Szegedi Tudományegyetem
Szoftverfejlesztés Tanszék

C++ Forráskód Modellezése és Visszatervezése

Ph.D. értekezés tézisei

Ferenc Rudolf

Témavezető:

Dr. Gyimóthy Tibor

Szeged
2004

Bevezetés

A szoftverrendszerek gyorsan növekednek és változnak, így a ma megírt forráskód nagyon rövid idő alatt elavul, úgynevezett *örökölt kóddá* válik. Ez főleg a piac gyorsan változó igényei és a folyamatosan változó, új technológiák következménye. A mindig szoros határidők miatt a legtöbbször nem sikerül tisztességesen kibocsátani a terméket naprakész dokumentációval (mint amilyenek például a tervezési leírások és a forráskód kommentek). Ezekben az esetekben az egyetlen érvényes dokumentáció maga a forráskód. Az előbb vázolt helyzet következménye lehet például sok klón a forráskódban, halott kódrészletek és a kód hibákra való hajlama. E problémák orvoslására törekszik az *újratervezés* (*reenginering*) tudománya, amely különböző módszereket, technikákat és eszközöket kísérletez ki a programmegértés és karbantartás megkönnyítésére. Az újratervezés első fele a *visszatervezés* (*reverse engineering*), amely „a vizsgált rendszer analizálásának a folyamata, hogy (a) beazonosítsuk a rendszer komponenseit és azok egymás közötti viszonyát és (b) megalkossuk a rendszer ábrázolását egy más formában vagy egy magasabb absztrakciós szinten" [3].

Az objektum-orientáltság lett az elmúlt években a legnépszerűbb paradigma a nagy szoftverrendszerek tervezésére és implementálására. Mára már sok objektum-orientált rendszer került olyan állapotba, hogy örökölt kódként lehet tekinteni rájuk, amelyeket újra kell tervezni. A régebbi programozási nyelvekkel (mint például a COBOL) ellentétben az objektum-orientált nyelvekhez kifejlesztett módszerek még nincsenek teljesen kidolgozva. Az igazán nagy és komplex rendszerek, mint például a telekommunikációs és irodai szoftverek általában C++ nyelven vannak megírva. Ez a programozási nyelv valószínűleg a legkomplexebb mind közül és így, nem meglepő módon, a legkevésbé támogatott újratervező módszerekkel és eszközökkel. Ez a nyelv adja nekünk a legizgalmasabb kihívásokat és lehetőségeket a kutatásra.

Hogy megértsünk egy ismeretlen szoftverrendszert sok különböző dolgot kell tudnunk róla. Ezeket az információkat a forráskódról szóló *tényeknek* nevezzük. Egy tény például a kód mérete. Egy másik tény az, hogy egy osztálynak van-e ősosztálya. Valójában minden információt, amely segít egy ismeretlen programkód megértésében, ténynek nevezünk az értekezésben. Magától értődő, hogy a tények kézzel történő feltárása csak aránylag kis forráskód vizsgálatakor lehetséges. Valós rendszerek, amelyek több millió programsorból állnak csakis szoftvereszközök felhasználásával analizálhatók.

A mi megközelítésünkben az *eszközzel támogatott tényfeltárás* egy automatizált eljárás, amely során a vizsgált rendszert fájlonként elemezzük analizáló eszközökkel, hogy beazonosítsuk a forráskód különféle tulajdonságait és egymás közötti viszonyait, valamint létrehozzuk a kinyert információ valamilyen magasabb szintű ábrázolását. Ezek az információk a későbbiekben felhasználhatók különféle újratervező eszközökben, mint például a metrikaszámítókban és szoftvervizualizálókban. Azonban az analizáló eszközök kimeneti formátuma sajnos nem szabványosított, a legtöbb eszköznek saját formátuma van ami együttműködési problémákhoz vezethet. Minden eszköz, amely szeretné felhasználni egy másik eszköz összegyűjtött információit, különféle konvertereket implementál az adatok elérésére.

Ezt a problematikus helyzetet felismerték a kutatók és jelentős erőfeszítéseket tettek, hogy javítsanak rajta. Egyik fő eredménye ennek a munkának a GXL [20] (Graph eXchange Language/Gráf Adatcserélő Nyelv). De a GXL használata még nem elegendő, mert nyújt ugyan egy szabványos csatornát gráfok (csomópontok és élek) adatcseréjére, de nem határozza meg, hogy hogyan ábrázoljuk a különféle programnyelv-specifikus entitásokat, mint például a C++ osztályokat és függvényeket.

A kutatók már foglalkoztak ezzel a területtel is (pl. [4–7; 16; 19; 22]), és több megoldást is javasoltak, ám egyik sem lett széleskörűen elfogadva és használva. Egy közös szabványos formátum (séma) nélkül nehéz a különböző C++ visszatervező eszközök közötti gördülékeny adatcserét megvalósítani. *Sémának* nevezzük az adatok formátumának a leírását attribútumokkal ellátott entitásokkal és relációkkal. Egy *séma példány* (más szavakkal *modell*) a séma egy megtestesülése, amely egy konkrét szoftverrendszert modellez. Ez hasonló az adatbázisokhoz, amelyek szintén rendelkeznek sémával (általában E-R diagramokkal megadva) és ennek megfelelő konkrét adatokkal (rekordok). Az értekezésben bemutatunk egy sémát a C++ nyelvhez.

A tényfeltárás eredményeire sok kutatás épül az újra- és visszatervezés területén. Ilyenek például a szoftvermérések (különféle metrikák), vizualizáció, dokumentálás és a programmegértés. De viszonylag kevés cikk foglalkozik a tényleges tényfeltárási eljárással C++ forráskódból. Mi bemutatunk olyan módszereket, amelyekkel megvalósítható a valós szoftverek automatikus tényfeltárása. Kifejlesztettünk egy *Columbus* [8–12; 14; 15] nevű keretrendszert ezen módszerek támogatására és a visszatervezésre általában, mely gondoskodik a tények ábrázolásáról, szűréséről és konverziójáról különféle formátumokra, hogy elősegítsük az eszközök közötti adatcserét. A keretrendszert mára már a világ számos egyetemén és kutatóközpontjában használják kutatási és oktatási célokra.

Hogy még egy szinttel magasabbra lépjünk az elemzett szoftverrendszer absztrakciójában, új módszereket fejlesztettünk ki a tervezési minták felismerésére a séma példányainkban, és így egyúttal a forráskódban is. A tervezési minták hasznos megoldásokat reprezentálnak objektum-orientált alakban és a legtermészetesebb lehetőséget nyújtják az elemzett rendszer mögött rejlő architektúrális tervezési megfontolások rekonstruálására a forráskódból.

Annak demonstrálására, hogy a módszereink nagyméretű, valós rendszereken is jól használhatók, analizáltuk a népszerű *Mozilla* [27] internetes programcsomag számos verziójának a forráskódját és különféle metrikákat számítottunk ki belőle, hogy előrejelezzük a kód hibára való hajlamát. Összehasonlítottuk a Mozilla hét különböző verziójának a metrikáit és bemutattuk, hogyan változott a szoftver hibára való hajlama a fejlesztése folyamán.

Az értekezés négy tézist tartalmaz, melyek a következők:

1. **Séma a C++ programozási nyelvhez.**
2. **C++ tényfeltáró eljárás és keretrendszer.**
3. **Tervezési minták felismerése C++ forráskódban.**
4. **Nyílt forráskódú szoftverek hibára való hajlamosságának vizsgálata.**

A következő fejezetekben röviden bemutatom ezeket a téziseket. Minden fejezet végén külön kiemelem a saját hozzájárulásomat az eredményhez.

