tervezési fogalom	Qt-fogalom
BaseModel	*QAbstractItemModel*
NodeProxyModel	*QAbstractProxyModel*-leszármazott
Node	*QModelIndex*
Szolgáltatás által nyújtott adat	szerep (*role*)

4.1. táblázat. A keretrendszer absztrakt fogalmainak leképezése a Qt modell-nézet alrendszerének fogalmaira

A szerepek kezelése

A szoftvermodellhez a felhasználó által bevezetett szolgáltatásokat is a Qt szerepei mentén szeretnénk kezelni. Ezeket egész típusú értékekkel azonosítva a dinamikus bővíthetőség megvalósítása nehézkes lenne, nehéz lenne elkerülni a különböző saját szerepek bevezetésekor az ütköző értékeket. Ezért kibővítettük a Qt lehetőségeit úgy, hogy karakterláncok (*stringek*) segítségével is azonosíthatóvá váljanak az egyes szerepek. A *string* szerepnevek ugyanis különböző adatokra valószínűleg különbözőek lesznek akkor is, ha azokat programozók egymástól függetlenül vezetik be[11].

A bővítéshez szolgáltat interfészt a 4.2. ábrán látható *StringlyRoledModel* absztrakt osztály, melyből mind a *NodeModel*, mind a *NodeProxyModel* származik. Ez a *QAbstractItemModel* metódusaihoz hasonló tisztán virtuális metódusokat (pl. `data`) definiál, amelyekben azonban a szerepek *stringgel* azonosítottak. A

[11]Ennek a megközelítésnek költsége van, hiszen szövegek összehasonlítása lassabb, mint egész értékeké.

StringlyRoledModel ezen kívül *string*-neveket tárol az alapértelmezett szerepekhez, valamint megvalósítást biztosít a függvények azon változatai számára, melyekben a szerepek egész számok (ezeket a szöveges megfelelőre képezi le).

Node-QModelIndex leképezés

A *QModelIndex* struktúra tartalmaz egy előjel nélküli egész azonosító számot (`internal_id`), ezzel azonosítja a hozzá tartozó entitást. A csomópontokat *QModelIndex*-re és ellenkező irányba leképező `QModelIndex nodeToIndex(NodePtr)` és `NodePtr indexToNode(QModelIndex)` metódusokat – mint tisztán virtuális függvényeket – a *StringlyRoledModel* segédosztályban helyeztük el, mivel ezekre a konverziókra a *NodeModel*-ben és a proxy-modellekben egyaránt szükség van.

A konverzió hátterében `std::unordered_map` adatszerkezetben tároljuk a csomópontok és a *QModelIndex*-beli azonosítók egymáshoz rendelését. A hozzárendelés optimalizációs okból lusta abban az értelemben, hogy egy csomóponthoz csak annak első lekérésekor rendelünk azonosítót, így csak azon csomópontokat tartjuk ilyen módon nyilván, melyeket legalább egyszer már lekérdeztek.

Proxy-modellek

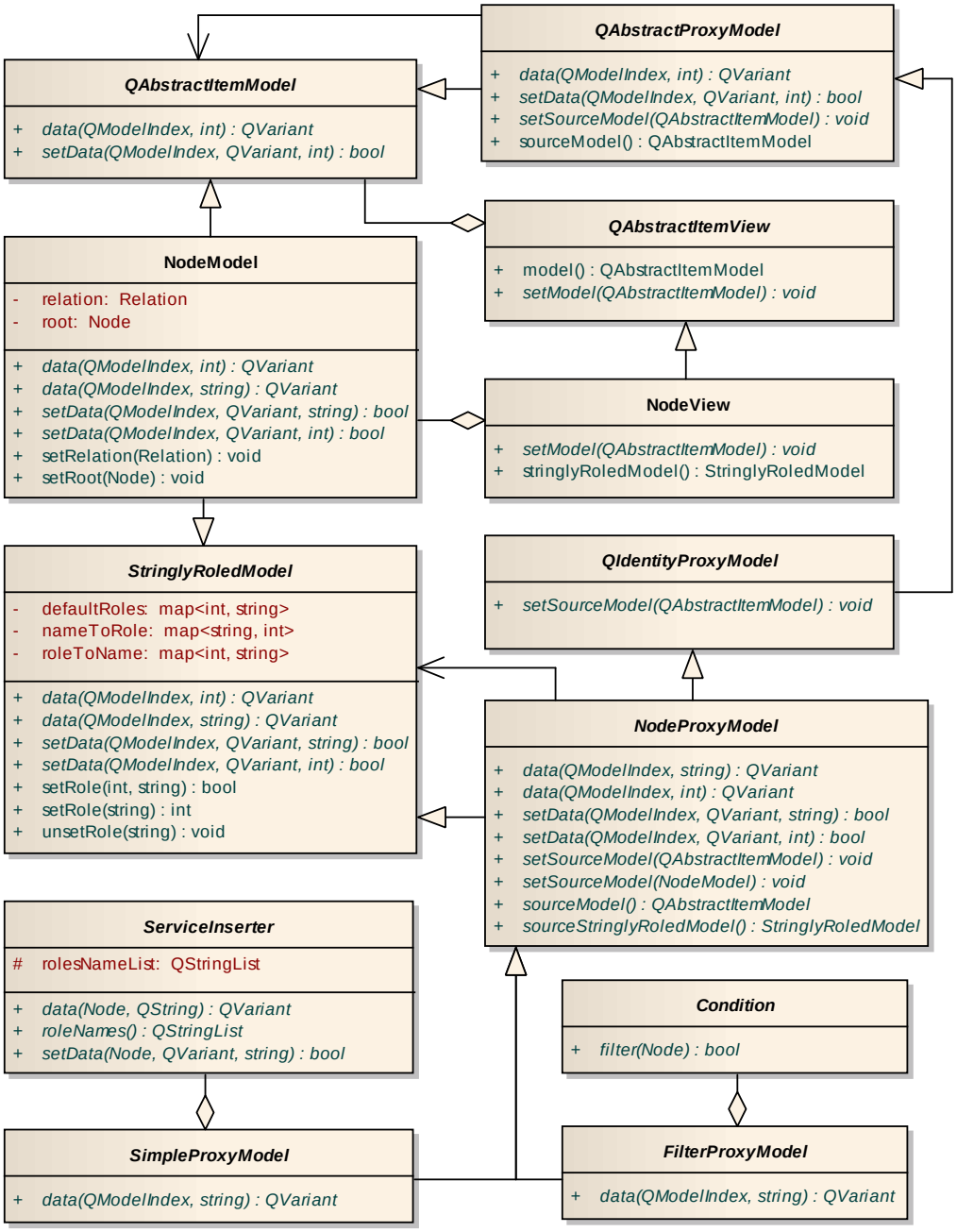
A proxy-modellek megvalósításához rendelkezésünkre állt a *QIndentityProxyModel* osztály, amelyet lényegében nem kellett módosítanunk, csupán kibővítenünk a *StringlyRoledModel* szolgáltatásaival. Így a *NodeProxyModel*, keretrendszerünk proxy-modelljeinek közös ősosztálya a *QIdentityProxyModel*-ből és a *StringlyRoledModel*-ből származtatva állt elő.

