Programok Struktúrális Bonyolultsági Mérőszámai

Porkoláb Zoltán

Eötvös Loránd Tudományegyetem
Természettudományi Kar
Informatikai Tanszékcsoport
Általános Számítástudományi Tanszék

Doktori értekezés

Budapest, 2002.

Témavezető: *Dr. Tőke Pál tudományos főmunkatárs*
Doktori program: **Informatika**
Programvezető: *Dr. Demetrovics János akadémikus*

Tartalomjegyzék

1. Bevezetés

Ebben a dolgozatban programozási nyelvek strukturális bonyolultságára definiálunk mértéket. A mérték egyaránt alkalmazható imperatív és objektum-orientált programozási nyelvekre, és az alkalmazott programozási paradigmától függetlenül méri a program bonyolultságát.

A szoftver életciklus egyre kisebb részét foglalja el az új rendszer specifikálása, megtervezése és implementálása. A fejlesztési idő nagyon nagy része, gyakran akár fele telik a teszteléssel, és az ott feltárt hibák javításával [2]. A szoftverre az élettartama alatt költött költség legnagyobb része pedig a rendszer karbantartásával kapcsolatos [3]. A szoftver tesztelésének, utólagos karbantartásának és módosításának költsége nagyrészben a program strukturális bonyolultságának függvénye. Egy jól használható szoftver bonyolultsági mérték még a fejlesztési fázisban felfedheti a szoftver kritikus pontjait, segítheti az áttekinthető és módosítható kód készítését, és becsléseket adhat a fejlesztési folyamat várható költségeire.

A szakirodalom számos módszert ismer programok strukturális bonyolultságának mérésére. A program bonyolultságát kiszámíthatjuk a programban szereplő operátorok és operandusok alapján (software science measure) [17], a predikátumok száma alapján (cyclomatic complexity) [30], vagy pl. az adatáramlás (data-flow) [21] alapján. Ezeket a mértékeket procedurális programok vizsgálatára vezették be, és a tapasztalatok szerint elsősorban azokon működnek kielégítően. Speciálisan az objektum-orientált programok bonyolultságát gyakran határozzák meg éppen bizonyos – csak az objektum-orientált módszertanra jellemző – mérőszámokkal [31]. Ezek a mérőszámok, (mint az osztályok attributumainak száma, az öröklési hierarchia mélysége, az egyes osztályok közötti kapcsolatok - binding), valóban jól reprezentálják az objektumelvű programok bonyolultságát, de többnyire nem alkalmazhatóak a hagyományos procedurális programokra.

Ugyanakkor a modern programozási nyelvekben egyre inkább előtérbe kerül a programozási paradigmák együttes alkalmazása [6]. Ezek a nyelvek egyrészt továbbra is erős procedurális elemekkel rendelkeznek. Emellett új, egyre inkább terjedő technikákat találunk, melyek legtöbbször ortogonálisak a hagyományos procedurális elvekre. Ezek között ma az egyik legelterjedtebb az *objektum-orientált programozás*, amely a feladat felbontásakor nem az alprogramok elsődlegességét, hanem az adatszerkezetek és a rajtuk végzett műveletek egyenrangúságát hirdeti. További új programozási paradigmaként jelenik meg a *generatív programozás* [8] és az *aspektus-orientált programozás* [25]. Mindkettő a procedurális és objektum-orientált alapok felett valósul meg; kiegészítve, de nem helyettesítve azokat.

Úgy véljük, egy adekvát bonyolultsági mérték a fentiek alapján nem kötődhet *egyetlen* programozási paradigmához. Olyan alapvető nyelvi elemeket, és konstrukciós szabályokat kell ehelyett felhasználnunk, amelyeket egyaránt alkalmazhatunk

az eltérő paradigmákra is. Ennek segítségével olyan mértékhez jutunk, melyet hatékonyan használhatunk fel a jelenlegi, tipikusan multiparadigmás programok mérésére. Az eddig alkalmazott mértékek legtöbbje egyoldalúan: vagy csak a program vezérlőszerkezetének, vagy csak az adatáramlásnak szentelt figyelmet. Javasolt mértékünk ezzel szemben három tényező együttesén alapul:

- A program vezérlőszerkezetének bonyolultsága. Miután a legtöbb mai program alapvetően ugyanazon vezérlőszerkezetek segítségével épül fel, mint a hagyományos procedurális programok[1], nemegyszer ez a vezérlőszerkezet határozza meg a program bonyolultságának jelentős részét.
- A használt adattípusok bonyolultsága. Az elmúlt évtized programozási elvei az egymásra épülő komplex adattípusok definiálásán alapulnak. Egy objektum-orientált könyvtár például különböző bonyolultságú osztályok (adattípusok) gyűjteménye. E típusok definíciója – explicite vagy implicite – módon része a programoknak, így komplexitásukat figyelembe kell venni a teljes program bonyolultságának kiszámításakor.
- Az adatkezelés komplexitása. E pont azt vizsgálja, milyen kapcsolat áll fenn a vezérlőszerkezet és a kezelt adatok között. Az adat a programban számos helyen kerülhet kapcsolatba a vezérlőszerkezettel. Létezhetnek globális és lokális adatok. A predikátumok olvassák, az értékadó utasítások írják az adatokat. Adatok a függvények, eljárások paraméterei is. Nyilvánvalóan más a bonyolultsági hatása egy read-only adatnak (például konstans, vagy érték szerint átadott paraméter), mint amit az adott alprogram meg tud változtatni, és ezen keresztül vissza tud hatni a hívó kódra.

1.1. A metrika definiálása

A metrikát absztrakt matematikai formában kívánjuk definiálni. Ehhez alapként a J. Howatt és A. Baker által bevezetett [22] szigorú formalizmust használjuk fel. Az adatok kezelésének módjánál a Fóthi–Nyéki féle *data-flowgraph* elvéből indulunk ki [15][49]. Az eredeti data-flowgraph elvét ugyanakkor megváltoztatjuk, finomítva az adatkezelés módját, megkülönböztetve az adatok input ill. output elérését.

Megmutatjuk, hogy ez az új metrika megfelelően működik a hagyományos, procedurális programok esetében. A mérték jól tükrözi a programok dekompozíciójánál tapasztalt bonyolultság-csökkenést is. Indokoljuk az adatkezelés kétirányú definiálásának szükségességét.

A további lépések során kiterjesztjük a mértéket az objektum-orientált programozási paradigmára. Az objektumelvű programok alapvető egysége az osztály.

[1] Talán az egyetlen figyelemreméltó újítás a *kivételkezelés* (exception-handling) széleskörű elterjedése

Definiáljuk az osztály bonyolultságát. Megmutatjuk, hogy a hagyományos adatstruktúrák és (procedurális) függvénykönyvtárak hogyan állnak elő, mint e definíció speciális esetei. Tárgyaljuk az enkapszuláció közvetlen és közvetett hatását a program bonyolultságára. Megmutatjuk, hogy az öröklődés és az aggregáció hogyan származtatható a metrika általános (nem objektumelvű) definíciójából. Megvizsgáljuk néhány nevezetes objektumelvű konstrukció komplexitását.

E feladatok során elsődleges célnak azt tekintjük, hogy alapvető, programozási paradigmáktól független eszközrendszert használjunk, és ezekből le lehessen vezetni az egyes speciális (procedurális, objektumelvű) konstrukciókat és azok bonyolultsági mérőszámait. Szeretnénk megmutatni, hogyan függenek (származtathatók) az egyes speciális esetekben alkalmazott mérőszámok az általános konstrukciókon definiált mértéktől.

1.2. A metrika validálása

A metrika validálása minden bonyolultsági mérték alapvető része. Ez a validálás két úton történhet: elméleti úton – a mérték viselkedésének elemzésével, és gyakorlati úton – a mérték valós programokra történő alkalmazásán keresztül. A dolgozat célja, hogy mindkét validálást elvégezze.

Az elméleti validálás első lépéseként megmutatjuk, hogy a bevezetett új a metrika teljesíti a Weyuker axiómák legtöbbjét [43]. Az Elaine Wuyeker által bevezetett 10 logikai állítás („axióma”) a program bonyolultsági mértékek általánosan elfogadott feltételei. Habár megmutatható, hogy az axiómák nem *elégséges* feltételek [6], az egyes állítások teljesülése komoly érv a metrika alkalmazhatósága mellett.

2. A dolgozat felépítése

Az elmúlt negyed évszázadban számos cikk foglalkozott szoftvertermékek bonyolultsági kérdéseivel. Ezen kutatások eredményeként olyan mértékek születtek, melyek egy- vagy több dimenzió mentén hatékonyan mérik a programok bonyolultságát. Hiba lenne elvetni az eddigi eredményeket, és teljesen új alapokról kezdeni egy metrika definiálását, még akkor is, ha időközben olyan fejlődés ment keresztül a szoftverfejlesztési folyamatban, ezen belül a programozási nyelvek területén is, melyek az eddigi eredményeket részben időszerűtlenné, de legalábbis újraértelmezendővé teszik.

Ebből a célból a *Szoftver mértékek áttekintése* c. fejezetben áttekintést adunk egyes szoftver metrika típusokról. Az általános jellemzés mellett részletesen elemezzük azokat az eredményeket, melyekre a jelen dolgozatban hivatkozunk.

Ezen munkák közül elsőnek Thomas J. McCabe ciklometrikus bonyolultságát vizsgáljuk meg részleteiben. McCabe [30] elsősorban Fortran programokat vizsgált, és azok kódminőségét szerette volna javítani mértéke segítségével, ajánlásokat dolgozva ki az egyes kódmodulok maximális bonyolultsági számára. Miután elsősorban a tesztelés költségeit tartotta szem előtt, McCabe mértéke a lehetséges tesztelési útvonalak számával kapcsolatos, és teljesen elhanyagolja a program vezérlő szerkezetén kívüli elemeit.

McCabe metrikájának másik komoly hiányossága, hogy nem veszi figyelembe az egyes utasítások beágyazottsági mértékét a bonyolultság meghatározásakor. Több javaslat született e hiányosság javítására, ezek közül matematikailag az egyik legprecízebb Howatt és Baker [22] javaslata. Az általuk bevezetett beágyazottsági mérték nem struktúrált programokra is értelmezhető és egyik alapjául szolgál a jelen dolgozatban definiált mértéknek is.

Számos mérték született a program nem-vezérlési szerkezetére koncentrálva is. Miután a jelen dolgozatban definiált mérték fokozottan figyelembe veszi a program adatstruktúráját és az adat áramlását, részletesen foglalkozunk Henry és Kafura információáramláson alapuló mértékével [21]. Dolgozatukban a szerzők amellett érvelnek, hogy a program bonyolultságát a program lehetséges információáramlási irányai határozzák meg. Információáramláson részben a modulok közötti adatáramlást, részben a vezérlés átadását értették.

A javasolt mérték c. fejezetben javaslatot adunk egy új, a programstruktúrán alapuló bonyolultsági mértékre, amely (csak) a program elemi vezérlési szerkezetétől, az adatszerkezetétől és az adatok és a vezérlési szerkezet kapcsolatától – az adatkezeléstől függ. E mérték olyan relatíve alacsony szintű programozási fogalmakon alapul, mint a predikátumok, szekvenciák, adatok írása és olvasása; és definiálása programozási nyelvektől független módon történik.

Az alapvető definíciók és jelölésrendszer c. alfejezetben bevezetünk néhány jelölést, amely mértékünk matematikai modellje alapjául szolgál. *A program vezérlési szerkezete* c. alfejezetben – jórészt Howatt és Baker, Piwowarski, valamint Harrison és Magel cikkei nyomán – definiáljuk a program vezérlési struktúráját, és annak bonyolultságát. Az összetett matematikai apparátus bevezetésének célja, hogy a vezérlési szerkezet bonyolultsága tükrözze a beágyazottsági szintet, nem strukturált programok esetében is.

Az alprogramok szerepe c. alfejezetben kiterjesztjük az összefüggő vezérlési szerkezetekre definiált mértéket egymást hívó alprogramok csoportjára is. Bevezetjük a programgráf fogalmát, Fóthi Ákos és Nyékyné Gaizler Judit munkáira alapozva. *Az adatok szerepe* c. alfejezetben bevezetjük az AV gráf fogalmát, ami az adatszerkezetek és az adatkezelés szerepének bonyolultságával egészíti ki a vezérlő szerkezetet. Példákon keresztül mutatjuk be, hogyan viselkedik az AV gráf bonyolultsági mértéke a program dekompozíciós átalakításaira.

A mai vezető programozási módszertan az objetum-orientált technológia. *Az osztály bonyolultsága* c. fejezetben bemutatjuk, hogy az előzőekben definiált AV gráf természetes módon kiterjeszthető objektum-orientált programokra, illetve az osztály bonyolultságának definiálására.

A metrika axiomatikus validálását a *Weyuker axiómák* vizsgálatával végezzük. Tisztázzuk az Elaine Weyuker által felvázolt logikai állítások szerepét, majd megvizsgáljuk a kilenc logikai állítás teljesülését. Végül összehasonlítjuk metrikánk viselkedését néhány neves bonyolultsági mértékkel.

Elkészült az e dolgozatban javasolt mérték Java forrásprogramokon történő meghatározását végző program. E program segítségével módunkban áll a gyakorlatban is tesztelni mértékünket. A *Mérési eredmények* c. fejezetben beszámolunk a mérték egy gyakorlati használatáról, és a kapott eredményeket összehasonlítjuk más metrikák eredményeivel.

3. A szoftver mértékek áttekintése

Egy szoftver mérték mérheti a fejlesztés során kialakult végterméket, ilyenkor termék mértékekről (product metrics) beszélünk. A termék mérték alkalmazható a fejlesztés bármely közbülső pillanatában is. Mérheti pl. a modell bonyolultságát, a kód méretét vagy akár a dokumentációt is.

Hasonlóan fontos lehet magának a fejlesztési folyamatnak a mérése is, ez utóbbiak a folyamat mértékek (process metrics). A folyamat mérték a termék létrehozásához szükséges fejlesztési folyamat jellemzőit méri, (például a termék kifejlesztéséhez szükséges ráfordítási időt) és/vagy a résztvevő fejlesztők (szükséges minimális) kvalifikációja alapján mér. A mérés történhet

- a fejlesztés költsége alapján (cost metrics)
- a szükséges kifejtendő erőfeszítés alapján (effort metrics)
- a projekt előrehaladása alapján (advancement metrics)
- a fejlesztési folyamat megbízhatósága (hibái) alapján (process non-reliability metrics)
- a folyamatban elkészült termékek újrahasználhatósága alapján (reuse metrics)

A termék mértékek egy része a termék azon tulajdonságait méri, melyek a termék felhasználója számára is láthatóak. E felhasználó lehet egy programot használó egyén, egy osztálykönyvtárat felhasználó további kód, stb. Az ilyen, kívülről is látható tulajdonságokat megjelenítő mértékek a külső termék mértékek (external product metrics). Ezek lehetnek:

- termék megbízhatósági mértékek (product non-reliability metrics)
- funkcionalitási mértékek (functionality metrics)
- tejelsítmény mértékek (performance metrics)
- használhatósági mértékek (usability metrics)

Fontos ugyanakkor mérni azokat a tulajdonságokat is, melyek a felhasználó számára nem, vagy csak korlátozott mértékben látszanak. A belső termék mértékek (internal product metrics) ilyen tulajdonságokat mérnek, melyek a terméket fejlesztők számára fontos részleteken alapulnak. Ilyenek a

- méret mértékek (size metrics)
- bonyolultsági mértékek (complexity metrics)
- stílus mértékek (style metrics)

Amikor egy szoftver termék értékesíthetőségét vizsgáljuk, elsősorban a külső metrikák eredményei az érdekesek. A belső metrikák és folyamat metrikák abból a szempontból érdekesek, hogy minősítik azt a folyamatot ill. eszközrendszert, melynek keretei között a termék létrejött. Ideális esetben a termék létrehozásába befektetett erőfeszítés (melyet folyamat mértékkel mérünk) összhangban áll az így létrejött termék belső jellemzőivel (melyet a belső termék mértékkel mérjük), és a termék felhasználók számára mutatott használhatóságával (külső termék mérték).

Az alábbiakban áttekintünk néhány gyakran használt metrikát, melyekre e dolgozatban többször is fogunk hivatkozni.

3.1. Méret metrikák

A méret metrikák a terméket valamely fizikai tulajdonsága alapán jellemzik. Elsősorban a forráskód méretét, (gyakran hosszát) mérik. Ezek a mértékek nem veszik figyelembe a vezérlési szerkezeteket, az egyes sorok jelentését. A forráskódot szemantikailag nem, csak szintaktikai szempontokból elemzik.

Az egyik legegyszerűbb - és emiatt gyakran használt - mérték a forráskód sorainak számát méri, de nem veszi figyelembe az üres, vagy csak megjegyzést tartalmazó sorokat. Miután a mérték nem veszi figyelembe a sorok szemantikáját, ezért egyaránt alkalmazhatjuk struktúrált vagy objektum orientált programokra is. A szemantika teljes figyelmen kívül hagyása egyben a LOC gyenge pontjait is kijelöli.

