Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Somogyi Norbert Zsolt

FORRÁSKÓD TRANSZFORMÁCIÓJA ECLIPSE KÖRNYEZETBEN

KÖVESDÁN GÁBOR

BUDAPEST, 2017

Tartalomjegyzék

Kivonat

Az elavult szoftverek modern környezetbe történő átültetése mindig bonyolult feladat. Sok ilyen szoftver még mindig használatban van, és a lecserélésük nehéz. Ezeket a szoftvereket úgy kívánatos modern környezetbe átültetni, hogy a korszerű szoftvertervezési konvenciókat kövessék, tehát jól olvasható és karbantartható kódjuk legyen, és a kompatibilitást is teljesen megőrizzék.

Dolgozatomban arra keresek megoldást, hogyan lehetséges procedurális forráskódot automatizált módon objektum-orientált struktúrájú, a kiinduló kódbázissal azonos szemantikájú kóddá alakítani. Ezt a problémát modellvezérelt eszközökkel vizsgálom, célom új eredmények elérése ezen a területen.

Ennek keretein belül bemutatom a felhasznált környezetet és technológiákat, illetve az alkalmazott megoldás mögött rejlő elméleti ismereteket, megfontolásokat.

Ezt követően ismertetem a kidolgozott módszeremet, amellyel a felvetett problémát vizsgálom. Bemutatom továbbá annak egy Java nyelvű, Eclipse technológiákat felhasználó prototípusát.

Végezetül összefoglalom és értékelem a téma kutatásán keresztül elért eredményeket, illetve kitérek a problémakör esetleges további felmerülő, nyitott kérdéseire és bővítési lehetőségeire.

Abstract

Modernizing outdated software is always a complicated task. Countless of them are still in use, and replacing them proves difficult. Transplanting such software into a modern environment should be done in such a way that transforms the original code to follow modern software development conventions, resulting in well readable and maintainable code that remains completely compatible with the former system.

The goal of this work is to examine this problem with regard to automatically transforming the structure of a given procedural source code into an object oriented structure without changing the semantics of the code. The problem is approached in a model driven perspective with the intention of obtaining novel methods in the field of model driven software development.

This paper presents the used environment and different technologies as well as the theoretic knowledge that the solution proposed here relies on. Following this comes the detailed description of how the transformation of the source code looks like, with regard to the structure of the transformed program.

Finally, a summary of results is included, which evaluates what was achieved and describes any remaining, open problems and possibilities of extension.

1 Bevezetés

1.1 Motiváció

Az elavult szoftverek modern környezetbe történő átültetése mindig bonyolult feladat. Sok ilyen szoftver még mindig használatban van, és a lecserélésük nehéz. Ezeket a szoftvereket úgy kívánatos modern környezetbe átültetni, hogy a korszerű szoftvertervezési konvenciókat kövessék, tehát jól olvasható és karbantartható kódjuk legyen, és az eredeti rendszerrel szemantikailag kompatibilisek maradjanak. A rendelkezésre állási követelmények miatt általában nem lehet az eredeti rendszert leállítani, és az új verziót éles környezetben tesztelni.

Ezeket a nehézségeket könnyíthetné meg egy transzformációs eljárás, amellyel a kódot automatikusan le lehetne képezni hasonló struktúrájú objektumorientált kódra, amely kézi refaktorálási lépésekkel könnyebben javítható lenne.

1.2 Procedurális és objektumorientált felépítés

A legacy szoftverek jellemzően procedurális szemléletben íródtak, melynek lényege, hogy definiáljuk a használandó adatstruktúrákat, és a programot funkcionális dekompozícióval globális függvényekre bontjuk, amelyek az adatstruktúrákon műveleteket végeznek. Egyes függvények logikailag tartozhatnak ugyan bizonyos adatstruktúrákhoz, de nyelv szinten az adatok és a rajtuk műveletet végző kód nincsenek egységbe zárva. Ez eltér az emberi szemlélettől, amelyben a programban szereplő fogalmakra (entitásokra) a hozzájuk kapcsolódó műveletekkel együtt gondolunk. Másrészt, nehéz a felelősségek elkülönítése (separation of concerns), és a globális állapotok és függvények használata nehezen átláthatóvá teszi a programkódot.

Ezzel szemben az objektumorientált felépítést követő programok az objektum fogalmának segítségével közelebb állnak az emberi gondolkodáshoz. A négy alap paradigma – egységbezárás, polimorfizmus, öröklés, absztrakció – betartásával a felelősségek jobban elkülöníthetők, és az egyes komponensek könnyebben újrafelhasználhatók, jobban olvashatók és egyszerűbben karbantarthatók lehetnek.

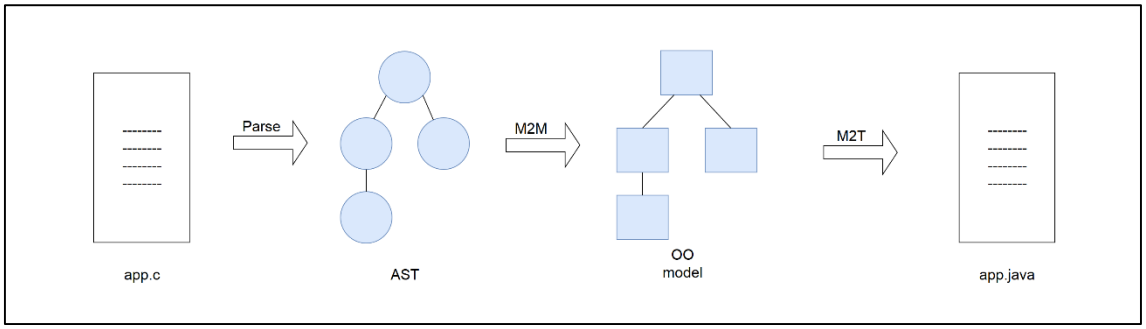
Az objektumorientált programozási nyelvek tehát a modern szoftverekben egyértelműen preferenciát élveznek a procedurális nyelvekkel szemben.

1.3 Célkitűzés

Dolgozatom célja a korábbiaknak megfelelően procedurális kódot automatizált módon objektumorientált kóddá alakítani. Mivel a procedurális programozási nyelvek legelterjedtebb és legjelentősebb képviselője a C nyelv, ezért ilyen nyelvű programok feldolgozását vizsgálom. A célnyelv jelen dolgozatban a Java lesz, viszont a felvetett megoldás részlegesen támogatni fogja a tetszőleges objektumorientált nyelvre való leképzést.

1.4 Megközelítés

A felvetett problémát dolgozatomban modell-vezérelt eszközökkel vizsgálom. Az adott C nyelvű forráskódot feldolgozom, majd programozásinyelv-független modellt építek belőle. Ez a modell fogja leírni a transzformáció eredményeképp kapott program strukturális és egyéb jellemzőit. Ezt bejárva modelltranszformációk sorozatával jutok el a kívánt eredményhez, majd kódgenerálás segítségével előállítom magát a forráskódot az adott célnyelven. Ezt a folyamatot szemlélteti az ***1. ábra***. A dolgozat részletesen bemutatja a transzformáció elvi folyamatát, illetve annak egy gyakorlati prototípusát is.



1. ábra: A transzformáció folyamata.

Fontos kiemelni, hogy maga a folyamat koncepciója nyelvfüggetlen módon történik. Bármely más forrásnyelv és célnyelv esetén is ugyanezeket a lépéseket végeznénk el. A nyelvfüggő lépések csak a modell kialakításának szempontjainál jelenik meg, maga a transzformáció elvi folyamata ugyanaz.

Megemlítendő, hogy a procedurális program szintaktikai átültetése egy objektumorientált nyelv szintaktikájára önmagában nem lenne célravezető. A transzformált szoftver struktúrája az, ami a feladat nehézségét és lényegét adja, hiszen pont azt szeretnénk elérni, hogy korszerű, jól karbantartható kóddá alakítsuk a kiinduló kódbázist.

Az objektumorientált nyelvre való áttérés tehát önmagában nem elegendő, a legfontosabb a megfelelő szerkezet kialakítása. Ez alatt azt értjük, hogy hogyan alakítsuk ki az osztályainkat, hogyan azonosítsuk a metódusainkat, illetve hogyan rendeljük azokat osztályokhoz.

1.5 A dolgozat felépítése

A ***2.*** fejezetben a kidolgozott módszer mögött rejlő elméleti ismereteket, megfontolásokat ismertetem. Bemutatom a forráskód elemzésének és feldolgozásának elméleti folyamatát, illetve röviden felvázolom a transzformáció felépítését. A ***3***. fejezetben a kutatott téma korábbi jelentős megoldási kísérleteiről és megközelítéseiről lesz szó. A ***4.*** fejezet a prototípus elkészítéséhez felhasznált technológiákat ismerteti röviden. Az ***5.*** fejezetben mutatom be a felvázolt probléma megközelítésére kidolgozott módszeremet. Itt fejtem ki részletesen a transzformáció fő lépéseit. A ***6***. fejezet az előző részben bemutatott módszer egy elkészített prototípusát ismerteti, ami a transzformáció fő lépéseit és a felhasznált algoritmusok működőképességét szemlélteti a gyakorlatban. A fejezet végén egy konkrét példán áttekintjük, hogy a kidolgozott módszer során megadott algoritmusok milyen szerkezetet generálnak a példakódból. Végezetül a ***7***. fejezetben összefoglalom a kidolgozott módszert, értékelem a téma kutatásának eredményeit, és megemlítek néhány továbbfejlesztési lehetőséget is.

2 Elméleti alapok

2.1 Formális nyelvek

A gyakorlatban programozási nyelvek leírására formális nyelveket [1] használunk, melyeket általában környezetfüggetlen (CF – context free) nyelvtanok segítségével írunk le. Ezek produkciós szabályok összességével specifikálják az adott nyelven generálható szavak halmazát. A szabályok nemterminális szimbólumokat (változók) és terminális szimbólumokat (szavak) tartalmaznak. A szabályok egymás után való alkalmazásával szavakat generálhatunk. Nyelvünk megengedett kifejezései azok lesznek, amelyeket a szabályok alkalmazásával generálni – szakkifejezéssel levezetni – tudunk.

Az adott programozási nyelv szintaktikáját a formális nyelven megfogalmazott parser szabályok, a lexer működését pedig a lexer szabályok írják le. A szabványos C nyelvhez íródott fordítóprogramok is általában így működnek.

2.2 A kód feldolgozása

A forráskód elemzése több lépésre bontható, ezek sorrendben: lexikai elemzés, szintaktikai elemzés, szemantikai elemzés [2]. Ebben a fejezetben ezeket a lépéseket nézzük meg részletesebben.

2.2.1 Lexikai elemzés

A forráskód elemzésének első lépése a lexikai elemzés. Ez azt jelenti, hogy a kódot – jellemzően ASCII karakterekből álló szövegként értelmezve – úgynevezett tokenekre bontjuk. Ezek egy-egy logikai egységet jelentenek, amikkel később dolgozni tudunk. A tokeneknek van típusa, illetve rendelhetünk hozzájuk attribútumokat is. Azt a programot, ami a lexikai elemzést elvégzi, lexernek nevezzük. Azt, hogy a lexer milyen szabályok alapján építi fel a tokeneket, általában az adott programozási nyelv fordítóprogramjának lexer szabályai határozzák meg. A lexer szabályokban a tokeneket reguláris kifejezések segítségével adjuk meg.

