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

Nagy Ákos

TERVEZÉSI MINTÁK PROGRAMNYELV-FÜGGETLEN FELISMERÉSE

KONZULENS

Dr. Kővári Bence

BUDAPEST, 2014

Tartalomjegyzék

Összefoglaló

Az objektumorientált szoftverfejlesztésben használt tervezési minták adott problémák adott kontextusbeli megoldásának alapötleteit írják le. Első dokumentációjuk [1] óta széles körben elterjedtek az ipari alkalmazásokban fejlesztésében. Javítják a kód karbantarthatóságát, bővíthetőségét, és a mérnöki gyakorlatban fontos kipróbált megoldások használatát is segítik. Programozási nyelvektől függetlenül írják le a megoldásokat, így minden, az objektumorientáltságot támogató programnyelven alkalmazhatóak.

Egy szoftver forráskódjában lévő tervezési minták elemzése segít megérteni a kód adott részének célját és az implementációs részletek mögött meghúzódó tervezési döntéseket. A karbantarthatóság és bővíthetőség elősegítése miatt a minták (vagy azok hiánya) a kód – és így a végleges szoftver – minőségére is utalhatnak. A kód manuális (a forráskód fejlesztő által történő) átvizsgálása viszont körülményes és időigényes feladat.

A minták automatikus felismerésének irodalma fokozatos térhódításuk nyomán egyre több terület ötleteit ötvözi (pl.: fuzzy rendszerek [2], formális nyelvek [3]). Bár a legtöbb alapötlet független a vizsgált programozási nyelvtől, a szerzők nem fektetnek valódi hangsúlyt a nyelvfüggetlenségre. Munkám során ezért egy olyan keretrendszert valósítottam meg, mely képes megragadni ezeket a nyelvfüggetlen elemeket és általánosan leírni az objektumorientált fejlesztésben használt alapfogalmakat.

A .NET Compiler Platform [4] (korábban Roslyn) megjelenésével a C# és a Visual Basic programozási nyelvek fordításhoz kapcsolódó szolgáltatások (pl.: szintaxisfa elemzése) közvetlenül elérhetővé váltak a fejlesztők számára. Ezt az új eszközt használtam fel arra, hogy a keretrendszer első konkrét implementációjaként C# programnyelven megírt forráskódból előállítsam a nyelvfüggetlen modellt.

A tervezési minták felismeréshez konkrét mintákat elemeztem. A felismerést egy rugalmas, jól bővíthető kritérium-ellenőrző rendszerként valósítottam meg C# nyelven. Eredményeimet nyílt forráskódú osztálykönyvtárak forráskódján validáltam és hasonlítottam össze más Java alapú rendszerek eredményeivel.

Mivel az irodalom elsősorban eddig a Java nyelvre fókuszált, ezért munkám fontos lépés a C# nyelven írt szoftverek elemzésében. Hosszabb távon a tervezési minták felismerése mellett egyéb szoftveres minták felismerése is implementálható (pl.: objektumorientált tervezési elvek betartásának vizsgálata, anti-patternek felismerése).

Abstract

Design patterns describe the basis of solutions for a given problem in a given context in object oriented software design. Since their introduction in [1], they have been widely adopted in the software engineering industry. Their use enhances maintainability, extensibility and it promotes reusability, one of the most important aspects of the engineering practice. They describe the solutions in a programming language neutral way, thus making them applicable in all languages that support object oriented design.

Analyzing design patterns in the source of a given software is helpful for understanding the purpose of a given code extract and the design decisions that have driven the implementation. As they promote maintainability and extensibility, their existence or lack are indicators of the quality of the code and indirectly the quality of the final product.

Because of their widespread adaptation, the literature of automated design pattern detection combines ideas from numerous other domains, e.g. fuzzy systems [2] or formal languages [3]. Although most of the concepts are language neutral, authors do not put an emphasis on elaborating the language neutral aspects of their solution. In my work I designed a framework that is capable of seizing these language neutral elements and describe object oriented concepts in a generalized way.

With the emergence of the .NET Compiler Platform [4] (formerly known as Roslyn), the services involved in the compilation and building of C# and Visual Basic code (e.g.: syntax tree analysis) have become available for software developers. I used this new tool to implement the first version of my framework in order to transform C# code into the language neutral model.

For the design pattern detection I analyzed concrete design patterns. I implemented the detection as a flexible, extensible criteria-checking system in C#. I validated my results on open-source libraries and compared them with the results of other, Java based works.

Since the literature has been mainly focusing on Java, my work is an important step in the analysis of software written in C#. In the long run, the detection of other software patterns (e.g. other object-oriented design considerations, anti-patterns) can also be implemented.

1 Bevezetés

A tervezési minták objektumorientált környezetben próbálják meg elősegíteni azt, hogy a végső forráskód karbantartható, könnyen bővíthető, tesztelhető legyen – egyszerűbben fogalmazva, hogy jó minőségű kód szülessen. Mivel elsősorban megoldási ötleteket írnak le az objektumorientáltság alapfogalmait használva, ezért minden, ezeket támogató nyelven használhatóak. Ezen előnyeik miatt használatuk általánosan bevett gyakorlat a szoftverfejlesztésben.

A tervezési mintáknak előnyei és széles körű használata miatt a minták felismerésének területe aktív kutatási terület. A tervezési minták használata a kódban egyértelmű hatást gyakorol objektumorientált minőségi metrikákra, így a minták felismerése a metrikák értékének jó előrejelzése. Szintén fontos, hogy mivel bevett gyakorlatokat reprezentálnak, ezért ha egy adott forráskódrészlet illeszkedik egy tervezési mintára, akkor annak célja és az implementáció mögötti tervezési döntések jól körvonalazhatóak további elemzés nélkül is. Ez rosszul működő rendszerek hibaelhárításakor, öröklött ősrendszerek (legacy rendszerek) karbantartásakor vagy újratervezésekor az elemzési fázisba fektetett energiákat közel felére is csökkentheti. Ezek mellett a tervezési minták részletes analízise segíthet feltárni olyan új kontextusokat, amikben egyes minták használhatóak (vagy éppen ellenkezőleg, nehezen használhatóak).

Munkám célja kettős. Egyrészt egy olyan programnyelv-független adatmodell kialakítása, amellyel tetszőleges, az objektumorientált paradigmát támogató programozási nyelven leírt forráskód leírható. Másrészt erre az adatmodellre építve egy tervezési mintákat felismerő rendszer architektúrájának és algoritmusának a megtervezése.

Dolgozatom több szempontból is hiánypótló a tervezési minták felismerésének irodalmában.

Bár az irodalomban felhasznált legtöbb ötlet szintén programozási nyelvektől függetlenül alkalmazható, és szintén egy programozási nyelvektől független bemeneti formátummal dolgozik, a szerzők legtöbbje nem fordít valódi hangsúlyt arra, hogy ezeket a nyelvfüggetlen aspektusokat bemutassa, alkalmazásukat megindokolja.

Másrészt az eddigi kutatások elsősorban Java nyelven megírt szoftverek forráskódját analizálták. Én az adatmodell és a mintafelismerő működésének

validációjaként elkészült első implementációm során C# nyelven megírt forráskódot elemeztem. Mivel a két nyelv eszköztára több területen eltérő, ezért későbbi kutatások alapja lehet saját eredményeim összehasonlítása Java nyelvű kódok elemzésének eredményével.

Harmadrészt a C# nyelvű forráskód használatához a .NET Compiler Platform osztálykönyvtárat használtam fel. Ez az osztálykönyvtár még jelenleg is fejlesztés alatt van, végső megjelenése a dolgozat írása után fél évvel várható. Ennek az eszköznek ilyen célú használata mindenképpen újszerű.

Dolgozatomban a következőkben röviden definiálom a tervezési mintákat általánosan, majd kitérek a modern programnyelvek fordítási modelljére a C# programozási nyelven keresztül. Ezek után bemutatom a .NET Compiler Plaformot.

A munkám szempontjából alapvető jelentőségű elemek rövid áttekintése után bemutatom a tervezési minták használatának és ezen keresztül felismerésének értékét, részletesebben kitérve a korábbi bekezdések állításaira. Ehhez bemutatok néhány metrikát, megmutatom a metrikák kapcsolatát a kód minőségével és a tervezési mintákkal. A mintafelismerés irodalmának rövid áttekintése zárja ezt a fejezetet.

A háttérinformációk leírása után bemutatom a nyelvfüggetlen adatmodellemet. Megmutatom, hogyan képes leírni tetszőleges, objektumorientált programozási nyelv elemeit és hogyan építhetők fel rá a mintafelismerő algoritmusok, melyek szintén ebben a fejezetben kerülnek leírásra. Mind az adatmodellnek, mind pedig a mintafelismerőnek C# nyelven készítettem el a referencia-implementációját, amit szintén röviden bemutatok.

Az általános keretrendszer bemutatása után megmutatom, hogyan alkalmazható a .NET Compiler Platform forráskódelemzésre és hogyan transzformáltam át ennek segítségével a forráskódot az általam definiált modellbe.

Végül bemutatom eredményeimet, amiket nyílt forráskódú, C# nyelvű osztálykönyvtárak elemzésével nyertem. Eredményeim validálásához összehasonlítom őket a Java nyelvű kutatások eredményeivel és ennek fényében értékelem őket. Szintén itt mutatom be, hogyan lehet integrálni a C# nyelven elkészült referencia-implementációt a programozási nyelv elsődlegesen használt fejlesztőkörnyezetébe.

Dolgozatom zárásaként előrevetítek néhány fejlesztési lehetőséget és még egyszer összefoglalom az eredményeimet.

1.1 Tervezési minták

Az objektumorientált tervezési minták legelterjedtebb definíciójuk szerint valamilyen, adott kontextusbeli problémára adják meg a megoldás általános, újrahasznosítható elemeit [1]. Bár a tervezési minták első koncepciója jóval korábbra nyúlik vissza, az objektumorientált fejlesztésben ismertté az 1995-ben megjelent, Design Patterns: Elements of Reusable Object-Oriented Software [1] című könyv tette őket.

Ebben a könyvben a szerzők dokumentálnak 23 tervezési mintát. A dokumentáció legfontosabb elemei:

- A tervezési minta neve. Ez, bár evidensnek tűnhet, nagyon fontos, hiszen elősegíti a fejlesztők közötti kommunikációt, ezzel magában hordozva az újrafelhasználhatóság lehetőségét.
- A kontextus és a probléma, amiben a minta használható, és amit a minta megold. Ez a definíció alapján a tervezési minták legfontosabb eleme.
- A probléma megoldásának alapötlete. Ennek strukturális leírására a szerzők az objektumorientáltság alapfogalmait használják (osztály, interfész, mező, asszociáció, aggregáció), míg az esetenként szintén szükséges, dinamikus viselkedést (értsd: adott metódus implementációjának részlete) is megadják.

A tervezési minták, mivel adott problémákra jól bevált megoldásokat jelentenek, értelemszerűen növelik a kód újrafelhasználhatóságát és javítják a robosztusságát. Ezek mellett az előnyeik mellett a kód minőségén egyéb szempontok szerint is javítanak; ezekre a 2.1 fejezetben térek ki részletesebben.

A későbbi fejezetekben a valóban vizsgált tervezési minták röviden bemutatásra kerülnek, de a minták részletes bemutatása nem célja a dolgozatnak. A továbbiakban egy adott tervezési mintán az [1]-ben leírt verzióját értem és az ott használt nevével hivatkozok rá, eltérés esetén ezt külön kiemelem. Ennek oka részben az, hogy a tervezési minták felismerésének kutatói szintén ezekre a minták hivatkoznak, így ez elősegíti az eredmények megismételhetőségét, részben pedig az, hogy ezek a minták és dokumentációjuk az iparban is a megoldások alapjait képzik.

A tervezési minták tehát a szoftvermérnökség egyik fontos alappillérei. Míg a legtöbb egyéb mérnöki tudományág művelői nagyon sok bevett gyakorlattal és időtálló

megoldással rendelkeznek bizonyos problémákra (az építészmérnökök száz vagy ezer éve álló épületekkel, a gépészmérnökök olyan évszázados találmányokkal, mint a csiga vagy az inga), addig a szoftverfejlesztés néhány tíz (vagy nézőponttól függően kevesebb) éve létezik, a szoftvermérnökök nem „óriások vállán állnak”. A tervezési minták leírása, ismerete és alkalmazása segít a programozást és a programírást valódi mérnöki tudománnyá emelni.

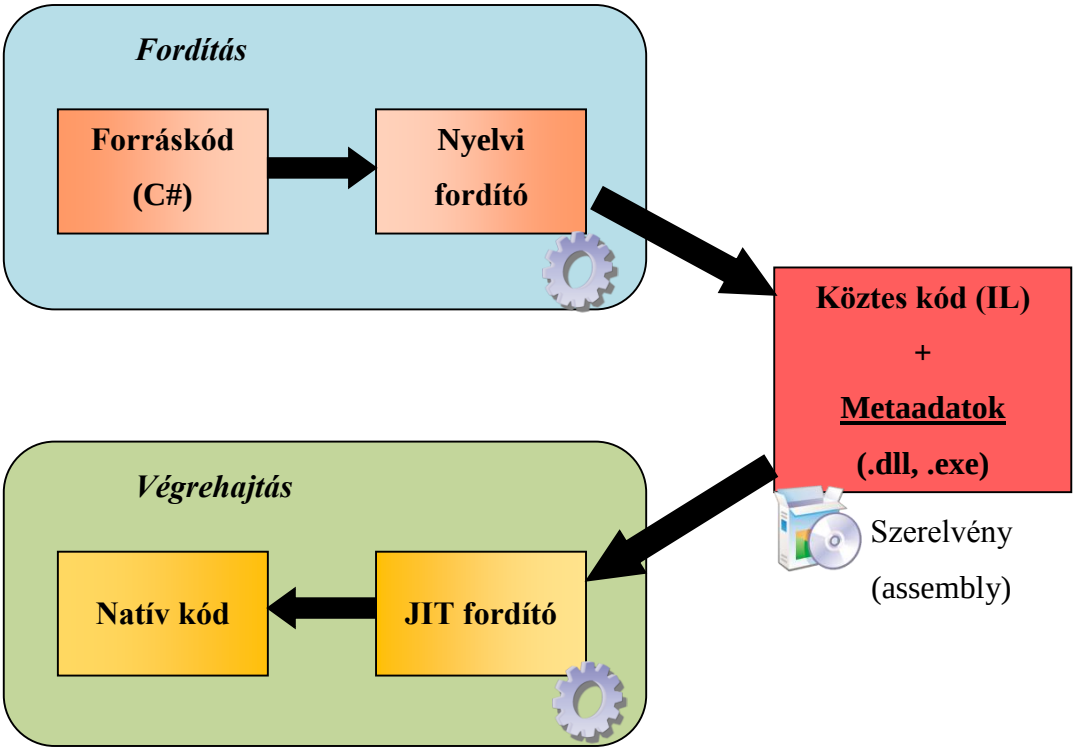
1.2 Modern programozási nyelvek fordítása

A fordítási folyamat célja, hogy adott programozási nyelven megírt forráskódból előállítsa az operációs rendszer és a processzor számára is értelmezhető, futtatható állomány(oka)t. A programkódok fordítása a nyelvek és automaták témakörének fontos aspektusai, ennek részletes bemutatása túlmutat a dolgozat keretein. A dolgozatomban elkészített, C# forráskód-elemző működésének és technológiai hátterének megértéséhez azonban szükséges a folyamat nagyvonalú ismerete, ezért ez a fejezet bemutatja ennek a nyelvnek a fordítási folyamatát.