1. Séma a C++ programozási nyelvhez

Ezt a munkát az a felismerés motiválta, hogy rendkívüli a fontossága annak, hogy a különböző újratervező eszközök (mint például elemzők, metrikaszámítók és klón felismerők) adatokat tudjanak cserélni egymással. E cél eléréséhez szükség van egy közös formátumra, melynek segítségével kommunikálni tudnak egymással ezen eszközök. Részletesen kidolgoztunk egy sémát, melynek a neve *Columbus Séma a C++*-hoz [7; 12; 16], amely C++ nyelven íródott forráskódról képes információkat ábrázolni. A séma moduláris, aminek következtében rugalmasan bővíthető és módosítható. Aprólékosan ábrázol minden fontos tényt a forráskódról úgy, hogy egy logikailag ekvivalens forráskód generálható a példányaiból. A séma kidolgozása egy hiánypótló munka, mert ilyen részletességgel még senki sem publikált sémát a C++ nyelvhez. Ennek oka valószínűleg a C++ nyelv rendkívüli komplexitásában keresendő. Az értekezés első tézise magában foglalja a séma megtervezése mellett annak implementációját is. Az implementáció, melyet a C++ elemzőnk is használ, tartalmaz még algoritmusokat névfeloldásra, típus ellenőrzésre, mentésre/betöltésre, osztálydiagram és függvényhívási gráf generálásra, hogy csak néhányat említsünk.

A séma leírása szabványos UML osztálydiagramokkal van megadva, amely által egyszerűen implementálható és fizikailag is könnyen ábrázolható (pl. GXL segítségével). Annak ellenére, hogy nem alkalmas formális leírásokra, az UML-t választottuk, mert mára egyeduralkodó szabvánnyá vált az objektum-orientált tervezésben és így könnyebben elsajátítható a séma a felhasználók számára.

A séma struktúrája

A C++ nyelv nagy komplexitása miatt úgy döntöttünk, hogy modularizáljuk a sémánkat hasonlóan a [16] cikkben leírtakhoz. Ez egyben lehetőséget biztosít a séma bővítésére és módosítására is. A sémát hat csomagra bontottuk, melyek a következők:

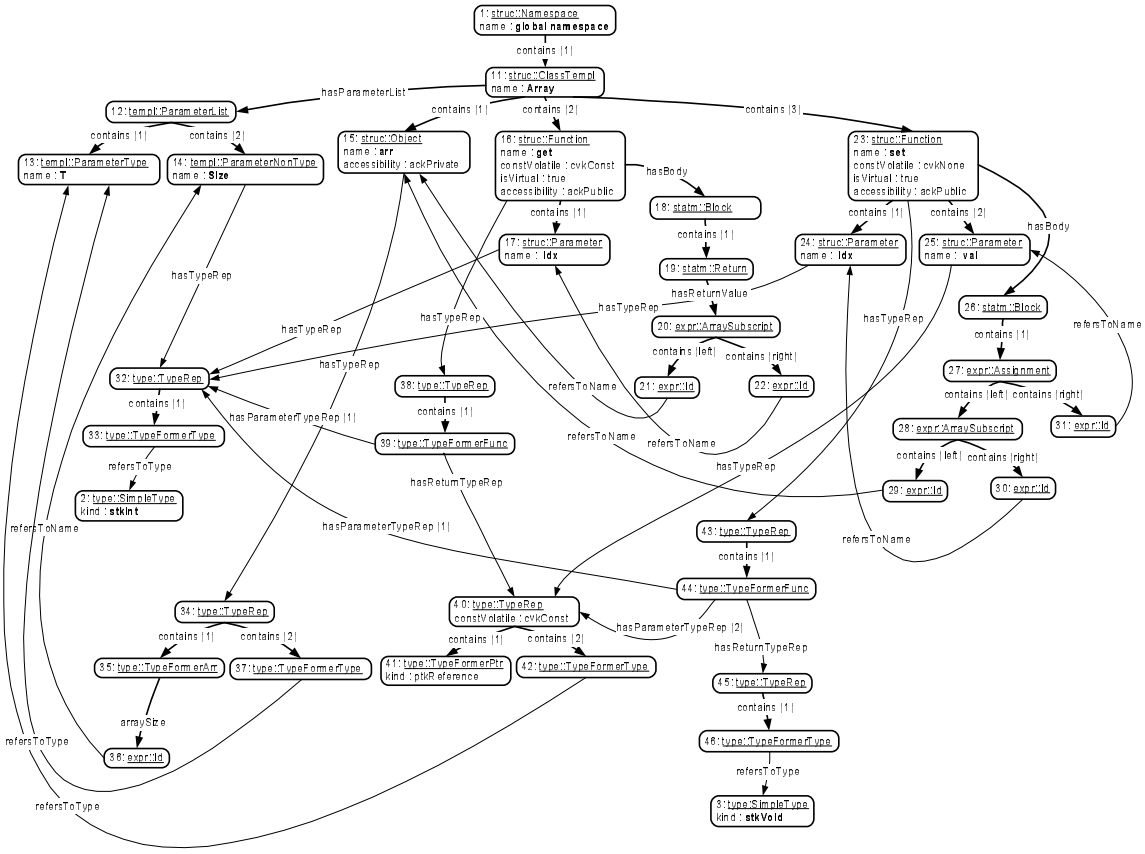
- *base*: alapcsomag, amely ősosztályokat és adattípusokat tartalmaz a séma többi része számára.
- *struc*: ez a csomag modellezi a fő program entitásokat a beágyazási struktúrájuknak megfelelően (pl. objektumok, függvények és osztályok).
- *type*: az ebben a csomagban található osztályok ábrázolják az entitások típusait.
- *templ*: ez a csomag ábrázolja a sablonparaméter és argumentum listákat.
- *statm*: ebben a csomagban találhatók az utasítások modellezésére szolgáló osztályok.
- *expr*: ez a csomag modellezi a kifejezéseket.

Ebben a tézisfüzetben csak a legérdekesebb diagramot mutatjuk be, a *struc* csomagét (lásd a 3. ábrát). A teljes séma leírása az értekezésben található.

A séma használatát egy példán keresztül illusztráljuk. Az 1. ábrán található példa C++ forráskódot használjuk fel erre a célra. A példához tartozó séma példány a 2. ábrán látható. Egy objektumdiagram-szerű jelölést használunk, ahol a csomópontok a séma osztályainak az objektumpéldányait ábrázolják, az őket összekötő élek pedig a különféle asszociációs és aggregációs relációk megtestesülését jelenítik meg. A diagramot leegyszerűsítettük a könnyebb érthetőség érdekében azzal, hogy elhagytunk bizonyos attribútumokat, mint például azt, hogy melyik fájl hányadik sorából származnak az entitások.

```
template <typename T, int Size>
class Array {
  T arr[Size];
public:
  virtual const T& get(int idx) const {
    return arr[idx];
  };
  virtual void set(int idx, const T& val) {
    arr[idx] = val;
  }
};
```

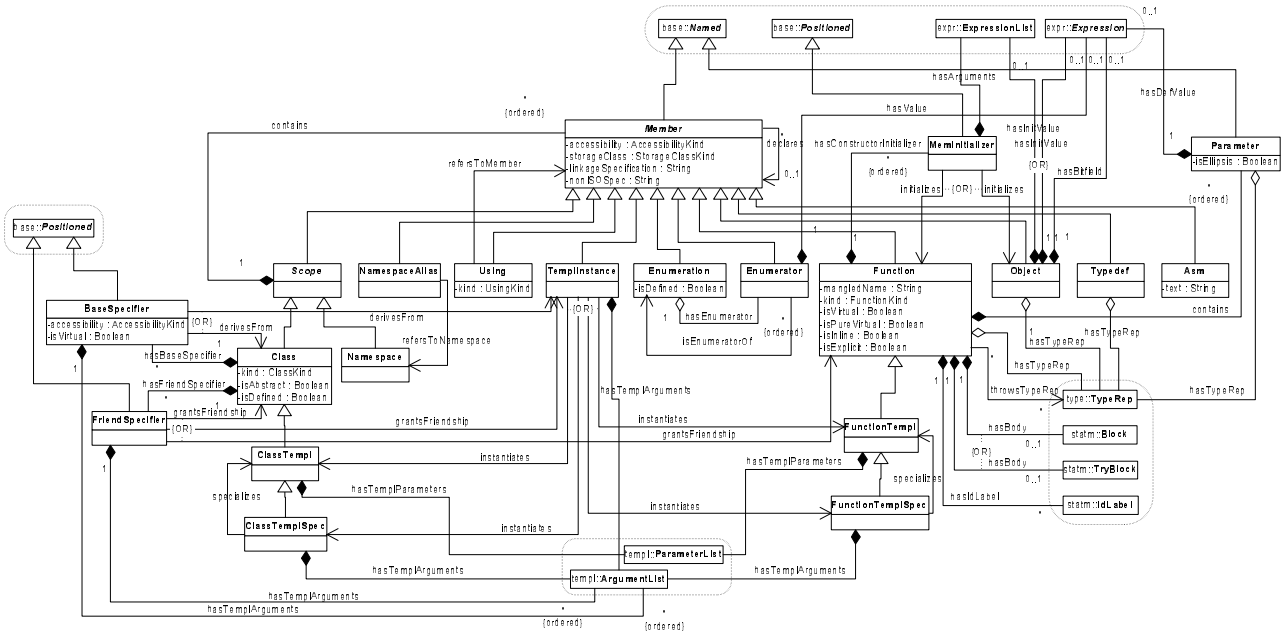
1. ábra. C++ forráskód példa.