2.2.2 Szintaktikai elemzés

A szintaktikai elemzés a lexikai elemzés során előállított tokeneket dolgozza fel. Célja a tokenek alapján egy feldolgozható struktúra – általában szintaxisfa – felépítése. Ezen felül a szintaktikai elemzés feladata a szintaktikailag hibás kifejezések felismerése is. Azt, hogy mi számít szintaktikailag helytelennek, általában a parser szabályok specifikálják. Azt a programot, ami a szintaktikai elemzést elvégzi, parser[1]nek hívjuk.

A korábbiakban említésre került, hogy formális nyelveket CF nyelvtannal is leírhatunk, melyek produkciós szabályok összességéből állnak. Minden, a nyelven leírható kifejezéshez levezetésekkel juthatunk el. Egy levezetés a produkciós szabályok egymás után való alkalmazása. Ha ezeket a levezetéseket fa struktúrában ábrázoljuk, akkor egy levezetési fát kapunk. Ez a fa tehát azt tartalmazza, hogyan vezettük le az adott szöveget. Ha ebből a fából elhagyjuk a fölösleges, a feldolgozás szempontjából lényegtelen információkat, akkor a kapott struktúrát Absztrakt Szintaxis Fának (Abstract Syntax Tree – AST) nevezzük. A fordítóprogramok jellemzően AST-t generálnak a programok szintaktikai leírására.

2.2.3 Szemantikai elemzés

A szemantikai elemzés célja a forráskód szintaktikailag helyes, de szemantikailag helytelen hibáinak észrevétele. Ilyen lehet például C nyelven egy nem definiált függvény meghívása, vagy egy integer változó sztring értékkel való inicializálása. Mindkét művelet szintaktikailag helyes (a nyelvtan támogatja), de C nyelven nem megengedett.

A szemantikai elemzést első ránézésre fölöslegesnek tűnik külön lépésben elvégezni, hiszen akár a szintaktikai elemzés során is végezhetnénk. Ez azonban azért nem lehetséges, mert sokszor nem áll rendelkezésre elegendő információ a szintaktikai elemzéskor ahhoz, hogy biztosan helyesen tudjunk dönteni. Ilyen például a példaként említett nem definiált függvény hivatkozása: a hívás helyén nem tudjuk biztosan eldönteni, hogy volt-e már definiálva az adott függvény. A szükséges információt az AST bejárásával nyerhetjük ki, így a program teljes szintaktikai elemzése mindenképpen meg kell, hogy előzze a szemantikai elemzést.

[1]A parsing kifejezés sokszor a lexikai, szintaktikai és szemantikai elemzés együttes folyamatára utal.

A szemantikai elemzés tehát az AST-t bejárva egy modellt készít, ami a feldolgozott forráskódot írja le. Ezt azonban nem minden fordítóprogram csinálja így, van olyan is, amely csak AST-t épít. Ezen felül természetesen készíthetők egyéb struktúrák is – pl. indexek –, amire külön nem térek ki, jellemzően azonban a feldolgozás célja a programot reprezentáló AST és/vagy modell felépítése.

2.2.4 Példa

Lássuk most egy egyszerű példán keresztül, hogy hogyan működik a lexikai, szintaktikai és szemantikai elemzés. A példa egy rövid forráskódrészlet elemzésén keresztül mutatja be a parszolás folyamatát.

Adottak a következő lexer szabályok reguláris kifejezések segítségével:

1. táblázat: A lexer szabályok.

Token	Szabály
T_intLiteral	`(+|-)?[0-9]+`
T_stringLiteral	`’[a-z]+’`
T_if	`if`
T_operatorGreater	`>`
T_operatorEquals	`==`
T_operatorLesser	`<`
T_operatorAssign	`=`
T_openParen	`(`
T_closeParen	`)`
T_openBrace	`{`
T_closeBrace	`}`
T_separator	`;`
T_identifier	`[a-z]+`

A definiált tokenek tehát az egész számok, a sztringek, az *if* kulcsszó, a kisebb, nagyobb és egyenlő operátorok, a nyitó és csukó zárójelek, azok kapcsos változatai, az utasításokat elválasztó pontosvessző, illetve a változó azonosítók. Nyelvünkben az egész számok opcionálisan + vagy – jellel kezdődhetnek, amit korlátlan számú (de legalább egy) 0-9-es karakter követ. A sztringek aposztróf karakterek között az angol ábécé kisbetűit tartalmazhatják, a változók azonosítói pedig ugyanezeket a karaktereket aposztróf nélkül.

A parszer szabályok EBNF[2] [3] formátumban az alábbiak:

```
statements: (statement)*;
statement: if | instruction;
instruction: variable, T_operatorAssign, operand, T_separator;
variable: T_identifier;
operand: variable | constant;
if: T_if, T_openParen, logical expression, T_closeParen, T_openBrace, body,
     T_closeBrace;
body: (instruction)*;
logical expression: operand, logical operator, operand;
constant: T_intLiteral | T_stringLiteral;
logical operator: T_operatorGreater | T_operatorEquals | T_operatorLesser;
```

Nyelvünk utasítások sorozatát írja le, amelyek elemi utasítások vagy *if* utasítások lehetnek. Az egyetlen megengedett elemi utasítás az értékadás. Ez két oldalból áll egyenlőségjellel elválasztva, a végén pontosvesszővel lezárva. A baloldalon csak változó azonosító szerepelhet, a jobb oldalon pedig operandus, ami vagy változó, vagy konstans lehet. Konstansból kétfélét engedünk meg: egész számot vagy sztringet.

Az *if* utasítás felépítése a következő: az *if* kulcsszót követően egy nyitó és egy csukó zárójel között egy logikai kifejezés szerepel, amely két operandusból és egy logikai operátorból áll. Az operandusok az előbbieknek megfelelően változók vagy konstansok lehetnek, az operátor pedig kisebb vagy nagyobb jel. Ezt követően nyitó és csukó kapcsos

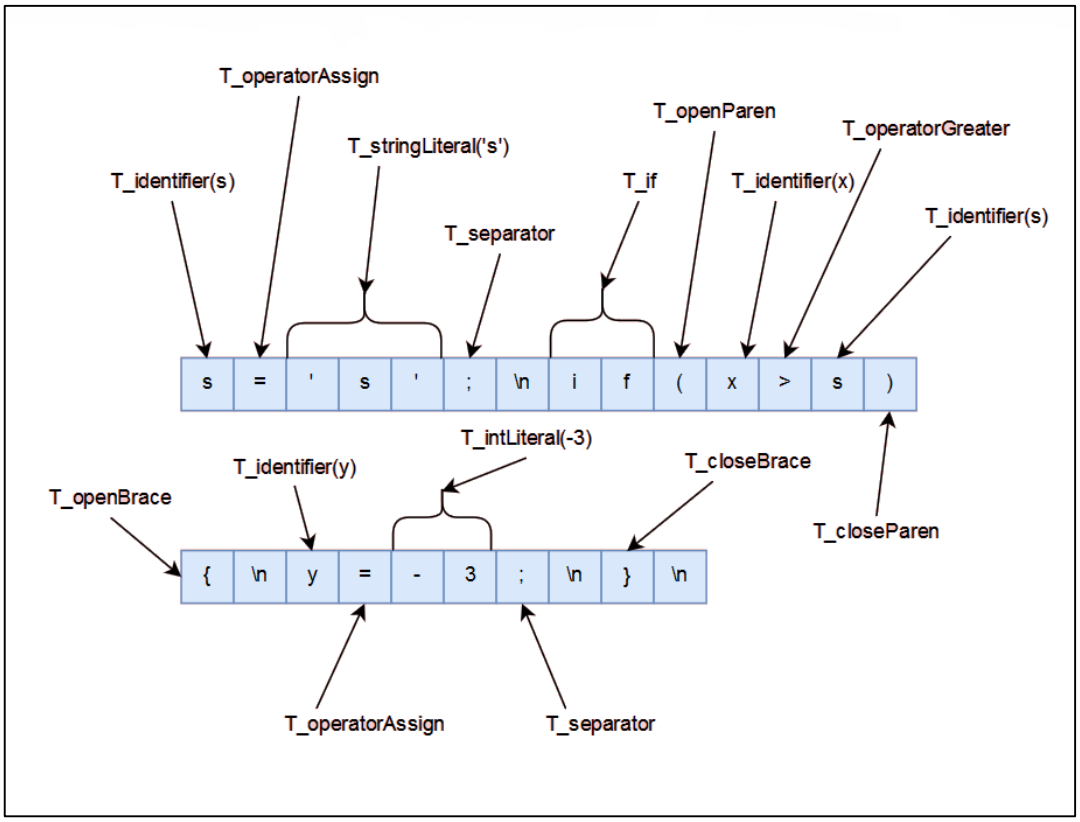
[2] https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

zárójelek között az utasítás törzse szerepel, amely tetszőleges számú utasítást tartalmazhat.

Vizsgáljuk az alábbi forráskódot:

```
s = ’s’;
if (x > s) {
     y = -3;
}
```

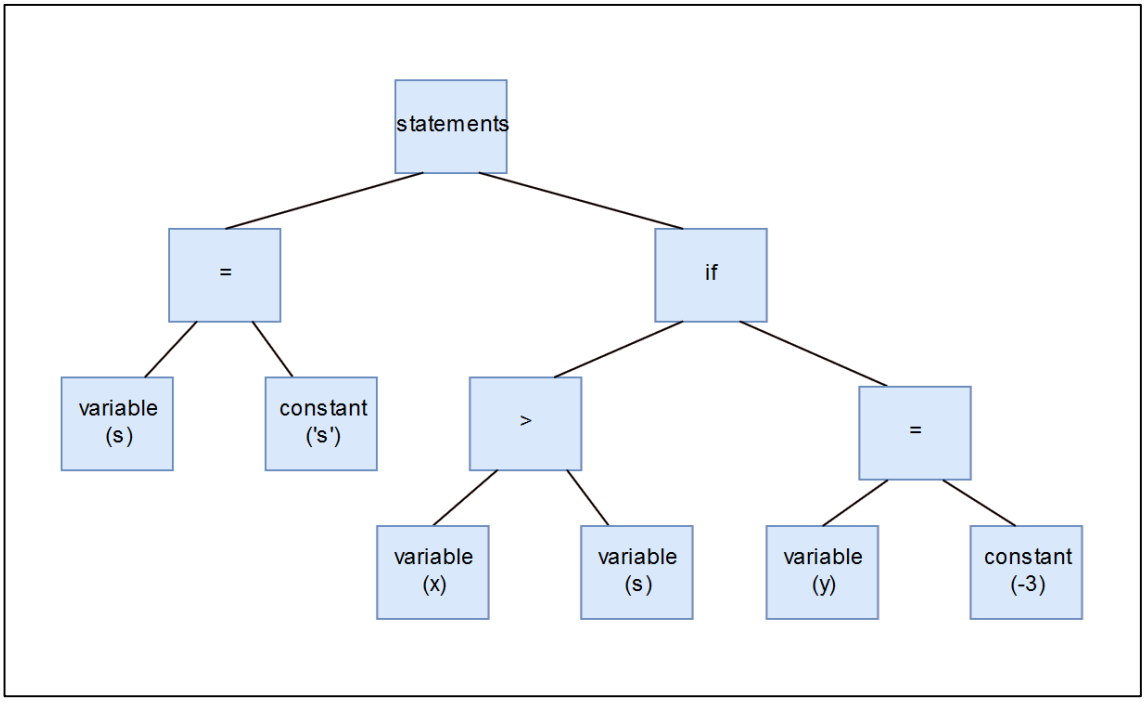
Az első lépés a lexikai elemzés. A forráskódot ASCII karakterekből álló szövegként értelmezve karakterenként vizsgálva feldolgozzuk, és a lexer szabályokra illeszkedő tokeneket keresünk. A whitespace karaktereket (szóközök és tabulátorok) az egyszerűség kedvéért nem tüntetem fel, mert a folyamatot nem befolyásolják. Az elemzés eredményét a ***2. ábra*** szemlélteti.