1.3 C# programozási nyelv fordítása

A C# programozási nyelven megírt kód fordítása és futtatása két lépcsőben történik. Első lépésben a C# nyelvű kódból a C# nyelvi fordító egy köztes nyelvű kódot hoz létre. Ez a kód ekvivalens a C# nyelvű kóddal, de az operációs rendszer számára még nem értelmezhető. Ezt a köztes nyelvű kódot a .NET keretrendszer dolgozza fel a

következő lépésben és készít belőle futtatható kódot. Ez a JIT (just-in-time) fordító feladata. Ezt mutatja be az 1. ábra:

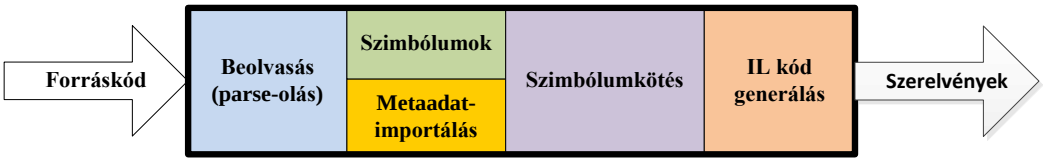


1. ábra - C# forráskód fordítása és futtatása

Ez a kétlépcsős megoldás a .NET keretrendszerhez hasonló, menedzselt környezetek programozási nyelveire jellemző. Az egyéb programozási nyelvek esetében általában már az első lépés végén előáll az operációs rendszer által futtatható kód, így a következő fejezetekben leírtak jelentős része igaz egyéb (fordított, és nem interpretált) programnyelvek esetében is.

1.3.1 Hagyományos megközelítés

A 2. ábra mutatja a C# forráskód nyelvi fordításának lépéseit:



2. ábra - C# forráskód nyelvi fordítása

Az első lépés a forráskód tokenizációja. Ebben a lépésben történik meg a kód beolvasása szintaxisfába. A szintaxisfa egy fagráf, ami a forráskód elemeit egy

fastruktúrába rendezve tartalmazza a programozási nyelv szabályainak megfelelően. A szintaxisfa mindig követi a teljes hitelesség elvét, azaz a forráskódban leírt összes információt tartalmazza, de egy strukturált, kategorizált elrendezésben. Ennek a beolvasásnak a célja, hogy egyrészt megkönnyítse a forráskód szintaktikai elemzését (értsd: a forráskód megfelel-e a programozási nyelv szabályainak), illetve hogy lehetővé tegye az ezutáni automatizált transzformációkat és feldolgozásokat.

A következő lépés a metaadatok importálása. Fejlesztés közben egyrészt a .NET keretrendszer komponenseit, másrészt külső osztálykönyvtárakat is használunk. Ezeknek a forráskódja nem áll rendelkezésünkre, de a benne lévő, lefordított típusok metaadatai (pl.: név) importálásra kerülnek, hogy lehessen dolgozni ezekkel a típusokkal.

Ezután a szintaxisfában lévő tokenek (karakterláncok) az importált metaadatok figyelembe vételével nevesített szimbólumokká képződnek le. A nevesített szimbólumok működését és jelentőségét legkönnyebben az alábbi példával lehet szemléltetni:

C# programnyelven egy egész szám változó deklarálásának több lehetősége is van:

- System.Int32 pelda=1 : Ebben az esetben a típus teljes nevét használjuk (AssemblyQualifiedName). A szintaxisfában megjelenő token a System.Int32.
- Int32 pelda=1 : Ebben az esetben a típusnak csak a rövid nevét használjuk, mert a kódban korábban importálva lett a System névtér. A szintaxisfában megjelenő token az Int32.
- int pelda=1 : Ebben az esetben a típusnak egy alternatív nevét (alias-át) használjuk. A szintaxisfában megjelenő token az int.
- A típusneveknek akár mi magunk is adhatunk alternatív neveket, ezzel lényegében végtelenre növelve az adott típust leíró tokenek számát.

A fenti esetekben a szintaxisfában megjelenő tokenek különbözőek, de ugyanahhoz a típushoz tartoznak. A szemantikai analízis feladata, hogy ezt felismerje és minden tokenhez a megfelelő nevesített szimbólumot hozzárendelje.

Egy másik jó példa a forráskódban használt kommentek és üres karakterek. Mivel a szintaxisfa a forráskódot teljes hitelességében tartalmazza, így ezek is elemei a

szintaxisfának. Ezek viszont nem kerülnek lefordításra, így szemantikus analízis során a fa ezen elemei eldobásra kerülnek.

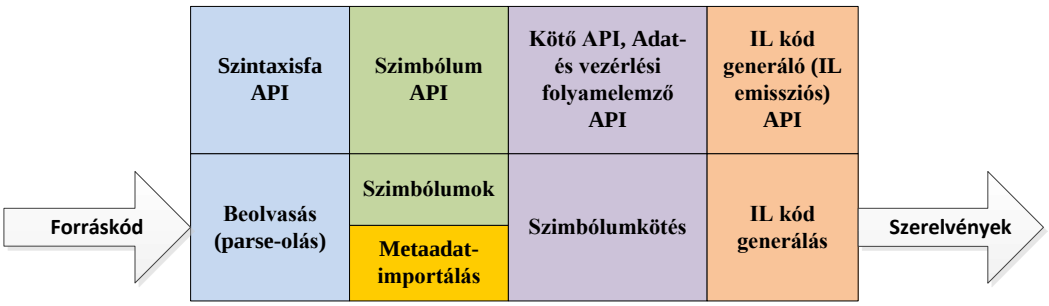
A következő lépés a szimbólumkötés. Ebben a fázisban a nevesített szimbólumokhoz a fordító kikeresi a megfelelő típusokat, metódusokat, mezőket. A fenti példa esetén használt szimbólumokhoz a System.Int32 típus fog tartozni.

Utolsó lépésben a kötés után kialakult állapotból az IL Emitter elkészíti a köztes kódot (**I**ntermediate **L**anguage kód), amit később a .NET keretrendszer tud futtatni. Egyéb, nem menedzselt programozási nyelvek esetében ebben a lépésben már közvetlenül futtatható kód készülhet.

A folyamatnak egy fontos aspektusa ebben a hagyományos megközelítésben, hogy az egész egy fekete doboz. Bemenetként megkapja a forráskódot, kimenetként pedig előállnak a köztes kódot tartalmazó állományok (szerelvények), de a fejlesztőknek nincs lehetősége sem arra, hogy a fordítási folyamatba belenyúljanak, sem arra, hogy a fordítás egyes szolgáltatásait, vagy az ezek során keletkező információkat (szintaxisfa, szemantikus modell) direkt módon elérjék.

1.3.2 A .NET Compiler Platform

A .NET Compiler Platform (korábban Roslyn) megközelítése szakít a korábbi hagyományokkal. A fordítás eddigi feketedoboz-szerű folyamatához biztosít API-t, amin keresztül a fejlesztők az egyes fázisok szolgáltatásait direkt módon is hívhatják akár tervezési időben, a fázisok során előálló információkat elérhetik, és saját maguk feldolgozhatják. Ezt szemlélteti a 3. ábra:



3. ábra - A .NET Compiler Platform API-ja

Az előző fejezetnek megfelelően a beolvasás során áll elő a szintaxisfa, ami ezentúl elérhető az új fordítási modellben. A következő lépések (metaadat-import és szimbólumfeldolgozás) során előáll a szimbólumtábla és a szemantikus modell egyes

elemei, míg a kötés során a szemantikus modell további elemei (pl.: vezérlési- és adatfolyamgráfok). Az utolsó lépésben az Emit API-n keresztül elérhetővé válik az előállított köztes kód is.

A folyamat egyes lépése során elérhető információk segítségével a C# kód feltérképezhető, meghatározhatóak a benne lévő osztályok, köztük a kapcsolatok, az osztályok metódusai, a hívási relációk alapján a hívásgráfok és még sok egyéb. A pontosan felépített modellt a következő fejezetek mutatják be.

Erre az API-ra építve elérhetőek további funkciók is, amik már nem csak a vizsgált programozási nyelvhez, hanem a Visual Studio fejlesztőkörnyezethez is kapcsolódnak. Ilyenek az automatikus kódelemzés és –átírás, a forráskódátrendezés, a forráskódszínezés (syntax highlighting). Ezek az eszközök korábban is részei voltak a környezetnek, de ezzel az új API-val egységes alapra helyezhetőek a fordítási folyamattal.

Megjegyzés: Munkám során különös hangsúlyt fektettem arra, hogy a tervezési minta felismerés valóban programnyelv független maradjon. Ez természetesen csak azoknak a programnyelveknek az esetében lehet hatékony, ahol a C#-hoz (és a Visual Basic-hez) hasonlóan létezik egy eszköz, ami a forráskód ilyen analízisét lehetővé teszi (egy ilyen eszköznek a megírása komoly feladat). A legtöbb modern, objektumorientált programnyelvhez viszont léteznek hasonló célú, még ha kevésbé kifinomult megoldások. Java programnyelv esetén ilyen a Recode, C++ esetén a flex++.

2 Tervezési minták felismerése

Ez a fejezet a tervezési minták használatának és felismerésének jelentőségét mutatja be, erősen építve a tervezési minták felismerésének és az objektumorientált fejlesztés szoftvermetrikáinak irodalmára.

2.1 Tervezési minták jelentősége

Az objektumorientált programozási paradigma megjelenésének elsődleges célja az volt, hogy a komplex rendszereket karbantarthatóbbá, egyszerűbben bővíthetővé tegye. Az erre használt fogalmak (öröklés, adatrejtés, polimorfizmus, interfész, felelősségek szétválasztása, csatolás, kohézió) azonban meglehetősen absztraktak, ezek ismerete önmagában nem garantálja azt, hogy a készülő rendszerekben megfelelően lesznek alkalmazva (ha egyáltalán alkalmazva lesznek).

Ennek a fejezetnek az a célja, hogy bemutasson néhány objektumorientált metrikát, amelyekkel számszerűsíthető az objektumorientált kód minősége, így konkretizálva a minőséget. Erre építve bemutatom a tervezési mintáknak a metrikákra – és így a végső termékre – gyakorolt hatását, ezzel alátámasztva a tervezési minták használatának fontosságát.

2.1.1 Az objektumorientált tervezés metrikái

Az objektumorientált kódok minőségének számszerű meghatározásának kutatásában alapműnek számít [5]. Ebben analitikus indoklással kerültek bevezetésre metrikák, amelyek az objektumorientált kódok esetében számíthatóak.

- Öröklés mélysége: Ez az adott kódelem helyének az öröklési fában a gyökérelemtől vett távolsága. A bonyolult öröklési fa megnehezíti a tesztelést és a karbantarhatóságot.
- Gyerekek száma: Egy osztályra (interfészre) vonatkozóan a belőle leszármazó osztályok (interfészek) száma. Ennek a metrikának nem csak a lokális, hanem az átlagos elemzése is igazán hasznos: ugyanazt a működést leírni kevesebb leszármazottal átlagosan globálisan áttekinthetőbbé teszi a kódot és segíti a tesztelést.

- Csatolás: Egy osztályra (interfészre) vonatkozóan azoknak az osztályoknak (interfészeknek) a száma, amelyekre az osztály hivatkozik Minél magasabb a csatolása egy osztálynak, annál nehezebben tesztelhető önállóan és annál kevésbé újrahasznosítható, hiszen a működéséhez nagyszámú egyéb osztályra is szükség van. A rendszer stabilitásában is fontos szerepe van, hiszen egy osztálynak minél nagyobb a csatolása, annál több más szolgáltatást használ, amelyek hibája esetén saját működése sem feltétlenül biztosított. Az interfészek használata csökkenteni a csatolás mértékét.

- Az [6] szerzői definiálták a metódusok számát metrikaként. Tesztelés szempontjából értelemszerűen azokat a metódusokat könnyebb kezelni, amelyeknél ez az érték kisebb.

- Stabilitás: A stabilitást definiáló képlet a csatolás alapján a következő:

$$S = \frac{Ca}{Ca + Ce}$$

ahol *Ca* az afferens csatolás (azoknak az osztályoknak a száma, amik függenek az adott osztálytól), *Ce* pedig az efferens csatolás (azoknak az osztályoknak a száma, amiktől függ az adott osztály). (A nevező 0 értéke csak olyan osztályok esetén fordulhat elő, ami nem hivatkozik semmire és rá sem hivatkoznak, de ez az eset a gyakorlatban nem bír jelentőséggel). Minél magasabb a stabilitás, annál kisebb az osztály függőségeinek aránya.(hasonló metrika került definiálásra [7]-ben is).

2.1.2 A metrikák és a tervezési minták kapcsolata

A tervezési mintáknak és a metrikáknak közös a céljuk: segítsék a fejlesztőket abban, hogy megfelelő minőségű kódot produkáljanak. Felhasználási lehetőségük azonban más: a tervezési minták proaktívak, azaz használatukkal elkerülhetőek a rossz programozási gyakorlatok problémái. A metrikák használata egy reaktív megközelítés: a kész rendszer (vagy rendszerrész) tulajdonságai alapján kiszámítjuk az adott metrikát, majd a metrika jelentésének megfelelően értékeljük a kapott eredményt és szükség esetén megpróbáljuk kijavítani a problémát.