2. ábra. A fenti kód séma példánya (modellje).

Adatcsere más eszközökkel

A Columbus rendszer által kinyert, a sémának megfelelő adatok cseréjét több gyakorlati alkalmazásban is megvalósítottuk. Az első ilyen felhasználás egy a Nokia Kutatóközponttal együttműködésben lezajlott K+F projekt keretein belül történt. Ekkor a Columbus keretrendszert a Nokia saját *TDE* [31] nevű UML tervező eszközéhez használtuk fel C++ analizátorként. A felépített sémapéldányokból osztálydiagramokat állítottunk elő és egy COM interfészen keresztül átszállítottuk azokat a TDE-be.



3. ábra. A **struc** csomag osztálydiagramja (egy a séma hat csomagja közül).

A Helsinki Egyetemen zajló *Maisa* projektben [25; 26] is sikeresen volt használva a Columbus kimenete tervezési minták felismerésére C++ forráskódban. A séma alkalmazásra került a FAMOOS projektben is a Crocodile metrikás eszközzel [29]. Egy fontos felhasználás még a jelenleg is zajló fejlesztés a Columbus és a GUPRO eszköz [6] közötti adatcserének a megvalósítására GXL segítségével. Sikeres adatcserét valósítottunk meg a *rigi* gráf vizualizáló eszközzel [24; 28] is. Nemrégiben a kanadai Waterloo Egyetemmel kezdtünk el egy közös kutatást, melynek keretein belül Columbus sémapéldányokat szeretnénk vizualizálni a PBS – Portable Bookshelf-ben [17].

Saját hozzájárulás

Ezt a munkát az a meglátás motiválta, hogy a sikeres adatcsere kulcsfontosságú az újra- és visszatervező eszközök számára. Ehhez szükség van egy olyan közös formátumra, amely egységesen használható a különféle eszközökben, mint amilyenek az elemzők és a metrikaszámítók. Egy szabványos séma felállítása még mindig várat magára. Ebben az értekezésben erre a célra egy adatcserére alkalmas sémát mutatok be a C++ nyelvhez, melynek a neve Columbus Séma a C++ nyelvre.

A Columbus sémát én terveztem, implementáltam és a Columbus visszatervező keretrendszer részévé tettem. Az implementáció tartalmaz még különféle algoritmusokat névfeloldásra, típus ellenőrzésre, mentésre/betöltésre, osztálydiagram és függvényhívási gráf generálásra, hogy csak néhányat említsek.

2. C++ tényfeltáró eljárás és keretrendszer

Egy kis program tényfeltárása aránylag egyszerű és könnyedén elvégezhető kézzel is. Az igazi kihívást egy olyan valódi szoftver elemzése jelenti, amely több millió programsorból áll. Az értekezésben bemutatunk egy eljárást [15], amely öt kulcsfontosságú lépésből áll, amelyek végrehajtásával sikeresen el lehet végezni egy C++ tényfeltáró eljárást. Az eljárás olyan fontos dolgokra tér ki, mint például a konfigurációk kezelése, a sémapéldányok összefésülése, a feltárt tények szűrése, majd konvertálása. Részletesen bemutatjuk a Columbus visszatervező keretrendszert [12] is, amely messzemenően támogatja az eljárást. A keretrendszert széleskörűen használják a világ különböző egyetemein, és az értekezés írásáig több mint 600 letöltést regisztráltunk.

A tényfeltáró eljárás

A tényfeltáró és prezentáló eljárás vázlata a 4. ábrán látható. Az eljárás öt egymást követő lépésből áll, ahol mindegyik lépés felhasználja az előző lépés eredményét. A következőkben ezeket a lépéseket mutatjuk be.

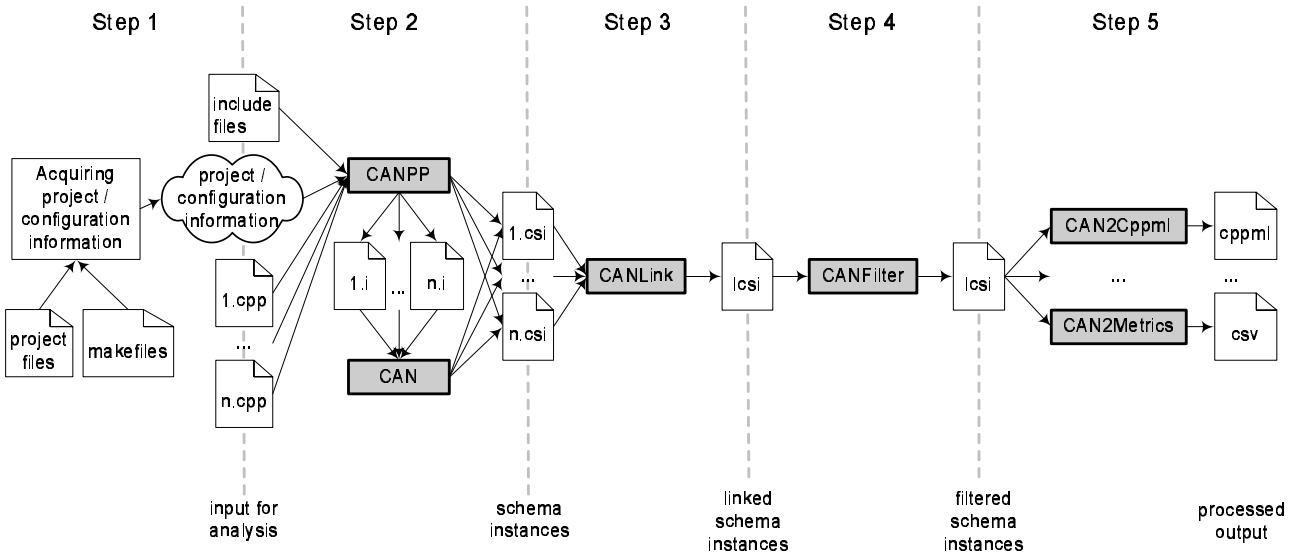
A javasolt módszer fontos előnye, hogy a lépések *inkrementálisan* végezhetők el, vagyis ha az egyes lépések eredményei készen állnak és a lépés bemenete nem változott, akkor az eredményeket nem kell újra létrehozni.

1. lépés: Projekt/konfigurációs információk kinyerése

A szoftverrendszerek forráskódja rendszerint több fájlra van osztva, és a fájlok is könyvtárakba és alkönyvtárakba vannak rendezve. Ráadásul különféle előfeldolgozási konfigurációk lehetnek érvényesek rájuk. Az az információ, hogy ezek a fájlok hogyan viszonyulnak egymáshoz és milyen beállítások érvényesek rájuk általában *makefile*-okban (amennyiben a szoftvert a *make* eszközzel fordítjuk) vagy különféle *projekt fájlokban* (amennyiben a szoftvert IDE – integrált fejlesztő környezetekkel fordítjuk) vannak eltárolva. Ezekből a fájlokból a projekt/konfigurációs információk kinyerése nem egy triviális feladat, mivel a különféle IDE-k különböző (és a legtöbb esetben dokumentálatlan) fájlformátumokat használnak. A makefile-ok további nehézségeket okoznak: az információ kinyerése belőlük különösen nehéz, mert nem csak kizárólag programfordításra használhatók, hanem egyéb feladatokra is alkalmasak. Bemutatunk egy úgynevezett *fordítóprogram elrejtési* módszert a makefile információk hasznosítására és két különböző módszert az IDE projekt fájlok kezelésére: az *IDE integrációt* és a *projekt fájl importálást*.