2. ábra: A lexikai elemzés eredménye.

A feliratok a tokenek típusát, a zárójelben szereplő értékek pedig azok attribútumait (jelen esetben az értékét) jelentik.

A második lépés a szintaktikai elemzés, amely során a parszer szabályok alapján a forráskódot reprezentáló AST-t építjük fel. Először a levezetési fát (parse tree) készítjük el, majd a lényegtelen szintaktikai részletek elhagyásával (pl. zárójelek, pontosvesszők, információt nem hordozó csomópontok) megkapjuk az absztrakt szintaxisfát, melyet a ***3. ábra*** szemléltet.



3. ábra: A példa kódhoz tartozó absztrakt szintaxisfa.

Ha a programkódunk szintaktikai hibás lenne, azt a lexikai és szintaktikai elemzés során vennénk észre fel nem ismert tokenek, illetve hibás szerkezetű AST formájában.

Az utolsó lépés a szemantikai elemzés. Ekkor az AST-t bejárva ellenőrizzük, hogy a forráskód betartja-e nyelvünk szemantikai szabályait. Esetünkben például célszerű lehet a nyelvünket úgy értelmezni, hogy logikai operátorokat csak ugyanolyan típusú változókon vagy konstansokon értelmezünk. Ha például x egy korábban létrehozott egész szám típusú változó, akkor a szintaxisfa bejárása során szemantikai hibát találunk, ugyanis az if utasítás logikai feltételében a másik operandus egy sztring.

2.3 Legacy kód transzformációjának folyamata

A folyamat, mely során a procedurális C kódból objektumorientált kódot készítek, három fő lépésből áll.

1. A kiinduló kódbázist a korábban ismertetett több lépésből álló elemzés (lexikai, szintaktikai, szemantikai) során modellé kell alakítani, amely a program struktúráját, működését leírja.
2. Az így létrejött modell a procedurális kódot írja le, a cél azonban egy olyan modell létrehozása, ami egy objektumorientált programot reprezentál. Ezért az eredeti modell bejárásával létrehozok egy másik, szemantikailag egyező modellt. Ennek a modellnek ugyanazt a programot kell leírnia, mint amit az eredeti modell írt le, viszont már objektumorientált szerkezetben. Ezt a lépést – amikor egy modell transzformációjával egy másikat hozunk létre – a szaknyelv **model-to-model**, röviden **M2M** transzformációnak nevezi. A pontosság kedvéért megemlítendő, hogy nem egy transzformációról, hanem transzformációk sorozatáról van szó. Ezekre azonban tekinthetünk úgy, mintha egy nagy transzformáció ment volna végbe.
3. Ezután már készen áll a végeredményt reprezentáló megfelelő modell. Az utolsó lépés a kódgenerálás, ez hozza létre a modellezett alkalmazás kódját. Ezt a folyamatot **model-to-text,** röviden **M2T** transzformációnak nevezzük.

3 Kapcsolódó irodalom

Az objektumorientált szemlélet elterjedésekor – látva annak hasznosságát – felmerült az igény, hogy a korábbi procedurális programokat objektumorientált programokra lehessen átalakítani. Ebben a fejezetben néhány korábbi megoldási javaslatot vizsgálok meg. Az idézett munkákban közös, hogy mindegyik az adott programot szintaktikailag reprezentáló AST-ből indul ki.

A [4] által bemutatott megközelítés első lépésként egy olyan XML állományt épít az AST-ből, amely egy programozási nyelv-független, procedurális alkalmazás szintaktikai sajátosságait írja le. A második lépés egy inkrementális algoritmus használata, ami a programot több, kisebb méretű összetevőre (cluster) bontja. Ezeket vizsgálva több kisebb, úgynevezett Domain Object Modelt (DOM) épít. Végül ezeket „összerakva” alakul ki az alkalmazás végső modellje. A megoldás végterméke tehát egy modell, ami a modernizált alkalmazást írja le.

[5] a megoldás első lépéseként az AST-t bejárva kigyűjt minden olyan információt, amire szüksége van. Ezekből entitásokat épít, amelyeken a második lépésben dolgozni tud. Ezután következik az osztályok kialakítása. Erre a célra a szerzők egy heurisztikus particionáló algoritmust dolgoztak ki, amely az entitások között úgynevezett távolságot (distance) definiál. Az algoritmus futása során az entitások vizsgálatán keresztül ciklikusan korrigálja a távolság értékeket, és azokból az entitásokból, amelyek távolsága egymáshoz képest nagyon kicsi, clustereket épít. Az algoritmus tehát az összes entitást tartalmazó kiinduló halmazt entitásokat tartalmazó részhalmazokra (clusterre) bontja. Ezek a részhalmazok adják az objektumorientált alkalmazás struktúráját.

[6] a parszolás elvégzése és az AST generálása után olyan keretrendszert készít, amely automatizált módon képes procedurális programot egy neki megfelelő modernizált programra transzformálni. Az itt bemutatott módszer elsősorban hibás tervezés nyomait keresi (pl. redundáns vagy duplikált megoldások), és ezeket kijavítva szervezi osztályokba és metódusokba az eredeti alkalmazást. Ehhez különböző adatstruktúrákat – pl. adatfolyam-gráfot (data flow graph – DFG), vagy állapotgépet – épít a programból, és ezek segítségével vizsgálja a kódot.

[7] a procedurális kód objektum-orientálttá való transzformálása során a minél magasabb szintű újrahasználhatóság elérésére törekszik. Ehhez feltételezi, hogy az alkalmazás követelmény analízise (requirement analysis) rendelkezésre áll, ugyanis ezt az információt felhasználja a transzformáció során. A folyamat különböző magasabb absztrakciós szintű elemeket készít a transzformáció segítésére (pl. ER[3] diagramm, dataflow[4] diagram), majd könnyen újra hasznosítható komponensekre próbálja bontani a programot, az objektum-orientált paradigmákat szem előtt tartva.

[5] és [4] közötti hasonlóság, hogy mindkettő automatizált módon, particionáló algoritmus használatával alakítja ki a modernizált szerkezetet. [5] algoritmusa a heurisztikus algoritmus miatt azonban nagyméretű alkalmazások esetén jobb számítási komplexitással képes működni, így gyorsabb a transzformáció. Az alapvető különbség, hogy [5] nem modellekkel dolgozik, ezért nem használhatja ki a különböző modell-vezérelt eszközök által nyújtott lehetőségeket (pl. kódgenerálás, automatikus transzformáció stb). [6] [4]-hez hasonlóan szintén modellt (objektummodellt) épít, és a megoldása célnyelv független, viszont nem tisztán objektum-orientált rendszert állít elő, ugyanis állapot-és eseményvezérelt elemeket ötvöz objektum-orientált szemlélettel. [7] megoldása jól újrahasznosítható, objektum-orientált rendszert épít, azonban a transzformáció nem teljesen automatizálható, ugyanis a különböző adatstruktúrák felépítéséhez olyan információra van szükség a programmal kapcsolatban, amelyhez emberi segítség szükséges.

Az idézett munkák írásakor a modell-vezérelt módszerek kevésbé voltak elterjedtek, és a modell-vezérelt eszközök terén is nagy fejlődést tapasztalhattunk az utóbbi években. Ezért aktuális feladatnak tekintem a problémakör megoldásának modell-vezérelt eszközökkel történő vizsgálatát.

[3] https://www.smartdraw.com/entity-relationship-diagram/

[4] https://en.wikipedia.org/wiki/Data_flow_diagram

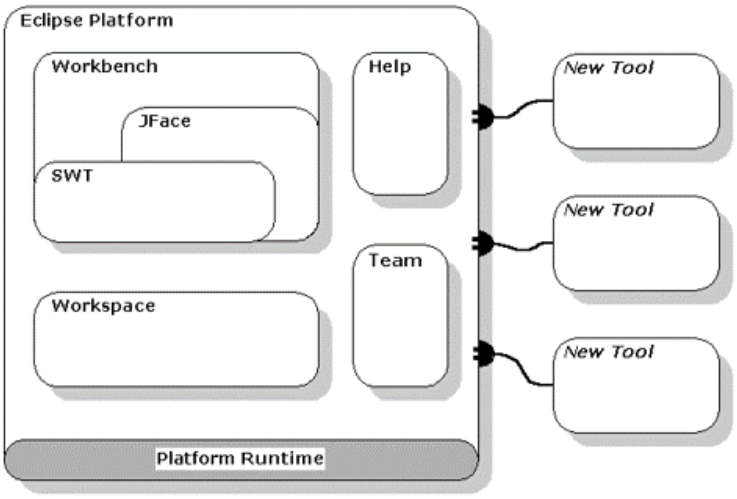
4 Kapcsolódó technológiák

A kidolgozott módszer prototípusának elkészítése során számos technológiát megvizsgáltam és felhasználtam. Ebben a fejezetben ezeket mutatom be röviden.

4.1 Eclipse és OSGi

Az Eclipse[5] [8] kezdetben egy integrált fejlesztőkörnyezet (IDE – Integrated Development Environment) volt. Az idő során azonban fejlődésen ment keresztül, sok komponense hasznosnak bizonyult más jellegű alkalmazások fejlesztéséhez is. Így alkalmazásplatform lett belőle, amely pluginek használatát támogatja. Ez azt jelenti, hogy az adott szoftver az Eclipse-en belül tud futni, kihasználva a platform nyújtotta szolgáltatásokat. Az Eclipse így rendkívül rugalmas és testreszabható, különböző disztribúciói eltérő plugineket tartalmazhatnak. Ráadásul saját pluginek is készíthetők és használhatók. A procedurális kód transzformációjának implementációja is Eclipse pluginként fog működni.

Az Eclipse felépítését a ***4. ábra*** szemlélteti.



4. ábra: Az eclipse moduláris felépítése.

[5] https://www.eclipse.org/

Az Eclipse az OSGi-re épül, amely egy nyílt szabvány moduláris rendszerek integrációjához. A pluginek bővíthetik más pluginek funkcióit, illetve függhetnek is egymástól. Ezeket a viszonyokat definiálni kell. Az OSGi szabvány ezt egy manifeszt fájlban (MANIFEST.MF) teszi meg. A pluginek kiterjesztési pontokat is megadhatnak, amelyek funkciókat szolgáltatnak más pluginek számára. Ezeket a plugin.xml állományban kell megadni.

4.2 Eclipse Modeling Framework (EMF)

Az EMF[6] [9] az Eclipse metamodellező keretrendszere. A metamodell a modell modelljét jelenti, amelynek egy konkrét példánya pedig egy modell lesz. A kód transzformációja során építendő programozásinyelv-független objektumorientált modell egy EMF segítségével készített metamodell példánya lesz.

Az EMF segítségével tehát metamodelleket tudunk létre hozni. Ha ezzel megvagyunk, akkor abból az EMF közreműködésével kódot kell generálnunk, hogy szoftverünkben használni tudjuk azt. A kódgenerálás paraméterezhető is, megadható pl. a generált kód Java-verziója. Ezek a paraméterek a metamodelltől függetlenül, egy másik modellben, a „genmodel” kiterjesztésű generátormodellben tárolódnak.