Mivel azonban közös a céljuk, ezért sok esetben jól karakterizálható a kapcsolat közöttük. Ismert tény, hogy tetszőleges projektben minél hamarabb kerülnek

feltérképezésre a hibák, annál olcsóbb a kezelésük [8]. A tervezési minták alapján korán, fejlesztés közben következtethetünk kód minőségére a metrikák kiszámítása nélkül.

A dolgozatnak nem célja, hogy minden minta hatását bemutassa az egyes metrikákra, de a most következő cikkekben bemutatott és a 2.1.2.1 fejezetben használt módszerrel a többi minta hatása is hasonlóan értékelhető az adott metrikára.

[9] szerzői igazolták, hogy a Bridge tervezési minta csökkenti az átlagos gyerekszámot. Ez a korábbi megfontolások alapján csökkenti a kód komplexitását és növeli az átláthatóságot, így a karbantarthatóságot.

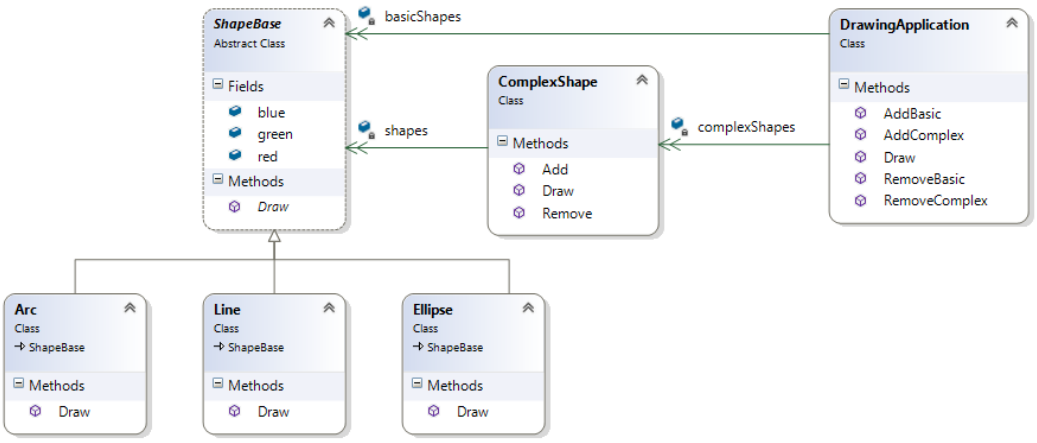
A Visitor tervezési mintának egyértelmű szerepe van az átlagos metódusszám csökkentésében (ha a Visitor-műveletek valóban nagyszámúak) [10]. A metódusszámot csökkentve kevesebb tesztelés kell a kód helyességének ellenőrzéséhez. Továbbá a működés egy részének absztrakciója a Visitor mintában szintén megkönnyíti a tesztelést.

A Mediator tervezési mintát elemzik [11] szerzői és megmutatják, hogyan csökkenti a csatoltságot a tervezési minta használta. Ezzel szintén megkönnyíti a tesztelést, növeli az újrahasznosíthatóságot és a stabilitást is definíciója szerint.

2.1.2.1 A Composite tervezési minta hatása

A dolgozatban később felismert Composite (kompozit) tervezési minta egyes metrikákra gyakorolt hatásait mutatja be részletesebben ez az alfejezet.

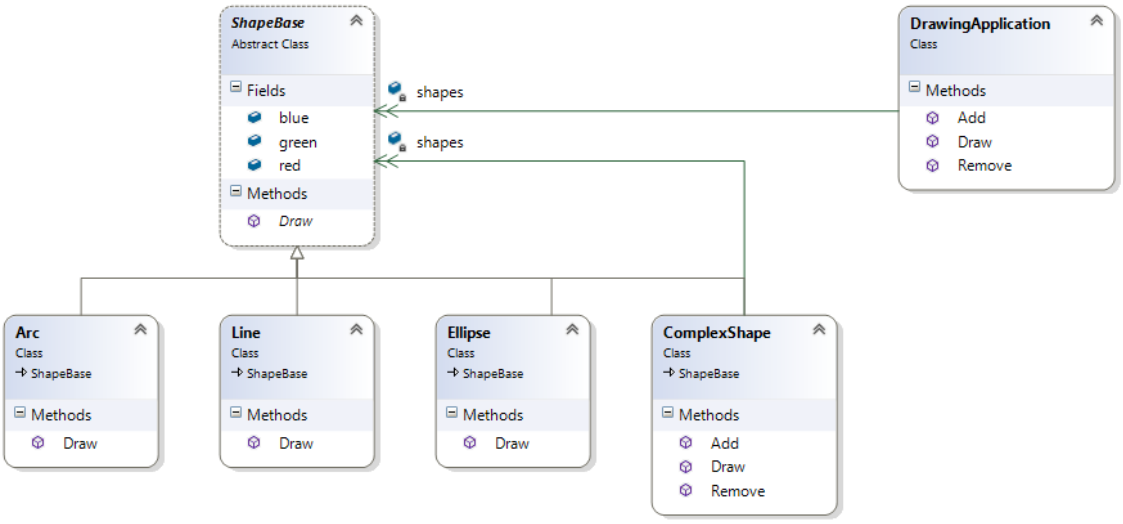
Feladatunk egy egyszerű rajzoló alkalmazás készítése, ami ismer alapformákat (vonal, körív, ellipszis) és ezekből az alapformákból összeállíthatunk bonyolultabb alakzatokat, valamint ezeket az alakzatokat kirajzolhatjuk. Első, tervezési minták nélküli megközelítésünk:



4. ábra - A rajzalkalmazás osztálydiagramja tervezési minták nélkül

A 4. ábraáról leolvasható, hogy a kliens osztály csatoltsága 2 (az alapformák közös őse és a bonyolult formák osztálya). Ráadásul mindkét csatolás efferens, tehát a kliens osztályt ezek a csatolások destabilizálják. Jól látszik az is, hogy metódusainak száma (a feladathoz képest) nagy, különösen akkor, ha figyelembe vesszük, hogy ugyanazért felelősek (listák karbantartása és feldolgozása).

Vezessük most be a Composite [1] tervezési mintát! Ezután az új struktúra:



5. ábra - A rajzalkalmazás a Composite minta bevezetése után

Az 5. ábraán látható új struktúrában a kliens osztály csatoltsága már csak 1 (ez a csatolás ugyan továbbra is efferens, de kevésbé van destabilizáló hatása a korábbi 2 efferens csatoláshoz képest). Fontos továbbá, hogy a metódusok száma felére csökkent.

A gyerekek átlagos száma ugyan nőtt, de ez egy olyan eset, amikor az osztályok száma nem nő meg, hiszen már meglévő elemek közé veszünk be öröklési relációt. Az öröklés bevezetése természetesen implicit csatolást jelent az ős és a leszármazott között, de ebben az esetben a csatoltság sem változik, hiszen a bonyolult alakzat osztály már eddig is függött az alakzat ősosztálytól.

Összességében tehát látszik a Composite tervezési minta használatának pozitív hatása a kód metrikáira, és így a korábbi fejezetek megfontolásai alapján – a metrikákon keresztül – a végső kód minőségére is.

2.2 Tervezési minták felismerésének jelentősége

A tervezési mintáknak nem csak a használata, hanem a felismerése is sokat segít a megfelelő minőségű szoftverek készítésében, a szoftverrendszerek evolúciójának elősegítésében.

A mintafelismerés legfontosabb és leginkább kézzel fogható haszna a korábbról öröklött ősrendszerek [12] (legacy systems) újratervezésekor jelentkezik. Ezek olyan idős szoftverrendszerek, amelyek karbantartását sokszor jóval a tervezett élettartam után is el kell látni. Gyakran kulcsfontosságú (mission-critical) rendszerekről van szó, amelyeket, amíg csak lehet, senki nem akar lekapcsolni. Mivel azonban a szoftverfejlesztés – mind paradigmáiban, mind eszköztárában – egy nagyon dinamikusan fejlődő tudományterület, minden ilyen rendszer idejében eljön az idő, amikor már nem csak elavult, hanem használhatatlan lesz (mert például a fizikai konfiguráció, amin fut túl régi és már nem kaphatóak hozzá pótalkatrészek, vagy, mert a szoftverfejlesztési platform támogatása megszűnt). Amikor elérkezik ez a pillanat, újra kell tervezni őket (reengineering). Erre több módszer is létezik, ám minden módszer egyes lépéseinek ugyanaz a lényege:

- Aktuális rendszer átvétele
- Kód, adatok, funkcionalitás, dokumentáció kinyerése/elemzése
- Új kód létrehozása, dokumentáció, tesztelés

Korábbi kutatások azt mutatják, hogy a teljes ráfordítás 43%-a a középső szakaszában van az ilyen célú projekteknek [13]. Ez érthető: a korábbi dokumentációk nem biztos, hogy elérhetőek vagy teljesek, az azóta történt módosítások, javítások pedig nem is feltétlenül dokumentáltak. Gyakran a forráskódot magát kell elemezni ahhoz, hogy a rendszer működését megértsük, ez viszont egy komoly feladat. Bizonyos kódrészletek céljának, vagy a kódrészlet megírása mögött tervezési döntések, követelmények azonosításában a tervezési minták viszont utat mutathatnak. A tervezési minták a definíció alapján adott problémák adott kontextusbeli megoldásának alapötleteit írják le. Azaz ha megismerünk egy mintát a kódban, akkor a megoldott problémát, így a követelményt és a tervezési döntéseket is fel tudjuk tárni. Ezt felismerve és tudva, hogy a teljes projekt ráfordításának közel fele ebben a szakaszban történik, jogosan merül fel az igény, hogy dolgozzunk ki automatizált módszereket a minták beazonosítására.

A kód működésének megértése a kód alapján (visszafejtés, reverse engineering) nem csak újratervezéskor lehet feladat, hanem működő, de hibás rendszerek kijavításában is. Az automatizált mintafelismerés ilyenkor is sokat segíthet.

Fontos érv még az automatizált felismerés mellett, hogy segíti a tervezési minták formális leírását. Bár jelenleg is léteznek formalizmusok a minták leírására, ennek mértéke elmarad az ideálistól. Ebből kifolyólag a minták alkalmazása is gyakran elsősorban empirikus úton történik.

Emiatt az empirikus, gyakran kísérletező alkalmazás miatt az iparban fejlesztett szoftverek egyfajta evolúciós folyamaton mennek keresztül. A mintákra tekinthetünk úgy a forráskódban, mint a génekre az élő szervezetekben. Az élő szervezet a génjei által meghatározott tulajdonságait használja fel a túlélésre környezetében. Ha ezek a tulajdonságok nem segítik a túlélést, a gének kihalnak az egyedekből az adott környezetben. Hasonlóan szoftverek esetében, a forráskódjukban található tervezési minták adnak nekik „tulajdonságokat”, amiket az adott kontextusban alkalmaznak. Ha a szoftver rosszul teljesít, az lehet annak az indikátora, hogy az alkalmazott tervezési minták mégsem használhatóak jól az adott kontextusban. A minták automatikus felismerésével ennek a folyamatnak az elemzését meggyorsíthatjuk, így is elősegítve minél jobb termékek készítését.

2.3 Tervezési minták felismerésének módszerei

Az előző fejezetben leírt megfontolások miatt a tervezési minták felismerése egy aktív és nagyon sok egyéb terület ötleteit ötvöző kutatási terület. Ennek a fejezetnek a célja, hogy betekintést adjon a felhasznált módszereibe, elemezze a korábbi kutatások és elhelyezze a saját munkámat ezek között.

A tervezési minták felismerésének alapvetően két különböző megközelítése létezik:

A **statikus elemzés** során a forráskód kerül elemzésre és az ebben lévő információk kerülnek feldolgozásra. Sokszor az irodalomban statikus elemzésen csak a strukturális elemzést értik, azaz a forráskód szerkezetének elemzése: osztályok, interfészek kapcsolata, metódusok deklarációja. Az egyértelműség kedvéért én statikusként hivatkozok minden olyan módszerre, amit a forráskód futtatása nélkül el lehet végezni (így a metódusok törzsének elemzése is ebbe a kategóriába tartozik).

A statikus megközelítések előnye, hogy nincs szükség a kód futtatására, ezzel erőforrásokat takaríthatunk meg és a kód külön instrumentálására sincs szükség. Hátránya viszont, hogy sok információ rejtve marad, különösen a kizárólag strukturális elemzések során (a State és a Strategy tervezési minták jó példák a strukturális hasonlóságra).

Ezzel szemben a **dinamikus elemzés** során a forráskód futtatása közben kerülnek információk kigyűjtésre. Gyakran az irodalomban dinamikus elemzésnek tekintik azt is, ha a metódusok forráskódját is kielemzik (nem csak a paramétereket és a visszatérési értéket), de nem futtatják a forráskódot (ennek oka, hogy a viselkedést tervezési időben is a metódusok írják le, a viselkedés és a dinamizmus pedig szorosan összekapcsolódnak). Az egyértelműség kedvéért én dinamikus elemzésen csak azokat a módszereket értem, amelyek a forráskód futtatásával szereznek információkat.

Ennek a megközelítésnek előnye, hogy bizonyos tervezési minták nagyon egyszerűen felismerhetőek segítségével. A Singleton (egyke) tervezési minta esetén annak meghatározása, hogy valóban egy példány jön létre a típusból a forráskód elemzésének nagyon mélyrehatónak kell lennie, de dinamikus elemzés során elég megbizonyosodni arról, hogy valóban csak egyszer fut le a konstruktor. Ez egy jó példája egyben a módszer hátrányának is – ezzel csak akkor kaphatunk pontos eredményt, ha végtelen ideig futtatjuk a kódot (másrészt honnan tudhatnánk, hogy valóban csak egy példány jöhet létre az objektumból). Ez természetesen nem megoldható, így csak statisztikai eredményeket kaphatunk. Emellett ennek a módszernek nagyobb erőforrásigénye is van a futtatás miatt és ehhez a kódot is instrumentálni (kiegészítésekkel adatgyűjtő célokra alkalmassá tenni) kell.