2. lépés: A forráskód analizálása – a sémapéldányok létrehozása

Ebben a lépésben a bemenő fájlokat egyesével fel kell dolgozni az előző lépésben kinyert információk alapján. Először a fájlok előfeldolgozása és az előfeldolgozással kapcsolatos információk feltárása történik meg az előfeldolgozó segítségével. Ezután az előfeldolgozott fájlokat a C++ analizáló leelemzi és feltárja belőlük a C++ nyelvvel kapcsolatos tényeket. Mindkét eszköz létrehozza a megfelelő sémapéldányokat.



4. ábra. A tényfeltáró eljárás.

3. lépés: A sémapéldányok összefésülése

Miután mindegyik sémapéldány fájl elkészült, ezen példányokat össze kell fésülni. Ezáltal, hasonlóan az igazi fordítóprogramokhoz, amelyek különálló fájlokat készítenek a logikailag összetartozó entitásoknak (mint például a könyvtármodulok és a futtathatók), az egymással kapcsolatban álló entitások megfelelően csoportosítva lesznek.

4. lépés: A sémapéldányok szűrése

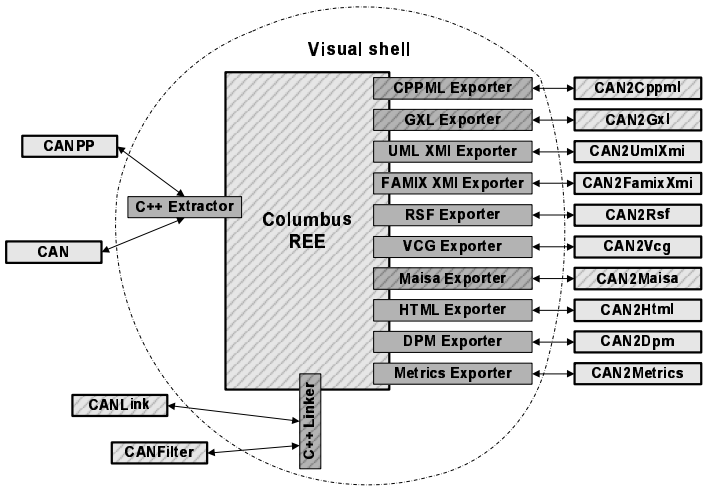
Nagy rendszerek esetén az előző lépések nagy sémapéldányokat tudnak eredményezni, amelyek óriási mennyiségű kinyert adatot tartalmaznak. Ezt nehéz használható módon prezentálni, ezért különféle szűrési módszereket kell alkalmazni, mint például csak bizonyos vizsgált modulokat meghagyni további feldolgozásra.

5. lépés: A sémapéldányok feldolgozása

Mivel a különféle C++ újra- és visszatervező eszközök különböző sémákat használnak az adataik ábrázolására, a (szűrt) sémapéldányokat konvertálni kell egyéb formátumokra, hogy széleskörűen fel lehessen azokat használni.

A Columbus keretrendszer

A tényfeltáró eljárást a visszatervező keretrendszerünk támogatja, amelyet a továbbiakban mutatunk be. A *Columbus Visszatervező Keretrendszert* [8–12; 14; 15] egy a Nokia Kutatóközponttal együttműködésben lezajlott K+F projekt keretén belül fejlesztettük ki. A fő célkitűzésünk az volt, hogy létrehozzunk egy keretrendszert, amely támogatja a tényfeltárást és egy közös interfészt biztosít egyéb visszatervező feladatok elvégzésére is. A főprogramot Columbus REE-nek (Reverse Engineering Environment/Visszatervező Környezet) hívják, amely egyben a grafikus felhasználói interfésze is a



5. ábra. A Columbus REE C++-specifikus konfigurációja.

keretrendszernek. A Columbus REE nincs a C++ nyelvre korlátozva, a C++-specifikus feladatokat a megfelelő bővítő modulok látják el. Ezáltal a REE könnyedén bővíthető más programnyelvekkel, és egyéb visszatervezési feladatokra is alkalmassá tehető. A konkrét analizálási és tényfeldolgozási feladatok különféle parancssori programokkal vannak megvalósítva, amelyeket a Columbus REE vezényel. A Columbus keretrendszer a következő eszközöket tartalmazza:

- *Columbus REE*. A keretrendszer grafikus felhasználói felülete.
- *Columbus IDE Add-inek*. A keretrendszer grafikus felhasználói felülete IDE-kben.
- *CANGccWrapper eszközkészlet*. GCC fordítóprogram elrejtő eszközkészlet.
- *CANPP*. C/C++ előfeldolgozó és sémapéldány építő eszköz.
- *CAN*. C/C++ analizáló és sémapéldány építő eszköz.
- *CANLink*. C++ sémapéldány összefésülő eszköz.
- *CANFilter*. C++ sémapéldány szűrő eszköz.
- *CAN2**. C++ sémapéldány konvertáló és feldolgozó eszközök.

Columbus REE (Reverse Engineering Environment)

A Columbus REE egy általános visszatervező környezet, melyben minden C++-specifikus feladatot valamely bővítő modul lát el. Ezek a modulok *extractor-*, *linker-* és *exporter modulok* lehetnek. Az 5. ábrán látható a rendszer aktuális C++-specifikus konfigurációja.

Columbus IDE Add-inek

A Columbus REE egy jelentős része a projekt (konfiguráció) kezelésével van elfoglalva. Ezt megteszik a népszerű IDE-k is, így logikus volt, hogy a Columbus REE maradék részét – azt, amely a tényfeltáró eljárással foglalkozik – becsomagoljuk egy különálló, *Columbus DLL* nevű komponensbe is, amely kommunikál az úgynevezett *Columbus IDE Add-inekkel*. Ezek az add-inek pedig olyan modulok, amelyek beépülnek az IDE-kbe és azok funkcionalitását bővítik.

Fordítóprogram elrejtés

A make program és a makefile-ok rugalmas eszközök a szoftverrendszerek konfigurálására és fordítására. A makefile-ok a fordítandó fájlokon és azok beállításain kívül tartalmazhatnak különféle más parancsokat is, mint például külső programok futtatása. Ezek a lehetőségek nyilván fejfájást okoznak a visszatervező szakembereknek, mert a makefile-okban található parancsokat valahogyan szimulálni kell a visszatervező eszközben.

Mi ehhez a problémához más irányból közelítettünk és úgy oldottuk meg, hogy ideiglenesen „elrejtjük" a fordítóprogramot egy eszközkészlettel. Az eszközünk megváltoztatja a PATH környezeti változót úgy, hogy a mi programjainkra mutasson, amelyek a fordító programjainak a neveit viselik. Így amikor az igazi fordítónak kellene elindulni, a mi programunk fog meghívódni helyette, amely miután lefuttatta az eredeti fordítót, elindítja a mi analizáló eszközeinket is. A módszert sikeresen alkalmaztuk a GCC fordítóprogrammal a nyílt forráskódú valós méretű *Mozilla* rendszeren, és ezzel igazoltuk a működőképességét.

Szűrés

A *CANFilter* eszköz három módszert tartalmaz melyek segítenek a sémapéldányok szűrésében: a *C++ entitások szerinti-*, a *bemenő fájlok szerinti-* és az *érvényességi kör szerinti szűrést*.

Séma példány konverziók

Mivel a különféle C++ újra- és visszatervező eszközök különböző sémákat használnak az adataik ábrázolására, a sémapéldányokat konvertálni lehet más formátumokra, hogy elérjük az eszközök közötti együttműködést. A sémapéldányainkat a következő formátumokra tudjuk átalakítani: *CPPML*, *GXL*, *UML XMI*, *FAMIX XMI*, *RSF*, *VCG*, *Maisa* és *HTML*.

Származtatott kimenetek

A séma példányainkat különféle származtatott kimenetek előállítására is használhatjuk. Ez további számítások elvégzését jelenti a példányokon. A következő kimenetek állnak rendelkezésre: *metrikák, tervezési minta felismerő* és a *SourceAudit* forráskód ellenőrző.