Megjegyzés. *Az implementáció egy kisebb megkötése volt, hogy a Qt az eseménykezelési mechanizmust felhasználó osztályai számára (amilyenek a* QAbstractItemModel *és általunk használt leszármazottai is) nem engedélyezi a sablon (generikus,* template*) konstrukció használatát, ezért a 3.8. ábrán látható* SingleProxyModel *és* FilterProxyModel *osztályok származtatás helyett aggregálva tartalmazzák a megfelelő* ServiceInserter*, illetve* Condition *objektumokat (ld. a 4.2. ábrát). Ezeket részletesen a 4.6. szakaszban mutatjuk be.*

A leképezés végeredményét, azaz a megjelenítést támogató réteg tényleges megvalósításának UML-osztálymodelljét a 4.2. ábra tartalmazza. Ezt összevetve a 3.8. ábrán bemutatott tervekkel, látható, hogy az csupán kis mértékben változott:

- legfelső szintű ősosztályként jelentek meg a Qt modell és proxy osztályai;

- a közös interfész biztosítására és a flexibilitás növelésére bevezettük a tervek között nem szereplő *StringlyRoledModel* osztályt;
- feltüntettük a nézetek Qt ősosztályát és ebből származtatott saját ősosztályunkat, melyből tovább származtatva keretrendszerünk felhasználója saját nézeteket hozhat létre.



4.2. ábra. A megjelenítést támogató réteg implementációja

NodeView

A *NodeView* osztályt a Qt nézet osztályából (*QAbstractItemView*) származtattuk. Célja, hogy a Qt modell osztályához képest a *NodeModel*-ben bevezetett bővítéseket kényelmesen lehessen használni. Az osztályról részletesebben a kiterjesztési lehetőségek között, a 4.5.2. pontban lesz szó.

4.5. Kiterjesztési pontok

Az eddigiek során bemutattuk, hogyan valósítottuk meg a dolgozat témáját képező, vizualizációt támogató keretrendszernek a 3. fejezetben kidolgozott tervét a választott környezetben. Ebben a szakaszban a végeredményt a szoftvervizualizációs alkalmazások fejlesztésére való felhasználás szemszögéből vizsgáljuk meg.

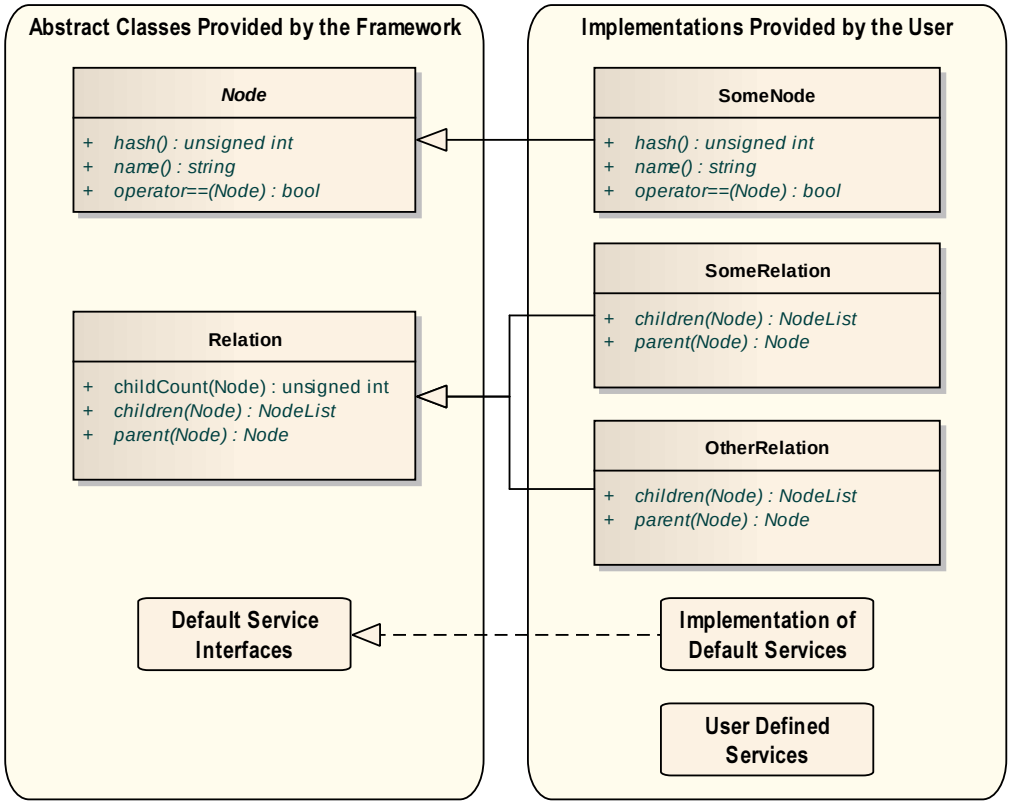
Egy szoftver kiterjesztési pontja a szoftver olyan aspektusa, melynek segítségével viselkedése anélkül változtatható meg, hogy módosítanánk az eredeti kódot. Kiterjesztési pontok mentén lehetséges például különböző alkalmazásokba beépülő más programok írása, vagy önállóan nem futtatható programegységek (komponensek, könyvtárak, keretrendszerek) felhasználása más szoftverekben.