A korábbi kettő megközelítés keverékeként áll elő a **hibrid módszer**. A kód bizonyos aspektusait könnyebb a forráskódból, másokat pedig a futó kódból elemezni, így próbálja meg ez a módszer a másik kettő előnyeit kombinálni.

Az irodalomban felhasznált módszerek jelentős része strukturális vagy dinamikus elemzéseket végez. A statikus viselkedésbeli elemzés meglehetősen nehézkes, hiszen a forráskód mindig tartalmazza a program összes lehetséges lefutását (ez a forráskód célja), futási időben viszont tipikusan csak ennek egy része valósul meg (pl.: a kivételkezelés blokkjai ideális esetben nem kerülnek futtatásra). Emiatt a statikus viselkedésbeli elemzés komplexitása - összehasonlítva a strukturálissal vagy a dinamikussal - meglehetősen nagy.

Munkám validációja során a P-MARt projektben [14] ismertetett eredményeket vettem alapul. A cikkben bemutatott kutatáshoz kapcsolódóan a szerzők hallgatókkal elemeztették néhány nyilvánosan elérhető Java osztálykönyvtár (többek között JHotDraw illetve a JUnit) forráskódját és arra kérték őket, hogy azonosítsanak benne tervezési mintákat. Mivel ez az egyetlen olyan kutatás, amiben a vizsgált kód tervezési mintái ellenőrzött módon dokumentálva vannak, az – eddig Java programozási nyelvre építő – irodalom előszeretettel használja eredményei validálására. A projekt honlapján [15] elérhető a tervezési minták tára, amiben egy jól strukturált formátumban megtalálhatóak az elemzett programok tervezési mintái (és eszközök ennek értelmezésére).

A különböző kutatások alapvetően arra épülnek, hogy egyéb területek ötleteit használják fel (esetleg ötvözik) a tervezési minták felismerésében. A [16] szerzőinek fontos felismerése a statikus/dinamikus módszerekkel kapcsolatban az általam is korábban felvázolt hátrányok, így ők a két módszer kombinációját használják: a statikus elemzéssel szűrik a kódot első körben lehetséges mintákra, majd dinamikus analízissel ezeket tovább szűkítik. Alapötletük egyedi, a kód szintaxis fáját analizálják, ennek elemei között keresnek relációkat a statikus elemzés során, majd a dinamikus analízisben az így meghatározott lehetséges minta-osztályok létrejövő példányait és hívásait figyelik (ezek szintén az osztályok közötti kapcsolatokat írják le).

A [17] szerzői a tervezési minták felismerésének fontosságát hozzám hasonlóan a szoftverek evolúciójával indokolják. Alapötletük, hogy a kódban mikro-architektúrákat keresnek (pl.: metódushívás, öröklés, metódus-felülírás), a tervezési mintákat pedig ezek alapján határozzák meg. Megközelítésük hasonló az enyémhez (az egyes tervezési mintákat én kritériumok alapján azonosítom be, ezeket keresem a kódban), a mikro-architektúrák halmazának megkötése azonban esetenként rugalmatlansághoz vezet. A tervezési mintákat a mikro-architektúrák alapján adatbányászati módszerekkel és eszközökkel (pl.: WEKA [18]) illetve neurális hálók segítségével keresik. Saját módszerem nem köti meg a mikro-architektúráknak megfelelő kritériumok halmazát, így elősegítve, hogy minden tervezési mintára meg tudjam határozni a rá legjobban jellemző tulajdonságokat.

A [19] szerzőinek meglátása az, hogy ahelyett, hogy egy jó eszközt próbálunk meg fejleszteni a tervezési minták felismeréséhez, inkább már meglévő eszközök eredményeit kellene megfelelő módon kombinálni („data fusion”-ként hivatkoznak megoldásukra). Megállapítják, hogy ez a megközelítés javít az egyes minták

eredményein. Mivel azonban meglévő módszerekből építkezik, így ezeknek hibáitól nem teljesen mentes; abban az esetben például, ha minden eszköz hamis pozitív vagy hamis negatív eredményt ad egy-egy minta esetén, ez a kombinált megoldás sem tud mást mondani.

A [20] szerzői egy modellalapú megközelítést alkalmaznak. A forráskód alapján elkészítik annak UML modelljét, majd ezen gráfpárosító algoritmusokat futtatnak (a szoftveres modellfeldolgozás területén ez egy szokásos módszer). Ez a módszer a saját definícióm szerint strukturális elemzés, így annak minden hátrányát magában hordozza.

Egy másik, szintén gráfpárosító algoritmust használó munka a [21] . Ebben a szerzők egy egyéb területeken már sikeresen használt algoritmus-variációt alkalmaznak a mintafelismerésben. Hasonló strukturális elemzést végeznek a [22] és [23] munkák is

A [24] munka az egyetlen olyan a szakirodalomban tudomásom szerint, ami felveti a .NET kódban a tervezési minták felismerésének lehetőségét. Sajnos azonban a szerzők ebben csupán nagyvonalakban vázolják fel saját, korábbi, Java nyelvre építő munkájuk portolási lehetőségeit, sem az implementációt, sem az eredményeket nem közlik és azóta sem közölték (minden bizonnyal felhagytak kutatásuk ezen ágával). Fontos különbség még, hogy nem C#, hanem IL kódban (ld. 1.3 fejezet) keresik a tervezési mintákat. Ennek a megközelítésnek hátránya, hogy mivel a C# fordító esetenként optimalizálja a kódot fordításkor, ezért nem biztos, hogy az eredeti kódban lévő összes fejlesztői szándék felismerhető.

A [25] munka során a szerzők egy olyan módszert mutatnak be, ami az csak az Observer (megfigyelő) tervezési minta felismerésére alkalmas. Megközelítésük a mesterséges intelligencia ontológia-fogalmára épít. Ez a cikk rávilágít az általános tervezésiminta-felismerés nehézségére (bár a szerzők további munkaként célul tűzik ki egyéb minták felismerését, ennek módjára nem térnek ki).

[26] szerzői a tervezési minták felismerését egy érdekes céllal vezetik fel. Állításuk szerint a tervezési minták használata a megírt kódoknak egyedi, intellektuális aspektusa, így ezek akár a programkódok beazonosítására is szolgálhatnak. Ettől az ötlettől vezérelve bemutatják, hogyan használták fel saját módszerüket plágiumdetektálásra.

A [27], [28] szerzői erős matematikai alapokra helyezik a felismerést. A szigma-kalkulusból [29] indulnak ki. Ezt kibővítik néhány egyéb operátorral az

objektumorientáltság függőségeinek leírására (ezt ró-kalkulusnak nevezik), majd egy formális bizonyító segítségével ismernek fel mintákat.

Az [30] szerzőinek munkája megfordítja a problémát: nem a forráskódban lévő tervezési mintákat próbálják meg detektálni. Ehelyett olyan, hibás programozási gyakorlatok felismerését tűzték ki célul, amik az egyes tervezési minták használatának hiányából erednek. Módszerük ugyan erősen épít az általuk megvizsgált egyetlen tervezési minta, az Abstract Factory (absztrakt gyártó) tulajdonságaira, mégis jól példázza, hogyan használható fel a mintafelismerés a kód minőségének javítására.

Az irodalom teljes feldolgozása a dolgozat keretein túl mutat. Sok egyéb kutatási terület ötletei köszönnek még vissza ezen a területen (pl.: gépi tanulás [31], hasonlóság-pontozás [32] vagy fuzzy rendszerek [2]).

Saját módszerem a statikus módszerek közé tartozik. A statikus módszerek mindkét aspektusát felhasználtam az információk gyűjtésére. A tervezési minták felismerésére meghatározott kritériumok egy része strukturális, egy része pedig viselkedésbeli információkból származtatható. Ennek részletei a 4.2 fejezetben olvashatóak.

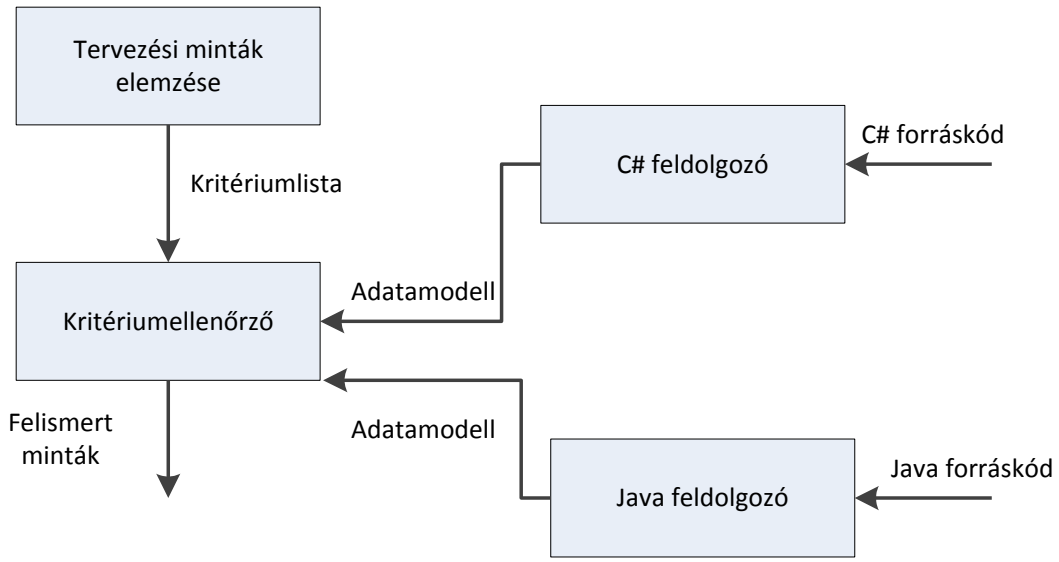
3 Programnyelv-független felismerési módszer

A tervezési minták felismerésének területén végzett korábbi kutatások szerzői írásaikban nem térnek ki részletesen munkájuk programnyelv-független aspektusaira. Bár a felhasznált alapötletek szintén valamilyen általános modellel dolgoznak, a modell leírása és előállításának lépései nincsenek bemutatva és dokumentálva. Munkám során különös hangsúlyt fektettem a nyelvfüggetlenségre, amelynek bemutatását a következő fejezetek tartalmazzák.

3.1 A nyelvfüggetlen felismerés architektúrája

Saját munkám első lépése az volt, hogy kidolgoztam egy olyan általános adatmodellt, ami képes leírni a modern objektumorientált programozási nyelvek konstrukcióit (osztály, interfész, metódus stb.) és a köztük lévő kapcsolatokat. Egy adott forráskódrészlet elemzésekor az első feladat az, hogy a forráskódot feldolgozva előállítsuk a neki megfelelő modellelemeket.

Ezután következhet a mintafelismerés már ezen az általános modellen, így garantálva a nyelvfüggetlenséget. A felismeréshez természetesen először a felismerni kívánt tervezési mintákat elemezni kell, és a definiáló karakterisztikáikat feltárni. A felismerés algoritmusa ezeket a jellemzőket keresi az általános modellben.

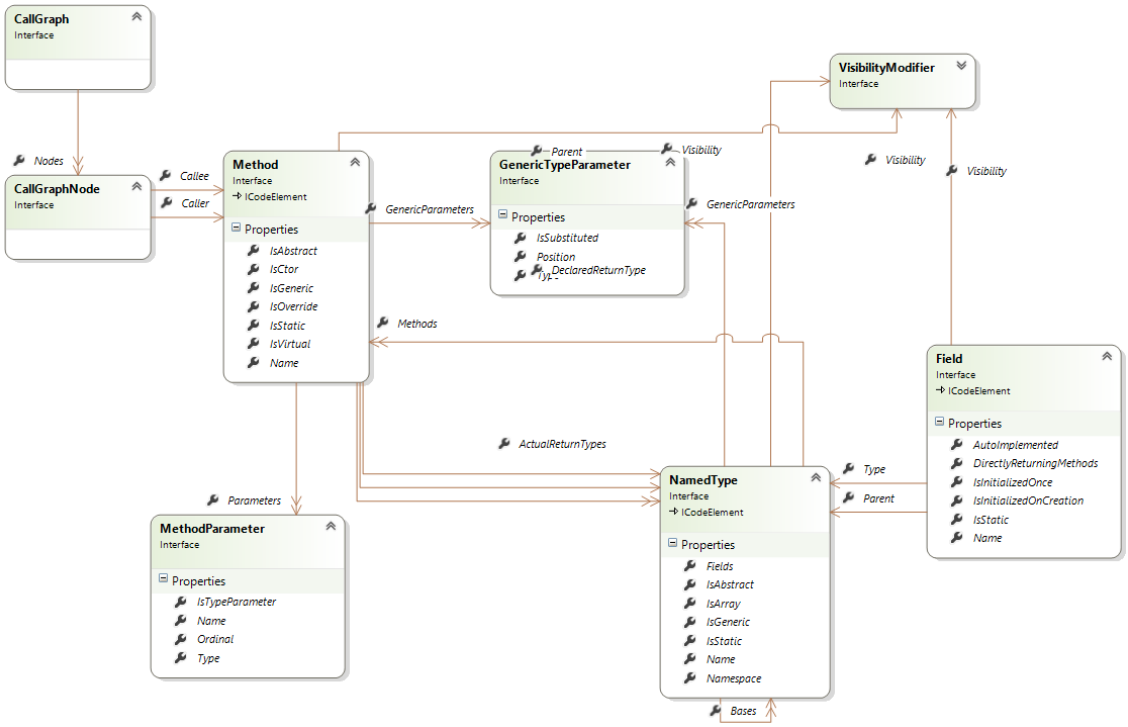


6. ábra - A tervezési mintákat felismerő rendszer architektúrája

A 6. ábraán bemutatott architektúra biztosítja tehát a nyelvfüggetlen felismerést. Természetesen minden programozási nyelvhez implementálni kell egy felismerőt, hiszen a nyelvi elemek, kulcsszavak és koncepciók eltérőek, ezután viszont az általános adatmodell és a mintafelismerő már használhatóak. A következő fejezetek ezeket mutatják be részletesebben koncepcionálisan és konkrét implementációban C# nyelven.

3.2 Adatmodell az objektumorientált fogalmak leírására

Az általános objektummodellt a 7. ábra szemlélteti:



7. ábra - Az objektumorientált nyelvek eszköztárát leíró általános adatmodell

Az objektumorientált eszköztár legfontosabb eleme az objektum. Az objektum azonban egy futási idejű konstrukció, tervezési időben osztályokról beszélünk, amelyeknek példányai lesznek az objektumok futási időben. Ennek megfelelően a modellünk központi eleme lehetne az „osztály”.