A LOC némi javítására szolgál az eLOC (Effective Lines of Code) és az lLOC (Logical Lines of Code). A „tényleges sorok száma" az üres és a megjegyzés sorok mellett nem veszi figyelembe azokat a sorokat, melyek a kód szempontjából közömbösek. Ilyenek pl. a csak parancszárójeleket (begin, end), idézőjeleket, stb. tartalmazó sorok, melyeket a programozó a kód olvashatósága miatt szokott külön sorba helyezni.

A LOC metrikából kiindulva természetesen jellemezhetjük a programot a szavak, utasítások, alprogramok (függvények és eljárások) vagy akár a forrásfájlok számával is.

3.2. A McCabe féle ciklomatikus bonyolultság

A struktúrális bonyolultsági metrikák egyik alapvető kiinduló pontja a Thomas J. McCabe által 1976-ban javasolt bonyolultsági mérték [30], amit szokás ciklomatikus (cyclomatic) bonyolultságnak is nevezni. McCabe Fortran programokat vizsgált és célja annak kutatása volt, hogyan lehet egy szoftverrendszert úgy modularizálni, hogy az a legkedvezőbb legyen a tesztelhetőség és karbantarthatóság szempontjából. A fejlesztési idő nagyon nagy része, McCabe szerint gyakran akár fele telik a

teszteléssel, és az ott feltárt hibák javításával [2]. A szoftverre az élettartama alatt költött költség legnagyobb része pedig a rendszer karbantartásával kapcsolatos [3]. McCabe szándéka olyan matematikai technika kidolgozása volt, amely számszerűleg fejezi ki az egyes modulok bonyolultságát, így azonosítva a kód kritikus részeit; azokat, amelyek a tesztelés vagy karbantartás során extra költséget jelenthetnek. Kifejti, hogy az akkor használatos modularizációs technikák a kód méretének a korlátozására szorítkoznak. Az IBM pl. alprogramonként legfeljebb 50 kódsor alkalmazását javasolta. McCabe rámutat, hogy egy 50 soros FORTRAN program 25 egymást követő **IF THEN** konstrukciót tartalmazva 33.5 millió eltérő végrehajtási ágat produkál, amit az akkori (és mai) gyakorlati lehetőségek között képtelenség tesztelni.

Elgondolásása a fentieknek megfelelően a kód tesztelésének költségével (a lehetséges összes végrehajtási ág bejárásával) kapcsolatos. Miután az olyan programok, amelyekben akárcsak egyetlen olyan utasítás is létezik, amely a kód egy megelőző részére adja át a vezérléset (legyen az valamilyen ciklus konstrukció, vagy **GOTO** utasítás), végtelen sok végrehajtási ággal rendelkeznek, ezért csak azokkal a minimálisan szükséges elemi útvonalakkal számol, melyek kombinációiból a programon belül tetszőleges végrehajtási útvonal előállítható. Ennek megfelelően a bonyolultság mértékét az alapvető utvonalak (basic path) fogalma segítségével definiálja.

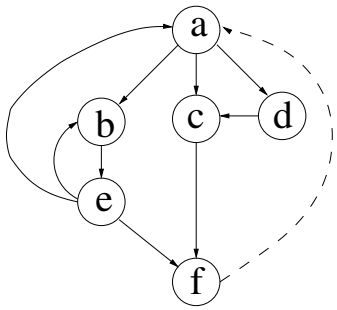
Definíció: Az n csomóponttal, e éllel és p komponenssel rendelkező G gráf $V(G)$ *ciklomatikus száma*:

$V(G) = e - n + 2p$

Tétel: Az erősen összefüggő G gráfban (azaz, ahol bármely két csúcs kötött létezik út) a *ciklomatikus szám* megegyezik a *lineárisan független körök* számával.

McCabe az alábbi módon használja fel a tételt. Egy adott programhoz hozzárendel egy olyan irányított gráfot, melynek egyetlen *belépési* ill. *kilépési* (entry/exit) csomópontja van. A gráf minden egyes csomópontja megfelel a program valamely szekvenciális blokkjának (elágazást, ciklust vagy ugró utasítást nem tartalmazó kódrészletének), míg az élek a vezérlésátadásoknak felelnek meg. (McCabe az elágazásokat és ciklusokat, mint vezérlési szerkezeteket - a FORTRAN szemléletnek megfelelően - ugró utasításoknak tekinti.) Minden irányított út a belépési és kilépési pont között egy lehetséges végrehajtási útvonalat (execution path) ábrázol. Ezt a gráfot Ledgard után [29] a program *vezérlési gráfjának* (control graph) nevezi, és feltételezi, hogy minden csomópont (azaz programblokk) elérhető a kezdőpontból, és minden csomópontból vezet út a végpontba.

Az alábbiakban egy vezérlési gráfra mutatunk példát. A kilépési csúcsból a belépési csúcsba vezető extra utat szaggatott vonallal ábrázoltuk.



A tételt McCabe a következőképpen alkalmazza G-re. Tegyük fel, hogy létezik él f csúcsból a csúcsba. Ekkor G erősen összefüggő (bármely két csúcsa között létezik útvonal). Ekkor a tételből következően

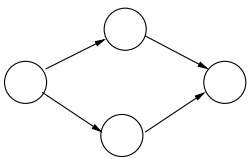
$$V(G) = e - n + 2p = 9 - 6 + 2 = 5$$

lineárisan független kör van G-ben. Legyen B egy ilyen maximális számú lineárisan független kör-halmaz. Ilyen B lehet pl. az alábbi:

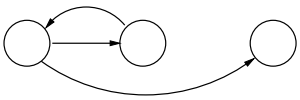
$$B = (abefa), (beb), (abea), (acfa), (adcfa)$$

B tartalmazza az összes kört G-ből és bármely G-beli útvonal kifejezhető az B-beli körök lineáris kombinációjaként, azaz B *bázisként* viselkedik. Természetesen választhatunk más körökből álló bázist is, azonban annak számossága is ugyanekkora lenne. Néhány tipikus példa a vezérlőszerkezetekre illusztrálja a McCabe mérték működését:

szekvencia, $v = 1 - 2 + 2 = 1$

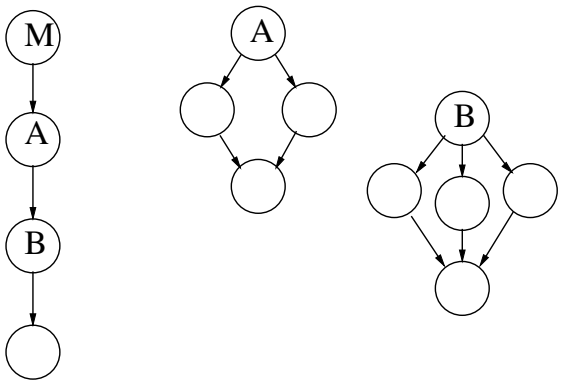


if-then-else vezérlőszerkezet, $v = 4 - 4 + 2 = 2$



while vezérlőszerkezet, $v = 3 - 3 + 2 = 1$

A $v = e - n + 2p$ formulában p értéke a G gráfbeli komponensek száma. Az eredeti feltevések (egyetlen belépési és kilépési csúcs, minden csúcs elérhető a belépési csúcsból, és minden csúcsból elérhető a kilépési csúcs) szerint minden programnak egyetlen komponensből álló összefüggő gráf felelne meg. Ésszerű azonban, ha a metrikát alprogramok halmazára is tudjuk értelmezni. Az alábbi példában az M főprogram két eljárást (A és B) hív meg.



Most $G = M \cup A \cup B$ és $p = 3$. Ezek alapján a ciklomatikus szám:

$$v(G) = v(M \cup A \cup B) = e - n + 2p = 13 - 13 + 2x3 = 6$$

McCabe ezt a számolási módot olyan programok bonyolultságának mérésére szánta, melyekben egy főprogram alprogramokat hív. Jegyezzük meg, hogy a módszer alkalmas arra is, hogy (főprogrammal nem rendelkező) alprogramok halmazának (pl. egy függvénycsomagnak) a bonyolultságát mérje.

Figyeljük meg a McCabe metrika egy „matematikailag kedvező" tulajdonságát; k darab G_i komponensekből álló G gráf komplexitása megegyezik az egyes G_i gráfok komplexitásának összegével.

$$v(G) = e - n + 2p = \sum_{i=1}^{k} e_i - \sum_{i=1}^{k} n_i + 2k$$

A ciklomatikus szám meghatározására adott $v = e - n + 2p$ kifejezés csak a programgráf alapján határozható meg, annak felépítése pedig esetenként nemtriviális feladat, ezért McCabe két egyszerűsítő formulát is megad. Mills struktúrált programokra bebizonyította [33], hogy $p = 1$ esetében az élek száma kifejezhető a függvény, predikátum és gyűjtő csomópontok számával, (melyek legyenek rendre θ, π és γ):

$$e = 1 + \theta + 3\pi$$

Mivel minden predikátum csúcshoz pontosan egy olyan csomópont tartozik, amely összegyűjti az éleket[2], és egy-egy belépési és kilépési pont létezik, ezért a csúcsok száma:

$$n = \theta + 2\pi + 2$$

Ebből következően, ha $p = 1$, akkor a $v = e - n + 2$ képlet kifejezhető:

$$v = (1 + \theta + 3\pi) - (\theta + 2\pi + 2) + 2 = \pi + 1$$

ahol π azon csomópontok (un. *predikátum csomópontok*) száma, melyek (legfeljebb) egy bemenő éllel, és egynél több kijövő éllel rendelkeznek. A számítástechnikai köztudatban inkább ez az egyszerűsített képlet terjedt el, amelyről fontos megjegyezni, hogy csak egy összefüggő gráfra alkalmazható. Ugyanakkor a több komponensből álló gráfok komplexitása (mint feljebb láttuk) az egyes összefüggő komponensek komplexitásának összege. Ez a kedvező számítási mód nagymértékben hozzájárult a McCabe metrika elterjedéséhez[3].

A predikátum csomópontok központi hangsúlyt kapnak tehát a McCabe metrikában. Nem mindegy ezért, hogy pontosan mit tekintünk egy predikátumnak. A gyakorlatban ugyanis egy predikátum csomópont tartalmazhat összetett feltételt is, pl.

IF c1 **AND** c2 **THEN**

Ez a valóságban két feltételt jelent;

IF c1 **THEN IF** c2 **THEN**

ezért McCabe szerint ésszerűbb a feltételek számát számolni, mint magukat a predikátum-csomópontokat. Az N értékű értékkiválasztásos (case) utasításokban $N - 1$ feltétel vizsgálata szükséges; ezt a szerkezetet ismét csak $N - 1$ IF utasítással szimuláljuk.

A ciklomatikus bonyolultsági számra több alapvető állítást mondhatunk ki.

1. $v(G) >= 1$
2. $v(G)$ a G-beli maximális számú lineárisan független körök száma.
3. Függvény- vagy eljáráshívások hozzávétele vagy elhagyása nem befolyásolja G bonyolultságát.
4. G-ben csak egyetlen út létezik, akkor és csak akkor, ha $v(G) = 1$.
5. G kiegészítése egy új éllel egységnyit növeli $v(G)$ bonyolultságot.

[2]Ez az állítás nyilvánvalóan csak struktúrált programokra igaz.

[3]McCabe egy másik egyszerűsítése az Euler formulán alapul. Ha egy n csúcsú, e élű gráf r tartmományra osztja a síkot, akkor $n - e + r = 2$. Ebből következően a ciklomatikus bonyolultsági szám az összefüggő programgráf által létrehozott tartományok számával egyezik meg.

6. $v(G)$ csak G döntési struktúrájától függ.

McCabe bemutatja azt is, hogyan használták a metrikát a gyakorlatban. A programozókat arra kérték, hogy a kód modularizálásában a kód hossza (programsorok száma) helyett a ciklomatikus számot vegyék figyelembe, és törekedjenek arra, hogy ez modulonként ne lépje túl a 10-es számot. McCabe nem indokolja a 10-es szám kiválasztását, mint írja: ez tűnik ésszerű, de nem mágikus felső korlátnak (seems like a reasonable, but not magical, upper limit). Amennyiben a bonyolultság túlhaladja e számot, a programozónak újra kell írnia vagy demodularizálnia kell a kódot. A cél az, hogy az egyes modulok karbantarthatóak legyenek és ezekben minden független vezérlési utat le lehessen tesztelni a gyakorlatban. Ugyanakkor McCabe maga is megjelöl egy kivételes esetet: nagyméretű értékkiválasztásos (case) utasításokban, sok érték esetén a 10-es felső korlát értelmetlennek, a modul felbontása pedig a bonyolultságot feleslegesen növelőnek tünt.

McCabe beszámol róla, hogy az általában jó programozási stílusúnak tartott programozók rendszerint eleve 3 és 7 közötti komplexitással hoztak létre modulokat, míg mások gyakran a 40 és 50 közötti tartományban. McCabe közeli kapcsolatot vél felfedezni a nagyobb komplexitási számú modulok, és a kód megbízhatatlansága között.

Cikkében McCabe kitér a struktúrált és nemstruktúrált programok viszonyára. Saját tapasztalataira és Knuth állítására hivatkozva bizonyos [26] speciális esetekben jogosnak tartja az eltérést a struktárált programkonstrukcióktól. Az „igazolható" eltéréseket kivételes esetnek tekinti.

3.3. Howatt és Baker

A McCabe féle ciklomatikus komplexitási mérték hiányosságait felismerve számos javítási kísérlet született a vezérlőszerkezetek egymásba ágyazottsága bonyolultságot növelő hatásának kifejezésére. A megoldások közös vonása, hogy az utasításokhoz rendelt beágyazottsági szám az adott utasítás végrehajtásához vezető döntési csomópontok (predicate node) száma. Ez struktúrált programoknál (melyek csak bizonyos vezérlő szerkezetek szekvenciájából és/vagy egymásba ágyazásából építünk) kielégítő. Ilyenkor ugyanis könnyen és egyértelműen azonosítani lehet az egyes utasításokra ható döntési csomópontokat – így a beágyzottsági mélységet is. Sajnos nem struktúrált programok esetében ez az informális definíció nem elégséges. Az pedig erős megszorítás lenne, ha egy mértéket csak strukturált programra lehetne alkalmazni.

A beágyazottsági szám pontos definíciójához vezető úton ki kell emelni James W. Howatt és Albert L. Baker 1989-es cikkét: Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting [22]. A cikkben a szerzők

kifejtik azon véleményüket, hogy a korábbi bonyolultsági metrika definíciók nem kellően formalizáltak, ami az analítikus kiértékelés teljes hiányával párosulva igen megnehezíti a publikált metrikák utólagos, független kutatók által történő tapasztalati igazolását. Az analítikus vizsgálat alapvető hiányosságokat tárhat fel. Természetesen a mérték definiálásakor alkalmazott matematikai szigor és az analítikus kiértékelés önnmagukban nem képesek garantálni, hogy a mérték pontosan méri a programok pszichológiai bonyolultságát. Ugyanakkor egy a formális alapokra jobban támaszkodó mérték biztosabb bázist képez a későbbi empírikus vizsgálatok számára.

Cikkükben Howatt és Baker formálisan definiálják a beágyazás (nesting) fogalmát imperatív programozási nyelvek számára. Feltételezik, hogy a mélyebben beágyazott vezérlő szerkezeteket tartalmazó programok bonyolultabbak, mint a kevésbé mély beágyazást alkalmazók. Véleményüket avval indokolják, hogy a mélyen beágyazott vezérlő szerkezetek megértése az alkalmazott predikátum csomópontok együttes megértését, a feltételek konjukcióját követelik meg a programozótól.

Miután az e dolgozatban bevezetendő metrika részben a Howatt és Baker által bevezetett bonyolultsági mértéken is alapul, tekintsük át a szerzők gondolatmenetét, definícióit és főbb megállapításait.

Howatt és Baker először az irányított gráf (directed graph), majd a vezérlési gráf (flowgraph) fogalmát definiálják. Az irányított gráf $G = (N, E)$ páros, ahol N a csúcsok, E az élek nemüres halmaza. A csúcsok egy rendezett (x, y) párja definiál egy élt, x-ből y-ba, ennek megfelelően x az y csúcs közvetlen megelőzője, míg y az x csúcs közvetlen rákövetkezője. Az x csúcs összes közvetlen megelőzőinek halmaza legyen $IP(x)$, míg összes közvetlen rákövetkezőjének halmazát jelölje $IS(x)$. Egy x csúcs bemenő fokszáma (indegree) $n = |IP(x)|$, kimenő fokszáma (outdegree) $m = |IS(x)|$. Azokat a csúcsokat, melyek kimenő fokszáma nagyobb, mint egy *predikátum* csomópontoknak nevezik. Az x-ből y-ba vezető út (path) fogalmát a szokásos módon definiálják; az x_1-ből x_k-ba vezető P út $(x_1, x_2), (x_2, x_3), ..., (x_{k-1}, x_k)$ élek sorozata, ahol

$\forall i(1 \leq i \leq k \Rightarrow (x_i, x_{i+1}) \in E$.

Minden egyes n csúcs, ami megjelenik P-ben a P úton fekszik (lies on P) $n \in P$.