Ha elkészültünk a generálás paraméterezésével is, akkor generálhatók a metamodellünk osztályaihoz tartozó modellosztályok. Minden osztályhoz tartozik egy interfész, és az azt implementáló Java osztály. Ezeknek az osztályoknak a példányai lesznek a metamodellünk példányai, tehát egy példány egy konkrét modellt fog jelenteni.

Előfordulhat, hogy egy generált osztályban kézzel módosítani szeretnénk valamit. Ennek tipikus esete lehet például metódusok definiálása. A metamodellen metódus szignatúrát megadhatunk ugyan, de annak törzsét kézzel kell megírni. Ez viszont felveti azt a problémát, hogy ha az osztályainkat újra generáljuk, akkor a módosításaink elvesznének. Ennek kiküszöbölésére az EMF annotációkkal jelöli meg a kódot. A generált (tehát újra generálandó) kódra alapértelmezetten egy kommenten belüli „@generated” jelölés kerül. Ha azt szeretnénk, hogy egy általunk írt kódrészlet megmaradjon, akkor ezt törölnünk vagy módosítanunk kell. Konvencionálisa a „@generated NOT” jelölést használjuk.

[6] https://www.eclipse.org/modeling/emf/

4.3 C Development Tooling (CDT)

A CDT[7] az Eclipse platformhoz szolgáltat funkcionalitásokat (plugineket), amelyek C és C++ projektek fejlesztéséhez biztosítanak integrált fejlesztőkörnyezetet. Habár önmagában nem tartalmaz C/C++ fordítót, lehetőséget biztosít azok Eclipse környezetbe való egyszerű integrációjához. A CDT a megszokott funkciókat biztosítja Eclipse környezetben C és C++ fejlesztés számára: pl. projektek létrehozása, kezelése, keresés a forráskódban, content assist támogatás. Nagy előnye, hogy Eclipse plugineket használhatunk C/C++ fejlesztés közben.

A CDT a korábbiakban bemutatott parszolás folyamatnak megfelelően fel tudja dolgozni az adott (C vagy C++) projektet, majd abból AST-t és szemantikus modellt tud építeni, melyek leírják a projekt szerkezetét és jellemzőit. Ez azért fontos, mert egy API-n keresztül el tudjuk érni az AST-t, szemantikus modellt és indexet. Így programozottan, futásidőben felhasználhatjuk és manipulálhatjuk a projektünk szinte minden tulajdonságát. Az AST és a szemantikus modell bejárásán keresztül elérhetjük a forrásfájlokat, az azokban definiált függvényeket, változókat, utasításokat. Ez lesz az a kiinduló modell, amely alapján a prototípus objektumorientált modellt épít.

4.4 Java Development Tools (JDT)

A JDT[8] a CDT-vel analóg módon plugineket szolgáltat Eclipse platformra, melyek egy Java projektek fejlesztésére alkalmas integrált fejlesztői környezetet valósítanak meg.

A készítők a JDT pluginjeit a következő kategóriákba sorolják:

1. A JDT APT pluginek annotációk feldolgozásához szükséges funkciókat biztosítanak Java 5-ös vagy későbbi verziójú projektek számára.
2. A JDT Debug a Java projektek hibakeresés funkcióját valósítja meg.
3. A JDT UI különböző refaktorálást elősegítő és áttekintő funkciókat biztosít grafikus felületek segítségével. Ilyen például maga a Package Explorer, a típus

[7] https://www.eclipse.org/cdt/

[8] https://www.eclipse.org/jdt/overview.php

hierarchia megtekintése, vagy elemek átnevezése (rename) az egész projektben található összes hivatkozás lecserélésével együtt.

4. A JDT Text pluginek a Java forráskód szerkesztőjét valósítják meg.
5. A JDT Core biztosítja a Java fordítót az Eclipse platformra. Ezen felül, ahogy a CDT C és C++ projektekre, úgy a JDT Core is szolgáltat API-t Java projektek programozottan történő elemzésére, manipulálására, létrehozására. Ezt használom fel arra a célra, hogy egy Eclipse Java projektbe szervezzem a transzformált alkalmazást.

4.5 Xtext

Az Xtext[9] [10] egy olyan keretrendszer, melynek segítségével szakterületi nyelveket hozhatunk létre. A szakterületi nyelv (Domain Specific Language – DSL) egy adott problémakör megoldására specializált nyelv.

Az Xtext egy, az ANTLR-en [11] alapuló parser generátort valósít meg. A parser generátorok egy nyelvtant várnak, amelyek megadják a lexer által generálandó tokenek leírását reguláris kifejezésekkel, illetve a nyelvünk szintaktikai szabályait. Ez alapján a generátor elő tudja állítani az adott nyelv lexerét és parserét. Ezt a nyelvtant, amely leírja a nyelvünket, metanyelvnek nevezzük. Tulajdonképpen a metanyelv is szakterületi nyelvnek tekinthető, amellyel nyelveket lehet leírni.

Az Xtext nem csupán egy egyszerű parser generátor, azon kívül többletfunkciókat is biztosít. Egyrészt, az Xtext nem szintaxisfát készít, hanem egyből egy EMF metamodell példányát adja vissza. Másrészt az elkészülő parsert Eclipse pluginokként generálja, és magán az elemzőn kívül mást is készít. Ilyen például a kódgenerátor, a formázó stb.

A transzformáció során az Xtext által nyújtott funkciók felhasználhatók lehetnek további vizsgálatok elvégzésére. Erről részletesebben a ***7.2*** fejezetben írok.

[9] https://www.eclipse.org/Xtext/

4.6 Viatra

A Viatra[10] [12] egy nyílt forráskódú modell-transzformációs Eclipse keretrendszer, ami EMF modellekkel dolgozik. Alapvetően két kategóriára bonthatók képességei:

1. Úgynevezett modell lekérdezéseket (model query) hozhatunk létre, amelyek a modell azon elemeinek halmazát adják vissza, melyek teljesítik a lekérdezésben megadott összes feltételt (pattern). Ennek legegyszerűbb esete lehet például, hogy olyan elemeket keresünk, amelyek valamely tulajdonsága valamilyen feltételt kielégít. Ezen felül automatikus validációkat is készíthetünk. Ezeket különböző EMF modell példányokon kiértékelhetjük: ha valahol nem teljesülnek a megadott feltételek, akkor arról jelzést – a validációban beállított értéknek megfelelően warningot vagy errort – kapunk az Eclipse-ben.
2. A Viatra segítségével M2M transzformációkat is definiálhatunk. Ennek alapvetően két típusa van: a batch és az esemény-vezérelt (event driven) transzformáció. Mindkét esetben deklaratívan megadhatjuk, hogy mi legyen a transzformáció eredménye, maga az átalakítás folyamata viszont automatikusan történik.

A procedurális kód transzformációja során érintett alapproblémák komplexitása egyelőre ugyan nem teszi szükségessé a Viatra használatát, azonban később olyan fejlesztéseket lehetne vele elvégezni, amelyek hatékonyabbá tehetnék a folyamatot (pl. modellek validációja, automatikus transzformáció).

[10] https://www.eclipse.org/viatra/

5 A kidolgozott módszer bemutatása

Ebben a fejezetben a procedurális kód objektum-orientált kódra való transzformálásának általam kidolgozott megközelítését mutatom be. Röviden kitérek a forráskód elemzésére és feldolgozására, majd részletesen ismertetem az objektum-orientált modell felépítésének lépéseit. A folyamat utolsó lépése a kódgenerálás, ami az alkalmazás tényleges kódját hozza létre. Ezt tetszőleges, a modellünkkel kompatibilis kódgenerálási technológiával (pl. EMF és Xtend) megvalósíthatjuk. Mivel ez technológiai részletkérdés, a továbbiakban erről nem lesz szó.

Az itt bemutatott módszer nem függ a transzformáció célnyelvétől, ezért minden nyelv esetén ugyanolyan elvek mentén, ugyanazokat az algoritmusokat kell alkalmazni.

5.1 C projekt parszolása

A transzformáció bemenete egy C projekt, ami tetszőleges számú forrásfájlból állhat. Az első lépés a kód feldolgozása, az azt reprezentáló AST felépítése. Ezt a korábbiakban ismertetett parszolás folyamata végzi el.

Az absztrakt szintaxisfa felépítése után annak bejárásával nyerhető ki a C kódról minden olyan információ, ami a modell felépítéséhez szükséges. Ezen információk alatt a projekt valamennyi forrásállományának tartalmát értjük, mint például a függvénydefiníciók, azok törzsei, a létrehozott struktúrák és tagváltozóik, a globális változók, stb. A továbbiakban a függvényekre, globális változókra és struktúrákra *entitásként* hivatkozok.

5.2 A modell felépítése

Ahhoz, hogy a procedurális kódbázist jól karbantartható, áttekinthető és könnyen bővíthető kóddá alakítsuk, nem elég ugyanazt a szerkezetet egy objektum-orientált nyelven leírni, mert azzal a szükséges refaktorálás csupán elenyésző részét végeznénk el Ennek érdekében kidolgozandó, hogyan épüljön fel a modernizált alkalmazás struktúrája, vagyis hogy az eredeti kódot milyen elvek mentén és milyen módszerek alkalmazásával szervezzük osztályokba.

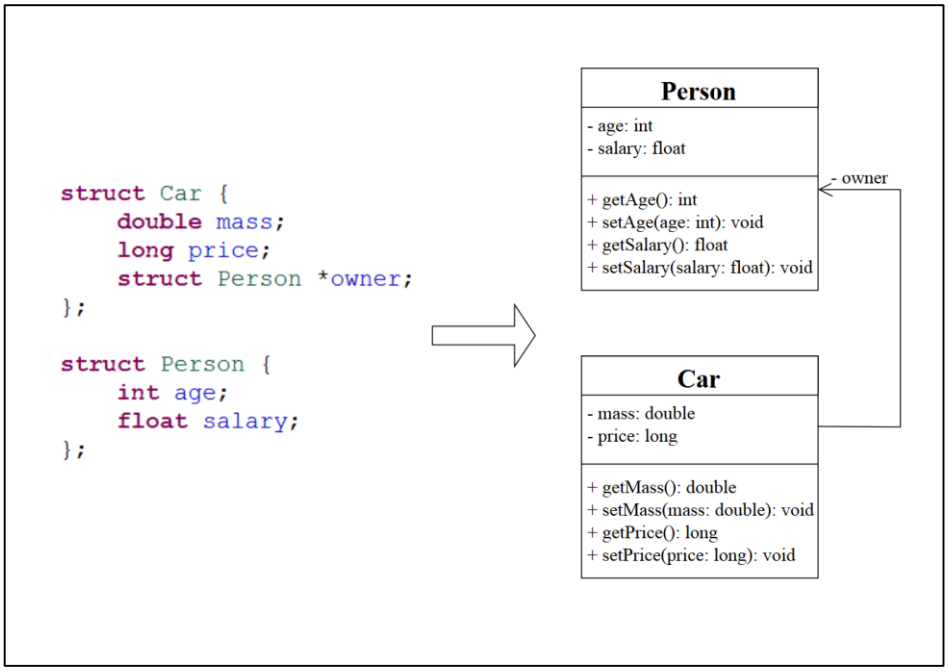
A továbbiakban bemutatok egy algoritmust lépésenként, amely az osztályhierarchia kialakítását végzi el különböző szempontok vizsgálata alapján. Az algoritmus több

lépésből álló vizsgálatot végez, és ha egy vizsgálat végén egy entitást már osztályhoz rendelt, akkor az átkerül a befejezett entitások halmazába. Így a következő vizsgálatot már csak azokon az entitásokon végzi el, amelyek még nem kerültek osztályba. A vizsgálatok tehát egy folyamatosan szűkülő adathalmazon dolgoznak. Fontos megemlíteni, hogy előfordulhat, hogy egy entitást több elemzés is besorolhatna, akár különböző osztályokba is. Ezért a vizsgálatokat eredményük prioritásának csökkenő sorrendjében végzem.