A dolgozatban bemutatott keretrendszer célja szoftvervizualizációs alkalmazások készítésének támogatása, így két irányban nyújt kiterjesztési lehetőséget: a szoftverelemző (jellemzően forráskódból absztrakt szintaxisfát előállító), valamint a megjelenítő komponensek felé (ld. a 3.2. ábrát).

4.5.1. Kiterjesztés a szoftverelemzés irányában

Keretrendszerünk kiterjesztési pontja a szoftverelemzés irányába a 4.3. szakaszban bemutatott absztrakt osztályok gyűjteménye, melyeket együtt *szoftvermodellnek* neveztünk. A felhasználó a modell absztrakt osztályaiból való származtatással és azok virtuális metódusainak felüldefiniálásával kapcsolódhat a keretrendszerhez (ld. a 4.3. ábrán).

Futtatható alkalmazás készítéséhez minimálisan a *Node* absztrakt osztály és egy *Reláció* (azaz összesen 5 egyszerű metódus) megvalósítása szükséges. Ezek megléte mellett a 4.4. példában látható néhány sor inicializáló kód elhelyezése egy Qt ablak konstruktorában már működő alkalmazást eredményez, mely a Qt beépített fanézetében mutatja számunkra a definiált modellt. A kódrészlet első sora létrehozza a modellt, a következő két sorban pedig a (szintaxis)fa gyökéreleme és relációja



4.3. ábra. A szoftvermodell felhasználása

kerül beállításra. Végül a beépített nézet példányosítása és a modell nézethez illesztése történik meg.

```
1  NodeModel *model = new NodeModel(this);
2  model->setRoot(NodePtr(new SomeRootNode));
3  model->setRelation(RelationPtr(new SomeRelation));
4  QAbstractItemView *tv = new QTreeView(this);
5  tv->setModel(model);
6  tv->show();
```

4.4. ábra. Minimális vizualizációs alkalmazás forráskódja

További relációk megvalósításával, illetve szolgáltatások definiálásával, és azok proxy-modellekbe ágyazásával bővíthetjük a fenti egyszerű program lehetőségeit.

A modellhez tartozó reláció cserélhető, a fölé épülő proxy-modellek lánca pedig szabadon variálható futás közben – a változásokra a nézetek a Qt eseménykezelő mechanizmusa mentén automatikusan reagálnak. Így a megjelenített adatok köre a felhasználói akciók függvényében változtatható. Tekintsük azt a példát, amikor

a felületen lehetővé szeretnénk tenni, hogy a végfelhasználó gombok segítségével választhasson egy rendszer osztályhierarchia- és fájlrendszer-nézete között. Ekkor a megfelelő kattintás eseményét a 4.5. kódrészben látható módon kezelhetjük le. A modellhez kapcsolt nézetek automatikusan frissülni fognak.

```
1 /* update() ensures redrawing of the whole window */
2 void on_showClassHierarhyButtonClicked()
3 {
4   model->setRelation(RelationPtr(new ClassHierarchyRelation));
5   update();
6 }
```

4.5. ábra. Reláció cseréje a modellben

A fentiek alapján az is látható, hogy a szoftvermodellhez különböző jellegű adatforrások kapcsolhatók: elképzelhető akár valamilyen perzisztens adattárolási forma – például egy XML leíró nyelvű állomány vagy egy relációs adatbázis –, akár olyan komponens, mely futási időben, közvetlenül a memóriában állítja elő a modell adatait.

4.5.2. Kiterjesztési lehetőségek a vizualizáció felé

A vizualizáció felé való kiterjesztési lehetőségeket elsősorban új nézetek definiálása jelenti a már létrehozott modellek megjelenítésére. Mivel keretrendszerünknek a szoftvermodell megjelenítését támogató, magas szintű rétege a Qt modell-nézet alrendszerére épül, új nézetek létrehozására a *QAbstractItemView* osztály leszármazottja, a 4.2. ábrán látható *NodeView* biztosít lehetőséget. A *QAbstractItemView* felhasználásához hatékony segítséget nyújt a Qt dokumentációja[12], ezért erre nem térünk ki részleteiben. A *NodeView* osztály bázistípusát a következőkkel egészíti ki.

- Felüldefiniálja a `void setModel(QAbstractItemModel *)` metódust olyan módon, hogy működése az eredeti megvalósítással egyezzen meg, ha a paraméter leszármazottja a *StringlyRoledModel*-nek, ám ha ez a feltétel nem teljesül, akkor a hatás ekvivalens legyen azzal, mintha a paraméter *null-pointer* lenne.

[12] A Qt modell-nézet architektúrájának felhasználásáról általában: `http://qt-project.org/doc/qt-4.8/model-view-programming.html` A *QAbstractItemView* osztály dokumentációja: `http://qt-project.org/doc/qt-4.8/QAbstractItemView.html`

- Bevezeti a `StringlyRoledModel *stringlyRoledModel() const` metódust, mely az éppen megjelenített modellt adja vissza *StringlyRoledModel* típusú mutatóként.