Egy vezérlési gráf (flowgraph) $G = (N, E, s, t)$ négyes egy irányított gráf, egy véges, nemüres N csúcsok halmazával, véges nemüres E élek halmazával, $s \in N$ kezdőcsúccsal (start node) és $t \in N$ végcsúccsal (terminal node). Minden vezérlési gráfban pontosan egy s kezdőcsúcs létezik, amelynek bemenő fokszáma 0, és pontosan egy végcsúcs létezik, amelynek kimenő fokszáma 0. Minden csúcs G-nek valamely s-ből t-be vezető útján fekszik.

Howatt és Baker a (nem feltétlenül struktúrált) vezérlési gráfon definiálja a predikátum csomópontok *hatókörét* (scope) és a csomópontok *predikáló halmazát*

(predicate set). A csomópont beágyazottsági száma a predikáló halmaz számosságának függvénye – végső soron attól függ, hogy a csomópont hány predikátum hatókörében van benne.

A vezérlési gráf beágyazottságának mértéke $ND(G)$ a gráf csomópontjai beágyazottsági mértékének az összege. A gráf bonyolultsága a gráf beágyazottsági mértéke és a csomópontok számának (N') összege:

$$SN(G) =\mid N' \mid +ND(G)$$

3.4. Halstead–féle termék mértékek

A legtöbb *termék mérték* csak egyetlen aspektusból méri a programot. M. H. Halstead egységes elméleti alapon olyan termék mértékeket javasolt [17] (Halstead's Product Metrics), melyek a program több tulajdonságát együttesen jellemzik. E termék mértékek közé tartozik a programszótár mérete (n), a programhossz (N), a program tárhely mérete (V) és a szükséges erőfeszítés (E) valamint a fejlesztési idő (T).

A *programszótár* metrika (program vocabulary) a programot tokenek sorozatának tekinti. Minden token vagy operandus, vagy operátor. A programszótár méretét a programban előforduló különböző tokenek számaként definiáljuk: $n = n_1 + n_2$, ahol n_1 a különböző operátorok száma, n_2 az eltérő operandusok száma.

A *programhossz* metrika (program length) a program bonyolultságát az előforduló (nem feltétlenül különböző) operátorok (N_1) és operandusok (N_2) összegeként definiálja: $N = N_1 + N_2$. Halstead adott egy képletet N értékének közelítő kiszámításához:

$$N' = n_1 log_2 n_1 + n_2 log_2 n_2$$

N egy egyszerű metrika, amit a kész programból számolunk ki, míg N' számított metrika, amit már a program fejlesztése közben is meghatározhatunk n_1 és n_2 becsült értékeiből.

A *program tárhely mérete* (volume) $V = N log_2 n$. A V érték a program fizikai megvalósítását jellemzi. Halstead szterint kapcsolat van a program absztrakciós szintje (L) és a szükséges tárterület között:

$$V^* = LV = konstans$$

$$L = V^*/V$$

.

Halstead egyik gyakran emlegetett mértéke a *szükséges erőfeszítés* (programming effort). Ez az érték arra ad becslést, hogy mekkora erőfeszítés a program implementálása egy adott nyelven. Elve a következő: minden V méretű program előállítása felfogható N darab választásnak az n elemű szókészletből.

$$E = V/L = V^2/V$$

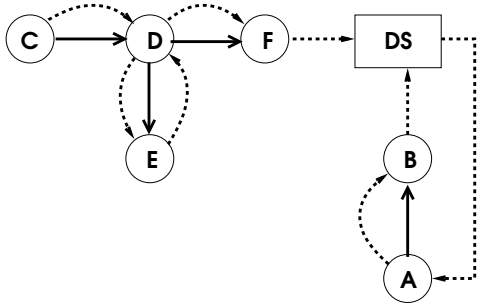
Curtis és társai érdekes cikkben hasonlítják össze Halstead és McCabe metrikáit [7].

3.5. A Henry- és Kafura-féle információáramlás metrika

Sallie Henry és Dennis Kafura *információ áramlás* alapú bonyolultsági mértéket definiált programokra. Az információ–áramlás metrikák a programot különböző alprogramok egységeként tekintik, és ezek közötti kapcsolatot vizsgálják. Konkrétan az a vizsgálat célja, hogy az egyes alprogramok között hogyan mozog az információ; részben a vezérlésátadások, részben az adatok segítségével: az objektumok írása, ill. olvasása révén.

Tekintsük az ábrát, ahol a programunk hat program-modulból, A, B, C, D, E, F modulokból áll, DS pedig egy adatszerkezetet reprezentál. A modulok lehetnek alprogramok (függvények vagy eljárások).

Az A modul olvassa DS-t, majd meghívja B modult. B modul aktivizálása után módosítja (írja) DS objektumot. A C modul meghívja D-t, az továbbhívja E modult. Az E modul értéket ad vissza D számára, amelyet az felhasznál, majd az eredményt átadja F modulnak. F a kapott eredmény alapján módosítja DS objektumot. Az ábrán folytonos vonallal jelöltük a vezérlésátadást, és szaggatott vonallal az információáramlást.



Henry és Kafura mértéke szerint információáramlásnak tekintjük az egyes modulok közötti aktivizációkat, azaz a vezérlés átadását és visszaadását, valamint az adatok írását és olvasását. Ennek megfelelően az alábbi közvetlen információáramlást (*termed direct local flow*) tudjuk megfigyelni:

$A \longrightarrow B$
$C \longrightarrow D$
$D \longrightarrow E$
$D \longrightarrow F$

Ezeket az információáramlási folyamatokat a vezérlésátadások határozták meg. Ezeken túl indirekt információáramlási folyamatokat (termed indirect local flow) is megállapíthatunk, mint pl. az E modul visszatérési értékének átadása D felé, illetve ennek továbbítása D-ből F modul felé:

$E \longrightarrow D$
$D \longrightarrow F$

Figyeljük meg, hogy az információáramlás nem feltétlenül valamely vezérlési kapcsolathoz, pl. vezérlésátadáshoz, kötődik. A DS adaton keresztül is valósul meg információáramlás, mivel mind a B, mind az F modulok írják a DS objektumot, az A modul viszont olvassa azt.

$B \longrightarrow A$
$F \longrightarrow A$

Henry és Kafura *globális* adatáramlásról beszél A és B modulok között, ha létezik egy olyan D adatszerkezet, melybe A információt helyez el (írja azt), és melyből B információt nyer ki (olvassa). *Lokális* adatáramlásról beszélnek A-ból B-be, ha (1) A meghívja B modult, vagy (2) B meghívja A-t és A értéket ad vissza B részére, melyet az felhasznál, vagy (3) ha létezik egy harmadik modul: C, mely meghívja sorrendben az A és B modult, és a B által visszaadott értéket továbbítja A-nak. A lokális adatáramlás (1) esetét *direkt* lokális információáramlásnak, míg a (2) és (3) eseteket *indirekt* lokális információáramlásnak nevezik. Az A eljárásba vezető lokális információáramlások száma és az A által inputra használt adatszerkezetek számának összege az A eljárás fan_{in} értéke, míg az A-ból kivezető információ folyamatok számának és az A által írt adatszerkezetek számának összege az A fan_{out} száma.

A fenti példák ismertetése jól tükrözi a Henry–Kafura mérték azon tulajdonságát, hogy elegendő a programot a modulok, alprogramok közti kapcsolatok (hívási lánc és adatkapcsolatok) vizsgálatára leszűkíteni. Általában az információáramlás az alábbi formában írható fel:

$$forras1, forras2, \ ... \ , forras3 \longrightarrow cel$$

A forrás illetve a cél definiálásához az alábbi jelölésrendszert vezeti be Henry és Kafura:

1. X.n.I jelölje az X eljárás n-ik paraméterének az értékét az eljárás elején.
2. X.n.O jelölje az X eljárás n-ik paraméterének az értékét az eljárás végén.
3. X.O jelölje az X függvény által visszaadott értéket, ha X függvény
4. X.D jelölje az X eljárás által hivatkozott D globális adatobjektumot.

Ez a jelölés alkalmas egy program információs folyamatainak leírására. Figyeljük meg, hogy nem különböztetjük meg az információátadás szempontjából, hogy azt az egyes eljárások paraméterei vagy globális objektumok segítségével valósítjuk meg.

Egy eljárás bonyolultsági mértékének meghatározásához Henry és Kafura szerint két összetevőt kell számításba venni: az alprogramot alkotó kód bonyolultságát és azt, hogy ez az eljárás milyen bonyolult kapcsolatban van környezetével. A programkód bonyolultságát a legegyszerűbb módon, a forrássorok számával (LOC) mérik, mivel állításuk szerint ez a mérték erős korrelációban van a kódban előforduló hibák számával. Az eljárás és környezete kapcsolatának bonyolultságát a fan-in és fan-out értékekkel jellemezzük. A szerzők által megadott képlet a bonyolultsági metrikára:

$$c = LOC_E * (fan_{in} * fan_{out})^2$$

ahol LOC az eljárás hossza, a $fan_{in} * fan_{out}$ szorzás pedig azt jelenti, hogy a bejövő és kimenő információ összes lehetséges kombinációját figyelembe vesszük. A második hatvány a képletben egyedül abból az okból került a képletbe, mert a szerzők szerint a bonyolultság több, mint az információmennyiség lineáris függvénye. Magának a képletnek egy gyenge pontja van, a LOC_E, azaz az alprogram hossza, melyet a pontosság romlása nélkül akár el is hagyhatnánk.

3.6. Objektum-orientált metrikák

Az objektum-orientált paradigma a nyolcvanas évek második fele óta a legelfogadottabb tervezési és programozási módszer. Mivel az objektum-orientált programok magasabb strultúrájukban eltérnek a procedurális programoktól, az alkalmazott metrikák is követték a paradigmaváltást[4].

Az objektum-orientált metrikák közül az egyik legnépszerűbb és leggyakrabban használt a Chidamber- és Kemerer által definiált hat szoftvermérték [5].

[4]Jelen dolgozat viszont éppen avval érvel, hogy mind a procedurális, mind az objektum-orientált programok rendelkeznek az elemi vezérlési- és adatkezelési szerkezetek egy közös készletével. Az, hogy mely program procedurális és mely objektum-orientált az attól függ, hogy e műveletek mely részhalmaza használata megengedett, ill. milyen gyakorisággal használják az egyes elemi szerkezeteket. Ennélfogva definiálható olyan metrika, amely képes paradigmafüggetlen módon mérni mindkét esetet.

- *Súlyozott metódusok osztályonként* (Weighted Methods per Class, WMC)

  Az osztály bonyolultságát az osztályon definiált metódusok bonyolultságának összegeként kapjuk meg. A mérték alkalmazhatósága melletti érv, hogy a bonyolultabb, és/vagy több metódussal rendelkező osztályok tesztelése és karbantartása több időt, és nagyobb erőfeszítést igényelnek. Megfigyelhető az is, hogy a több metódussal rendelkező osztályok nagyobb hatást gyakorolnak a belőlük származtatott osztályokra, (és ezen keresztül a teljes programra), hiszen a leszármaztatott osztályok *minden* metódust örökölnek a bázisosztálytól. Végül a tapasztalat azt mutatja, hogy minél több metódust tartalmaz egy osztály, annál kevésbé valószínű, hogy egy általános, jól definiált absztakciót megvalósító kódrészlettel van dolgunk, így ezen osztály újrafelhasználása nehezebb.

- *Az öröklődési hierarchia mélysége* (Depth of Inheritance Tree, DIT)

  A legtöbb objektum-orientált program jellemzően összetett osztályhierarchiával rendelkezik. Számos nyelv minden osztálya közvetve egy közös bázisosztályból származik (ilyen például a Java nyelv *Object* osztálya), ilyenkor az osztály bonyolultságát jellemzi, hogy milyen „távolságra" van a gyökér osztálytól. Minél mélyebben van egy osztály a hierarchiában, annál több metódust örököl őseitől, ennélfogva összetettebb lesz viselkedése. A mélyebb öröklődési fák nagyobb tervezési bonyolultságot eredményeznek, mivel több metódust és osztályt kell megértenünk.

  Bizonyos nyelvekben nem kötelező minden osztályt egyetlen közös bázisosztályból származtatni (C++); tetszőleges osztály lehet egy új öröklődési hierarchia kezdőpontja. Az ebből az osztályból indított öröklődési lánc hossza azonban szintén jellemzi a résztvevők bonyolultsági értékét. Némi gondot jelenthet azonban a többszörös öröklődés, amennyiben ez az adott nyelvben megengedett.

- *Leszármazott osztályok száma* (Number of Children, NOC)

  Ebben az esetben az osztályból közvetlenül leszármaztatott gyerekosztályok száma lesz a bonyolultság mértéke. Chidamber és Kemerer avval érvelnek, hogy minél nagyobb a leszármazott osztályok száma, annál több az újrafelhasználás, illetve annál nagyobb az esélye, hogy az osztályban nem a megfelelő absztrakciót alkalmaztuk az ősosztályban. Végül, minél nagyobb a származtatottainak száma, annál fontosabb szerepet tölt be egy adott osztály a tervezés és a tesztelés folyamatában.

  Ezzekkel az érvekkel nem tudunk teljes mértékben egyetérteni. Egy implementált osztály bonyolultsága nem függhet olyan kódrészletektől, melyeket vagy

megvalósítanak, vagy sem. Az öröklődés tipikusan olyan objektum-orientált nyelvi elem, amely a bázisosztályra semmilyen módon sincsen visszahatással[5]. Másrészt, ha ugyanazt a kódot többször tudjuk újrahasználni, az éppenhogy a kódminőség pozitív jellemzője, nem pedig negatív kritérium.

- *Összefonódás az osztályok között* (Coupling between object classes, CBO)

  Ezt a mértéket az adott osztályhoz kapcsolódó osztályok számával definiáljuk. Két osztály akkor kapcsolódik egymáshoz, ha az egyik felhasználja a másik metódusait vagy attributumait.

  Egy C osztály *fan-out* értéke azon különböző osztályok száma, melyekre hivatkozunk a C osztályban. A C osztály *fan-in* értéke azon más osztályok száma, melyek hivatkoznak a C osztályra. Az összefonódást a „kimenő" hivatkozások (*fan-out*) számával mérjük.

  Az osztályok túlzott összefonódása hátrányos a moduláris tervezés szempontjából, és akadályozza az újrafelhasználást. Minél függetlenebb egy osztály, annál kevesebb zavaró körülményt kell figylembe vennünk a kód újrafelhasználásakor, például örökléskor. Karbantartáskor, módosításkor a nagyobb kapcsolat törékenyebbé teszi a kódot is. Sűrűbb kapcsolat az osztályok között komolyabb tesztelési erőfeszítéseket igényel. Az osztályok közötti kapcsolat lehező legalacsonyabb szinten tartását szolgálja az *egységbezárás* (encapsulation).

- *Üzenetek által kiváltott metódusok száma* (Response for a Class, RFC)

  A bonyolultságot e mérték szerint az osztály egy példányához érkező üzeneteket kezelő metódusok, és az általuk meghívott további metódusok számával definiáljuk.

  Minél több metódus léphet működésbe egy metódushívás érkezésekor, annál nehezebb az osztály tesztelése, mivel nehezebb a vezérlési folyamat megértése. Az üzenetek kapcsán meghívott metódusok számának növekedése az osztály bonyolultságának növekedését jelzi.

- *Kohéziós hiány a metódusokban* (Lack of Cohesion in Methods, LCOM)

  Legyen adott C osztály az M_1, M_2, ... , M_k metódusokkal. Jelölje I_k az M_k metódus által használt adattagok halmazát.

  $P = \{(I_k, I_j) \mid I_k \cap I_j = 0\}$

  $Q = \{(I_k, I_j) \mid I_k \cap I_j \neq 0\}$

[5]Ez alól a virtuális függvények alkalmazása némileg kivételt képez.

Ekkor, ha $|P| > |Q|$ akkor $LCOM = |P| - |Q|$, különben 0.

Ez a metrika azon a tapasztalati tényen alapul, hogy az osztályon belüli kohézió előnyös, hiszen valószínűsíti, hogy az adott osztály helyes absztrakciót valósít meg. Az erős kohézó elősegíti az enkapszuláció alkalmazását is. A kohézió hiányában szenvedő osztályok valószínűleg szétbonthatók kettő vagy több alosztályra.

Az ismertetett mértékek az objektum-orientált tervezés természetes megközelítéséből adódnak. A NOC és RFC mértékek adnak valamelyest támpontot a teszteléshez szükséges erőfeszítésekről. Általában megfigyelhető, hogy a struktúrális mértékek a méret mértékekkel korrelálnak.

A Chidamber- és Kemerer féle mértékek együttesen jellemzik az osztály bonyolultságát. A fenti mértékek bajosan adnak előre jó „méret" és „szükséges erőfeszítés" becsléseket.

3.7. Gyakorlatban használt mértékek

A következőkben áttekintést adunk a ma alkalmazott szoftver bonyolultsági mértékekről. Annak érdekében, hogy valós, a gyakorlatban is széleskörűen használt szoftvermértékeket vizsgáljunk meg, az egyik legigéretesebb jelenlegi fejlesztési projekt, az *Eclipse* által alkamazott metrikákat vesszük alapul.