Minden olyan entitást, amit nem sikerül a vizsgálatok során osztályhoz rendelni, egy globális osztályba szervezek. Ide kerül természetesen az alkalmazás belépési pontja is.

5.2.1 Struktúrák leképzése

Az osztályok kialakításának első lépése, hogy minden, a C kódban definiált struktúrát egy osztályra képzek le az objektum-orientált kódban. Az osztály tartalmazza a struktúra összes tagját privát tagváltozóként, amelyek értékét az adott osztályban felvett publikus getter és setter metódusokon keresztül érhetjük el, illetve állíthatjuk más értékre. Ezt azért tartom célszerűnek, mert azzal, hogy nem engedjük a tagváltozók közvetlen elérését, objektumaink belső állapotait konzisztensen tudjuk tartani. Ez azt jelenti, hogy el tudjuk kerülni a logikailag nem megengedett állapotokat. A struktúrák osztályba való leképzését az ***5. ábra*** szemlélteti egy példán keresztül.



5. ábra: Struktúrák leképzése osztályokká.

A példán látható struktúrák tagjainak típusai tájékoztató jellegűek, csak a leképzés szemléltetését szolgálják. A C típusainak kezeléséről később, a ***6.4.*** fejezetben írok.

5.2.2 Paraméterek és visszatérési érték vizsgálata

Az első lépésben az entitások közül a struktúrákból készítettem osztályokat, a függvényekkel és globális változókkal azonban még nem foglalkoztam. A második lépés célja a függvények osztályokhoz rendelése.

Ennek érdekében a függvények paramétereit és visszatérési értékeit vizsgálom struktúra típusok szempontjából.

Egy függvény struktúra típusú paramétere azt jelzi, hogy a függvény valamire használja az adott típust. Mivel a függvény valamilyen szempontból felhasználja ezt a típust, érdemes lehet metódusként az ehhez a típushoz tartozó osztályba szervezni, hiszen az adott típussal dolgozó művelet logikailag összetartozhat az adatokkal.

Ha tehát egy függvény paraméterlistája struktúra típusú változót vár, akkor ez a függvény potenciálisan hozzáadható az adott struktúrához tartozó osztályhoz. Ezt viszont csak akkor lehet biztosan kijelenteni pusztán paraméterlista vizsgálattal, ha csak egyfajta ilyen változó szerepel a listán. Ha több is szerepelne, akkor nem lenne egyértelmű, hogy melyikbe célszerűbb szervezni a függvényt[11].

A struktúra típusú visszatérési érték arra utal, hogy a vizsgált függvény az adott típusból vagy frissíti egy változó értékét, vagy egy újat hoz létre és ad vissza. Mindkét esetben célszerű az ehhez a típushoz tartozó osztályba szervezni metódusként a vizsgált függvényt, ugyanis az objektum-orientált szemlélet szerint egy típus adatai és a típus állapotának megváltoztatásáért felelős műveletek egybetartoznak. Ezt nevezzük egységbezárásnak. A visszatérési érték ezért alapvetően erősebb jelnek tekinthető, mint a paraméter. Ha mindkét helyen van struktúra típus, akkor a visszatérési érték osztályába szervezem a függvényt.

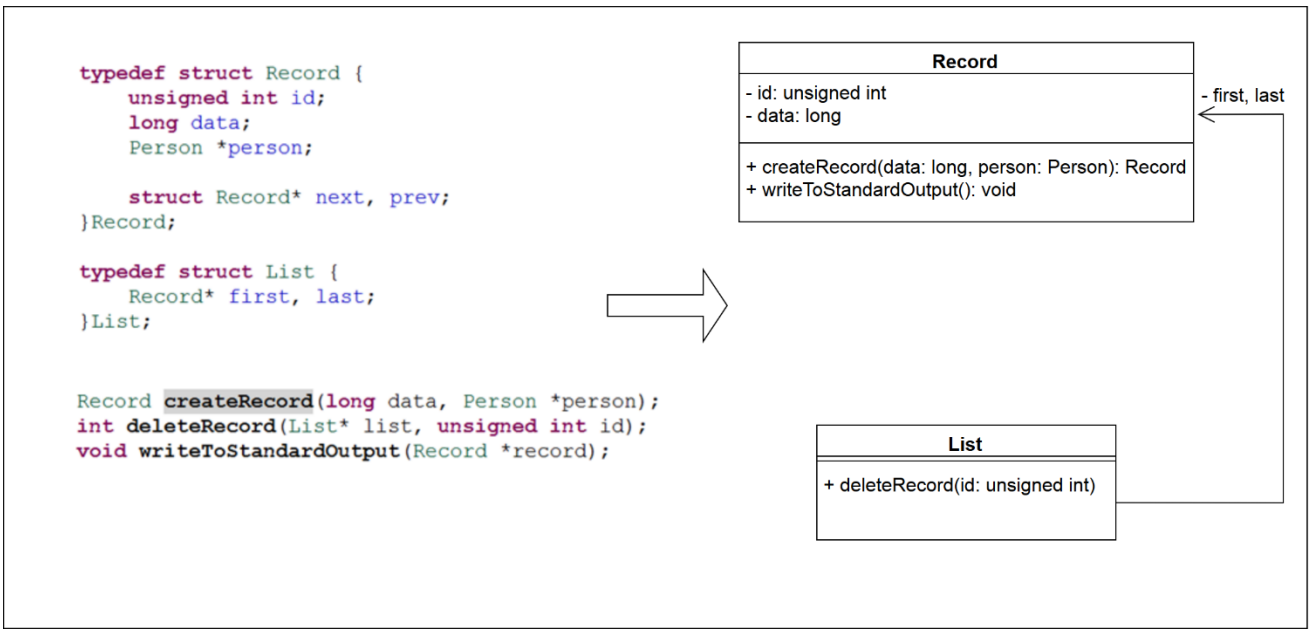
[11] Létezik olyan programozási konvenció, hogy a paraméterek fontosságuk típusában vannak felsorolva. Ez alapján ebben az esetben is lehetne dönteni, viszont ez egy erős feltételezés lenne, amit közel sem biztos, hogy betart a vizsgált alkalmazás.

Az előbbieknek megfelelően azt, hogy a visszatérési érték és paraméterlista vizsgálat eredményeinek különböző kombinációi esetén tud-e következtetni az algoritmus, és ha igen, akkor hogyan dönt, a ***2. táblázat*** foglalja össze.

2. táblázat: A paraméterlista és visszatérési érték vizsgálatának esetei.

	Paraméterlistában nincs struktúra típus	***Paraméterlistában egy struktúra típus van***	***Paraméterlistában több struktúra típus van***
Visszatérési érték nem struktúra típus	***Nem célszerű osztályba szervezni***	***Az adott struktúra típusba célszerű szervezni***	***Nem tud dönteni***
Visszatérési érték struktúra típus	***A visszatérési érték típusába célszerű szervezni***	***A visszatérési érték típusába célszerű szervezni***	***A visszatérési érték típusába célszerű szervezni***

Azokban az esetekben, amelyeknél sikerül azonosítani a célosztályt, az algoritmus el is végzi a metódus hozzáadását az osztályhoz. Ahol nem tud dönteni, vagy arra a következtetésre jut, hogy egyik meglévő osztályba se célszerű beszervezni az adott függvényt, ott nem csinál semmit, tehát a függvény további vizsgálat bemenete lehet.



6. ábra: Paraméterek és visszatérési érték vizsgálata.

A ***6. ábra*** egy egyszerű példán keresztül mutatja be a paraméterek és visszatérési érték vizsgálat mögötti ötletet. Adott egy láncolt lista és néhány, a rekordelemeken, illetve a listán műveletet végző függvény. Az első függvény egy konstruktorfüggvény lehet, a második töröl egy elemet a listából, a harmadik pedig kiír egy listaelemet a sztenderd

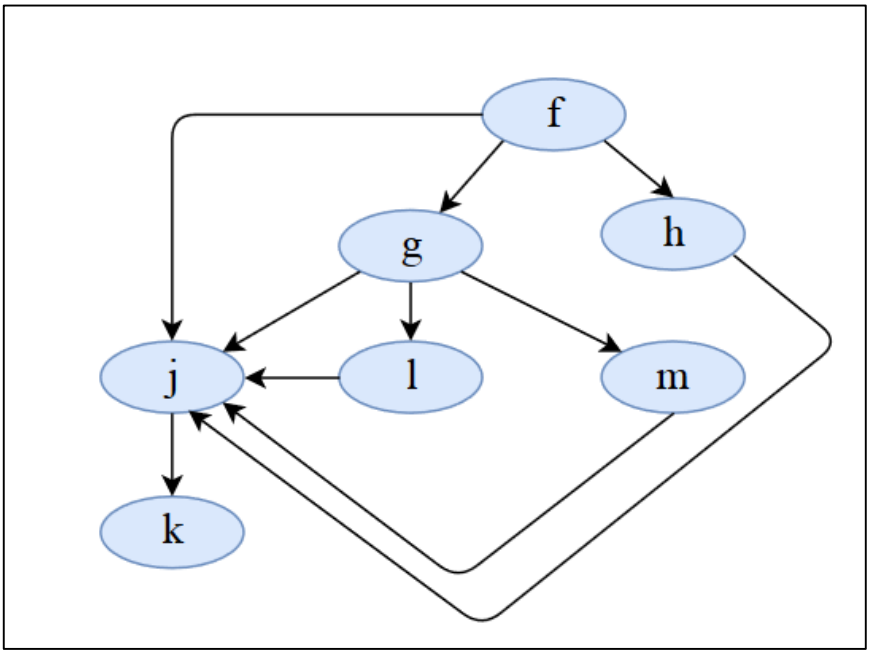
kimenetre. Az első a visszatérési érték miatt a Record osztályba szerveződik, a második és harmadik pedig a List és Record paramétertípusok miatt az adott osztályba kerülnek.

5.2.3 Hívási láncok keresése

A harmadik lépés bemenete minden olyan függvény, amely az előző lépésben nem került osztályba. A vizsgálat célja további, bizonyos szempontból összetartozó függvények osztályba szervezése.

Ennek érdekében egy vezérlésfolyam-gráfot[12] (control flow graph – CFG) építek, amely a program vezérlésének „útját” jelöli. Ez egy irányított gráf, amelynek csomópontjai a bemeneti függvények, egy él pedig valamely ***f*** csomópontból ***g*** csomópontba mutat pontosan akkor, ha ***f*** futása során legalább egyszer meghívja ***g***-t.

A ***7. ábra*** egy ilyen CFG-t szemléltet.



7. ábra: Egy példa CFG.

A CFG tehát minden bemeneti függvényre tartalmazza, hogy az mely függvényeket hívja meg futása során. A vizsgálat során hívási láncokat keresek, és bizonyos szempontok

[12] A szakirodalomban a vezérlésfolyam-gráf némileg [más]t jelent, mint dolgozatomban. Azért használom mégis ezt az elnevezést, mert jól kifejezi a gráf jelenlegi funkcióját.

alapján a gráf egy-egy összefüggő részgráfjából (hívási láncból) készítek egy-egy osztályt.