A fenti kiegészítések segítségével a *NodeView* leszármazottai csak *NodeModel*-lel vagy *NodeProxyModel*-lel képesek együttműködni, viszont kihasználhatják ezek mindkét közös ősosztálya, a *QAbstractItemModel* és a *StringlyRoledModel* által nyújtott lehetőségeket (melyekről bővebben a 4.4. szakaszban írtunk). Amennyiben a plusz lehetőségeket a nézet nem igényli, úgy az közvetlenül a *QAbstractItemView*-ból is származtatható.

A másik kiterjesztési lehetőséget a 3.4.2. pontban bemutatott proxy-modellek adják. A proxy-modellek közös őse a *NodeProxyModel* osztály. Céljuk az, hogy egy *NodeModel*-példányhoz adjanak hozzá különböző *szolgáltatások* (ld. a 3.3. pontban) által nyújtott adatokat, vagy azokat transzformálják a megjelenítés számára.

```
class SomeType : public Type
{
  public: std::string type(NodePtr n)
    { /* calculate some type name */; }
};

class TypeServiceInserter : public ServiceInserter, private
    SomeType
{
  public:
    TypeServiceInserter()
      { roleNames_.insert("type");
        rolesNames_.insert("toolTip"); }

    QVariant data(NodePtr n, const std::string &role)
    {
      if ("type" == role || "toolTip" == role)
        return QString::fromStdString(type(n));

      return QVariant();
    }
};
```

4.6. ábra. Típus-szolgáltatás és inserter megvalósítása

Adatok hozzáadása Példa lehet adat hozzáadására, ha szeretnénk a modellünk csomópontjaihoz szöveges típusmegjelölést (*variable reference*, *arithmetic expression* stb.) rendelni, mely ún. *tooltip*-ként[13] jelenítünk meg. Ehhez az alábbi három feladatot kell megoldani (ld. a 4.6 és a 4.7. kódrészletet).

```
1 /* Using above classes with SimpleProxyModel */
2 NodeProxyModel *proxy = new SimpleProxyModel(
3                            ServiceInserterPtr(
4                              new TypeServiceInserter));
5 proxy->setSourceModel(model);
6 view->setModel(proxy);
```

4.7. ábra. A *SimpleProxyModel* egy felhasználása

1. Definiálni kell a típus-szolgáltatást (például az azonos nevű előre definiált, absztrakt szolgáltatásból származtatva).
2. El kell készíteni a *ServiceInserter* osztály egy leszármazottját a `QVariant data(NodePtr n, const std::string &role)` függvény olyan felüldefiniálásával, mely a `"tooltip"` – illetve esetleg a `"type"` – szerepre az előbbi szolgáltatás által megadott típusnevet adja vissza.
3. Modellünk és az azt megjelenítő nézet közé egy *SimpleProxyModel*-példányt kell illesztenünk, melyet az előző pontban megadott típus egy példányával – mint konstruktorparaméterrel – hoztunk létre.

Transzformációk A modell transzformálásának egyik módja lehet például szűrés, melyet a *FilterProxyModel* segítségével, az előző bekezdésben leírtakhoz hasonlóan könnyen megvalósíthatunk. Az adatok transzformálása érdekesebb feladat.

Példaként tekintsük azt az egyszerű esetet, amikor egy osztály belső szerkezetét jelenítünk meg az egyes osztályok metódusaival együtt, és szeretnénk az UML-hez hasonló módon, a név előtt jelölni azok láthatóságát. Az egyes csomópontokra a 4.2. táblázatban felsorolt láthatósági szinteket definiáljuk. Feltesszük, hogy ezek az `"visibility"` szerepen keresztül – mint egy felsorolási típus értékei – már elérhetőek a modellben.

Például egy `doParallel()` nevű védett metódust `#␣doParallel()` alakban szeretnénk megjeleníteni. Ehhez nem érdemes új szolgáltatást bevezetnünk, hiszen

[13] A *tooltip* olyan szövegbuborék, amely akkor jelenik meg, mikor a felhasználó egy grafikus elem fölé mozgatja az egérkurzort. Jellemzően a kurzor alatti elemről közöl kiegészítő információkat.

Érték (konstans)	Jelentése	UML
`Public`	publikus	+
`Protected`	védett	#
`Private`	privát	-
`None`	nincs	

4.2. táblázat. A láthatósági szintek egy lehetséges megjelenése a modellben

a meglévő szolgáltatások már minden szükséges adatot biztosítanak. Elég, ha a *NodeProxyModel* egy megfelelő leszármazottját helyezzük a modellünk és a megjelenítés közé. Ebben a `QVariant data(const QModelIndex &index, const std::string &role)` tisztán virtuális metódust kell megvalósítanunk a 4.8. példában látható módon. Az elkészült proxy-modell felhasználása a 4.9. példához hasonlóan történhet.

```
class VisibilityProxyModel : public NodeProxyModel
{
public:
  QVariant data(const QModelIndex &index, const std::string &role)
  {
    if(role == "name")
    {
      QString result
        = stringlyRoledModel()->data(index, "name").toString();

       switch
         (stringlyRoledModel()->data(index, "visibility").toInt())
       {
         case Public: result.prepend("+ "); break;
         case Protected: result.prepend("# "); break;
         case Private: result.prepend("- "); break;
         case None: result.prepend("  "); break;
       }
       return result;
    }
    else return stringlyRoledModel()->data(index, role);
  }
};
```

4.8. ábra. Adatok transzformációja proxy-modellel

```
/* Using above proxy with an appropriate object of NodeModel */
NodeProxyModel *proxy = new VisibilityProxyModel();
proxy->setSourceModel(model);
view->setModel(proxy);
```

4.9. ábra. A *VisibilityProxyModel* egy felhasználása

A fent bemutatott egyszerű példák csak szemléltetni szeretnék keretrendszerünk alapvető lehetőségeit. Nagyobb szoftvervizualizációs feladatok megvalósításának részletes bemutatására a dolgozat terjedelmére tekintettel nincs módunk, de a 4.6. szakaszban áttekintés szintjén bemutatunk egy, a munkánkat valós környezetben felhasználó alkalmazást.