Saját hozzájárulás

Létrehoztam egy eljárást, amely öt pontban meghatározza azokat a lépéseket, amelyeket meg kell tenni ahhoz, hogy sikeresen el lehessen végezni egy tényfeltáró feladatot. A Columbus visszatervező keretrendszer több részét én terveztem és implementáltam (többek között az 5. ábrán a csíkos mintával ellátott részeket). A keretrendszer különféle eszközöket és bővítési mechanizmusokat tartalmaz, így megszabadítja a kutatókat attól a tehertől, hogy különböző feladatokra újabb és újabb elemzőket kelljen írni, így a saját konkrét feladatukra összpontosíthatnak.

Én terveztem és implementáltam a keretrendszer következő részeit: *Columbus REE*, *C++ linker modul*, *CPPML/GXL/Maisa exporter modulok*, *Columbus IDE Add-inek*, *CANLink*, *CANFilter* és a *CANGccWrapper* eszközkészlet, továbbá a következő konvertáló algoritmusokat: *CPPML – C++ Markup Language/C++ Jelölő Nyelv* (beleértve a nyelv megtervezését is), *GXL – Graph eXchange Language* és *Maisa* (az algoritmusok a *CAN2Cppml*, *CAN2Gxl* és *CAN2Maisa* eszközökben vannak implementálva). Részt vettem a *tervezési minta felismerő* modul fejlesztésében is.

3. Tervezési minták felismerése C++ forráskódban

A meglévő visszatervező eszközök sokrétű absztrakt szoftverábrázolást képesek jelenleg is előállítani. Az objektum-orientált programok absztrakt ábrázolásának egy természetes módja az UML diagramok használata, de amíg a forráskódból több eszköz is képes UML diagramokat előállítani, addig a *tervezési minták* [18] felismerésére gyakorlatilag nincs szoftveres támogatás. Pedig ahhoz, hogy megbízhatóan rekonstruálni lehessen a forráskód architektúráját és a mögötte rejlő döntéseket, a tervezési minták felismerése elengedhetetlen. Az értekezés harmadik tézispontja két módszert taglal a tervezési minták felismerésére C++ forráskódban. Először bemutatunk egy módszert [13] és eszközkészletet tervezési minták felismerésére a Columbus és a Maisa [25; 26] szoftverek integrációjával. Ez a módszer kibővíti a Columbus keretrendszer tényfeltáró képességeit a Maisa mintafelismerő képességével. A második módszer egy paraméterezhető, gyors gráfillesztő algoritmus, amely egy új megoldást ad a tervezési minta keresés problémájára [1]. Az algoritmus a sémapéldányainkban keresi a tervezési minták előfordulásait. A keresés tartalmazza a függvényhívások, objektumlétrehozások és operáció felüldefiniálások felismerését is. Ezek azok az elemek, amelyekkel képesek vagyunk pontosabban meghatározni a mintapéldányokat. A keresett minták az általunk definiált új, XML-alapú, *Design Pattern Markup Language/Tervezési Minta Jelölő Nyelv* (*DPML*) nevű nyelven vannak leírva. Ezáltal a mintaleírásokat szabadon lehet módosítani, hozzáilleszteni bizonyos helyzetekhez illetve akár új leírásokat is létre lehet hozni.

A Columbus és a Maisa integrációja

A Maisa egy a szoftverarchitektúrák analizálására szolgáló eszköz, amelyet a Helsinki Egyetemen fejlesztettek ki egy kutatási projekt keretében. A Maisa fő feladata a tervezési UML diagramok analizálása és architektúra szintű metrikák számítása a szoftverrendszer korai minőség-előrejelzésére. Továbbá a Maisa képes tervezési minta példányokat keresni a UML diagramokban. Az absztrakció szintje kulcsfontosságú az analízis sikeréhez: minél részletesebbek a diagramok, annál pontosabbak lesznek az eredmények. Így a forráskódból rendelkezésre álló részletes információból történő tervezési minták keresése egy biztató út a Maisa gyakorlati használhatóságának növelésére.

Mivel a Maisa teljes egészében Java-ban lett implementálva, nem tudja közvetlenül elérni a sémapéldányainkat a memóriában, így egy triviális utat választottunk a két eszköz összekapcsolására: a Columbus keretrendszerben egy exporter modul készít egy fájlt a Maisa bemenő formátumában, amelyet azután a Maisa be tud tölteni, és fel tud dolgozni. A Columbus által létrehozott fájl PROLOG tények formájában tárolja a forráskódból feltárt szükséges információt a fő programbeli entitásokról (osztályok, attribútumok, stb.) és a köztük levő kapcsolatokról (öröklődés, kompozíció, stb.).

Kísérletek

A módszer egyszerű kísérletekkel lett tesztelve. Leimplementáltunk néhány szabványos tervezési mintát C++-ban (Singleton, Visitor, Builder, Factory Method, Prototype, Proxy és Memento) és a Columbus segítségével analizáltuk a kódot és elkészítettük a Maisa bemeneti fájljait. Végül a Maisa-val elvégeztettük a keresést, ami sikeres volt ezekben az esetekben. Ezek az egyszerű kísérletek már jelzik a módszer potenciális képességeit, de ennél szélesebb körű, valós világból vett szoftvereken elvégzett kísérletekre van még szükség, hogy leellenőrizzük a módszer igazi hatékonyságát.

Mintabányászó algoritmus a Columbus-ban

A legtöbb a tervezési minták forráskódból történő felismerésével foglalkozó megközelítés csak a minták *alapvető struktúrájával foglalkozik*. Mi kifejlesztettünk egy új módszert, amely túlmutat ezen azáltal, hogy annyi hasznos információt hasznosít a forráskódból amennyi csak lehetséges. Először analizáljuk a C++ forráskódot a Columbus keretrendszerrel, amely felépíti a megfelelő séma példányt. Azután betöltjük a minta leírásainkat, amelyek DPML fájlokban vannak eltárolva. Végül az algoritmusunk hozzárendeli a forráskódban talált osztályokat a minta leírásban találtakhoz és ellenőrzi, hogy olyan kapcsolatban állnak-e egymással, mint ahogyan a leírás azt előírja. Itt kompozíciós, aggregációs, asszociációs és öröklődési kapcsolatokat figyelünk az osztályok esetében és függvényhívási, objektum létrehozási és operáció felüldefiniálási relációkat az operációk esetében. A függvénytörzs analízisének az eredménye a korábbi módszerekhez képest nagyobb pontosságot biztosít.

Kísérletek

Négy valós világból vett publikusan elérhető C++ projekten végeztünk kísérleteket. Ezek a következők:

- *Jikes* [21]. Nyílt forráskódú Java fordító rendszer az IBM-től.
- *LEDA* [23]. Hatékony adattípusok és algoritmusok könyvtára (library of efficient data types and algorithms).
- *StarOffice Calc* [30]. A StarOffice táblázatkezelője. Egy nagy C++ projekt, amely 6 307 forrás fájlból áll (több mint 1,2 millió nem előfeldolgozott nem üres programsor).
- *StarOffice Writer* [30]. A StarOffice szövegszerkesztője. Egy nagy C++ projekt amely, 6 794 forrás fájlból áll (több mint 1,5 millió nem előfeldolgozott nem üres programsor).

Az 1. táblázat mutatja a teszt projektekben beazonosított különböző tervezési minták számát. A legtöbb tervezési mintához elkészítettünk egy „lágy" („soft") leírást is, amelyben kissé enyhítettünk az eredeti, a [18]-ból vett specifikációkon (például nem követeltük meg némely osztálytól, hogy absztrakt legyen). A LEDA kivételével a többi projekt újabb, és észrevehető, hogy sokkal több tervezési mintát alkalmaztak bennük.

Saját hozzájárulás

Két módszert mutattam be a tervezési minták felismerésére C++ forráskódban.

Először leírtam egy módszert és eszközkészletet tervezési minták felismerésére a Columbus és a Maisa szoftverek integrációjával. Ez a módszer kibővíti a Columbus keretrendszer tényfeltáró képességeit a Maisa mintafelismerő képességével. A C++ kódot először analizáltam a Columbus segítségével, majd készítettem egy sémapéldány konvertáló algoritmust, amely adatokat gyárt a Maisa bemeneti formátumában, amely egy PROLOG-jellegű nyelv.