Jelölje egy ***f*** csomópontra valamely más ***g*** csomópontokból ***f*** felé mutató élek számát f_{in}, Ezzel a terminológiával élve az algoritmus olyan csomópontokat keres, amelyekre f_{in} alacsony. Ez lesz az algoritmus kiinduló (gyökér) csomópontja. Ha az algoritmus talál egy ilyen ***f*** függvényt, akkor ***f*** gyerekeit is megvizsgálja bemenő fokszám szempontjából, majd azok gyerekeit, stb. Egy csomópont gyerekeit akkor nem vizsgálja tovább, ha nincs neki, vagy ha az adott csomópont f_{in} értéke túl magas. Ha a hívási lánc függvényeinek száma meghalad egy minimális értéket, akkor ezek a függvények kerülnek egy osztályba. Ezt a minimális méretet jelölje s_{min}.

Az algoritmus tehát arra törekszik, hogy olyan leghosszabb hívási láncokat keressen, amelyek mindegyik tagját viszonylag kevés függvény hívja az alkalmazásban. Ez azért eredményez előnyös osztályszerkezetet, mert egyrészt a hívási lánc miatt ezek a függvények logikailag egybetartozhatnak, másrészt az alacsony f_{in} értékek biztosítják az osztály laza csatolását az alkalmazás többi része felé, így azok kevésbé fognak függeni az osztálytól. A minimális osztályméret biztosítja, hogy ne alakuljon ki sok kicsi, aránylag kevés függvényt tartalmazó osztály, ami „szennyezné” az alkalmazás struktúráját.

Azt kell még megadnunk, hogy mi számít elég alacsony határértéknek f_{in} esetén, illetve, hogy mennyi legyen a minimális osztályméret. Ezeket az értékeket a programban található összes függvény számától teszem függővé. Jelölje ezt a számot **N.** Ekkor az f_{in} értékek maximálisan megengedett értéke $maxf_{in} = \lceil 0{,}2 * \mathbf{N} \rceil$. Egy f_{in} tehát határértéken belül van, ha $f_{in} \leq maxf_{in}$, szemléletesen ha nem haladja meg a függvények számának 20%-át (felfele kerekítve). Hasonlóan, a minimális osztályméret $s_{min} = \lceil 0{,}1 * \mathbf{N} \rceil + 1$. A +1 azt a célt szolgálja, hogy nagyon kevés hívással rendelkező alkalmazások esetén legalább két metódus kerüljön egy osztályba.

Ezeket a határértékeket heurisztika alapján állapítottam meg, mert „elég jónak" tűnnek. Természetesen ezt további kutatásokkal később még finomítani lehet.

Az algoritmusom pontos működését a következő pszeudokód szemlélteti, ahol V(CFG) a csúcsok halmazát, E(CFG) az élek halmazát, $|H|$ egy H halmaz elemszámát, {} pedig az üres halmazt jelenti.

Algoritmus *hívási láncok vizsgálatára*

***Input**: Egy CFG=(V(CFG), E(CFG)) vezérlésfolyam-gráf*

***Output**: C = {c_1, c_2, …, c_n}, a függvények osztályainak halmaza*

```
1   N := FüggvényekSzáma()
2   max_fin := Maxfin(N)
3   smin := MinimálisOsztályméret(N)
4   F :=V(CFG) //A további vizsgálandó függvények halmaza
5   For ∀f ∈ F do
6   |     c := Vizsgál(f, {})
7   |     if |c| ≥ smin then
8   |     |     Hozzáad(C, c)
9   |     end
10  |     else do
11  |     |     For ∀g ∈ c do
12  |     |     |     Hozzáad(F, g)
13  |     |     end
14  |     end
15  end
16  return C
```

Vizsgál *eljárás*

***Input:** f ∈ V(CFG) függvény*

***Input:** c, a vizsgált lánc függvényeinek bővítendő halmaza*

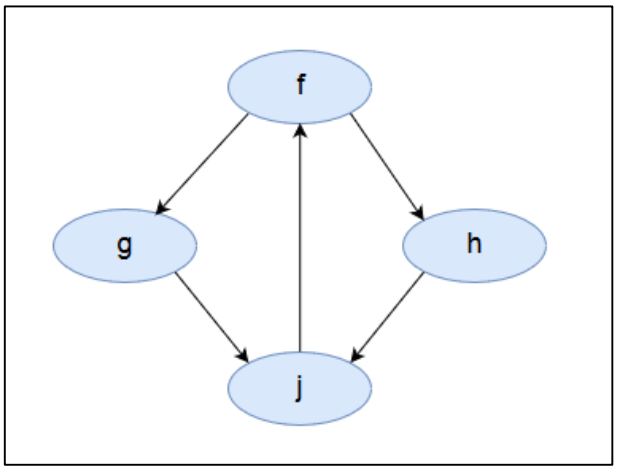
***Output:** c={f_1, f_2, …, f_k} a lánc osztályba szervezett függvényeinek halmaza*

```
1   Kivesz(F, f)
2   fin := BejövőÉlekSzáma(f)
3   if fin > max_fin then
4   |     return {}
5   end
6   Hozzáad(c, f)
7   For ∀g ∈ Leszármazottak(f) do
8   |     if g ∈ F then
9   |     |     c = Vizsgál(g, c)
10  |     end
11  end
12  return c
```

Egy csomópont esetleg több, különböző osztályhoz is hozzáadható lehet annak függvényében, hogy milyen sorrendben járjuk be a csomópontokat. Az algoritmus mohó módon működik, mert ha egy adott csomóponthoz talál osztályba szervezési lehetőséget, akkor azt azonnal megpróbálja végrehajtani. Ennek előnye, hogy ha létezik ilyen, akkor mindig meg fog találni egy lehetőséget. Hátránya azonban, hogy nincs garancia arra, hogy a legoptimálisabb eredményt kapjuk, ugyanis előfordulhat, hogy ha egy adott lépésben egy csomópontot nem adnánk hozzá az épülő osztályhoz, akkor azzal később egy jobb szerkezetet lehetne kialakítani.

Az algoritmus nem látogatja meg gyökérként azokat a csomópontokat, amiket egyszer leszármazottként már megvizsgált és azt tapasztalta, hogy a bejövő fokszáma túl nagy. Így fölösleges vizsgálatokat nem végez.

Előfordulhat az is, hogy a CFG-ben kör található. Ez jelen kontextusban azt is jelentheti, hogy a vizsgált alkalmazásnak létezik olyan vezérlési útja, ahol egy végtelen hívási láncba kerül. Egy ilyen esetet mutat be a ***8. ábra***, ahol két kör is található: az ***f-g-j-f***, illetve az ***f-h-j-f*** vezérlési úton.



8. ábra: Kör a CFG-ben.

Ez azért jelent problémát az algoritmus futása során, mert ha valóban végtelen hívásról van szó, akkor a függvények leszármazottjainak vizsgálata során maga az algoritmus is végtelen futásba fog kerülni. Az algoritmus ezért figyel arra, hogy a **Vizsgál** eljárás során egy csomópontot többször ne látogasson meg.

Lássuk most egy egyszerű példán az algoritmus futását. Tekintsük példaként a ***7. ábra*** vezérlés-folyam gráfját és az egyszerűség kedvéért feltételezzük, hogy a programban

található összes függvény bemenete lett a hívási láncok vizsgálatának. A csomópontok bejárása véletlen sorrendben történik, tegyük fel, hogy az algoritmus például ***f-g-h-j-l-m-k*** sorrendben látogatja meg a függvényeket gyökérként. Első lépésként az algoritmus kiszámítja a határértékeket:

- $max f_{in} = \lceil 0{,}2 * 7 \rceil = 2$
- $s_{min} = \lceil 0{,}1 * 7 \rceil + 1 = 2$

Ezután a csomópontok bejárása következik. Az algoritmus futását a ***3. táblázat*** mutatja be. Egy adott sor alatti vastag elválasztó vonal azt jelzi, hogy egy hívási lánc vizsgálatát az algoritmus befejezte. Ekkor a kialakítandó osztály cellában lévő ✓ azt jelenti, hogy az osztály mérete elegendő a létrehozáshoz, az ✘ pedig azt, hogy nem.

3. táblázat: Az algoritmus futása.

Lépészám	Vizsgált csomópont	Bejövő fokszám	Hátralévő csomópontok	Kialakítandó osztály
1.	*f*	0	*{g, h, j, l, m, k}*	*{f}*
2.	*g*	1	*{h, j, l, m, k}*	*{f, g}*
3.	*j*	5	*{h, l, m, k}*	*{f, g}*
4.	*l*	1	*{h, m, k}*	*{f, g, l}*
5.	*m*	1	*{h, k}*	*{f, g, l, m}*
6.	*h*	1	*{k}*	*{f, g, l, m, h}* ✓
7.	*k*	1	*{ }*	*{k}* ✘

Az első meglátogatott függvény ***f***. Azt látjuk, hogy $f_{in} = 0$, tehát megvizsgáljuk a gyerekeit is. Ekkor $g_{in} = 1$, tehát tovább mehetünk. A j csomópontra viszont $j_{in} = 5$, így ezt a részfát nem vizsgáljuk tovább. Megyünk tovább ***g*** következő leszármazottával, $l_{in} = 1$, tehát ***l*** gyerekei következnek. Mivel ***j***-t már egyszer megvizsgáltuk, és további gyerek nincs, ***m*** következik, aki szintén csak ***j***-t hívja. Így a ***g*** oldali részfát bejártuk, jöhet a ***h*** oldali. Itt $h_{in} = 1$, leszármazottja pedig megint csak ***j***, így ***f*** vizsgálata befejeződik. Az összegyűjtött függvények: ***f, g, l, m, h***. Mivel $5 > s_{min}$, ezért ebből alakítunk egy osztályt. Mivel j_{in} már egyszer túl nagynak bizonyult, így azt nem is látogatjuk meg többé. Az összes többi függvénnyel már végeztünk, így ***k*** következik, akinek viszont nincs több leszármazottja, így az algoritmus futása itt véget ér. A ***k*** csomópont osztályjelölt ugyan, de más függvény nem került mellé, így $1 < s_{min}$ miatt ebből nem lesz osztály. Egy osztályt sikerült tehát kialakítani: ***f, g, l, m, h*** tagokkal.

6 A kidolgozott módszer prototípusa

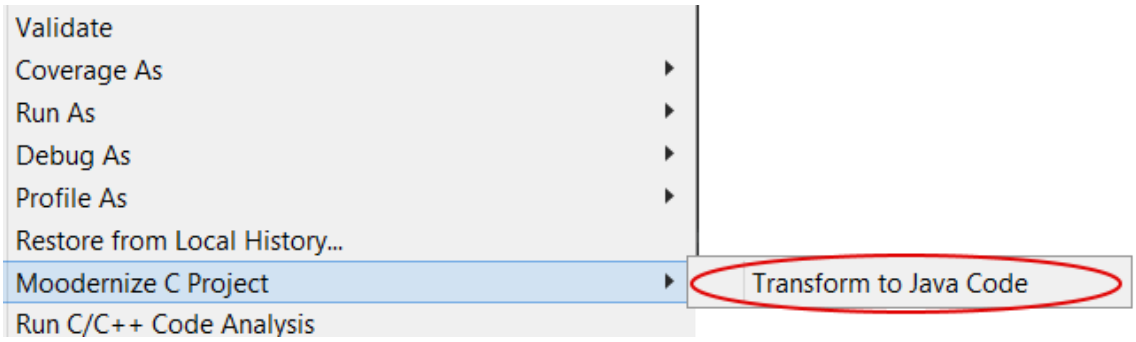
Ebben a fejezetben a korábbiakban leírt módszer egy prototípusát mutatom be. Ismertetem, hogy a transzformáció egyes lépései hogyan jelennek meg a megvalósításban. Végül egy átfogó példaalkalmazáson megnézzük, milyen Java kódot generál a transzformáció a C kódból.