4.6. A keretrendszer alkalmazása

A dolgozatunkban bemutatott generikus szoftvermodell-vizualizációs keretrendszer gyakorlati alkalmazására sor került az Eötvös Loránd Tudományegyetem Informatikai Karán jelenleg is folyó egyik projektben. A projekt az ipari szoftverek megértésének támogatását tűzi ki célul, hogy egy programozó jelentősen kevesebb erőforrás ráfordításával megismerhessen, megtanulhasson egy általa nem ismert forráskódot, majd abban hatékonyan tudjon dolgozni. Ugyan fejlesztőeszközök is használhatóak a kód megértéséhez, de csak korlátozott módon, hiszen eredetileg más feladatokra tervezték őket. A projekt célja tehát egy olyan alkalmazás elkészítése, amelyet dedikáltan a forráskód megértésének támogatására terveztünk.

A projekttel szemben támasztott igények

Ipari szoftverek esetén gyakori, hogy több programozási nyelven készülnek el, akár modulonként különbözőekben. Ezért az alkalmazás modelljét nyelvfüggetlen, a programozási paradigmáktól is kötetlen módon szükséges definiálni. Ezen kívül az adott platformhoz tartozó egyéb interfész és metainformációkat leíró nyelveken definiálhatják a modulok közötti kapcsolatokat. Egy kódmegértést segítő eszköznek a forráskód mellett fontos ezen információk feldolgozása és megfelelő, egységes módon történő kezelése is.

Egy valós, nagy méretű program forráskódjának elemzése során sok gigabájtnyi adat áll elő, melynek tárolására a rendszermemória kevés. Emiatt az adatokat per-

zisztens módon, például adatbázisban kell tárolnunk és csak az éppen szükségeseket tarthatjuk a memóriában.

A programmegértést támogató szoftver feladata, hogy a forráskódot – vagy egyes moduljait – magasabb absztrakciós szinteken tudja prezentálni. A felület tervezésekor figyelembe kell venni a különböző felhasználói igényeket is, hiszen például egy programozó más aspektusaira kíváncsi egy projektnek, mint egy menedzser. Ezen felhasználói igények kielégítéséhez különböző nézetek definiálására van szükség, amelyek a megfelelő szinten, a megfelelő szempontok szerint vizualizálják a modellt.

A keretrendszer alkalmazhatóságának vizsgálata

A fentebb megfogalmazott igények jól illeszkednek a dolgozatunk 3.1. szakaszában említett követelményekre. Mivel a keretrendszerünk ezen minőségi elvárásoknak eleget tesz, ezért megvizsgáltuk a projektbeli alkalmazhatóságát.

A szoftvermodell alkalmasnak találtuk tetszőleges programozási vagy leírónyelv szintaxisfájának reprezentálására, ugyanis a 3.2. fejezetben leírtak szerint a hierarchikus szerkezeten túlmenő speciális elvárásokat nem támaszt.

Az elemezett program méretéből adódó skálázhatósági követelményt a szoftvermodell azzal teljesíti, hogy működéséhez csak az éppen használt adatok betöltését igényli a memóriába.

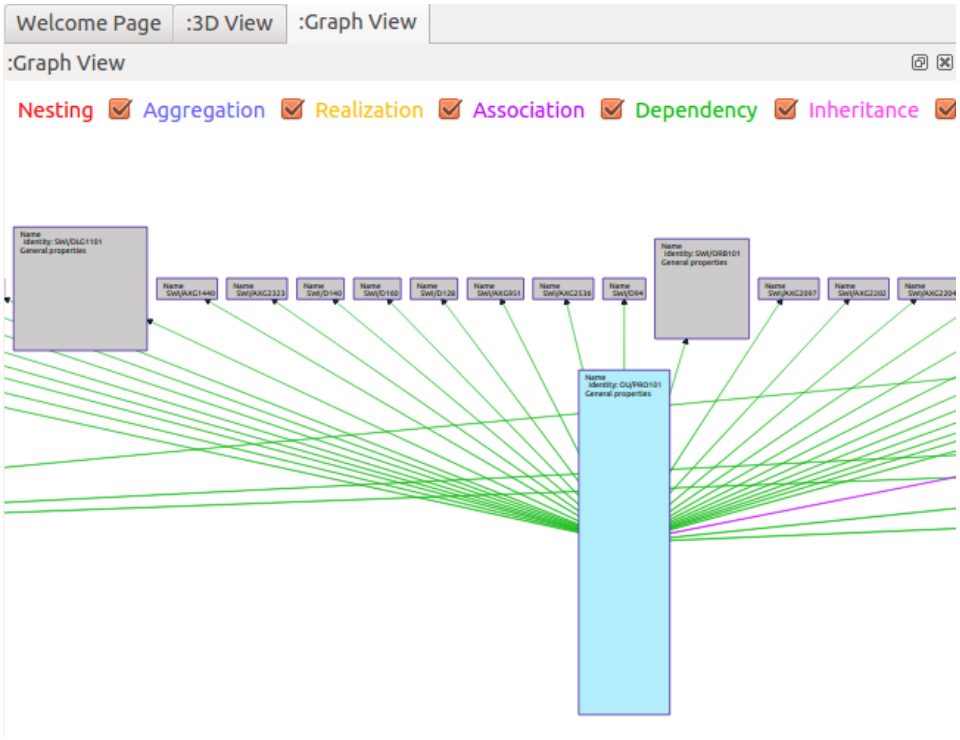
Keretrendszerünk megjelenítést támogató rétege a *proxy-modellek* segítségével lehetővé teszi ugyanazon modell különböző vetületekben és absztrakciós szinteken történő megjelenítését (ld. a 3.4.2. szakaszban). Az adott modell több nézetében történő megjelenítése mellett a 3.4.3. szakaszban definiált szinkronizációs interfész az e nézetek közötti automatikus kontextusmegőrzésre is lehetőséget nyújt.