A másik módszerrel egy új megoldást adtam a tervezési minta keresés problémájára, amely magában foglalja a függvényhívások, objektumlétrehozások és operáció felüldefiniálások felismerését is. Ezek azok az elemek, amelyekkel pontosabban meg lehet határozni a mintapéldányokat. A keresett minták az általam definiált új, XML-alapú, Design Pattern Markup Language (DPML) nevű nyelven vannak

Statistics	Jikes	LEDA	Calc	Writer
Abstract Factory	-	-	-	-
Builder	-	-	2	7
Builder soft	-	-	17	9
Factory Method	-	-	-	-
Factory Method soft	-	-	1	9
Prototype	1	-	-	1
Prototype soft	1	-	-	1
Singleton	-	-	-	-
Adapter Class	-	-	-	16
Adapter Class soft	-	-	13	16
Adapter Object	54	-	27	62
Adapter Object soft	62	-	153	135
Bridge	-	-	-	-
Bridge soft	-	-	73	80
Decorator	-	-	-	-
Decorator soft	-	-	-	-
Proxy	36	-	-	4
Proxy soft	44	-	-	5
Chain of Responsibility	-	-	-	-
Iterator	-	-	-	-
Iterator soft	-	-	1	-
Strategy	4	1	10	5
Strategy soft	12	2	20	32
Template Method	5	-	94	101
Visitor	-	-	-	-
Visitor soft	-	-	-	5
Sum total	235	6	442	525

1. táblázat. A beazonosított tervezési minta példányok száma.

leírva. Ezáltal a mintaleírásokat szabadon lehet módosítani, hozzáilleszteni bizonyos helyzetekhez illetve akár új leírásokat is létre lehet hozni. A módszer négy szabadon elérhető szoftveren lett tesztelve.

4. Nyílt forráskódú szoftverek hibára való hajlamosságának vizsgálata

Napjainkban a nyílt forráskódú szoftverek egyre fontosabbakká válnak. Sok nagy cég támogat nyílt forráskódú projekteket, és sok közülük használja is ezeket a szoftvereket a mindennapi munka során. Következésképpen, sok ilyen projekt rohamosan fejlődik és gyorsan nő a mérete. Mivel a nyílt forráskódú szoftvereket általában önkéntesek fejlesztik a szabad idejükben, a forráskód minősége és megbízhatósága kétséges lehet. Különböző kódmérések igazán hasznosak lehetnek, hogy többet tudjunk a kód minőségéről és hibára való hajlamosságáról. A fordítóprogram elrejtő eszközkészletünk segítségével kiszámítottuk az irodalomban [2] ismertetett, a forráskód hibára való hajlamát előjelző metrikákat a nyílt forráskódú, *Mozilla* [27] nevű internetes szoftvercsomag forráskódjából. Ezután összehasonlítottuk a kapott eredményeket a [2]-ben publikáltakkal. Az egyik célkitűzésünk az volt, hogy kiegészítsük az eredményeiket egy valós világból vett szoftver mérési eredményeivel. Ezen kívül összehasonlítottuk a Mozilla hét különböző verziójának (lásd a 2. táblázatot) mért értékeit hogy megvizsgálhassuk hogyan változott a hibára való hajlamossága a fejlesztése során.

ver.	NCL	TLOC	TNM	TNA	A metrikák definíciói
1.0	4 770	1 127 391	69 474	47 428	NCL: Az osztályok száma. (Number of Classes.)
1.1	4 823	1 145 470	70 247	48 070	TLOC: Az összes nem üres sor száma.
1.2	4 686	1 154 685	70 803	46 695	(Total number of non-empty lines of code.)
1.3	4 730	1 151 525	70 805	47 012	TNM: A rendszer összes metódusának száma.
1.4	4 967	1 171 503	72 096	48 389	(Total Number of Methods in the system.)
1.5	5 007	1 169 537	72 458	47 436	TNA: A rendszer összes attribútumának száma.
1.6	4 991	1 165 768	72 314	47 608	(Total Number of Attributes in the system.)

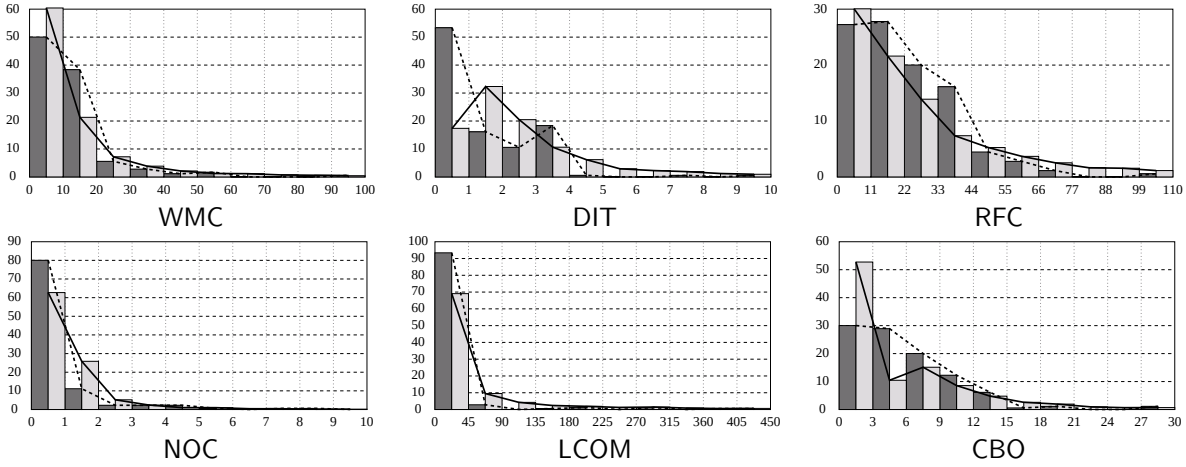
2. táblázat. Rendszer szintű metrikák a Mozilla hét verziójához.

Itt meg kell említenünk, hogy a Mozilla hét verziójának a teljes tényfeltárását elvégeztük és felépítettük a hozzájuk tartozó séma példányokat, amelyek felhasználhatók további újra- és visszatervezési célokra, mint például architektúra helyreállításra és vizualizálásra. Itt csak a metrikák kiszámítására használtuk őket. Nem osztályoztuk a metrikákat kifejezőképességük vagy használhatóságuk szerint, ehelyett felhasználtuk a Basili és társai által kapott eredményeket [2] és a metrikákat eszerint tanulmányoztuk.

Basili és társai diákok által C++ nyelven írt objektum-orientált rendszereket vizsgáltak. Egy kísérletet végeztek el, ahol a diákokat nyolc darab háromfős csoportba osztották, és minden csoportnak ugyanaz volt a feladata – egy kis/közepes méretű projekt fejlesztése. Mivel minden szükséges dokumentáció (például a fejlesztés során előforduló hibák jelentései és azok javítása) rendelkezésükre állt, képesek voltak a hibák gyakorisága és a metrikák közti kapcsolatok vizsgálatára. Erre a célra hat metrikát választottak ki és vizsgálták azok eloszlásait, valamint a köztük lévő korrelációkat. Ezután a metrikák és az osztályokban található hibák közti kapcsolatot elemezték. Erre a projektre a későbbiekben *referencia projektként* fogunk hivatkozni. Az általuk vizsgált hat metrika a következő:

- WMC – Metódusok súlyozott száma osztályonként (Weighted Methods per Class).
- DIT – Öröklődési fa mélysége (Depth of Inheritance Tree).
- RFC – Osztály válasza (Response For a Class).
- NOC – Gyerekek száma (Number Of Children).
- LCOM – Metódusok kohéziójának hiánya (Lack of Cohesion on Methods).
- CBO – Objektum osztályok közötti csatolások (Coupling Between Object classes).

A referencia projekt és a Mozilla összehasonlítása



Az X tengelyen a metrikák értékei találhatók. Az Y tengelyen azon osztályok számának százaléka található, amelyek az adott metrikaértékhez tartoznak.
A sötétebb oszlopok a referencia projekt értékeit mutatják [2], míg a világosabbak a Mozilla 1.6 értékeit ábrázolják.

6. ábra. A referencia projekt és a Mozilla metrikáinak eloszlása.

Összehasonlítottuk a Mozilla 1.6-ra kiszámolt metrikákat a referencia projekt metrikáival. A 6. ábra mutatja *metrikák eloszlásának* összehasonlítását. Látható, hogy a WMC, RFC, NOC és LCOM metrikák eloszlásai nagyon hasonlítanak egymásra, míg a DIT és CBO metrikák eloszlásai eléggé különböznek.