Minden objektum szolgáltatásokat nyújt az őket használó kódnak, ezeket metódusokon keresztül teszi. A publikusan hívható metódusok alkotják az objektum interfészét. Tervezési időben természetesen ezek a konstrukciók is az osztályban jelennek meg.

Problémát jelent azonban, hogy egyes programozási nyelvekben az osztály interfészét önálló kódelemként is le lehet írni, másképpen fogalmazva az interfészre explicit nyelvi támogatás van a nyelvben (Java, C#).

Viszont ennek a két konstrukciónak az egy kategóriába vétele nem megkerülhető, hiszen például a C++ programozási nyelv nem támogatja az interfészeket, ezeket absztrakt osztályokkal kell megvalósítani. A C++ ezen megoldása adja a probléma megoldását is: az interfészeket értelmezhetjük úgy is, mint absztrakt osztályok (hiszen nem példányosíthatóak) adattagok nélkül, csupa absztrakt metódussal (hiszen nincs implementációjuk a deklaráció helyén), amelyekben egyben virtuálisak is. Ezek persze nem valódi osztályok, így az „osztály" vagy „class" megnevezést nem célszerű használni, érdemesebb az objektumorientált terminológiában elterjedt típus fogalmát használni, ez tartalmazza az osztályok és interfészek kategóriáját is. A modell központi eleme így a nevesített típus (NamedType). Ebbe a kategóriába kerülnek tehát az osztályok, és abban az esetben, ha támogatottak, az interfészek.

A nevesített típusok legfontosabb jellemzője a nevük. Az objektumorientáltság egyik központi koncepciója az öröklés, ennek leírására minden típus tartalmaz referenciát az őseire (az ősök is nevesített típusok, így osztályok és interfészeket támogató nyelvek esetén ezek is benne vannak a gyűjteményben). A legteljesebb analízis érdekében nem csak a közvetlen ősök, hanem az ősök ősei is benne vannak ebben a listában. Fontos még tudni a típusokról a különböző módosítóikat (absztrakt, statikus) hiszen az interfészeket például ezek nélkül nem lehet leírni.

Ez az általános megközelítés lehetőséget ad arra is, hogy az olyan nyelv- vagy platformfüggő konstrukciókat, mint .NET esetében a delegate (típusos metódusreferencia) is fel tudjunk dolgozni ezen a kategórián keresztül.

Minden ilyen nevesített típusban vannak metódusok. Ezek a metódusok az objektumorientált metódusoknak felelnek meg a modellben. Legfontosabb tulajdonságaik a nevük, a paramétereik listája, a visszatérési értékük típusa és a különböző módosítóik (pl.: statikus, absztrakt, virtuális). Ez utóbbiak különösen fontosak, hiszen a legtöbb tervezési minta erősen épít a polimorfizmusra, amelyet jelentős részben ezekkel a módosítókkal lehet megvalósítani.

Metódusok esetében megkülönböztetünk deklarált és valódi visszatérésiérték-típusokat. Az objektumorientáltság lehetőségeit kihasználva gyakran élünk a

lehetőséggel, hogy egy adott metódus valódi visszatérési értékének típusa leszármazottja a metódus szignatúrjában deklarált típusnak Ezt a tervezési minták gyakran kihasználják, így ezek ismeret és megkülönböztetése nagyon fontos.

A metódusparaméterek lényegében név-nevesített típus párosok. Néhány metaadat még belekerült a modellbe (pl.: pozíció a paraméterlistában), de ezeknek legtöbbje a tervezési minták felismerésének szempontjából kevésbé fontos.

Az objektumok állapottal is rendelkeznek, amiket az adattagjaikban tárolnak. Tervezési időben ez is az osztályban jelenik meg (és csak az osztályban; interfészek nem rendelkeznek adattagokkal, hiszen ezek műveleteket írnak le). Minden adattagnak van neve és típusa. A későbbi analízis céljából tárolunk információkat arról, hogy az adattagok a létrehozásukkor kerülnek-e inicializálásra, illetve hogy pontosan egyszer kerülnek-e inicializálásra. Fontos, hogy a metódusok gyakran az adattagokon végeznek műveleteket; egy ilyen művelet például az adattag elérhetővé tétele az interfészen. Szintén a későbbi analízis céljából fontos ismerni azokat a metódusokat, amik az adott adattagot adják vissza valódi visszatérési értékként.

Az objektumorientáltság fontos elve az adatrejtés. Ez azt jelenti, hogy az objektumok adattagjaihoz csak szükséges esetben és ellenőrzött módon biztosítunk hozzáférést. Ezt láthatósági módosítókkal érjük el – ezek segítségével szabályozható az, hogy az egyes adattagok a kód mely részeiből érhetőek el. Privát (private) láthatóság esetén csak az osztály kódjában, védett (protected) láthatóság esetén csak az osztály és a leszármazottak kódjában, privát láthatóság esetén mindenhol elérhető az adattag. A legtöbb modern programozási nyelv definiál saját, speciális célú láthatóságokat. Ezek közül a modell egyet támogat: a szerelvény (.NET alapú nyelvek esetén internal, Java esetén package) láthatóságot. Az objektumorientált nyelvek a láthatóság fogalmát kiterjesztik metódusokra és típusokra is, így a modellem ennek megfelelően rendel láthatóságot a típusokhoz, metódusokhoz és adattagokhoz.

A modern programozási nyelveknek egyik lényeges szolgáltatása a sablonok vagy generikusok támogatása. Ennek lényege az, hogy a típusok egyes aspektusai nem tervezési, hanem (programozási nyelvtől függően) fordítási vagy futási időben dőlnek el. Ennek segítségével tudunk írni például általános adatszerkezeteket: írhatunk egy generikus binárisfa osztályt, amiben a fában tárolt elemek típusa a generikus típusparaméter. Később a generikus típusparaméter behelyettesítésével példányosíthatunk olyan bináris fákat, amelyek egész számokat tárolnak, és olyanokat,

amik karakterláncokat, anélkül, hogy külön típust kellene felvennünk emiatt. Mivel a modern programozási nyelvek ezt a koncepciót mind támogatják (és kibővítik típusokra és metódusokra), ezért ez is fontos eleme a modellemnek.

Saját modellem sok helyen hasonlít az UML osztálydiagramok metamodelljére, fontos azonban megjegyezni, hogy ez a modell egy kevésbé általános, a tervezési minták felismerésének gyakorlati igényeit szem előtt tartva kialakított modell. A legszembetűnőbb eltérés a generikusság kezelése. Ennek felvétele az adatmodellbe jelentősen megkönnyíti az elemzést. Sok egyéb attribútum is bekerült az egyes, UML-nek megfelelő elemekbe, amik az egyes tervezési minták elemzéséhez nélkülözhetetlenek lesznek. Ilyen a futási idejű visszatérési értékek listája metódusok esetén, vagy mezők esetén az inicializálás ténye.

3.2.1 Hívásgráf

Az adatmodellnek fontos eleme még a hívásgráf, ami a kódban található metódushívások alapján épül fel. Ez nem tartozik szigorúan az objektumorientált koncepciók közé (hívásgráfot egyéb megközelítésű programnyelveken is felépíthetünk a metódusoknak megfelelő kódolási egységekből), de fontos információkat hordoznak a kód viselkedéséről. A hívásgráf elemzésével kiszűrhetünk olyan, hamis pozitív eredményeket, amelyeket a csak strukturális elemzés valódi pozitívként észlel (részletesen ld. a 2.3 fejezetben).

A hívásgráfot éllistás formában tárolom. A lista minden elemének van két végpontja, ezekben a végpontokban pedig a hívó és a hívott metódus az általános objektummodell metódusaként van reprezentálva.

3.2.2 Az adatmodell implementációja

Az adatmodell – és az egész felismerő keretrendszer – implementációját C# nyelven készítettem el. Az adatmodell implementációját a 8. ábra látható négy szintben határoztam meg:

ICodElement réteg	**Modell elemeinek beazonosítása**
Interfész réteg (pl.: INamedType)	**Modell leírása**
Nyelvfüggetlen ősosztály réteg (pl.: NamedTypeBase)	**Mintafelismerés és alapszolgáltatások (pl.: egyenlőség meghatározása)**
Nyelvfüggő réteg (pl.: CShaprNamedTypeBase)	**Modellépítés**

8. ábra - Az általános adatmodell rétegei

A legfelső szinten egyetlen elem áll: az ICodeElement. Ezzel reprezentáltam az összes kódelem egyetlen közös aspektusát: azt hogy kódelemek, azaz objektumorientált programozási nyelvekben használhatóak.

A következő szint az interfészréteg: itt írtam le a korábbi fejezetben bemutatott modellelemek tulajdonságai.

A harmadik szint egy implementációs kényszer. Ezen a szinten a második szintben lévő interfészek absztrakt implementációja van. Ezekben az osztályokban valósítottam meg olyan műveleteket, amiket a felismerés során használtam (pl.: két típus összehasonlítása).

A negyedik szint már minden kódelemhez tartalmaz programozási nyelv specifikus tulajdonságokat. Munkám során C# forráskódok beolvasását és elemzését fogom megvalósítani, így ez a réteg olyan információkat hordoz, amik erre a nyelvre jellemzőek (pl.: a Compiler Platform által feldolgozott, egyes típusokhoz tartozó szimbóluminformációk). Fontos, hogy maga a mintafelismerés csak az előző 3 szinten található típusokkal dolgozik, így megőrizve a nyelvfüggetlenséget. Ezen a negyedik szinten lévő osztályok és a bennük lévő információk egyrészt a kód beolvasásának és

feldolgozásának hatékonyságát növelik, másrészt a hibakereséshez szolgáltatnak információkat.

Ezen az adatmodellen fut a felismerés, amit a következő fejezetek mutatnak be. Az adatmodell előállítását C# nyelvű forráskód alapján a 4.4 fejezet mutatja be.

4 Felismerés futtatása az objektummodellen

A tervezési minták felismeréséhez meg kell határozni a tervezési minták néhány kulcsfontosságú karakterisztikáját és ezeket a modell fogalmaival leírni. Az általános felismerő algoritmus ezeket a karakterisztikákat, mint kritériumokat tudja ellenőrizni az egyes kódelemeken, így beazonosítva a mintákat.

4.1 Tervezési minták kiválasztása

Munkám végső célja természetesen az összes, [1] által definiált tervezési minta felismerése. Először azonban az adatmodell és a felismerő validációjához ki kellett választanom néhány mintát, amelyek felismerését implementáltam.

A felismerendő tervezési minták kiválasztásakor komoly problémát jelentett az, hogy a szakirodalomban megjelent munkák eddig Java programozási nyelven megírt osztálykönyvtárak kódját elemezték. Bár munkámnak fontos aspektusa a nyelvfüggetlenség, a Java és a C# nyelvek eszköztára nem elhanyagolható mértékben tér el egymástól, amik egyes minták használatát könnyebbé/nehezebbé, vagy éppen fölöslegessé teszik (ld. később). Ez azonban saját, C# nyelvű forráskódomon végzett eredményeim validálását jelentősen megnehezítette. Megjegyzendő, hogy az eredmények validálásának a tervezési minták felismerésében még azonos nyelvű kódot elemző munkák esetében is olyan komoly probléma, hogy önálló kutatási területet alkot ([33], [34]).

C# nyelvre fókuszáló munkák hiányában a következő megközelítést választottam. Megvizsgáltam az irodalomban legtöbbször hivatkozott és legjobban dokumentált P-MARt projekt eredményeit. Ennek során a szerzők több, nyílt forráskódú, Java nyelvű osztálykönyvtárat elemeztek, de a JUnit nevű egységtesztelő könyvtárnak létezik .NET-re portolt, C# nyelven megírt verziója is (ennek szintén szabadon elérhető a forráskódja). A legjobb összehasonlíthatóság kedvéért tehát a saját felismerő algoritmusomat a .NET-es verzión futtattam és ezeket az eredményeket értékeltem a Javás verzió eredményeinek fényében.

Ez a megoldás – egyéb, C#-ra építő munkák hiányában – lehetőséget biztosít eredményeim validálására.

4.1.1 Tervezési minták kiválasztása

Először azokat a tervezési mintákat vizsgáltam meg, amelyeket a P-MARt szerzői a Javás verzió esetében:

- Composite
- Decorator (dekorátor)
- Iterator (iterátor)
- Observer
- Singleton

Ezek közül a tervezési minták közül .NET alapú programnyelvekben, mint amilyen a C# is, az Iteratort és az Observert kiváltják beépített nyelvi elemek (az enumerátorok és az események), így ezeknek az implementációja nagyon ritka. Ezt figyelembe véve csak a Composite, a Decorator és a Singleton tervezési minták felismerését valósítottam meg a listából.

Hogy az adatmodell teljes funkcionalitását jól le tudjam tesztelni, ezek mellett megvalósítottam még a Chain of Responsiblity (felelősséglánc) és a Factory Method (gyártó metódus) tervezési minták felismerését is. Ezek nem adnak ugyan összehasonlítható eredményt, de manuális ellenőrzéssel meg lehet bizonyosodni arról, hogy a felismert minták valódi pozitívak-e, ezzel is értékes visszajelzést nyerve az objektummodell és a felismerő algoritmus helyességéről.

4.2 Tervezési minták elemzése

A felismerő algoritmusnak az általános adatmodell példánya mellett bemenete még egy-egy kritériumlista minden, felismerni kívánt tervezési mintához. Ezeket a kritériumokat a tervezési minták strukturális és viselkedésbeli analízisével határoztam meg (ezek definícióját ld. 2.3 fejezetben). A következő felsorolásban a *strukturális információk dőlt,* **a viselkedésbeli információk félkövér** betűkkel szerepelnek. Fontos megjegyezni, hogy a két módszer együttes használata szükség esetén lehetővé teszi olyan követelmények megfogalmazását is, amelyek részben az egyik, részben a másik kategóriába tartoznak.