A fentiek alapján keretrendszerünk megfelelőnek bizonyult a projektben való felhasználásra.

A projekttermék bemutatása

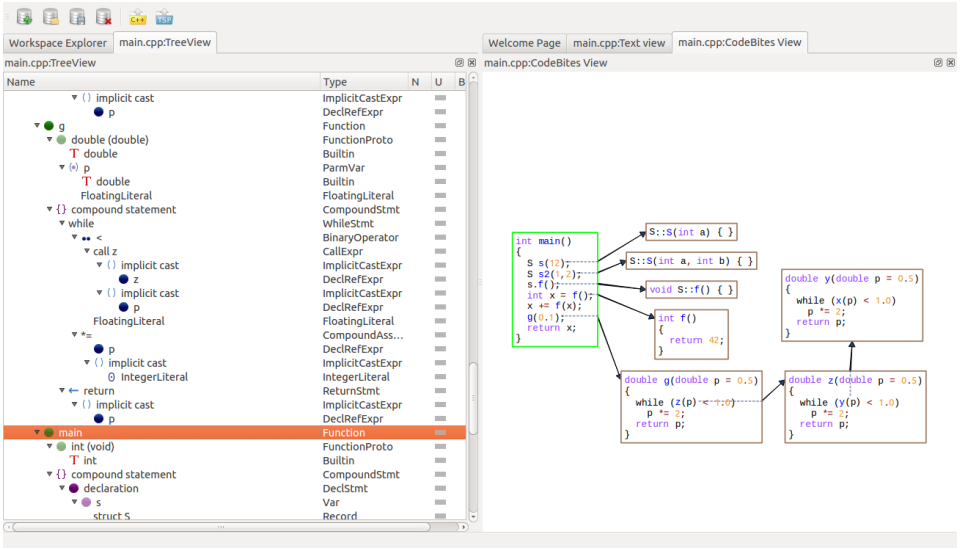
A projekt terméke még fejlesztés alatt áll, de már számos kódmegértést támogató funkcióval rendelkezik. Ezek közül a legjelentősebbek a következőek.

- Különböző programozási nyelveken írt forrásszövegek (pl. C++, Java) és a projekt metainformációinak elemzése, a feldolgozott adatok egységes, nyelvfüggetlen kezelése.
- Különféle adatbázisrendszerek támogatása az elemzés során nyert adatok perzisztens tárolására (SQLite, MySQL).
- Szoftvermodulok közötti összefüggések feltérképezése és megjelenítése (ld. a 4.10. ábrát).



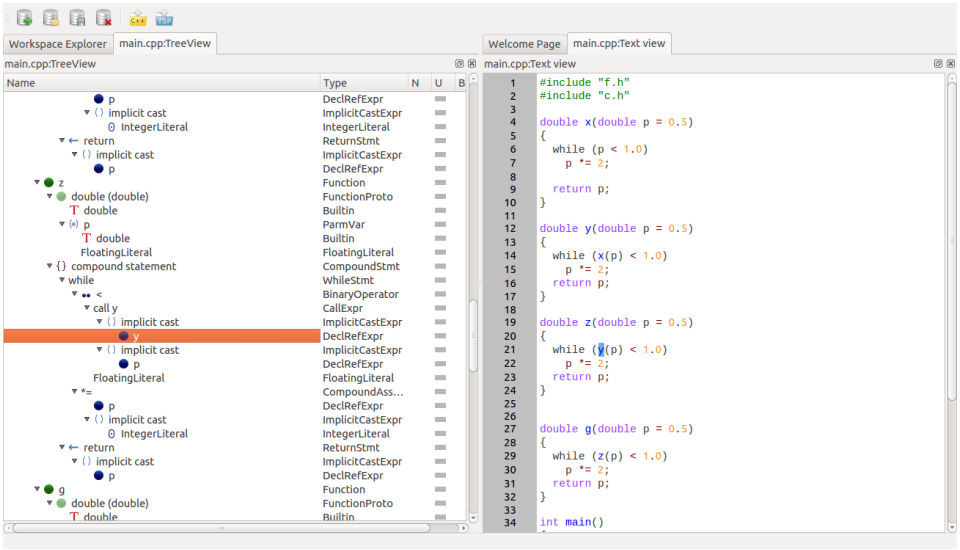
4.10. ábra. Forráskód alapján vizualizált UML diagram

- Fordítási információk – mint például egyes részegységek fordítási idejének vagy különböző fordítók egymáshoz viszonyított fordításkori futásidejének az – ábrázolása.
- A modell különféle nézettípusokban (pl. *prettyprint*, absztrakt szintaxisfa, UML, 3D) való vizualizációja (ld. a 4.11. ábrát).