Ref. | **Moz.**	WMC		DIT		RFC	
Maximum	99,00	**337,00**	9,00	**33,00**	105,00	**1 074,00**
Minimum	1,00	**0,00**	0,00	**0,00**	0,00	**0,00**
Medián	9,50	**7,00**	0,00	**2,00**	19,50	**21,00**
Várható érték	13,40	**14,12**	1,32	**2,39**	33,91	**48,95**
Szórás	14,90	**22,16**	1,99	**2,90**	33,37	**81,99**

Ref. | **Moz.**	NOC		LCOM		CBO	
Maximum	13,00	**1 213,00**	426,00	**55 198,00**	30,00	**70,00**
Minimum	0,00	**0,00**	0,00	**0,00**	0,00	**0,00**
Medián	0,00	**0,00**	0,00	**15,00**	5,00	**2,00**
Várható érték	0,23	**1,06**	9,70	**273,82**	6,80	**5,11**
Szórás	1,54	**17,44**	63,77	**1,597,53**	7,56	**7,49**

A félkövér számok a Mozilla 1.6 értékeit reprezentálják, míg a normális számok a referencia projekt értékeit.

3. táblázat. A referencia projekt és a Mozilla 1.6 osztályainak alapvető statisztikái.

Az eloszlások mellet más statisztikákat is összehasonlítottunk. A 3. ábra mutatja a két projekt alapvető statisztikáit. A *Minimum* értékek majdnem megegyeznek, de a *Maximum* értékek jelentősen nőttek, ami nem meglepő, ha figyelembe vesszük azt, hogy a Mozilla-nak megközelítőleg 30-szor több osztálya van, mint a referencia projektnek. Mivel az LCOM az osztályok méretének (pontosabban a tagfüggvények számának) a négyzetével arányos, ezért az ilyen mértékű növekedés várható volt. A

Mozilla-ban megközelítőleg ötezer osztály található, ezért első pillantásra a NOC különösen magas értéke meglepő lehet. De ha figyelembe vesszük azt, hogy a második legnagyobb NOC érték csak 115, akkor azt feltételezhetjük, hogy a nagy NOC értékkel rendelkező osztály egy közös ősosztály, amelyből a legtöbb osztály származik. A *Medián* és a *Várható érték* „egyfajta átlagot" fejez ki, ami többé-kevésbé azonos mindkét esetben, kivéve az LCOM esetében (hasonlóan a Maximumnál leírtakhoz). Mivel a Mozilla-ban sokkal több osztály található, és ezek sokkal változatosabbak, ezért a metrikák sokkal szélesebb skálán mozognak.

Referencia	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0,02	**0,24**	0	**0,38**	0,13
DIT		1	0	0	0,01	0
RFC			1	0	0,09	**0,31**
NOC				1	0	0
LCOM					1	0,01
CBO						1
Mozilla	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0,16	**0,53**	0	**0,64**	**0,39**
DIT		1	**0,54**	0	0,08	**0,23**
RFC			1	0	**0,31**	**0,51**
NOC				1	0	0
LCOM					1	0,16
CBO						1

A félkövér számok a szignifikáns korrelációkat jelölik.

4. táblázat. A referencia projekt és a Mozilla metrikáinak korrelációi.

Basili és társai [2] a metrikák *korrelációját* is kiszámolták (az eredmények a 4. táblázatban láthatók). Azt találták, hogy a lineáris Pearson-féle korreláció az általuk vizsgált metrikák esetében általában nagyon kicsi. Annak ellenére, hogy három együttható valamivel jelentősebb korrelációra utal, azt a következtetést vonták le, hogy ezek a metrikák statisztikai szempontból függetlenek. Mi is kiszámoltuk ezeket a korrelációkat a Mozilla 1.6 esetében, de eltérő eredményeket kaptunk. A NOC független a többi metrikától, hasonlóan, mint a referencia projekt esetében, de a többi metrika esetében valamekkora korreláció azért megfigyelhető. Három esetben ez a korreláció kicsi, de a többi esetben ez többé-kevésbé szignifikáns. Mi több, néhány esetben ez az érték igen nagy (például a WMC és az LCOM között), amiből az következik, hogy ezek a metrikák nem teljesen függetlenek és redundáns információt tartalmaznak. Ez azért meglepő, mert Basili és társai azt találták, hogy ezen metrikák közül némelyek használhatók a hibák észlelésére, míg mások nem.

A Mozilla-n mért metrikák változásának tanulmányozása

Basili és társai hat hipotézist állítottak fel (mindegyik metrikához egyet), amelyek kifejezik a várható kapcsolatot a metrikák és a kód hibára való hajlama között [2]. Ezeket a hipotéziseket ellenőrizték, és azt találták, hogy némelyik metrika jól használható a hibák felismerésére, míg mások nem.

A továbbiakban bemutatjuk az összes hipotézist és következtetést a metrikák a kód hibára való hajlamának felismerésre vonatkozó „jóságával" kapcsolatban, amelyek a [2]-ben találhatók, és ezeket felhasználva elemezzük a Mozilla változásait.

WMC hipotézis: *„Azon osztályok, amelyeknek jelentősen több tagfüggvényük van, mint a többinek, sokkal bonyolultabbak, ezért várhatóan nagyobb a hibára való hajlamuk."* A WMC-t valamennyire szignifikánsnak találták a [2]-ben. A Mozilla-ban a nagy WMC-vel rendelkező osztályok aránya kicsit csökkent, de nem jelentősen. Ez alapján csak azt mondhatjuk, hogy a Mozilla nem romlott e metrika szerint.

DIT hipotézis: *„Ha egy osztály mélyebben helyezkedik el az öröklődési hierarchiában, akkor várhatóan hajlamosabb lesz a hibákra a sok örökölt definíció miatt."* A DIT nagyon szignifikánsnak bizonyult a [2] szerint, ami azt jelenti, hogy minél nagyobb a DIT érték, annál nagyobb a hibának a valószínűsége. A Mozilla esetében azon osztályok aránya, amelyeknek legalább hét ősük van, kissé nőtt, de ezen osztályok aránya elhanyagolható a többihez képest. Viszont a kettő vagy annál kevesebb őssel rendelkező osztályok aránya jelentősen nőtt, míg a több mint kettő de kevesebb, mint hét őssel rendelkező osztályok aránya számottevően csökkent. Ez azt sugallja, hogy a Mozilla újabb verzióiban kevesebb hiba lehet.

RFC hipotézis: *„A nagyobb válaszadó halmazzal rendelkező osztályok összetettebb funkcionalitást valósítanak meg, ezért nagyobb a hibára való hajlamuk."* Az RFC a [2] szerint nagyon szignifikáns. Ez azt jelenti, hogy ha egy osztálynak nagyobb az RFC értéke, akkor nagyobb a hibára való hajlama is. A Mozilla esetében azon osztályok aránya, amelyeknek az RFC értékük nagyobb, mint tíz csökkent (az osztályok több mint 70%-a tartozik ide), míg a többi osztály aránya nőtt. Összességében ez a Mozilla minőségének javulását jelenti (vagyis kevésbé hajlamos a hibákra).

NOC hipotézis: *„Azt várjuk, hogy a sok gyerekkel rendelkező osztályok hajlamosabbak a hibákra."* A NOC nagyon szignifikánsnak bizonyult, de az iránya ellentétes volt azzal, amit a NOC hipotézisben állítottak, azaz minél nagyobb a NOC értéke, annál kevésbé hajlamos a hibákra az osztály. A Mozilla esetében a három vagy több gyerekkel rendelkező osztályok aránya elhanyagolható és ott nem is változtak jelentősen a metrikák. A nulla vagy kettő gyerekkel rendelkező osztályok aránya csökkent, míg az egy gyerekkel rendelkező osztályok aránya nőtt. Ennek megfelelően a Mozilla kicsit javult.

LCOM hipotézis: *„Azon osztályok, amelyeknek a metódusai között kicsi a kohézió feltehetően rosszul lettek megtervezve, ezért valószínűleg nagyobb a hibára való hajlamuk."* Az LCOM a [2] szerint nem szignifikáns, de a hipotézis szerint a Mozilla enyhén romlott, mert a legalább tizenegy LCOM értékkel rendelkező osztályok aránya nőtt, míg a többi alig változott.