2001 november 29-én a szoftveripar néhány vezető cége bejelentette az *Eclipse.org* konzorcium megalapítását [46]. Az alapítók között egyaránt voltak szoftverfejlesztő és integrátor cégek, mint a Borland, az IBM, és a Merant; tervező és CASE eszköz gyártók, mint a TogetherSoft és Rational Software (melyet azóta megvásárolt az IBM); valamint operációs rendszer rendszer fejlesztők, mint a QNX Software Systems, a RedHat és a Suse. A konzorcium célja, hogy nyílt forráskódú eszközök létrehozásával támogassa az Eclipse felületre történő fejlesztéseket. Megalapítása óta olyan neves cégek csatlakoztak a konzorciumhoz, mint a Hitachi, Telelogic, Oracle, Parasoft, SAP vagy a Hewlett-Packard. Társult tag lett a CORBA szabványok felügyeletét ellátó OMG (Object Management Group) is.

Az Eclipse fejlesztőfelület (IDE) nyílt forráskódú környezet, mely segítségével integrálhatóak az egyes gyártók szoftverfejlesztést támogató eszközei. Az Eclipse környezet egyaránt fut Microsoft Windows, Linux és QNX operációs rendszer környezetben.

Az Eclipse segítségével tetszőleges programnyelvi keretben végezhetünk fejlesztést, de az egyik kiemelt célkitűzés a Java nyelv támogatása. Ez a JDT (Java Development Tools) segítségével történik. Az IDE plug-in eszközökkel bővíthető. Plug-in eszközök segítségével válik lehetővé a code-refactoring illetve szoftver metrikák haszálata is, egyenlőre csak Java kódra alkalmazva.

Az Eclipse projekt metrika plug-in számos mértéket alkalmaz a kód különböző paramétereinek megállapítására. Ezek a következőek:

- McCabe ciklomatikus bonyolultság

  A metrika a klasszikus McCabe bonyolultság alkalmazása a Java nyelvre, amely a kód lineáris összetevőit, pedikátumot (azaz elágazást, ugrást) nem tartalmazó kódblokkokat azonosítja és számolja össze. Ily módon alkalmazása a tesztesetek összeállítását is támogatja. A plug-in a metrikát metódusonkét alkalmazza a kódra, predikátumként a **for**, **while**, **do**, **if** vezérlési szerkezeteket és opcionálisan a **case**, **catch** és aritmetikai elágazást tekinti. Az eredményt összegzi metódusonkét és osztályonként.

- Törékeny kapcsolatok (Efferent coupling)

  A metrika megszámolja azokat a típusokat, melyeket a mért osztály „ismer". Ezek közé tartoznak az örökölt *bázisosztályok*, a *megvalósított interfészek*, a metódusok *paramétereinek*, deklarált *változóinak*, eldobott és elkapott *kivételeinek* típusai. Összefoglalóan ide tartozik az összes típus, melyre a mért osztály forrása hivatkozik.

  Ha az osztály nagy számban hivatkozik idegen típusokra, az azt jelentheti, hogy az osztály túl sok szétszórt információt próbál összekapcsolni. E szám ugyancsak jelzi az osztály sérülékenységét, ami az általa használt típusok stabilitásán múlik. Az osztály sérülékenységét csökkenthetjük avval, ha szétbontjuk elemi(bb) osztályokra (*class decomposition*).

- Kohéziós hiány metódusokban (Lack of Cohesion in Methods)

  A kohézió az objektum-orientált programozási paradigma fontos eleme. A kohézió jelzi, hogy az osztály mennyire egyértelmű absztarkciót valósít meg. Ha az absztrakció nem egyértelmű, az osztályt fel kell bontanunk elemi összetevőkre, melyek már egyetlen fogalmat reprezentálnak.

  Amennyire egyértelmű az elv, annyira nehéz a gyakorlatban egyértelmű mérési mechanizmust kidolgozni. Ennek vélhetően az az oka, hogy a *jó absztrakció* ténye nagyfokú szemantikai megértést igényel, amit egy adott programozási nyelv szintaxisa legtöbbször nem képes kellően kifejezni.

  Miután az osztály magasabb kohézióját tekintjük előnyösebb esetnek, a „kohézió hiányát" mérjük, hogy – a többi bonyolultsági mértékhez hasonlóan – az alacsonyabb értékek jelentsék a jobb helyzetet.

  A kohézió hiánya mérésére Chidamber és Kemerer [5], valamint Henderson-Sellers [20] által adott definíciót használják legtöbbször.

Chidamber és Kemerer definíciója az osztály azon metódusait számolja meg, melyeknek egyetlen közösen használt osztálybeli adattagja sincsen, és ezekből levonja azok számát, melyek legalább egy mezőn osztoznak. Ha a különbség negatív, akkor a mérték nulla.

Henderson-Sellers a kohézió hiányát a következőképpen definiálja:

Legyen M az osztály által definiált metódusok, F az osztály adattagjainak száma. Az $r(f)$ jelölje azon metódusok számát, melyek elérik az osztályban definiált f mezőt, $f \in F$. Végül legyen $< r >$ az $r(f)$ értékek F-en számolt középértéke. Ekkor a mérték definíciója:

$$\frac{< r > - |M|}{1 - |M|}$$

A számolás során az egyes implementációk bevezetnek bizonyos mosításokat. Az *Eclipse* implementációja például az alábbi egyszerűsítésekkel él:

- Csak azokat a metódusokat számolják M-be, melyek legalább egy adattagot írnak vagy olvasnak. Azok a metódusok, melyek egyetlen mezőt sem érnek el, gyakran csak azért nem lettek statikusként deklarálva, hogy polimorfikusan viselkedjenek.
- Csak azokat a mezőket számolják F-be, melyeket az osztály legalább egy metódusa elér.

- Metódus sorainak száma (LOC)

  A mérték egyszerűen megszámolja az újsor karaktereket a metódusban. A sorok száma az egyik legrégebben használt szoftvermetrika, amely ráadásul erősen függ a kódolási stílustól. Mégis, még ma is elterjedten használják ezt az értéket.

  Amellett, hogy az eltérő kódolási stílus azonos metódusok esetében is eltérő értéket produkál, további gondot jelent, hogyan kezeljük az üreshely (whitespace) karaktereket, megjegyzés sorokat, stb.

- Mezők száma az osztályokban

  Az egyik legegyszerűbb mérték. Az osztályban definiált nagyszámú adattag jelenlétét nem tekinthetjük automatikusan a rossz kódminőség indikátorának, mégis egy olyan jelenség, melyre fel kell figyelnünk. Előfordulhat, hogy felül kell vizsgálnunk az osztály definícióját és fel kell bontanunk több osztályra.

Döntés kérdése, hogyan kezeljük az osztályban definiált szimbolikus konstansokat. Gyakran fordul elő, hogy egy osztály nagyszámban definiál konstansokat. Opcionális, hogy ezeket információhordozó adattagoknak, vagy a nyelv literáljaihoz hasonlóan figyelmen kívül hagyandó konstansoknak tekintjük őket.

- Szintek száma

  Ez a mérték egy adott metódus vezérlési szerkezetének maximális beágyazottági mértékéről ad információt. Alapelvként azt az általános megfigyelést tekintik, miszerint a magas beágyazottsági mérték nehezíti a kód megértését. Tapasztalatok szerint az ilyen metódusok általában alacsony absztrakciós szinten működnek.

  A magas beágyazottsági szinttel rendelkező eljárásokat gyakran több privát metódusra, esetleg eljárásosztályokra[6] bonthatjuk fel.

  Néha a beágyazottsági szint nem tükrözi egyértelműen a kód megértése szempontjából fontos információkat. A legtöbb programozó számára a következő két kódrészlet nem azonos komplexitásúnak tűnik:

```
// szokásos tabulálás:
if ( .. )
    f1();
else if ( ... )
    f2();
else if ( ... )
    f3();
else
    f4();
```

```
// valójában ezt a beágyazottságot jelenti:
if ( .. )
    f1();
else
{
    if ( ... )
```

[6]Eljárásosztálynak (Method Object) nevezzük az olyan osztályokat, melyek egy műveletet és az azok által használt temporális változókat csomagolják be [47].

```
        f2();
    else
    {
        if ( ... )
            f3();
        else
            f4();
    }
}
```

- Paraméterek száma

  Ez a mérték megszámolja az egyes eljárások paramétereinek számát. Nagy paraméterszámmal rendelkező metódusok jelenléte számos esetben arra utal, hogy tervezéskor nem azonosítottunk egy osztályt. Az össztartozó paraméterek csoportosításával megteremthetjük ezeket az osztályokat. Előfordulhat, hogy az így létrejövő új osztályok kiválthatnak korábban (hibásan) definiált osztályokat is. Összességében ezen folyamat végén egyszerűbb és könnyebben karbantartható kódhoz jutunk.

- Utasítások száma

  A sorok számához (LOC) hasonlóan a méret mértékek (size metrics) csoportjába tartozó mérőszám, mely azonban sokkal megbízhatóbb a sorok megszámolásánál. Ennek oka, hogy kevésbé érzékeny az eltérő kódolási konvenciókra. A metódusonként elvégzett utasításszámlálás nem ad információt az alkalmazott vezérlési szerkezet jellemzőiről, így pl. a metódus beágyazottsági mélységről, de a nagyszámú utasítást tartalmazó eljárások alaposabb kódrevíziót igényelnek. Az ilyen eljárások törzsének egyes összefüggő kódrészletét gyakran lehet kiemelni új eljárásként. Ekkor a kódrészlet saját nevet kap, esetleg paraméterlistaként fejezzük ki explicit függését a környező kódtól, ezáltal a progarmrészlet áttekinthetőbbé és könnyebben módosíthatóvá válik.

  Az Exclipse-ben alkalmazott implementáció utasításnak tekinti a *vezérlési szerkezeteket*, az explicit *konstruktorhívásokat*, a *bázisosztály konstruktora* explicit hívását, kivételek dobása kapcsán a **throw**, **try**, **catch**, **finally** kulcsszavakat, az *értékadásokat* és *eljáráshívásokat*, stb. Amennyiben *beágyazott osztály* definíciójával találkozik, akkor annak bonyolultsági értékével növeli a tartalmazó osztály metrikus értékét.

- Osztályonkénti súlyozott metódusok

  A fenti mértékek jó része metódusonként számolható. Ezekből gyakran állítanak elő egy számított értéket a teljes osztály bonyolultságának kifejezésére.

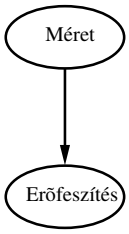
Ez legtöbbször a az egyes metódusokra számított értékek összege, ami jól reprezentálja az osztály előállításához (és későbbi karbantartásához) szükséges erőfeszítést (effort metrics). Az Eclipse által használt metrikák közül jelenleg a McCabe féle ciklomatikus bonyolultág metódusonkénti összegzése történik meg, mint a tejes osztály bonyolultságát jellemző szám kiszámítása.

3.8. A jelenlegi mértékek kritikája

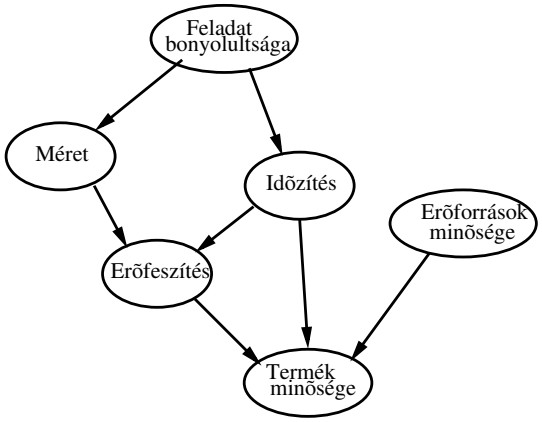
A szoftver mértékek kutatása már több mint 30 éve van jelen a számítástudományban. Az eddig elért kétségbevonhatatlan eredmények ellenére többen erős kritikát fogalmaznak meg a jelenleg használt mértékekkel szemben. Norman Fenton és Martin Neil cikkükben [14] felvetik, hogy a legtöbb szoftvermetrika éppen a velük szemben felmerülő legalapvetőbb igényt nem teljesíti: a szoftver életciklusa alatt nem biztosít a döntéshozók számára kellő mennyiségben számszerű adatokat a termékről.

Részben a fentek miatt a szoftvermetrikák használata ill. a metrikákkal kapcsolatos fejlesztések jellemzően nem a szoftvertechnológiában érdekelt cégek tudatos minőségjavító elhatározásainak eredménye. Sokszor egy fejlesztés elakadása, az elérendő célok veszélybe kerülése, vagy csak valamely külső előírásnak való megfelelés szándéka miatt kerül előtérbe valamely metrika alkalmazása. Az Egyesült Államokban a metrikákkal kapcsolatos ipari tevékenység döntő része például a *CMM*-el (Capability Maturity Model) kapcsolatos [23], mivel valamely metrika használata előfeltétele a vállalat magasabb CMM szinten való elismerésének. (A CMM keretein belül bevezetett metrikák magyarországi gyakorlati tapasztalatairól írt nagyszerű könyvet Balla Katalin [1].)

Cikkükben a szerzők megkülönböztetik a szoftver mértékek két eltérő modelljét: a *regressziós* (regression) és az *ok-okozati* (casual) modellt. A regressziós modell egyszerű statisztikai összefüggést keres a *mérhető* adatok és a metrika értéke között, függetlenül attól, hogy létezik-e közvetlen ok-okozati összefüggés közöttük. Számos esetben szeretnénk megjósolni például a feladat elvégzéséhez szükséges befektetett munkát, ez motiválja a szükséges erőfeszítés–mértékek (effort metrics) használatát. Ezek tipikusan regresszió-alapon működnek, ahol a kifejtendő erőfeszítés a termék méretének valamely függvénye. Ez a megközelítés azonban félrevezető. Miután a termék *mérete* nagy valószínűséggel nem *oka* a termék előállításához szükséges erőfeszítéseknek, az ilyen modellek alkalmatlanok a döntéshozók számára szükséges információk fejlesztés közbeni szolgáltatására:



Az ok-okozati modell megpróbálja feltérképezni azokat az összetevőket (és a köztük levő kapcsolatokat), melyek befolyásolhatják a rendszer általunk megfigyelhető paramétereit. Ez a modell kifejezi az egyes komponensek egymásra gyakorolt hatását:



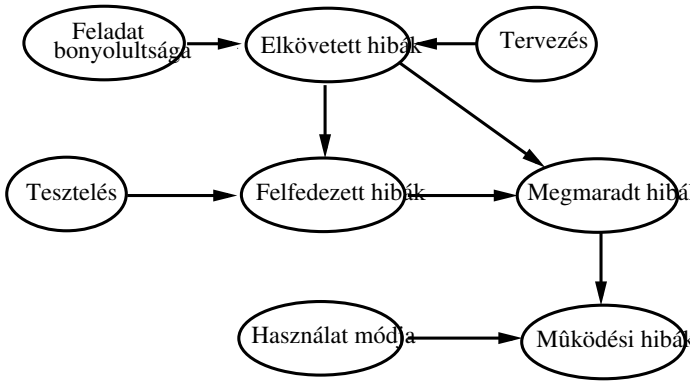
Ezen kapcsolatok segítségével Fenton és Neil olyan kérdések megválaszolását is elképzelhetőnek tartja, melyek a komponensek közötti kapcsolatokból számíthatóak, mint például:

- Adott bonyolultságú specifikáció és adott korlátozott erőforrások mellett milyen valószínűséggel érhető el a megfelelő minőségű termék kifejlesztése?
- Mennyire csökkenthetjük a felhasználandó erőforrások mennyiségét, amennyiben elfogadjuk a kifejlesztendő termék alacsonyabb minőségi paramétereit?
- A modell előrejelzése szerint 4 fejlesztő 2 éves munkája szükséges a termék kifejlesztéséhez, de csak 3 fejlesztő áll rendelkezésre 1 évig. Ugyanazon termékminőség célkitűzése mellett mennyivel jobb minőségű fejlesztőcsapatra van szükség?

Egy regressziós modell természetesen nem tartalmaz információt e kérdések megválaszolásához. Az ok-okozati modell legfőbb előnye, hogy képes kezelni a különböző fejlesztési- és termék paramétereket, a tapasztalati tényeket és szakértői döntéseket, hiteles ok-okozati összefüggéseket, a bizonytalan vagy hiányos információkat.

Mindezek mellett nem szabad sem a fejlesztés, sem a kész termék tekintetében olyan új terheket eredményeznie, amely számottevő többletköltséggel járna.

Fenton és Neil az ok-okozati modell megvalósítását Bayes-i hálóval (Bayesian belief net, BBN) képzelik el. A BBN egy grafikus háló, melyhez valószínűségi táblázatok tartoznak. Az ábra csomópontjai bizonytalan értékű változókat reprezentálnak, míg az őket összekötő nyilak az ok-okozati összefüggéseket ábrázolják. Az egyes valószínűségi táblák, melyeket a csomópontokhoz rendelünk, az adott változó egyes állapotaihoz tartozó valószínűséget adják meg. A szülő nélküli csomópontoknál ezek a kiinduló valószínűségek, a szülővel rendelkező (okozati) csomópontok esetében ez a szülő állapotainak kombinációjához rendelt feltételes valószínűségek. Mint minden BBN-nél, a valószínűségi táblázatok tapasztalati tényeket, és szakértői döntéseket tartalmaznak.