4.11. ábra. Függvények és metódusok hívási hálója

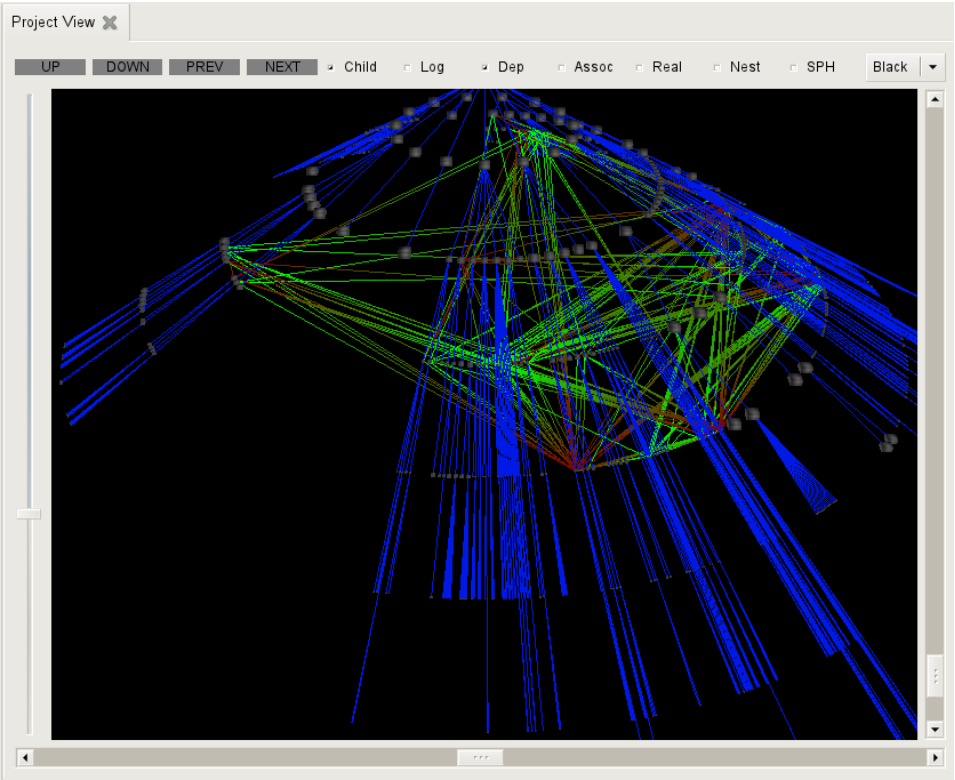
- Egyazon modell különböző nézeteinek azonos időben történő szinkronizált megjelenítése (ld. a 4.12. ábrát).



4.12. ábra. Szintaxisfa és forráskód szinkronizált megjelenítése

- Tervezési minták, a megadott kódolási konvenciótól való eltérések és tipikus hibák heurisztikus felismerése.

- A megértési folyamat nyomon követése (ld. a 4.13. ábrát).



4.13. ábra. Programkomponensek 3 dimenziós megjelenítése a megértettség mértékének ábrázolásával

- Modulok, függvények bonyolultságának mérése, metrikák készítése.
- A modell megjegyzésekkel történő annotálása.

5. fejezet

Összefoglalás és eredmények

A szoftverrendszerek komplexitásának növekedésével egyre nagyobb igény jelentkezik az ezek megértését segítő vizualizációs alkalmazások iránt. Ahogy a 2. fejezetben láttuk, a jelenleg rendelkezésre álló eszközökkel ez az igény csak komoly kompromisszumok árán elégíthető ki, ilyenek például: az elemzett programozási nyelv specifikussága, a folyamatos fejlesztés hiánya, a szoftvermodellezési és vizualizációs feladatok egységesen magas szintű támogatása, vagy a magas költségek.

Dolgozatunk célja egy olyan nyílt forrású keretrendszer megtervezése és megvalósítása volt, amely jelentősen megkönnyíti magas szintű szoftvervizualizációs alkalmazások készítését. A megoldás során egy két rétegből álló komponenst dolgoztunk ki, mely újrafelhasználható és kiterjeszthető módon képes egy tetszőleges programozási nyelven készült forráskód alacsony szintű absztrakt szintaktikus modelljét könnyen megjeleníthető, magas szintű formára alakítani.

A komponens alsó, szoftvermodell rétege a szintaxisfa absztrakt, nyelvfüggetlen, bővíthető interfészeként funkcionál. Ez a felület többek köz lehetőséget nyújt a forráskód eltérő hierarchikus szinteken történő leírására. A szoftvermodell felett elhelyezkedő, a vizualizációt támogató réteg nyújt támogatást a modell további szűrésére, transzformálására, illetve megjelenítés-specifikus információk, attribútumok hozzáadására. Ezen kívül biztosítja a különböző nézetek közötti kontextus megőrzését automatikus szinkronizációval.

Eredményeink gyakorlati alkalmazhatóságát igazolja, hogy a keretrendszer felhasználásra került az Eötvös Loránd Tudományegyetem Informatikai Karán fejlesztett, – a 4.6. szakaszban bemutatott – kódmegértés támogató szoftvervizualizációs programban.

6. fejezet

Továbbfejlesztési lehetőségek

A dolgozatunkban bemutatott szoftvervizualizációs keretrendszer továbbfejlesztésére számos lehetőség adódik. Ebben a szakaszban ezek közül vesszük sorra a leglényegesebbeket.

A rendszer tervei, elméleti alapjai több irányban is kiterjeszthetők a szoftvermodell rétegében újfajta elemzési szempontok támogatásával. Ilyen lehet például a 2.4. szakaszban bemutatott *Visual Code Navigator* eszköz által is támogatott szoftverevolúció-modellezés, vagy az elemzett szoftver dinamikus tulajdonságainak (viselkedésének) modellezése.

A rendszer a 2. fejezetben megvizsgált eszközök mintájára további komponensekkel is bővíthető. Ezek közül a legfontosabb talán a 2.2.2. pontban bemutatott *SolidFX*-hez hasonló lekérdező komponens (*query engine*), mely lehetővé teszi tetszőleges bonyolultságú lekérdezések futtatását a modellen, ezáltal új nézetek vagy akár adatok (pl. metrikák) egyszerű, dinamikus előállítását a felhasználó számára.