A megvalósítás jelenlegi állapota egy prototípus, amely célja, hogy a kidolgozott elméleti módszer működőképességét szemléltesse a gyakorlatban.

6.1 A megvalósítás kontextusa

A prototípus Java nyelven, Eclipse pluginként lett megvalósítva. Futtatásához tehát Eclipse fejlesztőkörnyezetre, és JavaSE-1.8[13]-ra van szükség. Ezen felül a sikeres futáshoz elérhetőnek kell lennie a CDT és JDT technológiáknak is. Ezek az Eclipse beépített „Install New Software" funkciójával telepíthetők.

A transzformáció indítását a plugin telepítése után az Eclipse-ben C projektek kontextus menüjében érhetjük el. Ezt a ***9. ábra*** mutatja.



9. ábra: A transzformáció indítása.

Minden .c kiterjesztésű forrásfájlt az Eclipse projekten belül source mappákban kell elhelyezni, hogy a transzformáció felismerje őket forrásállományként. Ha nem ilyen

[13] http://www.oracle.com/technetwork/java/javase/downloads/index.html

mappába (hanem pl. sima könyvtárba) helyezzük őket, akkor a transzformátor figyelmen kívül fogja azokat hagyni. A prototípus a mappaszerkezetet egyelőre nem másolja le, mindent egy packagebe szervez.

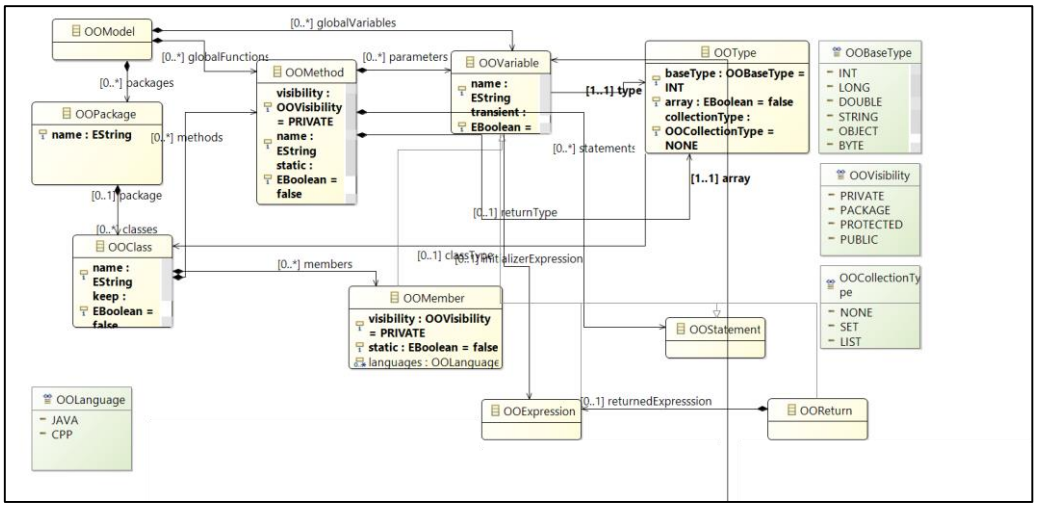
6.2 A forráskód elemzése és feldolgozása

A 2.2. fejezetben bemutatott parszolás folyamatát a C Development Tooling (CDT) segítségével valósítom meg. A folyamat elindításakor programozottan, Java kódból végrehajtom az elemzést, és felépítem a kiválasztott projekt forrásfájljait reprezentáló AST-ket.

Felmerülhet a kérdés, hogy mi lenne a helyzet akkor, ha nem C nyelvű programot kellene feldolgozni. A megoldás ugyanígy nézne ki: vagy valamely más, az adott nyelvhez készült keretrendszer segítségével, vagy ha nem találunk ilyet (vagy mindenképpen saját megoldást szeretnénk használni), akkor akár elkészíthetjük a saját megoldásunkat is, pl. Xtext segítségével.

6.3 OOGen – az alkalmazás modellje

Az objektum-orientált alkalmazás modellje egy EMF metamodell példánya, amely egy többnyire nyelv-független, objektum-orientált programot ír le. A modell tartalmazza az OO nyelvek közös vonásait, illetve ezen felül támogatást nyújt nyelvi különlegességek kezelésére is. Ezt a metamodellt nem én készítettem el, egy másik projekt keretein belül született. Én tehát csak használom ezt a modellt, illetve szükség esetén kiegészítem / módosítom azt.



10. ábra: Az OOGen metamodell részlete.

A ***10. ábra*** az OOGen egy részletét mutatja. A leírt alkalmazások modellje csomagokból (*package*) áll, amelyek osztályokból állnak (*OOClass*). Az osztályok metódusokat (*OOMethod*) és tagváltozókat (*OOMember*) tartalmaznak. A metódusok egyéb tulajdonságaikon felül – pl. láthatóság, statikusság – paraméterlistával, illetve visszatérési értékkel rendelkeznek. Törzseiket utasítások (*OOStatement*), illetve kifejezések (*OOExpression*) alkotják. A láthatóságok (*OOVisibility*) private, protected, public, és csomagszintű értékeket vehetnek fel. A használható, objektum-orientált típusokat az *OOBaseType* enumeráció definiálja. Az egyes kollekciók lista, vagy halmaz interfészűek lehetnek. Az esetleges nyelvi sajátosságok kezelésére be van vezetve az *OOLanguage* enumeráció, ami jelenleg Java és C++ nyelveket támogat.

6.4 Nyelvi elemek konverziója

Egy C forráskódú program konverziójakor felmerülnek különböző nyelvi sajátságok is, amikkel foglalkozni kell a transzformáció során. Ilyen például a C típusainak konverziója, a pointerek kérdése, vagy a külső könyvtárak kezelése is.

6.4.1 A C beépített típusai

A transzformáció során le kell képezni a C beépített típusait a célnyelv típusaira. Ez nyilvánvalóan csak nyelvfüggő módon történhet meg, így esetünkben meg kell adni minden C típusra, hogy azoknak mi legyen a Java megfelelőjük. A megfeleltetést a ***4. táblázat*** szemlélteti:

4. táblázat: A C típusainak konverziója Java típusokra.

C típus	*Java típus*
int	*int*
short	*int*
long	*long*
char	*byte*
unsigned int	*int*
unsigned short	*int*
unsigned long	*long*
unsigned char	*byte*
signed char	*byte*
float	*double*
double	*double*
long double	*double*
void	*void / Object*

A C *char* típusának képe azért a *byte* és nem a *char*, mert a *byte* értéktartománya (-128 és +127 között) felel meg a C-s karakter típusok tartományainak. A Java-s *char* ugyanis 0 és +65535 közötti értéket vehet fel.

A *void* típus a C nyelvben több kontextusban is előfordulhat, ezek más-más kezelési módot igényelnek:

- Ha függvény visszatérési értékéről van szó, akkor a *void* képe *void* lesz.
- Ha függvény paramétereként szerepel, az C-ben azt jelenti, hogy nincs paramétere a függvénynek. Ekkor nincs mit konvertálni, a leképzett metódusnak egyszerűen nem lesz paramétere.
- Ha *void** szerepel – azaz egy olyan pointer, ami adatra hivatkozik, de az adat típusáról nem ad információt – akkor a konvertált típus *Object* lesz, mert ez az összes létező Java osztálynak őse, így pontosan kifejezi a *void** szerepét.

6.4.2 Pointerek leképzése

Mivel Java-ban minden változó referencia, ezért egy C pointer Java megfelelője egy, a pointer által mutatott érték típusának megfelelő Java típusú egyszerű változó. Ez azonban csak az egyszeres indirekció esetét kezeli le, a többszörös esetet kollekciók vagy tömbök használatával lehet megoldani. A kétszeres indirekció például adott típusú változók kollekciójára képezhető le, a háromszoros az ilyen kollekciók kollekciójára, stb. Az OOGen jelenlegi formájában csak az egyszeres indirekciót támogatja, ezért a prototípus jelenlegi verziójában is csak ennyit kezelek. Azonban a modell kibővítése után, mivel a többszörös indirekció kezelésének koncepciója már meg van (kollekciók), az indirekciók számának kibővítése nem lenne bonyolult feladat.

A pointerek kapcsán a másik nagy probléma a memóriakezelés kérdése. C-ben ugyanis egyrészt pointer műveleteket is végezhetünk (pl. pointerek odébb mozgatása a memóriában, dereferálás), másrészt közvetlenül „belenyúlhatunk” a memóriába azzal, hogy egy pointert beállítunk egy közvetlenül megadott memóriacímre. Ezeket a műveleteket a transzformáció nem tartja meg. Egyrészt azért, mert a Java egyáltalán nem is támogatja őket semmilyen formában, másrészt pedig azért, mert a memóriakezelést a

garbage collector[14]-ra bízva felszabadítjuk a felhasználót a memória manuális kezelése alól. Így egyrészt kényelmesebb és hatékonyabb lehet a fejlesztés, másrészt a fejlesztés során egyáltalán nem ritka memóriakezelési hibákat (pl. memóriaszivárgás[15]) is elkerülhetjük.

További probléma, hogy függvények pointer típusú paramétere esetén – mivel a C-ben tömb típusú paramétereket a tömb elejére mutató pointerként veszünk át – nem tudjuk egyértelműen eldönteni, hogy az adott pointert tömbként használja-e az alkalmazás vagy nem. Így a tömb típusú paraméterek leképzése jelenleg nem támogatott, ezt később további (komplexebb) vizsgálatokkal lehetne megállapítani.

6.4.3 Külső könyvtárak kezelése

Az olyan külső könyvtárak leképzését, amelyek forráskódja nem áll rendelkezésre, a transzformáció nem tudja támogatni. Ezek lecserélése egy, a célnyelven elérhető megfelelő rendeltetésű könyvtárra a transzformáció elvégzése után kézi refaktorálással a felhasználó feladata marad.

6.5 Példa

A következő példa egy egyszerű listakezelő alkalmazást valósít meg C nyelven. A program könyvek adatait kezeli, és a sztenderd inputon / outputon kommunikál a felhasználóval. A könyvek adatait láncolt listában tárolja. Ennek az alkalmazásnak a transzformációját nézzük most meg.

Az alkalmazás entitásait a ***11. ábra*** mutatja be. Ide tartoznak a struktúrák, a rajtuk műveleteket végző függvények, és az általánosabb célú függvények. Globális változót nem használ a program.

[14] http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[15] https://en.wikipedia.org/wiki/Memory_leak

```c
typedef struct Rekord {
    char nev[41];
    char foglalkozas[41];
    char telefonszam[13];
    int sorszam;
    struct Rekord *elozo, *kov;
} Rekord;

typedef struct Strazsak {
    Rekord *elso;
    Rekord *utolso;
} Strazsak;

void kepernyoreir(Strazsak *strazsak, int hanyadik);
void fajlbair(FILE *f, Rekord *rekord);
void hozzafuz(Strazsak *strazsak, char nev[],
              char foglalkozas[], char telefonszam[], int *sorszam);
void modosit (Strazsak *strazsak, char nev[],
              char foglalkozas[], char telefonszam[], int hanyadikat);
int torol(Strazsak *strazsak, int hanyadikat);
void keres(Strazsak *strazsak, int mialapjan, char keresett[]);
void beolvas(FILE *f, Strazsak *strazsak, int *sorszam);

void entert_urit();
int hibas_input(int parancs, int alsohatar, int felsohatar);
void nem_megadott(char adat[]);
void bemasol(char dest[], char source[], char mennyit);
void kivag(char sztring[], int mennyit);
int hanyadik_karakter(char sztring[], char c);
void beker(char nev[], char foglalkozas[], char telefonszam[]);
```

11. ábra: A példaalkalmazás entitásai.