A Bayes-i hálón alapuló rendszerek régi és jól kidolgozott matematikai alapokkal rendelkeznek, és számos alkalmazásuk van az informatika terén, a spam-filterektől a szakértői rendszerekig. A BBN alapú rendszerek közül a legismertebb a Microsoft Office súgó-varázslója, amely a felhasználó által detektált jelenségek alapján tesz javaslatot a hiba elhárítására.

Teljesen más utakon kívánja a szoftver bonyolultsági mértéket definiálni Kokol, Podgorelec és Zorman [27]. Jelenlegi mértékekkel szemben megfogalmazott kritikájuk szerint az egyik fő probléma, hogy a metrikák erőteljesen nyelvfüggőek. Más metrikákat kell alkalmaznunk a magasszintű nyelvekre és az alacsony szintű megvalósításokra (pl. egy assembly kódra). Gyakran még az egyes magasszintű nyelvekre is külön-külön kell definiálnunk a mértéket. (A jelen dolgozatban javasolt mérték részben megoldást kínál erre a problémára: az AV gráf nyelvfüggetlen, és egyaránt alkalmazható alacsony és magasszintű implementációkra. Ugyanakkor a konkrét programozási nyelv leképezése az AV gráfra nem tudja elkerülni a függést a konkrét nyelv definíciójától. Lásd: C++ és Java függvényhívások).

Egy program, vagy információs rendszer sok formában jelenhet meg. A követelményrendszer, a specifikáció, a rendszerterv, a felhasználói dokumentáció,

a felhasználói interfész mind a rendszer egy-egy reprezentációja. Függetlenül attól, hogy ezek milyen megjelenési formában elérhetőek, (text dokumentáció, formális specifikációs nyelv, UML diagramm, programkód, stb.) egyaránt meg kell tudnunk mérni bonyolultságukat (és a bonyolultság változását, mikor ezek között a reprezentációs formák között tranzformációk történnek). A bonyolultsági mértéket – szélső esetként – magára a végrehajtható kódra is alkalmazhatjuk. A fenti érvelés alapján a szerzők egy univerzális metrikát javasolnak, amely tetszőleges elekronikus dokumentumra méri a bonyolultságot.

Kiindulópontjuk azon alapul, hogy mind a számítógép programok, mind a természetes nyelven íródott dokumentumok a kommunikáció egy adott formáját képezik. Mind a számítógépes rendszer egyes összetevői között végzett kommunikáció, mind a humán kommunikáció *kódot* használ az információcseréhez. A szerzők azt feltételezik, hogy egy (számítógépes vagy humán) dokumentum információtartalma a nyelvi szimbolumok közötti kapcsolaton alapul.

A bonyolultság egyik értelmes mérési módja a Shannon által bevezetett *entrópia* fogalmára vezethető vissza [38]. (Miután hosszabb dokumentumok esetén a Shannon-i információs entrópia igen nehezen számítható, Grassberger egy közelítő számítást javasolt az entrópia becslésére [16].) Egy másik módszer a nagy távolságú korreláció, *Long Range Correlation (LRC)*, melyet Schenkel javasolt [36], az entrópia fogalmának általánosításán alapul, és rendkívül hasznosnak bizonyult az (elsősorban humán) dokumentumok bonyolultságának mérésére. Jelek sorozatának LRC mérésére az egyik legáltalánosabban elfogadott módszer a sorozat megfeleltetése véletlen bolyongási problémának. Ennek során a jelsorozatot Brown-mozgásként fogjuk fel. Az input dokumentum karaktereit egy rögzített kódtábla alapján bináris sorozatnak feleltetjük meg[7]. Az így kapott bináris sorozat feleltethető meg Brown-mozgásként, a 0 és 1 bitek az ellentétes irányú egységnyi lépések.

A Brown-mozgás fontos statisztikai jellemzője az *átlagos négyzetes fluktuáció négyzetgyöke* (root of mean square fluctuation) F, ahol:

$$F^2(l) \equiv \overline{[\Delta y(l, l_0)]^2} - [\overline{\Delta y(l, l_0)}]^2$$

ahol l két pont távolsága, és

$$\Delta y(l, l_0) \equiv y(l_0 + l) - y(l_0)$$

Ha a jelsorozat nem korrelál (egy normális véletlen bolyongás), vagy csak egy meghatározott távolságig ható helyi korrelációk léteznek (pl. Markov láncok), akkor

$$F(l) \approx l^{0.5}$$

[7]Belátható, hogy a kódtábla megválasztása nem befolyásolja a végeredményt.

Ha nincsen ilyen meghatározott távolság, és a korreláció „végtelen", akkor

$$F(l) \approx l^{\alpha} \ , \ \alpha \neq 0.5$$

A fenti mértékkel Kokol és társai az alábbi következtetésekre jutottak:

- **Elméleti előrejelzések.** A fent ismertetett hipotézis azon állítása, hogy a véletlen generált programok α értéke 0.5 körüli kell legyen bebizonyosodott. Véletlenül generált FORTRAN és Pascal programok átlagértéke 0.497 ill. 0.502 volt, míg az egyes programok értéke 0.48 és 0.51 között mozgott.
- **Programozási nyelvek összehasonlítása.** Az egyes programok bonyolultságának mértéke nagymértékben függött az implementációs nyelvtől. FORTRAN, Pascal, és C++ programok vizsgálatakor kiderült, hogy a legmagasabb értéket a C++ programok (0.68), a legalacsonyabb értékeket pedig a FORTRAN programok(0.51) mutatták. Köztük helyezkedtek el a Pascal programok (0.59). Hasonló kapcsolatot lehetett felfedezni a formális specifikáció és az azt leíró nyelv között.
- **Bonyolultság növekedés.** A forrás lefordításakor a mért bonyolultság növekedése történt. C forráskódú programok átlagos értéke 0.63 volt. Ugyanezen programok lefordított, gépi kódú értéke 0.69
- **Verziók közötti bonyolultság változása**. A Windows kernel egyes verzióinak bonyolultságának változása jó példa arra, hogyan változik a mérték a szoftver fejlődése során. Az LRC alapú mérték a Windows 3.1 esetében 0.766, míg a Windows 95 értéke 0.806.
- **A kódhibák számának előrejelzése** A szerzők hipotézise szerint a metrika az entrópiát méri, ezért fordított arányban kell állnia a hibák számával. A mérések szerint a hibák számának jóslása kb. 80 százalékos pontossággal történhet e metrika alapján.
- **Hagyományos szoftver bonyolultsági mértékek** A szerzők által javasolt mérték semmilyen korrelációt nem mutatott a hagyományos szoftver metrikákkal.

Bár Kokol és társai megállapításait számos ponton lehet vitatni, munkájuk rávilágít arra, hogy a szoftver bonyolultság vizsgálata terén ma is születnek kísérletek új megközelítések elfogadtatására.

4. A javasolt mérték

A metrikák áttekintése során láttuk, hogy azok gyakran egyoldalúan – csak a program vezérlési stuktúrája vagy csak az adatok viselkedése szempontjából – mér hatékonyan. A mai programozási paradigmák használata esetében ez különösen veszélyes, hiszen az adatszerkezet és a rajta végzett műveletek egyenrangú szerepet játszanak. Ezért olyan metrika bevezetését javasoljuk, mely egyaránt érzékeny a program vezérlőszerkezetének bonyolultságára, a használt adatok bonyolultságára és az adatkezelés komplexitására.

A program vezérlőszerkezetének bonyolultságát a beágyazottsági mérték segítségével határozzuk meg. A bonyolultság meghatározásánál figyelünk arra, hogy az értelmezhető legyen nem struktúrákt programok esetében is. Erre mai programozási nyelveink (C#, C++, Java) használatakor ismét érzékenyen kell figyelnünk, lévén a *kivételkezelés* egyre elterjedtebb hibakezelési mechanizmus e nyelvekben.

A vezérlési szerkezet bonyolultságának definiálása után rátérünk az adatok és az adathasználat bonyolultságának meghatározására. Az adatok önnmagukban vett bonyolultsága mellett figyelembe kell venni, hogy a vezérlési szerkezet mely pontjaiból hivatkozunk az adatokra. Az adatkezelés bonyolultsága így tükrözi kapcsolódásukat a vezérlési szerkezethet. A definiált mérték figyelembe veszi azt is, hogy az adatot olvassuk, írjuk, vagy mindkettő megtörténik. Megmutatjuk, hogy evvel a megkülönböztetéssel ki tudunk mutatni egészen finom bonyolultsági különbségeket.

4.1. Alapvető definíciók és jelölésrendszer

Az új metrika definiálásához a fentiek szellemében meg kell határoznunk a program vezérlőszerkezetét, az adatszerkezetét, majd a vezérlőszerkezet és az adatszerkezet kapcsolódási pontjait. Első lépésben tekintsünk néhány alapvető definíciót és jelölést a vezérlőszerkezettel kapcsolatban. A vezérlőszerkezetet az általánosan elterjedt módon irányított gráf segítségével definiáljuk. A definíciók és jelölések során alapvetően a J. Howatt and A. Baker [22] által bevezetett apparátusra építünk, de a beágyazási szint meghatározása Harrison és Magel [18] munkáin alapul.

Definíció 4.1. Az **irányított gráf** (directed graph) $G = (N, E)$ rendezett pár, ahol N a csúcsok (csomópontok), E az élek halmaza. Egy $e \in E$ él csúcsok egy $(x, y) \in (N \times N)$ rendezett párja, azaz $E \subset (N \times N)$.

Definíció 4.2. Ha $(x, y) \in E$ egy él a $G = (N, E)$ irányított gráfban, akkor az x csúcs az y **közvetlen megelőzője** (*immediate predecessor*). Hasonlóan az y csúcs az x **közvetlen rákövetkezője** (*immediate successor*).

Jelölés 4.3. Egy $y \in N$ csúcs közvetlen megelőzőinek halmazát jelölje $IP(y) = \{x \in N | (x, y) \in E\}$ Hasonlóan, egy $x \in N$ csúcs közvetlen rákövetkezőinek halmaza: $IS(x) = \{y \in N | (x, y) \in E\}$

Definíció 4.4. Egy $y \in N$ csúcs **bemenő fokszáma** (*indegree*) $|IP(y)|$. Hasonlóan, y csúcs **kimenő fokszáma** (*outdegree*) $|IS(y)|$.

Definíció 4.5. Egy p **út** (*path*) a $G = (N, E)$ irányított gráfban élek egy olyan $(x_1, x_2), (x_2, x_3), \ldots, (x_{k-1}, x_k)$, sorozata, ahol $\forall i[1 \leq i < k] \Rightarrow (x_i, x_{i+1}) \in E$. Ilyenkor szokás p-t az x_1-ből x_k-ba vezető útnak is nevezni.

Jelölés 4.6. A $P = path(x_1, x_k)$ jelölje az x_1 csúcsból x_k-ba vezető útat. Ha x_i csúcs rajta van az $x_1, \ldots x_k$ úton, annak jelölése: $x_i \in path(x_1, x_k)$. Ha x_i többször is előfordul az x_1-ből x_k-ba vezető úton, akkor az előfordulások számát jelöljük $sum(x_i \in path(x_1, x_k))$-val.

4.2. A program vezérlési szerkezete

Definíció 4.7. A $G = (N, E, s, t)$ négyes **vezérlési gráf**, ahol N a csúcsok véges, nemüres halmaza, E az irányított élek véges, nemüres halmaza. Az $s \in N$ a **kezdőcsúcs**, amelynek bemenő fokszáma $IP(s) = 0$. A $t \in N$ a **végcsúcs**, amelynek kimenő fokszáma $IS(t) = 0$

Jelölés 4.8. A $G = (N, E, s, t)$ vezérlési gráf esetén jelölje $N' = N \setminus \{s, t\}$ a gráf *nemtriviális csúcsainak* halmazát.

A vezérlési gráf a fentiek alapján olyan vezérlési szerkezeteket ír le, melyek egyedi belépési ponttal (s) és egyedi kilépési ponttal (t) rendelkeznek. Ez pusztán technikai konstrukció, amely lehetővé teszi, hogy kódblokkokról, alprogramokról (eljárásokról, függvényekről) és hasonló programkonstrukciókról beszéljünk. A gyakorlatban számos esetben az eljárásoknak több látszólagos visszatérési pontja létezik. (Ilyen valósul meg egy eljáráson belüli több **return** utasítás használatakor, vagy egy kiváltott és le nem kezelt kivétel esetében). Ugyanakkor ezek többnyire valóban csak látszólagos visszatérési pontok: a modern programozási nyelvek egy kódblokk elhagyásakor számos tevékenységet definiálnak: ilyen pl. az esetleges destruktorok meghívása, vagy az eljárás által specifikált kivételek ellenőrzése. Ezen tevékenységek elvégzése logikailag egyetlen visszatérési pontnak tekintendő, másrészt a gyakorlatban a fordítóprogram által generált kód is így viselkedik.

Az eljárásokban, alprogramokban alkalmazott többszörös belépési pont kevésbé használatos a hagyományos programozási nyelvek körében. Talán legismertebb klasszikus példája a PL/I programozási nyelv **entry** konstrukciója, melynek segítségével egy eljárást több belépési ponton keresztül is meghívhatunk, sőt az egyes belépési pontok eltérő számú ill. típusú paramétereket definiálhatnak. A PL/I

nyelv ezen konstrukcióját elsősorban objektum-szerű konstrukciók megvalósítására használták, ahol az egyes belépési pontok reprezentálták az egyes metódusokat, míg az eljárás lokális adatterülete az objektum attribútumait. Mivel a konstrukció egyrészt nem vált széleskörűen használt programozási technikává, másrészt viselkedése szimulálható a vezérlési gráf fenti definíciójával, ezért az egyedi belépési és kilépési pont feltételezése elfogadható és nem sérti az általánosságot.

A többszörös belépési és kilépési ponttal rendelkező kódblokkok egy másik nevezetes esete a Simula programozási nyelv blokkja, amelynek futását bizonyos előre meghatározott pontokon meg lehet szakítani a **detach** eljárás hívásával, majd a futását ismételten folytatni lehet ettől a ponttól [9]. Ez a mechanizmus azért is különlegesen érdekes, mert a Simula a mai objektum orientált programozási nyelvek egyik közvetlen előzményének tekinthető és az osztály mai fogalmára erőteljes hatást gyakorolt. A példányosult blokk az első detach pontig hasonlóan fut, mint a „hagyományos" kódblokkok, ott azonban átadja a vezérlést, de lokális adatait (lényegében belső állapotát) megtartja, és látható belső eljárásai ill. attribútumai is elérhetőek a külvilág számára.

Könnyű észrevenni a mai objektum orientált nyelvek osztály megvalósításával való párhuzamot. A blokk lokális változóiból keletkeztek az osztály attribútumai, belső eljárásaiból pedig metódusai. A példányosított Simula blokk az első detach pontig futva inicializálja lokális változóit – állapotát. Mindezt ma egy külön nevesített eljárás: a konstruktor végzi. A mai objektum orientrált nyelvek a Simula megközelítésével szemben különválasztják az osztály adatszerkezetét és eljárásait, ide értve konstruktorait és (amennyiben létezik) a destruktorát. A Simula által bevezetett modell alapvetően ma is érvényes, de a több belépési és kilépési ponttal rendelkező, saját lokális adatterületen állapotát tárolni képes kódblokk helyett több metódust definiálunk, melyek mindegyike egy-egy belépési és kilépési ponttal rendelkezik.

A modern programozási nyelvek megengedik a függvények és az operátorok paraméter alapján történő túlterhelését *overloading* (ADA, C++, Java nyelvek), illetve az alapértelmezett paraméterértékek használatát (ADA, C++ nyelvek). A túlterhelt függvények és operátorok használatát szokás **név szerinti** polimorfizmusnak, míg az alapértelmezett paramértékek használatát **érték szerinti** polimorfizmusnak is nevezni [6].

A függvények és operátorok túlterhelése esetében egyértelműen több, különböző paraméterekkel definiált, de azonos nevű alprogramról beszélhetünk, melyek mindegyike egy–egy belépési és kilépési ponttal rendelkezik. Ez a programnyelvi konstrukció egyértelműen illeszkedik a felvázolt modellre.

Az érték szerinti polimorfizmus (alapértelmezett paraméterek definiálása, és

használata alprogramok esetében) vizsgálatára tekintsük az alábbi C++ függvénydeklarációt, mely alapértelmezett paraméterekkel rendelkezik:

```
date create_date( int year, int month =1, int day =1);
```

A *create_date* függvény egy (év, hó, nap) hármasból hoz létre egy dátum objektumot. A hó és nap paramétereket elhagyhatjuk a függvény meghívásából, ebben az esetben ezek értéke 1 lesz. Az alábbi hívások tehát mind érvényesek:

```
date x1 = create_date( 2000, 11, 21);
date x2 = create_date( 2000, 11);
date x3 = create_date( 2000);
```

és rendre $(2000, 11, 21)$, $(2000, 11, 1)$ illetve $(2000, 1, 1)$ értékű dárumot hozza létre. Ez a három hívás azonban ugyanazt az *egyetlen* függvényt aktivizálja, méghozzá ugyanazon belépési ponton keresztül. Valójában a *create_date* eljárás egy hagyományos háromparaméteres függvény, mely nem is szerez tudmást arról, hogy a három felsorolt hívási forma közül melyik aktivizálta. Az alapértelmezett paraméter–értékeket a *hívó* eljárás és nem a *hívott* értelmezi. Ebből egyértelműen következik az is, hogy egy alprogram komplexitása nem növekszik amiatt, hogy alapértelmezett értékű paraméterekkel rendelkezik.