Az egyes, elemzett mintákhoz tartozó kritériumlista:

- Singleton:
    - *Nevesített típus*.
    - *Minden konstruktora privát*.
    - *Tartalmaz egy statikus mezőt, amelynek a típusa saját maga és ez a mező publikus* vagy **van egy publikus metódus, ami ezt közvetlenül visszaadja** (ebben az esetben a metódus csak statikus lehet, ezért ezt nem vizsgáltam külön).
    - **A mező pontosan egyszer van inicializálva.**
- Decorator:
    - Ebben az esetben a mintában szereplő konkrét decorator típusokat kerestem meg.
    - *Nevesített típus*.
    - *Valamelyik konstruktorának a paramétere vagy egy őstípusa, vagy a származási hiearchiában testvértípusa a vizsgált típusnak.*
    - **A hívási gráfban a típus és a konstruktorban megjelenő típus között van metódushívás. Ebben a metódushívásban a típus a hívó, a paramétertípus a hívott, a metódusok pedig:**
        - *abban az esetben, ha a konstruktorparaméter őstípus, akkor a hívó metódus felülírja (override) a hívottat.*
        - *abban az esetben, ha a konstruktorparaméter testvértípus, akkor a hívó és a hívott ugyanazt a metódust írják felül*.
    - *Nem kivételtípus:* C# nyelven a program nem várt működése közben keletkező hibákat a modern programozási nyelvek gyakorlatának megfelelően kivételekkel jelezzük. Fontos azonban, hogy a dobott kivétel csak a keretrendszerbeli Exception, vagy ennek leszármazottjának egy példánya lehet. A C# programozási ajánlások miatt a saját kivételtípusok egyben Decorator tervezési mintaként is működnek, viszont ennek csak egy primitív implementációi, így ezeket nem tekintem valódi Decoratornak

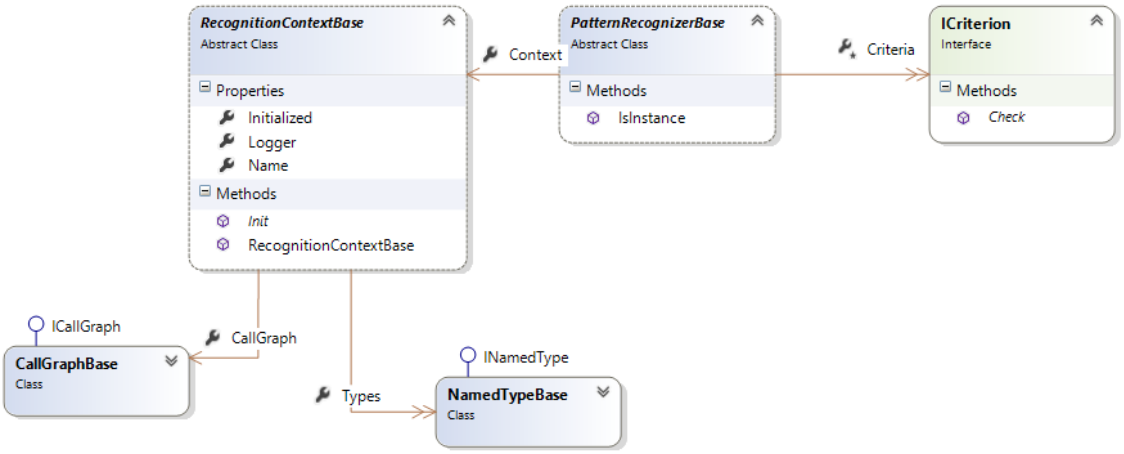
(emellett a kimenetben is túl sok zajt generál ezeknek a felismerése).

- Composite
    - Ebben az esetben a tervezési mintában szereplő közös komponens-interfészt kerestem meg.
    - *Nevesített típus.*
    - *Követelmény, hogy legalább két leszármazottja legyen, legalább két leszármazottjában legyen egy referencia, ami a típusra mutat vissza (szülő) és legalább az egyikben legyen még egy gyűjtemény ugyanilyen típusú elemekből (ez a kompozitosztály) és legalább az egyikben ne legyen gyűjtemény (ez a levélosztály). Ezek a referenciák jelen lehetnek mezők vagy metódusvisszatérések formájában.*
    - **A decorator mintához hasonlóan követelmény, hogy a kompozitosztály és a tartalmazott osztály között legyen a hívási gráfban él. A hívó a kompozit, a hívott a komponens, a hívó metódus pedig a hívott metódus felülírása.**
- Chain of responsiblity
    - Ebben az esetben a Chain of responsiblity osztályok közös ősét kerestem meg.
    - *Nevesített típus.*
    - *Követelmény, hogy legalább 3 leszármazottja legyen…*
    - **… és ezek között legyen él a hívási gráfban egy olyan metódus mentén, amely felülírja a közös ős egy metódusát.**
- Factory method
    - *Metódus.*
    - *Van egy őse, amelynek egy metódusát felülírja.*
    - **A felülírt metódus deklarált visszatérési értéke őstípusa az összes lehetséges valódi visszatérési érték típusának.**

4.3 Felismerő-algoritmus

A felismerő algoritmus bemenetei egyrészt a felépített adatmodell, másrészt pedig az előző fejezetben meghatározott kritériumok implementációja. A felismerő algoritmust C# programozási nyelven implementáltam.

A mintafelismerés saját, operatív adatszerkezete a 9. ábraán látható:



9. ábra - A mintafelismerő operatív adatszerkezete

A RecognitionContextBase osztály adja a felismerési kontextust – ez inicializálja és tárolja a modell elemeit. A modell előállítása az abstract void Init() metódus feladata. Egy adott programnyelvű forráskód elemzésekor le kell származni ebből az osztályból és az Init() metódust megfelelően felülírni, hogy előállítsa a Types és a Callgraph listákat (ezek a forrás típusait és hívási gráfját tartalmazzák; a metódusok és mezők a típusokon keresztül érhetőek el a modellben).

Az ICriterion interfész ír le egy kritériumot, maga az ellenőrzés pedig a bool Check(ICodeElement codeElement, RecognitionContextBase ctx) metódusban történik. Ha a leírt követelmény teljesül a bemenetként kapott kódelemre, akkor a visszatérési érték igaz, egyébként hamis. Az egyes mintákhoz meghatározott kritériumokat ilyen ICriterion interfész implementációkként valósítottam meg. A Factory Method esetén az alábbi kritérium-implementációk születtek:

```
//A vizsgált elem metódus és felülírja az ős egyik metódusát
public class OverrideCriterion : ICriterion
{
  public bool Check(ICodeElement elementToAnalyze,
                    RecognitionContextBase context)
  {
    MethodBase m = elementToAnalyze as MethodBase;
    if (m==null)
    {
      throw new ArgumentException();
    }
    return m.IsOverride;
  }
}
```

```
//A vizsgált elem metódus és az összes lehetséges visszatérési értékének
//típusa leszármazottja a definiált visszatérési érték típusnak
public class ReturntypeCriterion : ICriterion
{
  public bool Check(ICodeElement elementToAnalyze,
                    RecognitionContextBase context)
  {
    MethodBase m = elementToAnalyze as MethodBase;
    if (m == null)
    {
      throw new ArgumentException();
    }
    if (m.ActualReturnTypes==null
                || m.DeclaredReturnType==null || !m.ActualReturnTypes.Any())
    {
      return false;
    }
    return m.ActualReturnTypes
                 .All(at => at.Bases
                              .Any(atb => (NamedTypeBase)atb ==
                               (NamedTypeBase)m.DeclaredReturnType));

  }
}
```

Fontos, hogy az ICriterion interfész implementációk a paramétereikből csak olyan információkat használnak, amelyek még a programnyelv-független modell szintjén vannak.

A PatternRecognizerBase ősosztály egy minta felismerésének absztrakciója. Tartalmaz egy listát, amelyben ICriterion implementációk vannak, a bool IsInstanceOf(ICodeElement codeElement) metódusban pedig ebben a listában tárolt kritériumokat futtatja végig. Konstruktorának bemeneti paramétere egy RecognitionContextBase példány, ami a modell elemeit tartalmazza – később ez az objektum kerül továbbadásra az egyes ICriterion implementációknak. Ha minden kritérium igaz, akkor a kódelem a tervezési minta egy példánya.

```
public abstract class PatternRecognizerBase
{
  protected List<ICriterion> Criteria { get; private set; }

  public PatternRecognizerBase(RecognitionContextBase context)
  {
    this.Context = context;
  }

  public RecognitionContextBase Context { get; set; }

  public virtual bool IsInstance(ICodeElement elementToAnalyze)
  {
    foreach (var criterion in this.Criteria)
    {
      if (!criterion.Check(elementToAnalyze, Context))
      {
        return false;
      }
    }
    return true;
  }

}
```

4.4 C# forráskód beolvasása

A következőkben az 1.3 fejezet fogalmaira építve bemutatom, hogyan oldottam meg néhány, a munkám során felmerülő problémát a .NET Compiler Platform segítségével.

Az érthetőség és átláthatóság kedvéért a bemutatott forráskód-részletek nem tartalmazzák a különböző robosztussági ellenőrzéseket és a folyamat naplózásához használt hívásokat.

4.4.1 Valódi visszatérési érték típusok meghatározása

Az öröklés miatt egy metódus nem csak a definiált visszatérési típusának megfelelő objektumot adhat vissza, hanem bármit, ami ebből a típusból leszármazik. Ez az információ a Factory Method tervezési minta felismeréséhez nélkülözhetetlen. Az alábbi kódrészlet tárja fel ezeket az információkat egy metódusban:

```
public static List<INamedType> GetActualReturnTypes(
                this IMethodSymbol method,
                List<NamedTypeBase> types, SemanticModel model)
{
  List<INamedType> returntypes = new List<INamedType>();
  //a metódus definiálásnak helye a szintaxisfában
  foreach (var location in method.Locations)
  {
    //ha forráskódban van a deklaráció
    if (location.IsInSource)
    {
    //A szintaxisfából kikeresem azt a részt,
    //amit meghatároz a metódus helye (SourceSpan)
    var ms = model.SyntaxTree.GetRoot()
             .FindNode(location.SourceSpan) as MethodDeclarationSyntax;

    //Ha a metódus nem üres
    if (ms != null && ms.Body != null && ms.Body.Statements.Count != 0)
    {
       //megvizsgálom a metódusban a vezérlést
       //az első és az utolsó utasítás között
       var cf = model.AnalyzeControlFlow(ms.Body.Statements.First(),
                                         ms.Body.Statements.Last());

      //a ReturnStatements gyűjtemény visszaadja az összes
      //return utasítást a vizsgált szakaszban
      foreach (var returnStatement in cf.ReturnStatements)
      {
        var ret = (ReturnStatementSyntax)returnStatement;

       //null és void kifejezések kiszűrése
       if (ret.Expression != null
          && ret.Expression.CSharpKind() != SyntaxKind.NullLiteralExpression)
       {
       //meghatározom, hogy milyen típusú lesz a kifejezés végeredménye
       var type = model.GetTypeInfo(ret.Expression);

       //átalakítjuk a szimbólumot modellelemmé
       var rtype = types.Single(t => t ==
                  ((ITypeSymbol)(type.Type)).ToCSharpNamedType(logprovider));

       returntypes.Add(rtype);
       }
      }
     }
   }
  }
  return returntypes;
}
```

Ez a kódrészlet jól mutatja a Compiler Platform két fontos szolgáltatását: az átjárást a szemantikus modell és a szintaxisfa között (mindkét irányban) illetve a kód egy részletéhez tartozó vezérlési folyamok feltérképezését.

A bemeneti paraméterként megkapott IMethodSymbol egy szimbólumot reprezentál a szemantikus modellben. Ennek Locations tulajdonsága egy listát ad vissza, ami megadja, hogy a forráskód mely fájljainak szintaxisfájában vannak definiálva a metódusok (azért gyűjtemény a tulajdonság, mert a C# nyelv támogatja a parciális metódusok koncepcióját, így akár több részletben is definiálható egy metódus). Ennek a helyjelölőnek a SourceSpan tulajdonságából tudtam meghatározni azt, hogy a fájlban hol található a deklaráció (karakterszámtól karakterszámig).

A szemantikus modell SyntaxTree tulajdonsága visszaadja azt a szintaxisfát, amiből a modell készült, ennek GetRoot() metódusa megadja a fa gyökerét. A FindNode() metódus pedig megadja a paraméterként kapott hely (karakterszámtól karakterszámig) által definiált szintaxisfabeli elemet. Ennek a metódusnak adtam át a korábban kinyert értéket és így megkaptam, hogy a metódusszimbólumnak melyik szintaxisfabeli elem felel meg.

Az AnalyzeControlFlow metódust használtam arra, hogy a metódus első és utolsó utasítása között előállítsam a vezérlési gráfot (az első és utolsó utasításokat szintén a Compiler Platform API-jának segítségével határoztam meg).

Ebből a vezérlési gráfból kerestem meg a visszatérő utasításokat (ReturnStatements) a szintaxisfában. Ezután pedig a szemantikus model GetType() metódusával meghatároztam a kifejezés értékének megfelelő típusszimbólumot. Ez adja az átjárási lehetőséget a szintaxisfa és a szemantikus modell elemei között.