Az első lépés a struktúrákból osztályok létrehozása. Ezt a ***12. ábra*** szemlélteti.

```c
typedef struct Rekord {
    char nev[41];
    char foglalkozas[41];
    char telefonszam[13];
    int sorszam;
    struct Rekord *elozo, *kov;
} Rekord;
```

```
public class Rekord {
    private  byte[] nev;
    private  byte[] foglalkozas;
    private  byte[] telefonszam;
    private  int sorszam;
    private  Rekord elozo;
    private  Rekord kov;
}
```

```c
typedef struct Strazsak {
    Rekord *elso;
    Rekord *utolso;
} Strazsak;
```

```
public class Strazsak {
    private  Rekord elso;
    private  Rekord utolso;
}
```

12. ábra: A struktúrák osztállyá alakítása.

A második lépés a globális függvények osztályokhoz rendelése paraméter és visszatérési érték vizsgálat alapján. Ebben a lépésben a struktúrákon műveletet végző függvényeket adjuk hozzá metódusként a struktúrához tartozó osztályhoz. Mivel a visszatérési érték sehol sem struktúra típusú, így minden esetben a paraméterek típusa alapján választjuk ki a hozzá tartozó osztályt. Ennek az eredményét a ***13. ábra*** szemlélteti.

```
public class Strazsak {
    private  Rekord elso;
    private  Rekord utolso;

    public  void kepernyoreir(int hanyadik) {
    }
    public  void hozzafuz(byte nev, byte foglalkozas,
            byte telefonszam, int sorszam) {
    }
    public  void modosit(byte nev, byte foglalkozas,
            byte telefonszam, int hanyadikat) {
    }
    public  int torol(int hanyadikat) {
    }
    public  void keres(int mialapjan, byte keresett) {
    }
    public  void beolvas(Object f, int sorszam) {
    }
}
```

```
public class Rekord {
    private  byte[] nev;
    private  byte[] foglalkozas;
    private  byte[] telefonszam;
    private  int sorszam;
    private  Rekord elozo;
    private  Rekord kov;

    public  void fajlbair(Object f) {
    }
}
```

13. ábra: A paraméter és visszatérési érték vizsgálat eredménye.

Az utolsó lépés a maradék, általános célú függvények közötti hívási láncok vizsgálata. Ezek közül mindössze egy hívás található: a *beker* függvény hívja a *nem_megadott* függvényt. A függvények száma viszont 14, így a minimális osztályméret $s_{min} = \lceil 0{,}1 * 14 \rceil + 1 = 3$. Mivel csak ezt a két függvényt tudnánk osztályba szervezni, és a számuk nem haladja meg a minimális 3-mat, ezért a hívási láncok vizsgálata ebben az esetben nem alakít ki több osztályt. Így a maradék függvények (és a belépési pontként szolgáló *main* függvény) egy globális osztályba kerülnek. Ezt mutatja a ***14. ábra***.

```
public class ModernizedCProgram {

    public  void beker(byte nev, byte foglalkozas, byte telefonszam) {
    }
    public  int main() {
    }
    public  void entert_urit() {
    }
    public  int hibas_input(int parancs, int alsohatar, int felsohatar) {
    }
    public  void bemasol(byte dest, byte source, byte mennyit) {
    }
    public  void nem_megadott(byte adat) {
    }
    public  void kivag(byte sztring, int mennyit) {
    }
    public  int hanyadik_karakter(byte sztring, byte c) {
    }
}
```

14. ábra: A globális osztály.

6.6 Elérhetőség

A prototípus két részből áll: 1) magából a transzformációból, 2) valamint a dependenciaként használt OOGen metamodellből.

A futtatáshoz importáljuk be a megadott projekteket, majd Runtime Eclipseben futtassuk a *hu.bme.aut.moodernize.ui* projektet. Ezt követően a ***6.1***. fejezetben leírtak szerint kell eljárnunk.

A prototípus komponensei a következő URL-eken érhetők el:

- https://github.com/gaborbsd/Moodernize/tree/master
- https://github.com/gaborbsd/OOGen/tree/master

7 Összefoglalás

Dolgozatomban bemutattam egy módszert, amely procedurális felépítésű kódot képes automatikusan objektum-orientált szerkezetű kódra átalakítani. A módszert modell-vezérelt szemlélettel közelítettem meg. Ez azt jelenti, hogy az eredeti alkalmazásból egy olyan modellt készítettem, amely egy ugyanolyan szemantikával rendelkező, objektum-orientált szemléletű alkalmazást ír le. Az átalakítás során a vizsgált kód nyelve C volt.

A transzformáció kidolgozása során felmerült legfőbb probléma az volt, hogy hogyan szervezzük osztályokba a C program különböző elemeit annak érdekében, hogy jól áttekinthető, karbantartható, és bővíthető forráskódot kapjunk. Ezt a felépített modellen alkalmazott modell-transzformációk sorozatával értem el.

Az alkalmazott transzformációk a C nyelv struktúráit osztályoknak feleltetik meg, valamint a globális függvényeket próbálják meg ezekhez az osztályokhoz rendelni, mivel általában a procedurális felépítés miatt az adatstruktúrákon végzett műveletek különálló függvényekbe vannak szervezve. Ez a paraméter-és visszatérési érték vizsgálat célja.

Azokat a függvényeket, amelyeknél a hozzárendelés nem sikerült, további vizsgálatnak vetem alá, hogy úgy szervezzem őket egy külön osztályba, hogy az alkalmazás minél kevesebb komponense függjen ettől az osztálytól. Ezt a függvények közti hívási láncok elemzésével vizsgálom.

A dolgozatomban bemutattam a kidolgozott módszer egy Java nyelvű prototípusát is. A prototípus az ismertetett transzformációk végrehajtásával alakítja ki a C nyelvű program Java nyelvű, objektum-orientált szerkezetét. A program működését példákon keresztül is szemléltettem.

7.1 Az eredmények értékelése

A módszerem modell-vezérelt megközelítése a következő előnyökkel rendelkezik a régebbi kísérletekkel szemben:

1. Az objektum-orientált nyelvek közös, az OO paradigmák betartását eredményező tulajdonságait ki lehet emelni egy közös metamodellbe, az adott nyelv további nyelvi sajátosságait pedig a vizsgált nyelv függvényében tudjuk támogatni. Ezt követően az alkalmazás szerkezetét kialakító M2M

transzformációk egy többnyire nyelvfüggetlen modellen dolgoznak, aminek az az eredménye, hogy maga a transzformáció elvi folyamata (parszolás, modell építése és transzformációja, kódgenerálás) teljesen független a vizsgált nyelvektől. A módszer megvalósításakor tehát csak a különböző nyelvi sajátosságok – pl. típusok konverziója, memóriakezelés– kezelése a változó, a modell létrehozása és transzformációja ugyanolyan elvek mellett, ugyanolyan algoritmusok segítségével történhet.

2. A különböző modell-vezérelt eszközök számos hasznos funkcionalitással – pl. kódgenerálás, automatikus M2M transzformációk, modellek validációja, stb. – erősíthetik a modellek feldolgozását, ezáltal a transzformáció lehetőségeit és hatékonyságát. Ilyen például a dolgozatomban bemutatott Xtext és Viatra, de természetesen más keretrendszerek is használhatók.

7.2 Kitekintés és továbbfejlesztési lehetőségek

Az ismertetett transzformációs módszert több irányból is meg lehet közelíteni bővítések és továbbfejlesztések szempontjából. Ebben az alfejezetben röviden megemlítem ezeket a lehetőségeket.

7.2.1 Modell-vezérelt eszközök használata

A korábbiakban ismertetett modell-vezérelt eszközök – pl. Xtext, Viatra – használatával a transzformációt hatékonyabbá lehetne tenni. Egy ilyen lehetőség lehetne például a vizsgált nyelvek beépített könyvtári hívásainak – mint például matematikai függvények, fájlkezelés, stb –vizsgálata szakterületi nyelvekkel. Ehhez például az Xtext használatával az EMF modellen olyan M2M transzformációt lehetne végrehajtani, ami szakterületi nyelvek segítségével lecserélné az eredeti nyelv hívásait a célnyelv megfelelő hívásaira.

7.2.2 További vizsgálatok

A transzformáció végrehajtása során (az osztályok kialakításakor) további vizsgálatokat lehetne elvégezni, pl. a következőket:

- Építhetnénk egy, a CFG-hez hasonlóan adatfolyam-gráfot (dataflow graph) is, amely a függvény által létrehozott struktúra típusokat reprezentálná. Egy függvény abba az osztályba kerülhetne, amely nagyjából ugyanolyan típusú

más osztályokat referál, mint amit a függvény is használ, így csökkentve az adott osztályok közötti függőségeket.

- A hívási láncok vizsgálata során osztályba szervezett függvényeket redundancia szempontjából tovább lehetne elemezni. Ha például duplikált kódot találunk, akkor azt kiszervezhetjük egy külön metódusba, vagy ha több függvény ugyanazt a változót használja (paraméterként átadva egymás között), akkor abból osztályszintű változót generálhatnánk.
- A struktúrákat egymáshoz képest is vizsgálhatnánk. Ha valamely struktúrák néhány tagja ugyanolyannak bizonyul, akkor ezeket kiemelhetjük egy közös ősosztályba, és az adott osztályok ennek lehetnének a leszármazottai. Ugyanezt megcsinálhatnánk az osztályok viselkedése szempontjából is, ha a metódusok osztályokba szervezése már befejeződött. Ha különböző osztályok pontosan egyező szignatúrájú metódusokkal rendelkeznek, akkor felmerülhet, hogy ezek egy ősosztálybeli absztrakt metódust implementálhatnak.

Ábrajegyzék

Irodalomjegyzék

[1] B. Iván, Formális nyelvek, Budapest: Typotex, 2002.

[2] A. Aho, R. Sethi, J. Ullman és M. S. Lam, *Compilers: Principles, Techniques, and Tools*, Pearson Education Inc, 2006.

[3] N. Wirth, *What can we do about the unnecessary diversity of notation for syntactic definitions?*, Federal Institute of Technology (ETH), Zürich, 1977.

[4] Y. Zou és K. Kontogiannis, *A Framework for Migrating Procedural Code to Object-Oriented Platforms*, Waterloo.

[5] I. G. Czibula és G. Czibula, *Unsupervised transformation of procedural programs to object-oriented design*, 2011.

[6] P. Newcomb és G. Kotik, *Reengineering Procedural Into Object-Oriented Systems.*

[7] H. Gall és R. Klösch, *Program Transformation to enhance the Reuse Potential of Procedural Software*, New York, 1994.

[8] L. Vogel, Eclipse Rich Client Platform, 2015.

[9] D. Steinberg, F. Budinsky, M. Paternostro és E. Merks, Eclipse Modeling Framework, 2nd Edition, Addison-Wesley Professional., 2009.

[10] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Birmingham: PACKT Publishing, 2013.

[11] T. Parr, *The Definitive ANTLR 4 Reference*, 2007.

[12] G. Bergmann, I. Dávid, H. Ábel, Á. Horváth, I. Ráth, Z. Ujhelyi és D. Varró, *VIATRA 3: A reactive model transformation platform*, Springer Verlag, 2015.