Definíció 4.9. A $G = (N, E, s, t)$ vezérlési gráfban $n \in N$ **predikátum csúcs**, ha $\mid IS(n) \mid > 1$.

Nyilvánvaló, ha $n \in N$ predikátum csúcs, akkor $n \in N'$.

Figyeljük meg, hogy a modern programozási nyelvekben a predikátum fogalma jelentősen meghaladja a hagyományos programozási nyelvek predikátum jelentését, ahol azt lényegében leszűkíthettük az egyszerű vezérlési szerkezetek (elágazás, ciklus) használatára.

A következő programrészlet egyetlen elágazást tartalmaz, melynek igaz ága egy (két eljáráshívást tartalmazó) szekvencia, hamis ága pedig üres.

```
if ( t[i] + b > 0 )
{
    f();
    g();
}
```

E programrészlet látszólag egyetlen predikátum csúcsot tartalmaz. Mi történik azonban, ha a + operátort a programozó túlterhelte, és ez a függvény kivételt dob? Ugyancsak kivételt dobhat (méghozzá akár más típusút) az indexelés operátora is, pl. az indexhatárok megsértésekor. Nyilvánvalóan ezen kódrészlet komplexitása nem felel meg a McCabe cikkében felvázoltaknak.

A fentiek miatt predikátum csomópontnak kell tekintenünk minden olyan kódrészletet, ahonnan a vezérlés több irányban folytatódhat. Ilyen kódrészlet lehet egy eljáráshívás is, ahol a hívott eljárás kivételt dobhat. Miután számos programozási nyelvben lehetőség van operátorok túlterhelésére (pl. Ada, C++, C#), mely operátorok maguk is kiválthatnak kivételeket, nyilvánvaló, hogy a hagyományos vezérlő szerkezeteken (elágazás, ciklus) keresztül történő predikátum definiálás meglehetősen kevessé használható [52].

Definíció 4.10. Egy $G = (N, E, s, t)$ vezérlési gráfban **szekvenciális kódblokknak** nevezzük az élek egy olyan $(x_1, x_2), (x_2, x_3), \ldots (x_{k-2}, x_{k-1}), (x_{k-1}, x_k)$, sorozatát, ahol

$$\forall i[1 \leq i \leq k] \Rightarrow IS(x_i) = 1$$

A P program egy szekvenciális kódblokkjának tehát azon P-beli tokenek sorozatát nevezzük, melynek végrehajtása az első tokennel kezdődik, majd folyamatosan a szekvenciában egymást követő tokenek végrehajtásával folytatódik, és közben nem tartalmaz predikátumot. A predikátumról tett megjegyzés alapján a szekvenciális kódblokk egyes tokenjeinek végrehajtása nem válthat ki kivételt. A szekvenciális kódblokk nem egyértelmű, a leghosszabb ilyen sorozatot ezért külön definiáljuk:

Definíció 4.11. Egy $G = (N, E, s, t)$ vezérlési gráf adott $n \in N$ csomópontján áthaladó szekvenciális kódblokkjai közül a maximális hosszúságú sorozatot **alapblokknak** nevezzük.

A következőkben az egyes kódrészletek beágyazottságának mélységét kívánjuk definiálni. Tekintettel kell azonban lennünk a predikátum csomópontokról tett fenti megállapításokra. Míg Howatt és Baker definícióiban a beágyazottság mértékét a nem struktúrális konstrukciók (pl. goto utasítás) lehetősége miatt kellett indirekt módon, a *dominátorok* és a *predikátum csúcs legnagyobb alsó határa* segítségével definiálni, jelen esetben ennek a definíciós rendszernek átvételét a kivételkezelés megengedése motiválja.

A *scope* szám definíciójának soron következő bevezetése Harrison és Magel [18] munkáin alapul.

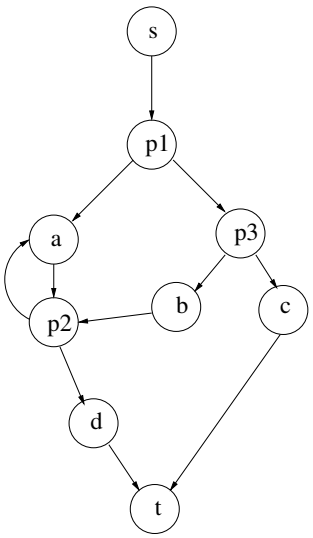
Definíció 4.12. Legyen $G = (N, E, s, t)$ vezérlési gráf, és $p, q \in N$ csúcsok.

- A p csúcs **dominátora** q-nak G-ben, ha $\forall P[P = path(s, q)] \Rightarrow p \in P$.
- A p **valódi dominátora** q-nak, ha dominátora és $p \neq q$.
- A p **közvetlen dominátora** q-nak, ha p valódi dominátora q-nak és $\forall r \in N$ esetén, ha r valódi dominátora q-nak, akkor r dominátora p-nek.

Definíció 4.13. $G = (N, E, s, t)$ vezérlési gráf, $p, q \in N$ csúcsok. Az p-ből q-ba vezető **első előfordulás utak halmaza** (*first occurence path*):

$$FOP(p, q) = \{path(p, q) | sum(q \in path(p, q)) = 1\}$$

Az első előfordulás utak halmaza p-ből q-ba $FOP(p, q)$ azokat az utakat tartalmazza, melyekre q pontosan egyszer fordul elő a $path(p, q)$ úton. Figyeljük meg, hogy a McCabe metrikához hasonlóan az önálló vezérlési utak kiválasztása a cél.



A fenti vezérlési gráf esetében tekintsük a p_1, p_2 csomópontokat.

$$FOP(p_1, p_2) = \{(p_1, a, p_2), (p_1, p_3, b, p_2)\}$$

Ugyanakkor FOP nem tartalmazza azokat a csomópontokat, melyek a p_2 csomóponton is áthaladó ciklus tartalmazna. Így nem része az első előfordulás utak halmazának: (p_1, a, p_2, a, p_2).

Definíció 4.14. Adott $G = (N, E, s, t)$ vezérlési gráf, és $p, q \in N$, Legyen $MP(p, q)$ azon csúcsok halmaza, melyek rajta vannak valamely $FOP(p, q)$-beli úton:

$$MP(p, q) = \{v \mid \exists P \, [\, P \in FOP(p, q) \wedge v \in P \,] \,\}$$

Az előző vezérlési gráf példáján $MP(p_3, t) = \{a, b, c, d, p_2, t\}$ Figyeljük meg, hogy egy $G(N, E, s, t)$ esetében $MP(s, t) = N$.

Definíció 4.15. Adott $G = (N, E, s, t)$, és $p \in N, IS(x) > 1$ predikátum csúcs. Az p predikátum **alsó határai** (*lower bounds*)

$$LB(p) = \{v \mid \forall r \forall P \, [\, r \in IS(p) \wedge P \in FOP(r, t) \;\Rightarrow\; v \in P \,]\}$$

Miután egy vezérlési gráf egyetlen t végponttal rendelkezik, ezért szükségszerűen létezik egy olyan v pont a gráfban, ahol valamely p predikátum csomópontból kiinduló útvonalak ismét találkoznak. Ez a v csomópont minden egyes olyan útvonalon rajta van, amely a p predikátum csomópont közvetlen rákövetkezőiből a t végpontba vezet. Ez a tulajdonság azonban nem egyedi, minden q-ból t-be vezető úton elhelyezkedő csomópontra igaz. Ezért ki kell emelnünk az *első* ilyen csomópontot.

A p predikátum alsó határai lényegében a *valódi dominátor* (Definíció 4.12.) inverz fogalma. Tekintsük azt a G' gráfot, amely úgy lett konstruálva, hogy az eredeti G vezérlési gráfban minden egyes él irányát megfordítjunk. Ebben az esetben p alsó határa G-ben pontosan p valódi dominátorai G'-ben. Ez a csomópont G'-ben a p csomópont *közvetlen dominátora*, amit a G gráfban p *legnagyobb alsó határa*-ként nevezünk.

Definíció 4.16. Adott $G = (N, E, s, t)$, és $p \in N, IS(x) > 1$ predikátum csúcs. A p predikátum **legnagyobb alsó határa** (*greatest lower bound*)

$$GLB(p) = \{q \mid \; q \in LB(p) \wedge \forall r \, [\, r \in (LB(p) \setminus \{q\}) \Rightarrow r \in LB(q)]\}$$

Példánkban $LB(p_1) = LB(p_3) = t$, és $LB(p_2) = d, t$. Ugyanakkor $GLB(p_1) = GLB(p_3) = t$ és $GLB(p_2) = d$.

Definíció 4.17. Adott $G = (N, E, s, t)$ vezérlési gráf, és $p \in N$ predikátum csomópont, a p csúcs által **predikált csúcsok halmaza**, vagy **hatóköre**:

$$Scope(p) = \{n \mid \exists q \, [\, q \in IS(p) \wedge n \in MP(q, GLB(p)) \,]\,\} \setminus \{\, GLB(p) \,\}$$

A *hatókör* fogalma definiált minden $x \in N$ csúcsra, de ez a halmaz csak a predikátum csomópontokra lesz nemüres. Ennek oka, hogy a nem predikátum csomópontok esetében $IS(x) = \{GLB(x)\}$. A példában szereplő vezérlési gráf esetében a hatókörök az alábbi halmazok lesznek:

$Scope(p_1) = \{a, b, c, d, p_2, p_3\}$
$Scope(p_2) = \{a, p_2\}$
$Scope(p_3) = \{a, b, c, d, p_2\}$

Definíció 4.18. Adott $G = (N, E, s, t)$, vezérlési gráf, és $x \in N$ csúcs. Az x csúcsot **predikáló csúcsok halmaza**:

$$Pred(x) = \{p \mid x \in Scope(p)\}$$

.

Definíció 4.19. A $G = (N, E, s, t)$ vezérlési gráf $x \in N$ **csomópontjának beágyazottsági mélysége** (*nesting depth*):

$$nd(x) = \mid Pred(x) \mid$$

Definíció 4.20 A $G = (N, E, s, t)$ **vezérlési gráf teljes beágyazottsági mélysége**:

$$ND(G) = \sum_{n \in N'} nd(n)$$

A példában szereplő vezérlési gráf esetében az alábbi predikáló halmazokat tudjuk megállapítani:

$Pred(p_1) = 0$
$Pred(p_2) = \{p_1, p_2, p_3\}$
$Pred(p_3) = \{p_1\}$
$Pred(a) = \{p_1, p_2, p_3\}$
$Pred(b) = \{p_1, p_3\}$
$Pred(c) = \{p_1, p_3\}$
$Pred(d) = \{p_1, p_3\}$

A teljes beágyazottsági mérték pedig $ND = 13$.

Dunsmore és *Gannon* [10] azt javasolják, hogy a vezérlési gráf komplexitását a gráf csomópontjainak átlagos beágyazási mélységeként definiáljuk. Cikkükben kizárólag strukturált programokkal foglalkozva a csomópontok beágyazási mélységét rekurzívan definiálják. Egy csomópont, mely nincsen beágyazva, 0 beágyazottsági mértékű. Amennyiben egy p predikátum csomópont beágyazottsági mértéke i, akkor egy p-t közvetlenül követő $Scope(p)$-beli csomópont beágyazottsági mértéke $i+1$.

Korábban láttuk, hogy egy adekvát mértéknek kielégítően kell működnie *nem struktúrált* programokra is. Dunsmore és Gannon mértékét Howatt és Baker [22] kiterjeszti nem stuktúrált esetekre is:

$$AVG(G) = \frac{ND(G)}{\mid Pred(x) \mid}$$

Howatt és Baker beágyazási mélysége strukturált programok esetén megegyezik a Dunsmore és Gannon által javasolttal.

Harrison és *Magel* [18] a vezérlési gráf teljes beágyazottsági mélységeként az egyes csomópontok komplexitási mértékének összegét adják meg. Ezzel lényegében a McCabe metrika filozófiáját követik, kiegészítve a beágyazás fogalmával. A beágyazottsági mérték alkalmazása azon a feltételezésen alapul, hogy egy predikátum megértése (részben) a predikátum csomópont hatókörében álló csomópontok megértését feltételezi.

Definíció 4.21. Egy $G = (N, E, s, t)$ vezérlési gráf **hatókör–száma** (*scope number*):

$$SN(G) = \sum_{n \in N'} (\mid Scope(n) \mid +1)$$

Az analógia nyilvánvaló a McCabe mértékkel. A predikátum csomópontok egyszerű összegzése helyett azonban az egyes predikátum csomópontok beágyazottsági mélységét összegezzük.

Howatt és Baker [22] megmutatja, hogy szoros kapcsolat létezik a vezérlési gráf teljes beágyazottsági mértéke és a hatókör–szám között:

$$SN(G) = \mid N' \mid + ND(G)$$

4.3. Az alprogramok szerepe

Ahogy az előző szakaszból látható, a vezérlési gráf egyetlen összefüggő, irányított gráfként értelmezett a McCabe-i definícióktól Howatt és Baker kompexitási metrikájáig. Ez a metódus a gráf egyedi kezdő- és végcsúcsával alkalmas a beágyazottsági mélység hatékony kifejezésére, ahogy azt az előző fejezetben láttuk. Ez a gráf a program futási idejű hívási láncát követi, és nem tükrözi a forráskód struktúráját. A vezérlési gráf úgy viselkedik, mintha a teljes program egyetlen eljárásként kerülne implementálásra.

Ugyanakkor vegyük észre, hogy a program megértését az egyes tevékenységek világos, elkülöníthető részekre, *alprogramokra* bontása nagymértékben elősegíti. A vezérlési gráf és az azon definiált mértékek segítenek az egyes eljárások bonyolultságának megméréséhez, gyakorlati, tapasztalati felső korlátot adva az alprogramok bonyolultságának, de nem adnak támpontot a program nagyobb léptékű felosztásához: eljárásokra bontásához.

Ennek az ellentmondásnak feloldására vezetjük be a **programgráf** (*programmgraph*) fogalmát. A programgráf fogalma – eredetileg Fóthi és Nyéki–Gaizler [15] által bevezetve – alkalmazható az eljárásokra bontott program leképezésére. A program eljárásokra bontása – dekompozíciója – legtöbbször csökkenti a teljes program

bonyolultságát. Programgráf segítségével ki tudjuk mutatni a bonyolultság csökkentését a különböző dekompozíciós technikák alkalmazása esetén.

Ábrázoljuk a programunkat – egyetlen összefüggő vezérlési gráf helyett – vezérlési gráfok halmazával. Az egyes vezérlési gráfok megfelelnek a program forráskódja által definiált függvényeknek és eljárásoknak. Az egyes vezérlőgráfok kezdőcsúcsát az eljárások neveivel címkézzük. Egyes csúcsok címkéi hivatkozásokat tartalmazhatnak más vezérlőgráfok kezdőcsúcsaira: ezzel ábrázoljuk az alprogramok hívását. A meghívott alprogramok vezérlési gráfjának végcsúcsát elhagyva a vezérlés a hívó gráfba tér vissza.

Definició 4.22. Egy $\mathcal{P} = \{G \mid G = (N, E, s, t)\}$ **programgráf** (*program-graph*) vezérlési gráfok halmaza, ahol az egyes vezérlési gráfok kezdőcsúcsait egyedi címkékkel jelöljük.

Jegyezzük meg, hogy Fóthi és Nyéki–Gaizler programgráf definíciója [15] tartalmaz két további megkötést a programgráfra vonatkozóan. Előírták egy főprogram létezését, *main* kezdőcsúcs címkével. A program végrehajtása ezen a ponton kezdődik. Az összes többi vezérlési gráf kezdőcímkéjére pedig van legalább egy hivatkozás valamely vezérlési gráfból.

Az első feltétel a program végrehajtási sorrendjének meghatározásához szükséges. A második feltétel garantálja, hogy a programgráf olyan eljárásokat tartalmaz, amelyeket valóban meg is hívunk valamely vezérlési ágon. Ezeket a feltételeket azért hagyom el, hogy a programgráf fogalmát változtatás nélkül tudjam alkalmazni az osztályok bonyolultságának definiálásához. Egy osztály metódusai között nincsen explicite hívási sorrend definiálva, és az sem szükségszerű, hogy egy programból egy osztály mindegyik tagfüggvényét meghívjuk.

Definíció 4.23. Egy $\mathcal{P} = \{G \mid G = (N, E, s, t)\}$ programgráf bonyolultsága a programgráfot alkotó G vezérlési gráfok hatókör–számainak összege:

$$\mathcal{C}(\mathcal{P}) = \sum_{G \in \mathcal{P}} SN(G)$$

A programgráf bonyolultságára adott definíciónk azt a tapasztalatot tükrözi, hogy egy–egy eljárás vagy függvény vezérlési struktúráját maradéktalanul megérthetjük kizárólag az adott alprogram forráskódjának vizsgálatával. A teljes program bonyolultsága ezen alprogramok bonyolultságának összege.