4.4.2 Interfész metódusok meghatározása

A tervezési mintáknak egyik kulcseleme az öröklődés és a többalakúság, amelyeket részben a metódusok felülírásával valósíthatunk meg. A legtöbb minta épít erre valamilyen formában, így a metódusok ezen aspektusát fontos feltárni. Az ősosztályok felülírt metódusait egyszerűen megkapjuk a metódusszimbólum egyik tulajdonsága alapján (IMethodSymbol.Override), de az interfészek metódusainak implementációit csak több API híváson keresztül érhetjük el. A 4.2 fejezet megfontolásai alapján erre az információra is szükségünk van.

```
private static void BuildInterfaceImplementationList(
                        List<NamedTypeBase> types,
                        IEnumerable<IMethodSymbol> methodSymbols,
                        Dictionary<ISymbol, SemanticModel> model)
{
  foreach (var methodSymbol in methodSymbols)
  {
    //Compiler Platform API: A DeclaringSyntaxReferences gyűjtemény
    //megadja a szimbólum definiálásának helyét a szintaxisfában

    //Ha a metódust reprezentáló szintaxisfa elem szűlője
    //a kódban egy interfészdeklaráció…
    if (methodSymbol.DeclaringSyntaxReferences.Length > 0 &&
            methodSymbol.DeclaringSyntaxReferences[0]
                        .GetSyntax().Parent.CSharpKind() ==
                                    SyntaxKind.InterfaceDeclaration)
    {

       //Az összes típuson végigmegyünk a modellben,
       //ami implementálja ezt az interfészt
       foreach (var sType in types.Where(t => t.Bases != null
            && t.Bases.Any(b => b.Name == methodSymbol.ContainingType.Name)))
       {
         var type = sType as CSharpNamedType;

         // megkeressem, hogy a SourceSymbol által reprezentált szimbólumban
         //melyik metódus implementálja a methodSymbol
         //által reprezentált metódust
         var implementor = type.SourceSymbol
                         .FindImplementationForInterfaceMember(methodSymbol);

         //Megkeressük a modellben az adott metódust
         //és override metódusként beállítjuk
         var parentType = types.Single(t => t ==
                                implementor.ContainingType
                                .ToCSharpNamedType(logprovider))
         SemanticModel mo = null;
         if (model.TryGetValue(implementor, out mo)
             || model.TryGetValue(implementor.ContainingSymbol, out mo))
         {
           var o = type.Methods
               .Single(m => (MethodBase)m == ((IMethodSymbol)implementor)
               .ToCSharpMethod(types, mo, parentType, logprovider));
           o.IsOverride = true;

         }
       }
    }
  }
}
```

Ez a kódrészlet bemutatja a szemantikus modell és a szintaxisfa elemei közötti átjárás egy másik lehetőségét. A DeclaringSyntaxReferences gyűjtemény visszaadja azokat a szintaxisfabeli elemeket, amik a szimbólumhoz tartoznak. Ennek természetesen csak akkor lesznek elemei, ha forráskódban van definiálva a szimbólum, és nem egy külső osztálykönyvtárban. Ez egy interfész-metódus esetén mindig az interfészben lévő

deklarációra mutat, nem pedig az egyes implementálókban lévő felülírásokra. Így lényegében a feladatom az volt, hogy egy adott metódus esetén, ha annak deklarációja interfészbe mutat, akkor megkeressem az összes típust, ami implementálja ezt az interfészt és megkeressem a felülíró interfészmetódust az implementáló típusban.

Az implementáló típusok megkeresése a saját modellemen történik, a felülíró metódust pedig a FindInterfaceImplementationForMember() API hívással kapom meg. Ezt az implementáló típusnak megfelelő szimbólumon kell meghívni, átadni pedig azt a metódusszimbólumot kell, aminek implementációját keressük.

5 Eredmények elemzése

A következő fejezetek a tervezésiminta-felismerő keretrendszerem eredményeit mutatják be.

5.1 Felismert minták

A mintafelismerő helyességének értékeléséhez a 4.1.1 fejezetben bemutatott megközelítést választottam. Az egyes tervezési minták esetében megvizsgáltam a P-MARt projekt által a JUnit keretrendszerben felismert minták számát és helyét. Eldöntöttem, hogy valóban a tervezési minta egyik példányáról van-e szó. Ha igen, akkor megvizsgáltam, hogy az NUnit kódjában az ennek megfelelő kódelem létezik-e és itt is valódi pozitív-e, valamint hogy a saját keretrendszerem felismerte-e a tervezési minta egyik példányaként.

Természetesen a fordított szituációt is elemeztem. Megvizsgáltam, hogy az előző lépésben nem lefedett, általam felismert minták valóban minták-e az NUnit kódjában, van-e megfelelőjük a JUnit kódjában, és ott tervezési minták-e vagy sem.

A következő leírás tartalmazza az összehasonlítást a Singleton, a Composite és a Decorator tervezési mintákra. Mind a JUnit [35], mind az NUnit [36] forráskódja elérhető, így az elemzés referencia kedvéért tartalmazza az osztályok nevét és ahol szükséges, értelmezi az osztályok szerepét.

Elsőként a **Singleton** mintát vizsgáltam. A P-MARt projekt két példányt ismert fel a JUnit forráskódjában, ezek az Assert és a Version osztályok. Ezeket megvizsgálva azonban látható, hogy mindkét minta hamis pozitív: a konstruktoruk láthatósága korlátozott (bár az eredeti tervezési minta privátként köti meg a konstruktorokat, modern értelmezések védett láthatóságú konstruktort is megengednek), nem tartalmaznak saját magukból példányt, így tetszőleges számban példányosíthatóak. A Version osztálynak nincs megfelelője az NUnit-ban (a .NET keretrendszer a verzió jelölésére beépített megoldással rendelkezik), így ennek felismerése nem elemezhető tovább. Az Assert osztálynak viszont az NUnit.Framework.Assert osztály felel meg. Ez az NUnit esetében sem Singleton, és a saját módszerem nem is ismeri fel, az én eredményem tehát valódi negatív.

A saját megoldásom az NUnit forráskódjában felismert további egy Singleton példányt a ServiceManager osztályban. Ennek az osztálynak nincs megfelelője a JUnitban, de manuálisan megvizsgálva megbizonyosodtam róla, hogy valóban Singletonról van szó, tehát valódi pozitív az eredmény.

Második mintám a **Composite.** Itt a P-MARt megoldás a minta konkrét kompozit osztályát ismeri fel, míg saját megoldásom a komponens interfészt. Ezeknek összeegyeztetése után a P-MARt egyetlen Composite mintát talál a JUnit kódjában, a junit.framework.Test osztályban. Ezt megvizsgálva ez valóban Composite minta (bár a típusosság hiánya miatt ennek megállapítása nem egyértelmű). Ennek az osztálynak a megfelelője az NUnit-ban az Nunit.Framework.Test osztály, ami szintén kompozit (de a típusosság hiánya miatt ebben az esetben is csak a kommentezés alapján biztos, hogy valóban kompozit mintáról van szó). Ezt a saját megoldásom is felismerte, helyes eredményt adva.

Emellett az osztály mellett még két interfész, az IErrorParser és az IProjectConvert került felismerésre. Ezekre az erősen típusos minta illeszkedik, így ezeknek megítélése egyértelmű, valódi pozitív eredményekről van szó. Ezeknek nincs megfelelője a JUnit keretrendszerben.

A harmadik összehasonlított minta a **Decorator**. Ebben az esetben a konkrét dekorátor komponensek kerülnek felismerésre mind a P-MARt projekt, mind a saját módszerem által. A P-MARt projekt 4 dekorátort jelent, ezek a TestDecorator, RepeatedTest, TestSetup, TornDown osztályok. Ezek mind valódi pozitívak, de az NUnit-ban csak a TestDecorator osztálynak van megfelelője (a TestSetup és a TornDown osztályok funkcionalitását a .NET keretrendszer biztosítja). Ezt saját keretrendszerem is felismerte.

Ezen kívül még két osztály került dekorátorként felismerésre, a LongRunningOperationDisplay és a TipWindow. Mindkettő kielégíti a minta követelményeit, de mivel ezek a grafikus felület részeit írják le, így szorosan kapcsolódnak a .NET keretrendszerhez és nincs megfelelőjük a JUnit-ban.

Tervezési minta	Kódelem a JUnitban	P-MARt eredmény	NUnit megfelelő	Saját eredmény
Singleton				
	Assert	**hamis pozitív**	Assert	**valódi negatív**
	Version	**hamis pozitív**	N/A	
Composite				
	Test	**valódi pozitív**	Test	**valódi pozitív**
Decorator				
	TestDecorator	**valódi pozitív**	TestDecorator	**valódi pozitív**
	RepeatedTest	**valódi pozitív**	N/A	
	TestSetup	**valódi pozitív**	N/A	
	TornDown	**valódi pozitív**	N/A	

1. táblázat - Saját eredményeim és a P-MARt eredményeinek összehasonlítása

Tervezési minta	NUnit kódelem neve	Saját eredményem
Singleton	ServiceManager	**valódi pozitív**
Composite	IErrorParser	**valódi pozitív**
	IProjectConverter	**valódi pozitív**
Decorator	LongRunningOperationDisplay	**valódi pozitív**
	CPWindowsForms.TipWindow	**valódi pozitív**

2. táblázat - Általam felismert tervezési minták, amiknek nincs megfelelője a JUnitban

Az 1. táblázat mutatja a P-MARt által, a JUnitban felismert tervezési mintákat, a hozzájuk tartozó osztályokat, valamint saját felismerési eredményeimet a JUnit osztályainak megfelelő osztályokkal kapcsolatban. A 2. táblázat olyan, általam felismert tervezési mintákat és az osztályaikat tartalmazza, amiknek nincs megfelelője a JUnitban.

Az 1. és 2. táblázatok alapján látható, hogy felismerő módszerem a P-MARt projekt eredményeinek fényében jól ismeri fel az egyes tervezési mintákat, illetve a P-MARt által nem tartalmazott, .NET verzió-specifikus minták esetén is valódi pozitív eredményeket ad e három tervezési minta esetében. Vannak olyan kódelemek is, amiket a P-MARt hamis pozitív mintaként ismert fel, míg ennek NUnit megfelelőjét saját módszerem valódi negatívként detektálta.

A P-MARt projekt eredményei az irodalomban is közkedvelt összehasonlítása alapnak számítnak. Az alábbi táblázat darabszámok szintjén hasonlítja össze a P-MARt, a saját és néhány egyéb kutatócsoport munkáját (fontos, hogy a legtöbb cikk csak a darabszámokat közli eredményei tárgyalásakor), szintén a JUnit illetve NUnit keretrendszereken:

	P-MARt	[19]	[20]	[23]	[37]	Saját eredményem
Singleton	2	N/A	1	3	N/A	1
Composite	1	N/A	1	34	1	3
Decorator	4	13	N/A	31	1	3

3. táblázat - Saját eredményeim összehasonlítása az elemezet irodalom eredményeivel

A 3. táblázatból látható, hogy saját eredményeim az egyes irodalombeli munkáknál konzisztensebb módon követik a P-MARt eredményeit.

A fenti 3 tervezési minta mellett foglalkoztam még a Chain of Responsiblity és a Factory Method tervezési minták felismerésével.

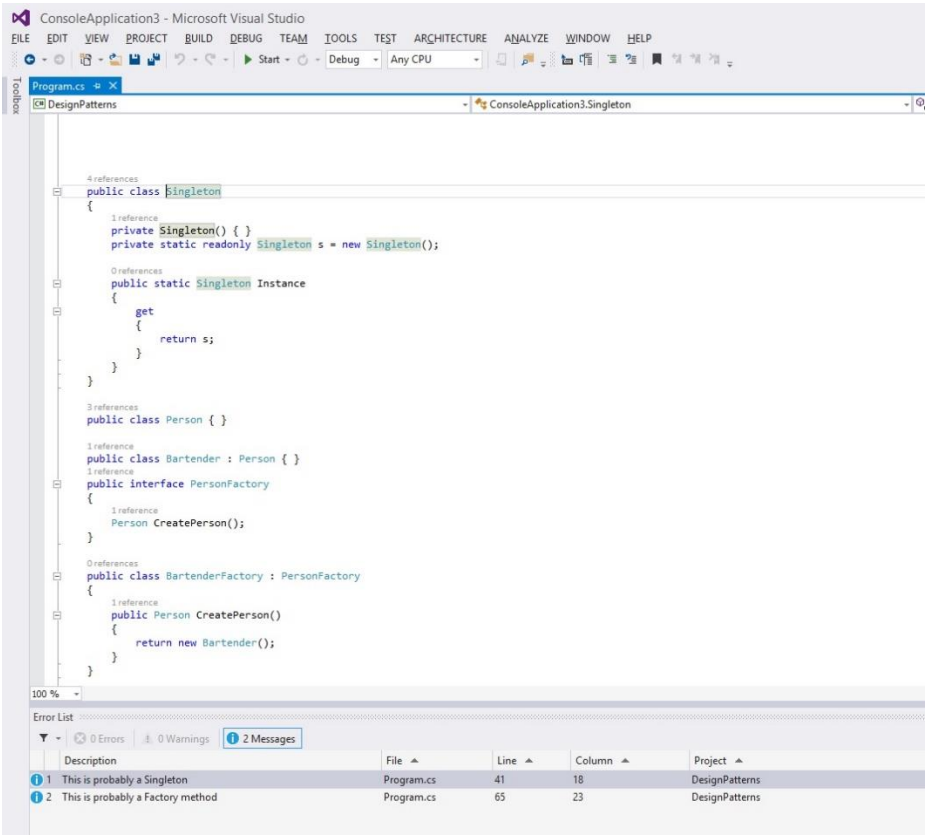
A **Chain Of Responsibility** mintának egy példányát találta meg a felismerő rendszerem az NUnit-ban (ez a Test osztály; itt a lánc ősét ismeri fel az algoritmus). A kódot átvizsgálva látszik, hogy ez egy valódi pozitív. A későbbi kutatások szempontjából ez is fontos eredmény, mert ezt a tervezési mintát az általam feldolgozott szakirodalom egyik cikke sem próbálta meg felismerni a JUnit keretrendszeren.

Szintén az NUnit keretrendszeren kerestem a **Factory Method** minta példányait. Az összesen 26 találatot az A Függelék: Factory Method minták az NUnit kódjában tartalmazza. Fontos megjegyezni, hogy minden felismert elem valódi pozitív volt. Tudomásom szerint nincs olyan korábbi kutatás, ami a Factory tervezési mintát a JUnit keretrendszer kódjában kereste volna, így az eredmények összehasonlítása itt nem lehetséges.

5.2 Visual Studio integráció

A .NET Compiler Platform a fordítási folyamatot kiajánló API-jára egyéb szolgáltatások is épülnek. Ezek segítségével a C# programozási nyelv etalon fejlesztőkörnyezetéhez, a Visual Studio-hoz is fejleszthetünk különböző beépülő modulokat, amelyek kódelemző funkciókat valósítanak meg.

Egy ilyen beépülő modulba építettem bele a tervezésiminta-felismerő C# implementációját is. A beépülő modul mindig a fejlesztőkörnyezet aktuálisan megnyitott fájlt dolgozza fel és keres benne tervezési mintákat (természetesen ehhez felhasználja a többi fájlban található kódelemek információit is).



10. ábra - Integráció a Visual Studio fejlesztőeszközbe

Ennek legfontosabb előnye, hogy a fejlesztők számára is egy felhasználóbarát, könnyen kezelhető módon jelenítjük meg a felismert tervezési mintákat. További előny, hogy a Visual Studio meglévő funkcionalitását is megkapjuk, így például a 10. ábra látható információs sávra kattintva automatikusan a felismert tervezési minta helyére ugrunk.