A keretrendszer már meglévő elemeinek a továbbfejlesztése az implementációs oldalon elsősorban a gyakori felhasználási esetekhez tartozó minta megvalósítások készítésével történhet, melyek segítségével a keretrendszer a gyakorlatban gyorsabban bevezethetővé és alkalmazhatóvá válik. Ilyen irányú bővítésre lehetőség van a keretrendszer mindkét rétegében. Egyrészt a *szoftvermodellnek* elkészíthetjük olyan implementációit, amelyek a leggyakrabban használt szintaktikus elemzőkhöz[1] nyújtanak kapcsolatot. Másrészt a *megjelenítést támogató réteg* kiterjeszthető további vizualizációs komponensek referencia implementációival. Ilyen lehet például a 2. fejezetben megvizsgált eszközöknél is bemutatott gráf nézet, *treemap* nézet, *radi-*

[1] Szintaktikus elemzők alatt értjük például a már létező fordító infrastruktúrákat, leírónyelv feldolgozókat stb.

al view vagy különféle 3 dimenziós nézetek. A gyakran használt transzformációs műveletekre további proxy-modell megvalósítások is készíthetők.

Végül a keretrendszer elterjedését segítheti, ha a különböző integrált fejlesztői környezetekbe való beépülését megkönnyítő csatoló komponensekkel bővítjük azt. Így ugyanis a végfelhasználó fejlesztők munka közben még kényelmesebben böngészhetik az éppen fejlesztett kód modelljét a különböző vizualizációk segítségével.

Irodalomjegyzék

[1] R. Baecker, „The early history of software visualization," 1996.

[2] P. Eades and K. Zhang, *Software Visualisation*. Series on Software Engineering and Knowledge Engineering, World Scientific, 1996.

[3] T. Ball and S. G. Eick, „Software Visualization in the Large," *Computer*, vol. 29, pp. 33–43, Apr. 1996.

[4] J. Jones, „Abstract Syntax Tree Implementation Idioms,"

[5] S. Ducasse, M. Lanza, and S. Tichelaar, „MOOSE: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," in *In proceedings of the second international symposium on constructing software engineering tools (COSET*.

[6] D. Reiners, L. Voinea, O. Ersoy, and A. Telea, „A Visual Analytics Toolset for Program Structure, Metrics, and Evolution," in *Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, pp. 146–160, 2010.

[7] A. Telea and L. Voinea, „An interactive reverse engineering environment for large-scale C++ code," in *Proceedings of the 4th ACM symposium on Software visualization*, SoftVis '08, (New York, NY, USA), pp. 67–76, ACM, 2008.

[8] J. Feigenspan, „Requirements and design for a language-independent IDE framework to support feature-oriented programming," January 2009. BSc. thesis.

[9] J.-M. Favre, „G$^{\text{SEE}}$: a Generic Software Exploration Environment," in *9th International Workshop on Program Comprehension, IWPC'2001*, pp. 233–244, IEEE, 2001.

[10] J.-D. Fekete, „The InfoVis Toolkit," in *Proceedings of the 10th IEEE Symposium on Information Visualization*, InfoVis'04, pp. 167–174, IEEE Press, 2004.

[11] „Oracle." `http://www.oracle.com/`.

[12] M. Lanza and S. Ducasse, „Codecrawler - an extensible and language independent 2d and 3d software visualization tool," in *In Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series*, p. 74, 2005.

[13] T. Panas, T. W. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc, „Architectural Visualization of C/C++ Source Code for Program Comprehension."

[14] „Babel – Language Interoperability Tool." `https://computation.llnl.gov/casc/components/`.

[15] „Rose fordító-infrastruktúra." `http://rosecompiler.org/`.

[16] „Vizz3D – a 3D Information Vizualization System." `http://vizz3d.sourceforge.net/`.

[17] D. Cruz, M. Berón, P. Henriques, and M. J. Pereira, „Strategies for program inspection and visualization," 2008.

[18] J. Malnani, „X-Ray, an open-source software visualization plug-in." `http://xray.inf.usi.ch/`, 2008.

[19] M. Skurla, „Javac AST Visualization on NetBeans." `http://netbeans.dzone.com/nb-javac-ast-visualization`, 2012.

[20] „A DOT irányított gráfleíró nyelv." `http://www.graphviz.org/content/dot-language`.

[21] „Graphviz." `http://www.graphviz.org/`.

[22] IBM Technologies, „IBM ManyEyes." `http://www-958.ibm.com/software/data/cognos/manyeyes`.

[23] G. Lommerse, F. Nossin, L. Voinea, and A. Telea, „The Visual Code Navigator: An Interactive Toolset for Source Code Investigation,"

in *Proceedings of the Third International Workshop on Academic Software Development Tools and Techniques (WASDeTT-3)*, INFOVIS '05, (Washington, DC, USA), pp. 4–, IEEE Computer Society, 2005.

[24] B. Shneiderman, „Treemaps for space-constrained visualization of hierarchies," *ACM Transactions on Graphics*, 1992.

[25] F. Chevalier, D. Auber, and A. Telea, „Structural analysis and visualization of C++ code evolution using syntax trees," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE '07, (New York, NY, USA), pp. 90–97, ACM, 2007.

[26] M. Sloan, „Visualising the Haskell AST." `http://www.mgsloan.com/wordpress/?p=36`.

[27] A. Hidayat, „ECMAScript parsing infrastructure for multipurpose analysis." `http://esprima.org/`.

[28] E. C. M. A. International tech. rep., Geneva, Switzerland, 6.

[29] H.-J. Schulz and H. Schumann, „Visualizing Graphs - A Generalized View," in *Proceedings of the conference on Information Visualization*, IV '06, (Washington, DC, USA), pp. 166–173, IEEE Computer Society, 2006.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA: Addison Wesley, 1995.

[31] OMG, „OMG Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.4.1)," Tech. Rep. OMG Document Number: formal/2011-08-06, Object Management Group, 8 2011.

Ábrajegyzék

Táblázatjegyzék