Joggal vethető fel, hogy vannak olyan esetek, amikor egy alprogram megértéséhez szükség van a teljes program más részleteinek ismeretére is. Tekintsük az alábbi programrészletet:

```
extern void g(), h();
int k;

int f(int i)
{
    if ( i + k > 0 ) {
        g();
        return 1;
    }
    else  {
        h();
        return -1;
    }
}

int main()
{
    k = 2;
    if ( f(-1) > 0 )
        printf("A");
    else
        printf("B");
    return 0;
}
```

Az f függvény if utasítása predikátum az f alprogramban. A feltétel kiértékelése – és így az f eljárásban aktuálisan végrehajtott utasítások – a *main* függvény kódjától függnek. A *main* függvényben végrehajtott utasítások az f visszatérőértékétől függenek.

Az egyes alprogramok között négy úton történhet adatcsere: globális változók segítségével, paraméterátadás segítségével, függvények visszatérési értékén keresztül vagy az alprogramban kiváltott és valamely más eljárásban elkapott kivételek útján. A 4.4 fejezetben látni fogjuk ezek szerepét a bonyolultsági mérték kiszámításában.

Amennyiben progamunk egyetlen eljárásból áll, bonyolultsági mérőszámunk megegyezik a *Howatt–Baker* és *Harrison–Magel* szerzőpárosok által a vezérlési gráf bonyolultságára adott mértékkel. A programgráf az egyes eljárások bonyolultságának mérésekor figyelembe veszi a beágyazottsági mértéket.

Fóthi és Nyéky–Gaizler [15] részletesen elemzi a programgráf viselkedését a program dekompozíciójára vonatkozóan. Induljunk ki egyetlen vezérlési gráfból, mely-

nek kezőcsúcsát a *main* cimkével láttuk el. Ez megfelel annak, hogy a programot függvények és eljárások nélkül, egyetlen monolitként implementáljuk.

Válasszunk ki egy olyan részgráfot *main*-ből, melynek egyetlen kezdő–, és egyetlen végcsúcsa van, ugyanabban a beágyazottsági mélységben. Helyettesítsük ezt a részgráfot egyetlen csomóponttal, melynek cimkéje egyedi, pl. f_1. Képezzünk egy programgráfot, melynek elemei az előbbi módon módosított *main* eljárás, és a kivágott részgráf, melyet egy f_1 cimkéjű kezdőcsúccsal és egy végcsúccsal egészítünk ki.

Amennyiben egy olyan f_1 kódrészletet emeltünk ki a teljes programból, melyben nincsen predikátum csomópont, a programgráf bonyolultsága nő. Ennek az az oka, hogy az n számú kiemelt csomópont mindegyikének hatókör halmaza üres, (lásd Def. 4.17.), ezért e csomópontok eltávolításával az $ND(G)$ nem csökken. A *main* gráf komplexitását adó:

$$SN(G) =\mid N' \mid + ND(G)$$

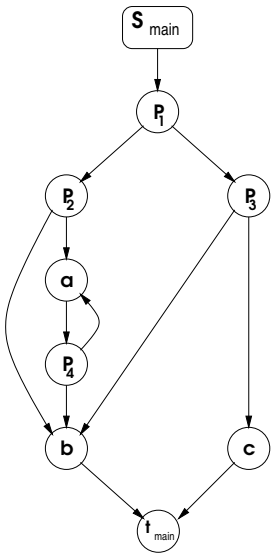
képletben $\mid N' \mid$ csökken $n-1$-el: n kivágott csomópontot helyettesítettünk egy darab „eljáráshívási" csomóponttal. Megjelenik ugyanakkor a programgráf új komponense: f_1, melynek komplexitása – lévén mindegyik csomópont beágyazottsági száma 0 – megegyezik f_1 csomópontjainak számávalm n-nel. A programgráf bonyolultsági mértéke pontosan egyel – a bevezetett extra eljáráshívás csomóponttal – megnő.

Meg kell jegyeznünk, hogy egy szekvenciális blokk kiemelése is eredményezheti a teljes programgráf bonyolultságának csökkenését. Ez fordul elő abban a gyakori esetben, amikor egy olyan kódsorozatot emelünk ki, ami több helyen is előfordul az eredeti vezérlési gráfban. Ez a programozók azon tapasztalatát tükrözi, mely szerint kívánatos az un. *block–copy programozás* elkerülése[8].

Egyes szekvenciális blokkok külön eljárásba történő kiemelése még akkor is ajánlatos lehet, ha evvel a teljes programgráf – a teljes program – bonyolultsága megnő. Szempont lehet ugyanis az egyes alprogramok bonyolultsági mértékének csökkentése, még a teljes komplexitás növelése árán is. Emlékezzünk arra, hogy McCabe eredeti tapasztalatai is az egyes eljárások bonyolultságának maximalizálásáról szólnak.

Amennyiben f_1 tartalmaz predikátum csomópontot is, akkor a program dekompozíciója eredményezheti a programgráf bonyolultságának csökkenését.

[8]Azé a helytelen programozói magatartásé, amikor egy kódrészletet átmásolnak a forrás egy másik helyére, és ott esetileg módosítják azt. Ezek a kódrészletek rövidtávon szemantikai hibákat, hosszútávon nehezen karbantartható forrást eredményeznek.



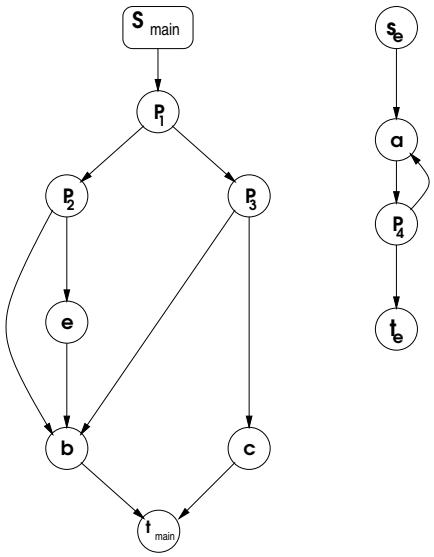
Tekintsük a fenti ábrákat. Az eredeti *main* eljárás négy predikátumn csomópontot tartalmaz, ezen belül a p_4 predikátum csomópont és az a csomópont ciklust alkotva ideális célpontja a dekompozíciónak.

$Scope(p_1) = \{p_2, p_3, p_4, a, b, c, \}$
$Scope(p_2) = \{p_4, a\}$
$Scope(p_3) = \{b, c\}$
$Scope(p_4) = \{p_4, a\}$

$Pred(p_1) = 0$
$Pred(p_2) = \{p_1\}$
$Pred(p_3) = \{p_1\}$
$Pred(p_4) = \{p_1, p_2, p_4\}$
$Pred(a) = \{p_1, p_2, p_4\}$
$Pred(b) = \{p_1, p_3\}$
$Pred(c) = \{p_1, p_3\}$

Ennek megfelelően $ND(main) = 12$, $\mid N'_{main} \mid = 7$, így: $SN(main) = \mid N'_{main} \mid + ND(main) = 19$.

Most emeljük ki az (a, p_4) kódblokkot egy külön eljárásba, amit e cimkével látunk el. Az alábbi két komponensből álló programgráfot kapjuk:



A hatókör és predikátum halmazok a következőképpen alakulnak:

$Scope(p_1) = \{p_2, p_3, b, c, e\}$
$Scope(p_2) = \{e\}$
$Scope(p_3) = \{b, c\}$
$Scope(p_4) = \{p_4, a\}$

$Pred(p_1) = 0$
$Pred(p_2) = \{p_1\}$
$Pred(p_3) = \{p_1\}$
$Pred(b) = \{p_1, p_3\}$
$Pred(c) = \{p_1, p_3\}$
$Pred(e) = \{p_1, p_2\}$
$Pred(p_4) = \{p_4\}$
$Pred(a) = \{p_4\}$

Ennek megfelelően $ND(main) = 8$, $\mid N'_{main} \mid = 6$, $SN(main) = 14$, és $ND(e) = 2$, $\mid N'_e \mid = 2$, $SN(e) = 4$

$$\mathcal{C}(\mathcal{P}) = \sum_{G \in \mathcal{P}} SN(G) = 18$$

4.4. Az adatok szerepe

A komplexitási mérték kitrerjesztését egyetlen vezérlési gráfról a vezérlési gráfok halmazából álló programgráfra az motíválta, hogy képesek legyünk a program dekompozíciójából adódó komplexitás–változást kifejezni. Ez a dekompozíció az eddigiekben csak a program *vezérlési szerkezetén* volt értelmezve. Ennek korlátaira már utaltunk, amikor az egyes alprogramok kölcsönös összefüggéseit tárgyaltuk.

A program vezérlőszerkezete dekompozíciójának egyik fontos következménye, hogy a globális változók helyett az alprogramra, illetve kódblokkra lokális változókat használhatunk. Valójában a változók hatókörének csökkentése az eljárások, függvények alkalmazásának egyik legfőbb indoka. Ugyanakkor a mai programozási nyelvek – különösen az objektum orientált paradigmát követők – fokozottan építenek a típusok dekompozíciójára is. Általánosságban véve a modern programozási nyelvek legalább olyan, vagy nagyobb hangsúlyt helyeznek az adatszerkezetre, mint a vezérlési struktúrára.

A következőkben kiterjesztem az eddig tárgyalt bonyolultsági mértéket, hogy az képes legyen kifejezni a program vezérlőszerkezete által felhasznált adatok és az adattípusok bonyolultságát is.

A program vezérlőszerkezetét és adatstruktúráját együtt kezelő bonyolultsági mérték ötlete Fóthi Ákos és Nyékyné Gaizler Judit cikkében [15] jelenik meg. Az ott definiált *data–flowgraph* a program vezérlési szerkezetét egészíti ki adatcsomópontokkal (*data node*) és azokat a vezérlési szerkezethez kapcsoló adatkapcsolati élekkel, *data reference edge*-ekkel. Az adatcsomópontok azokat a változókat jelképezik, melyeket a program felhasznál egy adott függvényen, vagy eljáráson belül. Az adatkapcsolati élek tartalmazzák az információt arról, hogy az egyes változókat mely vezérlési csomópontok használták.

Fóthi és Nyéky–Gaizler az adatkapcsolati élek definícióját *nem irányított* élként adják meg. Ez a megközelítés azonban erősen leegyszerűsíti az adatkezelés mikéntjét. Elvesztjük az információáramlás irányára vonatkozó ismeretünket, ez pedig befolyásolhatja a program, vagy alprogram megértésére irányuló tevékenységünket. E miatt a fenti szerzőkkel közösen írt cikkemben [48] finomítom az adatkapcsolati él fogalmát:

Definíció 4.24. Legyenek N és D két véges halmaz, sorrendben egy program vezérlési és adat csomópontjai. Az **adatkapcsolati él** (*data reference edge* egy (x_1, x_2) pár, ahol $(x_1 \in N \wedge x_2 \in D)$ és $(x_1 \in D \wedge x_2 \in N)$ állítások közül pontosan az egyik igaz.

Az adatkapcsolati él tehát *irányított*, iránya az adatáramlás irányára utal. Amenynyiben $x \in N$ egy vezérlési csomópont és $d \in D$ egy adat csomópont; (x, d) adatkapcsolati él azt jelenti, hogy az x csomópontban *módosítottuk* a d adatot, míg

(d, x) azt, hogy x olvasta d-t. Természetesen előfordulhat olyan eset is, amikor egy utasítás írja és olvassa is ugyanazt az adatot: ekkor két ellentétes irányú adatkapcsolati él köti össze x és d csúcsokat.

Definíció 4.25. Legyen N a vezérlési csúcsok halmaza, $E \subseteq (N \times N)$ a vezérlési élek halmaza. D az adatcsúcsok, $R \subseteq (N \times D) \cup (D \times N)$ az adatkapcsolati élek halmaza. Egy **adat és vezérlési gráf** (továbbiakban AV gráf) egy $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ négyes, ahol $\mathcal{N} = N \bigcup D$, a csúcsok nemüres halmaza, $\mathcal{E} = E \bigcup R$, az élek halmaza, $s \in N$ és $t \in N$ pedig a vezérlési szerkezet opcionális kezdő– és végcsúcsa.

Itt ismét eltérünk a Fóthi–Nyéky féle data–flowgraph definíciójától. Fóthi és Nyéky célja a procedurális programokra jól működő bonyolultsági mértékek kiterjesztése objektum–orientált programokra. Ennek céljából javasolják az adatok használatának vizsgálatát, mint a mérték egy összetevőjét. Mértékük azonban csak teljes programokra alkalmazható, nem teszi lehetővé, hogy egyes komponenseket, pl. önálló osztályokat, az őket hívó kliens kód[9] nélkül mérjünk. Az általunk javasolt mérték megengedi, hogy szélső esetben akár vezérlőszerkezet nélküli adatstruktúrákra ugyanúgy alkalmazzuk, mint adatszerkezet nélküli hagyományos vezérlési gráfokra. Ennek érdekében nem kötjük ki, hogy N (a vezérlési csúcsok halmaza) vagy D (az adatcsúcsok halmaza) külön–külön nemüres halmazok legyenek.

Jelölés 4.26. Jelölje s az AV gráf vezérlési struktúrájának opcionális kezdő csomópontját, és t a végcsomópontját. Szokás szerint $\mathcal{N}'$ jelölje $\mathcal{N} - \{s, t\}$ halmazt, az AV gráf nemtriviális csomópontjainak halmazát.

Tekintsük az alábbi (C–szerű programnyelven írt) kódrészletet. Figyeljük meg, hogy a felhasznált változók deklarációit nem tüntettük fel a kódban.

```
void main()
{
    if ( d1 > 0 ) {
        do {
            d2 = a();
        } while ( d4 < 10 );
        printf("%d", d3);
    }
    else {
        ++d3;
    }
}
```

[9]Kliens kód alatt az adott osztályra hivatkozó programrészletet értem.

Mértékünk nem foglalkozik az egyes konkrét programozási nyelvek szintakszisával. Nem tudjuk, hogy hogyan néznek ki a deklarációk és hol van helyük az adott nyelv programjaiban. Csak globális változók léteznek? Fordítási egységre lokális deklarációk? Hol van a deklarációk helye egy alprogramon belül? Számos blokk-struktúrált programozási nyelvben a deklarációk helye a kódblokkok eleje, és az első végrehajtható utasítás után már deklarációk nem következhetnek. Más nyelvek (mint pl. a C++) megengedik, hogy tetszőleges ponton, akár két végrehajtható utasítás között, egy **for** ciklus, vagy **if** elágazás fejében is, új változókat hozzunk létre.

Mértékünk fontos tulajdonsága, hogy az adatkezelést *nem a deklaráció helye* alapján, hanem az adatok *konkrét használati pontjain* méri. Természetesen ez *implicite* méri a deklaráció helyét is: egy lokális láthatóságú változót csak az adott kódkörnyezetben (pl. alprogramon belül) tudunk használni.

Tekintsünk a fenti programnak megfelelő AV gráfot az alábbi ábrán. A gráf egyetlen komponensből áll (miután egyetlen eljárást vizsgálunk), két predikátum csomóponttal: $p_1 : if(d1 > 0)$ és $p_2 : while(d4 < 10)$, és három további vezérlési csomóponttal: a egy (paraméter nélküli) függvényhívás, b függvényhívás egyetlen nemkonstans paraméterrel ($d3$), és c az inkrementáló operátor. A vezérlőszerkezetet a d_1, d_2, d_3, d_4 adatcsomópontok egészítik ki, melyek a hasonló nevű vátozókat reprezentálják:

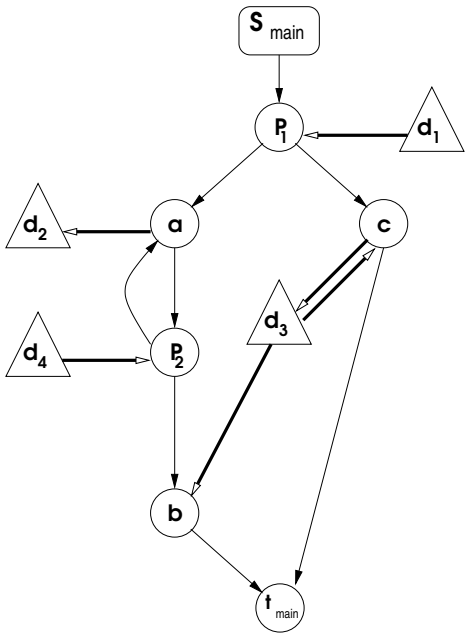


Fig.3.

A p_1 predikátum csomópont a d_1 adatot olvassa, p_2 pedig a ciklusfeltételben d_4-et olvassa. Az a csomópont egy függvényt hív meg, és a visszatérő értékét értékül adja d_2 változónak, azaz írja azt. A c vezérlési csomópont inkrementálja d_3-at. Az inkrementálás[10] szükségszerűen olvassa, és írja is a cél adatterületet. Ezt jelzi a két, ellentétes irányú adatkapcsolati él c és d_3 között. Végül a b csomópont a *printf* függvény hívását reprezentálja, melynek egyetlen adatparamétere d_3, melyet a paraméterátadáskor olvasunk.