CBO hipotézis: *„A szorosan csatolt osztályok hibára való hajlama nagyobb, mint a gyengén csatoltaké."* A [2] szerint a CBO szignifikáns, de e metrika alapján nehéz bármit is mondani a Mozillaról, mert nőtt azon osztályok aránya, amelyeknek a CBO értéke egy és három között van és csökkent a négy és hat közötti értékű osztályok aránya, ami jó és minőségbeli javulást jelenthetne. Másfelöl a nagy CBO értékkel rendelkező osztályok aránya is nőtt, ami romlást jelenthet.

Saját hozzájárulás

Ez a munka három fő eredményt mutat be, amelyek közül az első és a harmadik a saját eredményeim: (1) bemutattam, hogy a fordítóprogram elrejtő eszközkészletem megvalósítja a tényfeltáró eljárást valós világból vett szoftveren; (2) az összegyűjtött tények felhasználásával objektum-orientált metrikák lettek kiszámítva és egy előző munka [2] ki lett egészítve a valós világból vett Mozilla szoftveren végzett mérésekkel; és (3) a kiszámított metrikák segítségével tanulmányoztam, hogy hogyan változott a Mozilla hibára való hajlama hét verzión keresztül, ami másfél év fejlesztést jelent.

$\mathcal{N}o.$	[16]	[7]	[12]	[13]	[1]	[15]
1.	•	•	•			
2.			•			•
3.				•	•	
4.						•

5. táblázat. A tézisek és a velük kapcsolatos publikációk viszonya.

Konklúziók

Az értekezésben bemutattunk egy rendszert, amely segítségével lehetőség nyílik nagy, valós világból vett, C++ programozási nyelven írt szoftverrendszerek forráskódjának a visszatervezésére. Mi több, ehhez nem kell módosítani sem a forráskódot, sem a makefile- vagy projekt fájlokat.

A tényfeltáró eljárás során a keretrendszerünk elkészíti az analizált C++ rendszer egy modelljét egy jól meghatározott sémának megfelelően. Egy ilyen séma birtokában lehetőség nyílik az újra- és visszatervező eszközök továbbfejlesztésére, valamint újak létrehozására, amelyek már zökkenőmentesen képesek lesznek a tanulmányozott rendszerről információt cserélni. A keretrendszerünket kibővítettem egy a sémára alapozott programozói interfésszel is, amellyel a feltárt tények könnyedén elérhetők és felhasználhatók. Különböző formátumú kimeneti fájlok is készíthetők, hogy még inkább elősegítsük az eszközök közötti együttműködést.

Azzal, hogy a keretrendszer különféle algoritmusokat is tartalmaz származtatott kimenetek készítésére (pl. objektum-orientált metrikák), egy teljes megoldást nyújt C++ kód visszatervezésére, így már nem kell a kutatóknak elemzőket készíteni különböző célokra és a saját konkrét visszatervezési feladatukra összpontosíthatnak.

A tényfeltáró- és ábrázoló technológiánk segítségével elértem két további eredményt is. Először is, új módszereket dolgoztam ki a tervezési minták felismerésére C++ forráskódban. Másodszor, analizáltam a nyílt forráskódú Mozilla internetes programcsomag különböző verzióit és tanulmányoztam a hibára való hajlamának a változásait.

Végül, az 5. táblázat összefoglalja, hogy mely publikációk tartalmazzák az értekezés téziseit.

Köszönetnyilvánítás

Először is, szeretném megköszönni témavezetőmnek, Dr. Gyimóthy Tibornak a szakmai támogatását, valamint azt, hogy a nagyon motiváló légkörű Szoftverfejlesztés Tanszéken dolgozhatok. Szeretnék köszönetet mondani kollégáimnak és barátaimnak, Beszédes Árpádnak, Magyar Ferencnek, Vidács Lászlónak, Szokody Fedornak, Lóki Gábornak, Siket Istvánnak, Siket Péternek, Balanyi Zsoltnak és Müller Lászlónak – akik mind részt vettek a Columbus keretrendszer fejlesztésében és tesztelésében – hogy olyan nagy odaadással dolgoztak ezen a szoftveren. Köszönettel tartozom még David Curleynek az értekezés angol nyelvű ellenőrzéséért és Kocsor Andrásnak a hasznos tanácsaiért.

Végül, de nem utolsósorban, őszinte köszönettel tartozom feleségemnek, Györgyinek, a türelméért és a biztos családi háttérért.

Ferenc Rudolf, 2004. november 20.

Hivatkozások

[1] Zsolt Balanyi and Rudolf Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, September 2003.

[2] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In *IEEE Transactions on Software Engineering*, volume 22, pages 751–761, October 1996.

[3] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software 7*, pages 13–17, January 1990.

[4] Thomas R. Dean, Andrew J. Malton, and Ric Holt. Union Schemas as a Basis for a C++ Extractor. In *Proceedings of WCRE'01*, pages 59–67, October 2001.

[5] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization. In *Proceedings of WCRE'99*, 1999.

[6] J Ebert, R Gimnich, H H Stasch, and A Winter. GUPRO – Generische Umgebung zum Programmverstehen, 1998.

[7] Rudolf Ferenc and Árpád Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, March 2002.

[8] Rudolf Ferenc and Árpád Beszédes. Az Objektumvezérelt Szoftverek Elemzése. In *VIII. Országos (Centenáriumi) Neumann Kongresszus Elõadások és Összefoglalók*, pages 463–474. Neumann János Számítógép-tudományi Társaság, October 2003.

[9] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Extracting Facts with Columbus from C++ Code. In *Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 4–8, March 2004.

[10] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Fact Extraction and Code Auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, page 513. IEEE Computer Society, September 2004.

[11] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. *Tools for Software Maintenance and Reengineering*, chapter Extracting Facts with Columbus from C++ Code, pages 16–31. Franco Angeli Milano, 2004.

[12] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

[13] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybernetica*, 15:669–682, 2002.

[14] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss, and Mikko Tarkiainen. Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 16–27. University of Szeged, June 2001.

[15] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.

[16] Rudolf Ferenc, Susan Elliott Sim, Richard C Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a Standard Schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 49–58. IEEE Computer Society, October 2001.

[17] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. In *IBM Systems Journal*, volume 36, pages 564–593, November 1997.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[19] Ric Holt, Ahmed E. Hassan, Bruno Laguë, Sébastien Lapierre, and Charles Leduc. E/R Schema for the Datrix C/C++/Java Exchange Format. In *Proceedings of WCRE'00*, November 2000.

[20] Ric Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, November 2000.

[21] IBM Jikes Project.
`http://oss.software.ibm.com/developerworks/opensource/jikes`.

[22] E Mamas and K Kontogiannis. Towards Portable Source Code Representations Using XML. In *Proceedings of WCRE'00*, pages 172–182, November 2000.

[23] K. Mehlhorn and S. Naeher. LEDA: A Platform for Combinatorial and Geometric Computing. In *Cambridge University Press*, 1997.

[24] Hausi A Müller, Kenny Wong, and Scott R Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proceedings of ACFAS*, 1994.

[25] L. Nenonen, J. Gustafsson, J. Paakki, and A.I. Verkamo. Measuring Object-Oriented Software Architectures from UML Diagrams. In *Proceedings of the 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 87–100, 2000.

[26] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A.I. Verkamo. Software Metrics by Architectural Pattern Mining. In *Proceedings of he International Conference on Software: Theory and Practice (16th IFIP World Computer Congress).*, pages 325–332, 2000.

[27] Christian Robottom Reis and Renata Pontin de Mattos Fortes. An Overview of the Software Engineering Process and Tools in the Mozilla Project. In *Proceedings of the Workshop on Open Source Software Development*, pages 155–175, February 2002.

[28] The Rigi Homepage. `http://www.rigi.csc.uvic.ca`.

[29] Claudio Riva, Michael Przybilski, and Kai Koskimies. Environment for Software Assessment. In *Proceedings of ECOOP'99*, 1999.

[30] The StarOffice Homepage.
`http://www.sun.com/software/star/staroffice`.

[31] A Taivalsaari and S Vaaraniemi. TDE: Supporting Geographically Distributed Software Design with Shared, Collaborative Workspaces. In *Proceedings of CAiSE'97*, LNCS 1250, pages 389–408. Springer Verlag, 1997.