6 Fejlesztési lehetőségek

Folyamatos feladat a mintafelismerésben, hogy egyrészt a már megvizsgált minták felismerésének heurisztikáit tovább finomítsuk, másrészt további minták felismerését is implementáljuk. Ez, ahogyan a 4.2 fejezetben is bemutatásra került, a tervezési minták alapos elemzésével oldható meg.

Fontos lépés további programozási nyelvek beolvasásának és az általam definiált modellbe történő átalakításának a megvalósítása. Java nyelvű forráskódok elemzésével az eredmények könnyebben összevethetőek az irodalomban már ismert módszerekkel, míg például Visual Basic nyelvű kódok beolvasásával lehetőség nyílik arra, hogy olyan .NET platformra építő projekteket elemezzünk, amelyek egy része C#, másik része Visual Basic nyelven íródott.

Az egyes programnyelvű forráskódok elemzéséből sok érdekes következtetést lehet levonni. Néhány minta adott nyelven gyakori lehet, míg mások lényegében hiányozhatnak (ld.: 4.1.1 fejezet, Iterator és Observer minták). Ezek az információk segíthetnek az egyes programozási nyelvek előnyeinek és hátrányainak feltárásában, ezzel jobb képet nyerve arról, mely nyelvek milyen feladatokra alkalmasak.

Mivel az irodalom eddig kifejezetten a Java programozási nyelvre fókuszált, ezért munkám fontos lépés a C# programkódok ilyen irányú analízisében. Megvizsgálhatjuk, mely programozási nyelvek használtak és melyek kevésbé a nyelvben. Ennek aztán megvizsgálhatjuk az okait és esetlegesen javaslatokat tehetünk az általános programozási gyakorlatokra.

Az egyes tervezési mintákat akár projekttípusonként is megvizsgálhatjuk. Valószínűleg egyes minták bizonyos célú projektekben gyakrabban előfordulnak, míg másokban kevésbé. Ezzel szintén célzott programozási tanácsokat lehet adni a fejlesztőknek.

Végül, de nem utolsó sorban a tervezési minták mellett tetszőleges szoftveres minták implementációja megvalósítható. Így akár anti-patternek vagy code smell-ek detektálása, amelyeket az 5.2 fejezetben bemutatottakhoz hasonlóan akár az egyes fejlesztési környezetekbe is tudunk integrálni, így segítve a fejlesztőket.

7 Összefoglalás

Dolgozatomban a tervezési minták programozási nyelvtől független felismerését mutattam be. Kidolgoztam egyrészt egy nyelvfüggetlen adatmodellt a forráskód repzentációjára, másrészt egy olyan mintafelismerő-architektúrát, ami ezen a modellen fut és felismeri a tervezési mintákat.

Munkám során C# nyelven megírt forráskódot elemeztem. Ehhez az új, .NET Compiler Platform eszközt használtam. Dolgozatomban így elsőként a C# programozási nyelv fordítását és a .NET Copiler Platform eszközöket mutattam be.

Dolgozatom második fejezetében kitértem a tervezési minták használatának és felismerésének jelentőségére, valamint röviden áttekintettem a korábbi kutatásokat. Fontos felismerés, hogy az eddigi kutatások lényegében kizárólag Java nyelvű forráskódot elemeztek. Saját, C# nyelvet elemző dolgozatom így hiánypótló az irodalomban, és későbbi kutatások és eredmények validációjának alapja lehet.

Az áttekintő fejezetek után bemutattam a kidolgozott adatmodellt objektumorientált forráskódok reprezentálására az egyes elemek szükségességének indoklásával. Fontos eleme még a modellemnek a hívásgráf, ami szintén bemutatásra került. Végül kitértem az adatmodell implementációjára, amit C# nyelven végeztem el.

Ezután megindokoltam, hogy miért éppen a vizsgált tervezési mintákat választottam arra, hogy keretrendszerem eredményeit értékeljem. A kiválasztott tervezési minták és a kiválasztott eredmény-validációs módszer szorosan összefüggenek, így az is bemutatásra került. A kiválasztott minták elemzése után bemutattam a tervezésiminta-felismerő rendszer C# nyelvű implementációját. Szintén bemutattam néhány problémát, amivel a C# nyelvű forráskód beolvasása során találkoztam, és ezek megoldásának módszerét a .NET Compiler Platform segítségével.

Végül pedig értékeltem a keretrendszerem által produkált eredményeket és bemutattam, hogyan integráltam a mintafelismerőt egy, az iparban használt szoftverfejlesztő-környezetbe. A mintafelismerő keretrendszer eredményei – összehasonlítva korábbi kutatásokkal – ígéretesek, a forráskód reprezentálásra kidolgozott adatmodell, a tervezési minták elemzésének módszere és a mintafelismerő algoritmus mind alkalmasnak tűnnek arra, hogy komplex osztálykönyvtárakban tervezési mintákat keressünk ezzel a módszerrel.

Munkám nem csak azért fontos, mert az irodalomban még nem vizsgált C# nyelven keres tervezési mintákat. A tervezési minták felismerése forráskódban sokat segít azokban az esetekben, amikor valami miatt egy már meglévő rendszert újra kell tervezni, vagy vissza kell fejteni. Ez különösen az ősrendszerek (legacy systems) esetében hasznos, hiszen egyrészt pusztán ezeknek a rendszereknek a kora, másrészt a gyakran változó követelmények miatt nem mindig érhető el minden dokumentáció.

A tervezési minták a tudás újrafelhasználhatóságát és átadását is szolgálják, így egyfajta fejlesztői tudásbázis alapjául is szolgálnak. A minták felismerése forráskódban hozzájárul ennek a tudásbázisnak a tovább építéséhez: egyes tervezési mintákról kiderülhet, hogy bizonyos típusú rendszerekben gyakrabban használatosak, így a minták alkalmazhatóságának egy új, dokumentált lehetőségét felfedve. Ezzel nem csak a fejlesztői tudásbázishoz lehet hozzájárulni, hanem a szoftverek evolúcióját is elő lehet segíteni, így megteremtve a lehetőséget minőségi szempontból jobb termékek előállítására.

Irodalomjegyzék

[1] E. Gamma, R. Helm, R. Johnson és J. Vlissides, Design patterns: elements of reusable object-oriented software, Pearson Education, 1994.

[2] S. Alhusain, S. Coupland, R. John és M. Kavanagh, „Design Pattern Recognition by Using Adaptive Neuro Fuzzy Inference System” *2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI),* , 2013.

[3] H. Zhu, I. Bayley, L. Shan és R. Amphlett, „Tool Support for Design Pattern Recognition at Model Level” *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, 2009.

[4] „.NET compiler platform,” [Online]. Elérhető: http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx. [Hozzáférés dátuma: 2014. október 18.].

[5] S. Chidamber és C. Kemerer, „A metrics suite for object oriented design,” *IEEE Transactions onSoftware Engineering, ,* 20. kötet, 6. szám, 476-493 oldal, 1994 június.

[6] M. Lorenz és J. Kidd, Object-oriented software metrics: a practical guide, Prentice-Hall, Inc., 1994.

[7] R. Martin, „OO design quality metrics” *An analysis of dependencies,* 1994.

[8] B. Boehm, Software risk management, Berlin: Springer, 1989.

[9] A. Ampatzoglou, G. Frantzeskou és I. Stamelos, „A methodology to assess the impact of design patterns on software quality” *Information and Software Technology,* 54. kötet, 4. szám, 331-346. oldal, 2012.

[10] B. Huston, „The effects of design pattern application on metric scores” *Journal of Systems and Software,* 58. kötet, 3. szám, 261-269 oldal, 2001.

[11] N.-L. Hsueh, P.-H. Chu és W. Chu, „A quantitative approach for evaluating the quality of design patterns” *Journal of Systems and Software,* 81. kötet, 8 szám, 1430-1439 oldal, 2008.

[12] I. Sommerville, Software Engineering. International computer science series, Addison Wesley, 2004.

[13] L. H. Rosenberg és L. E. Hyatt, „Software re-engineering,” *Software Assurance Technology Center,* 2-3. oldal, 1996.

[14] Y.-G. Guéhéneuc, „P-mart: Pattern-like micro architecture repository” *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories,* 2007.

[15] „P-MARt Home,” [Online]. Elérhető: http://www.ptidej.net/tools/designpatterns/. [Hozzáférés dátuma: 2014 október 18.].

[16] D. Heuzeroth, T. Holl, G. Hogstrom és W. Lowe, „Automatic design pattern detection”, *11th IEEE International Workshop on Program Comprehension, 2003.*, 2003.

[17] F. Arcelli és L. Cristina, „Enhancing Software Evolution through Design Pattern Detection” *2007 Third International IEEE Workshop on Software Evolvability,* 2007.

[18] „WEKA Home,” [Online]. Elérhető: http://www.cs.waikato.ac.nz/ml/weka/. [Hozzáférés dátuma: 2014. október 18.].

[19] G. Kniesel és A. Binun, „Standing on the shoulders of giants - A data fusion approach to design pattern detection” *2009. ICPC '09. IEEE 17th International Conference on Program Comprehension,* 2009.

[20] M. L. Bernardi és G. A. Di Lucca, „Model-driven detection of Design Patterns”*2010 IEEE International Conference on Software Maintenance (ICSM), ,* 2010.

[21] A. Pande, M. Gupta és A. Tripathi, „DNIT - A new approach for design pattern detection”, *2010 International Conference on Computer and Communication Technology (ICCCT),* 2010.

[22] Y. Dongjin, J. Ge és W. Wu, „Detection of design pattern instances based on graph isomorphism”, *2013 4th IEEE International Conference on Software Engineering and Service Science (ICSESS),* 2013.

[23] M. Bernardi, M. Cimitile és G. Di Lucca, „A model-driven graph-matching approach for design pattern detection”, *2013 20th Working Conference on Reverse Engineering (WCRE),,* 2013.

[24] F. Arcelli, D. Franzosi és C. Raibulet, „.NET Reverse Engineering with MARPLE”, *2010 Fifth International Conference on Software Engineering Advances (ICSEA),* 2010.

[25] W. Ren és W. Zhao, „An observer design-pattern detection technique,” *2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE),* 2012.

[26] A. Asadullah, M. Basavaraju, I. Stern és V. Bhat, „Design Patterns Based Pre-processing of Source Code for Plagiarism Detection”, *2012 19th Asia-Pacific Software Engineering Conference (APSEC),* 2012.

[27] J. Smith és D. Stotts, „SPQR: flexible automated design pattern extraction from source code”, *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003.*. 2003.

[28] J. Smith and D. Stotts, "Extending SPQR to Architectural Analysis by Semi-Automated Training", *WICSA 2005. 5th Working IEEE/IFIP Conference on Software Architecture, 2005.*, 2005.

[29] M. Abadi, A theory of objects, springer, 1996.

[30] C. Jebelean, „Automatic detection of missing abstract-factory design pattern in object-oriented code”, *Proceedings of the International Conference on Technical Informatics*, 2004.

[31] S. Alhusain, S. Coupland, R. John és M. Kavanagh, „Towards machine learning based design pattern recognition”, *2013 13th UK Workshop on Computational Intelligence (UKCI),* 2013.

[32] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides és S. Halkidis, „Design Pattern Detection Using Similarity Scoring,” *IEEE Transactions on Software Engineering,* 32. kötet, 11. szám, 896-909 oldal, 2006 november.

[33] N. Pettersson, W. Lowe és J. Nivre, „Evaluation of Accuracy in Design Pattern Occurrence Detection” *IEEE Transactions on Software Engineering, ,* 36. kötet, 4. szám, 575-590. oldal, 2010 július.

[34] L. Fulop, R. Ferenc és T. Gyimothy, „Towards a Benchmark for Evaluating Design Pattern Miner Tools” *CSMR 2008 12th European Conference on Software Maintenance and Reengineering,* 2008.

[35] „JUnit Home,” [Online]. Elérhető: http://junit.org/. [Hozzáférés dátuma: 2014. október 18.].

[36] „NUnit Home,” [Online]. Elérhető: http://www.nunit.org/. [Hozzáférés dátuma: 2014. október 18.].

[37] J. Dong, Y. Sun és Y. Zhao, „Design pattern detection by template matching”, *Proceedings of the 2008 ACM symposium on Applied computing,* 2008.

A Függelék: Factory Method minták az NUnit kódjában

A saját mintafelismerőm az alábbi metódusokban ismerte fel a Factory Method tervezési mintát:

- NUnitCoreBuilders.CombinatorialStrategy.GetTestCases
- NUnitCoreBuilders.SequentialStrategy.GetTestCases
- NUnitCoreBuilders.PairwiseStrategy.GetTestCases
- NUnitCoreBuilders.DatapointProvider.GetDataFor
- NUnitCoreBuilders.LegacySuiteBuilder.BuildFrom
- NUnitCoreBuilders.TestCaseParameterProvider.GetTestCasesFor
- NUnitCoreBuilders.TestCaseSourceProvider.GetTestCasesFor
- NUnitCoreBuilders.ValueSourceProvider.GetDataFor
- NUnitCoreExtensibility.TestCaseProviders.GetTestCasesFor
- NUnitCoreExtensibility.TestCaseProviders.GetTestCasesFor
- NUnitCoreExtensibility.DataPointProviders.GetDataFor
- NUnitCoreExtensibility.DataPointProviders.GetDataFor
- NUnitCore.DomainAgent.CreateRunner
- NUnitUtil.MultipleTestDomainRunner.CreateRunner
- NUnitUtil.MultipleTestProcessRunner.CreateRunner
- NUnitUtil.InProcessTestRunnerFactory.MakeTestRunner
- NUnitUtil.MemorySettingsStorage.MakeChildStorage
- NUnitUtil.RegistrySettingsStorage.MakeChildStorage
- NUnitUtil.RemoteTestAgent.CreateRunner
- NUnitCoreTests.RemoteRunnerTests.CreateRunner
- NUnitCoreTests.SimpleTestRunnerTests.CreateRunner
- NUnitCoreTests.ThreadedTestRunnerTests.CreateRunner
- NUnitUtilTests.ProcessRunnerTests.CreateRunner
- NUnitUtilTests.TestDomainRunnerTests.CreateRunner
- NUnitMocks.MockInterfaceHandler.Invoke
- NUnitProjectEditor.ProjectModel.AddConfig