Hogyan definiálhatjuk az AV gráf bonyolultságát? Az eddigi példákban a bonyolultság kiszámításához meghatároztuk a predikátumcsomópontok hatókörét, és ebből a vezérlési csomópontok predikáló halmazát. Ez a tevékenység nem volt más, mint annak megállapítása, hogy az egyes vezérlési csomópontok végrehajtása mely predikátumcsomópontokban hozott döntésektől függ. A predikátumcsomópont legnagyobb alsó határa fogalmának bevezetése is lényegében arról szólt, hogy az e pont alatti vezérlési tevékenység már nem függ a predikátumban történt döntéstől.

Ezt a gondolatmenetet természetes módon ki lehet terjeszteni az adatok kezelésére. Egy predikátumcsomópontban döntés születik arról, hogy mely vezérlési ágat választjuk ki az aktuális futás során: mely csomópontokat hajtjuk végre. Ez a döntés ugyanakkor arról is szól, hogy mely adatokat kívánjuk kiolvasni, vagy módosítani. Ilyen értelemben a predikátumcsomópont hatóköréhez adatcsomópontok is tartoznak, melyek olvasása/írása a csomópont döntésétől függ. Ezek pont azok az adatok, melyeket a predikátumcsomópont hatókörében levő vezérlési csomópontok írnak/olvasnak. (Az adatok kezelése mindig vezérlési csomópontokon keresztül történik.)

Ugyanakkor figyeljük meg, hogy az egyes vezérlési csomópontok végrehajtása mindig kiértékeli a hozzá kapcsolt adatcsomópontokat. Ilyen értelemben beszélhetünk a nem-predikátum csomópontok hatóköréről, amely az adott vezérlési csomóponthoz kapcsolódó adatokból áll. Ez eltérés a vezérlési gráfoktól, ahol csak a predikátumcsomópontoknak van nemüres hatóköre.

Definíció 4.27. Adott $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ AV gráf, $\mathcal{N} = N \cup D$, ahol N vezérlési-, D az adatcsomópontok halmaza, $p \in N$ (nem feltétlenül predikátum) csomópont. A p csúcs **adat hatóköre** (D–hatóköre):

$$DScope(p) = \{d \in D \mid \exists n \in N \wedge n \in Scope(p) \cup \{p\} \wedge ((n,d) \in \mathcal{E} \vee (d,n) \in \mathcal{E})$$

Definíció 4.28. Adott $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ AV gráf, $\mathcal{N} = N \cup D$, ahol N vezérlési-, D az adatcsomópontok halmaza, $p \in N$ predikátum csomópont. A p csúcs **adat és vezérlési hatóköre** (AV hatóköre):

[10] A C és C++ nyelvekben szintaktikai és szemantikai különbség van a prefix és a postfix inkrementáló operátorok között, ami a két művelet bonyolultsági mértékében is kimutatható. Ebben a példában az egyszerűség kedvéért elhanyagoljuk ezt a tényt.

$$AVScope(p) = Scope(p) \cup DScope(p)$$

Mivel az író és az olvasó adatkapcsolati éleket külön szeretnénk figyelembe venni a komplexitás kiszámításakor, technikai okokból érdemes minden d adatcsomópontot két technikai csomóponttal (író és olvasó) helyettesíteni, így meg lehet különböztetni azt a jelenséget, hogy egy predikátumcsomópont döntése egy d adat írását, olvasását, esetleg mindkettőt eredményezi. Ennek a bonyolultság kiszámításánál van szerepe. Mivel ez pusztán technikai konstrukció, mely csak arra szolgál, hogy a hatókör halmazokkal és ne *bag* adatszerkezettel[11] legyen reprezentálható, ezért az ábrákon és számításainkon ezt a helyettesítést nem ábrázoljuk. Mindig vegyük figyelembe azonban, hogy a hatókör kiszámításakor nem az adott vezérlési csomóponttal kapcsolatban álló adatcsomópontok számát, hanem az adatkapcsolati éleket számoljuk.

Amennyiben a predikátumcsomópontok hatókörét kiterjesztjük az adatcsomópontokra is, értelmes az adatcsomópontok predikáló halmazáról beszélni: azon predikátumcsomópontok halmaza, amely egy adat olvasását/írását eldönti. Ezen predikátumhalmazon keresztül pedig a vezérlési csomópontok használatához teljesen hasonlóan definiálhatjuk az AV gráf bonyolultságát.

Definíció 4.29. Adott $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$, AV gráf, és $x \in \mathcal{N}$ csúcs. Az x csúcsot **predikáló csúcsok halmaza**:

$$AVPred(x) = \{p \mid x \in AVScope(p)\}$$

.

A fenti definícióban x egyaránt lehet vezérlési vagy adatcsomópont.

Definíció 4.30. A $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ AV gráf $x \in \mathcal{N}$ csomópontjának beágyazottsági mélysége:

$$nd(x) = \mid AVPred(x) \mid$$

Definíció 4.31 A $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ **AV gráf teljes beágyazottsági mélysége**:

$$ND(\mathcal{G}) = \sum_{n \in \mathcal{N}'} nd(n)$$

Definíció 4.32 A $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ **AV gráf bonyolultsága** legyen

$$\mathcal{C}(\mathcal{G}) = \mid \mathcal{N}' \mid + ND(\mathcal{G})$$

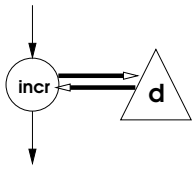
[11] Olyan halmaz, amelyben számontartjuk, hogy az egyes elemek *hányszor* szerepelnek a halmazban.

Az AV gráf bonyolultsága tehát a vezérlőszerkezetétől és az adatkezelésétől függ. A vezérlőszerkezet – a predikátumcsomópontok segítségével – meghatározza a vezérlési csomópontok, és ezen keresztül az egyes adatok kezelésének beágyazottsági mélységét. A teljes komplexitás ezen beágyazottsági mértékek és a csomópontok (mind adat, mind vezérlési) számának kifejezése.

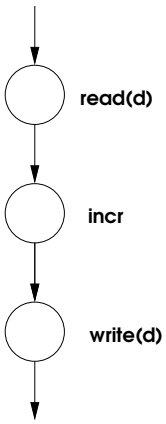
Ugyanehhez az eredményhez vezetne egy másik meggondolás is. Tekintsünk egy x vezérlési csomópontot, amely egy d adat inkrementálását végzi, pl. az alábbi kódrészletben:

```
int d = 1;
//....
++d;  // x csomópont
```

Az alábbi AV gráf részlet a fenti kód alapján készült:



Tekinthetjük a fenti kódrészlet adatcsomópontjait olyan speciális „vezérlési csomópontoknak”, melyek egyetlen tevékenysége az illető adat értékének kiolvasása vagy írása. Ebben az esetben ezek az új vezérlési csomópontok közvetlenül megelőzik, illetve követik azt a csomópontot, amely az adatkezelést végzi.



Az így számolt hatókör ill. komplexitás érték megegyezik azzal, ahogyan azt a **4.30** – **4.32** definíciókban megadtuk.

Figyeljük meg, hogy a definíciók során sehol sem tettük fel, hogy az AV gráf összefüggő, illetve vezérlési vagy adatszerkezete nem üres. (Ugyanakkor feltettük, hogy nem lehet egyszerre üres mindkettő. Ez esetben is definiálhatnánk a komplexitást: természetesen nullának.)

A nem összefüggő AV gráfok esetében a teljes gráf komplexitását ki tudjuk számolni az egyes részgráfok segítségével, az alábbi képlettel:

Lemma 4.33. Adott $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ AV gráf, ahol $\mathcal{N} = N \cup D$, N a vezérlési csúcsok, D az adatcsúcsok halmaza. Ha $\mathcal{G} = \bigcup \mathcal{G}_i$, ahol $\mathcal{G}_i$ vezérlő csúcsainak halmaza N_i, és $N = N_1 \cup N_2 \cup ... \cup N_k$ úgy, hogy $\forall (i,j)\ i \neq j \Rightarrow N_i \cap N_j = 0$ akkor

$$\mathcal{C}(\mathcal{G}) = \mid \mathcal{N}' \mid + \sum_{\mathcal{G}_i} ND(\mathcal{G}_i)$$

Az AV gráf képes leírni a programokat abban az esetben is, ha egynél több eljárásból állnak. A több alprogramból álló programok vezérlőszerkezete (definíció szerint) mindig diszjunkt. A vezérlési gráfok esetében tehát egyszerűen összeadtuk az egyes alprogramok bonyolultsági értékeit. Az adatszerkezetek figyelembe vételekor azonban ez a tulajdonság megváltozik.

Elképzelhető, hogy több alprogram, egy vagy több közös adatszerkezettel rendelkezik. Ilyen eset fordul elő, amikor pl. egy *globális változót* több eljárás írhat vagy olvashat. Ilyenkor az adat több vezérlőszerkezethez kapcsolódik adatkapcsolati éllel. Ezen adatok tehát diszjunkt vezérlési ágakon elhelyezkedő predikátumok hatókörében is szerepelhetnek, és a bonyolultság számításakor egynél többször kell számolnunk őket. Az $ND(\mathcal{G}_i)$ értékek tehát összegződnek. Ugyanakkor ezek az adatcsomópontok csak egyszer jelennek meg, mint $\mid \mathcal{N}' \mid$ részei.

Amennyiben ezen eljárások diszjunkt adatszerkezettel rendelkeznek, úgy komplexitásaikat egyszerűen összeadva most is megkapjuk az egész program bonyolultságát.

Első ránézésre feleslegesnek tűnik, hogy megkülönböztetjük az adatszerkezetek írását és olvasását, illetve az, hogy abban az esetben, ha az adatcsomópontot egyszerre írjuk és olvassuk is, akkor kétszeresen számoljuk ezt az eseményt bonyolultsági mértékünkben. Ugyanakkor egy kellően finom felbontású mértéknek szükségszerűen törődnie kell evvel a különbséggel.

Tekintsük az alábbi (C++ programozási nyelven írt) példát, melyben az f eljárást a d paraméterrel hívjuk meg.

```
int d = 1;

//....

f(d);  // f csomópont
```

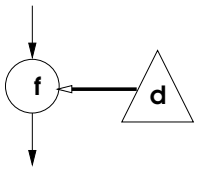
Abban az alprogramban, ahol ezt a kódrészletet vizsgáljuk, nem ismerjük az f függvény kódját, de ezen a ponton nem is szeretnénk, hogy egy másik kódmodulban megírt függvény impelmentációs részleteit ismernünk kelljen. Így nem tudhatjuk, hogy az f függvény csak olvassa, (kezdőértékétől függetlenül) csak írja a paraméterét, esetleg írja is és olvassa is. Márpedig az ismertetett kód bonyolultsága nagymértékben függhet ettől:

```
int d = 1;

//....

f(d);  // f csomópont

if ( d > 1 )
    // ...
else
    // ...
```

Látszik, hogy amennyiben d értéke megváltozhat az f csomópontban, más a kódrészletünk bonyolultsága, mint akkor, ha d semmiképpen sem változhat meg. Erre az információra természetesen a fordítóprogramnak is szüksége van; tipikusan más optimalizációs stratégiákat kell kiválasztania a két esetben. Így a modern programozási nyelvek megkövetelik ezen információ fordítási idejű biztosítását – tipikusan az f eljárás deklarációjában:

```
int d = 1;
void f( int );  // érték szerinti paraméterátadás
//....

f(d);  // f csomópont
```

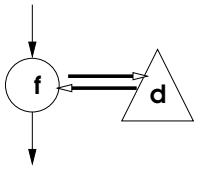
Érték szerinti paraméterátadás esetén az $f(int)$ eljárás az aktuális paraméter pillanatnyi értékét lemásolja, és az f eljárás ezen másolaton dolgozik. A másolaton történő módosítások nem hatnak vissza az eredeti aktuális paraméterre. Ebben az esetben az alábbi AV gráfhoz jutunk:



A d adatcsomópontot csak olvassa az f vezérlési csomópont, így csak egyetlen adatkezelési él kapcsolja d-t f-hez, a d csomópontot egyetlen egyszer számoljuk az f-et környező programrészlet hatókör, és ebből következően bonyolultsági számításaiban.

```
int d = 1;
void f( int& );  // referencia szerinti paraméterátadás
//....

f(d);  // f csomópont
```

A referencia szerinti (más programozási nyelvek esetében cím szerinti) paraméterátadás esetében az aktuális paraméter nem másolódik le, hanem az alprogram az eredeti tárterületen végez műveleteket. Ebben az esetben az eljárás által a paraméteren végzett módosítások permanensek, és láthatóvá válnak a hívó kódrészletben.



Az AV gráf e verziójában a d csomópont kettős adatkapcsolati éllel csatlakozik az f vezérlési csomóponthoz. A hatókör és bonyolultság számításakor ezt a kapcsolatot

kétszer vesszük számításba. Az előző példával összehasonlítva ez a kód magasabb bonyolultsági mérőszámot fog produkálni.

Az előző példában azt is érdemes megjegyezni, hogy a második tipusú hívás alkalmazása magasabb bonyolultságot eredményez attól függetlenül, hogy mi az f eljárás tényleges kódja. Így még az is elképzelhető, hogy az f ténylegesen nem kísérli meg a paraméter módosítását. Ez egyben egy jellemző példa Weyuker 6. axiómájára (63. oldal): különböző tartalmú, de egyforma bonyolultságú bevezető kódrészletek, (itt az f eljárás különböző módú paraméterátadással definiált kódjai) után ugyanazon programkód bonyolultsági mérőszáma eltérő lehet.

A következőben vizsgáljuk meg az AV gráf viselkedését a program dekompozíciója esetén. Az alábbi programot vizsgáltuk a 35-36. oldalakon. Most abból a szempontból vizsgáljuk meg, hogyan alakul a gráf bonyolultsága, ha a monolit kódot eljárásokra bontjuk.

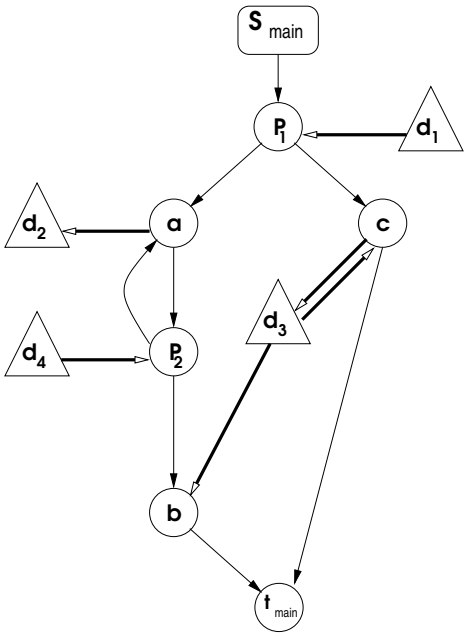


Fig.3.

Először írjuk fel a gráf csomópontjaira a *hatókör* halmazokat. Ne felejtsük el, hogy a hagyományos (adatmentes) vezérlési gráffal ellentétben itt a nem-predikátum csomópontoknak is lehet nemüres (adatokat tartalmazó) hatókör halmaza.

Figyeljünk arra, hogy d_3 adatcsomópontot háromszor is számoljuk, c írja és olvassa, b olvassa, erre a többszörös használatra a d_{3_i} technikai jelöléssel utalunk.

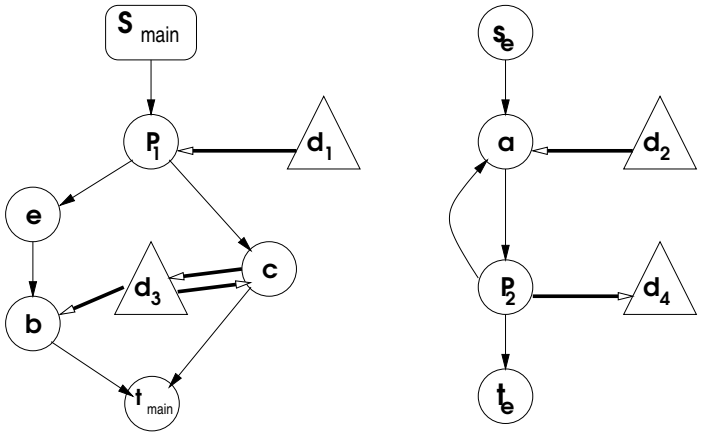
(Ezt a későbbiekben – lévén tisztán technikai okokból van rá szükség – el fogjuk hagyni.)

$Scope(p_1) = \{d_1, a, d_2, p_2, d_4, c, d_{3_1}, d_{3_2}, b, d_{3_3}\}$
$Scope(p_2) = \{a, d_2, p_2, d_4\}$
$Scope(a) = \{d_2\}$
$Scope(b) = \{d_{3_1}\}$
$Scope(c) = \{d_{3_2}, d_{3_3}\}$

$Pred(p_1) = 0$
$Pred(p_2) = \{p_1, p_2\}$
$Pred(a) = \{p_1, p_2\}$
$Pred(b) = \{p_1\}$
$Pred(c) = \{p_1\}$
$Pred(d_1) = \{p_1\}$
$Pred(d_2) = \{p_1, p_2, a\}$
$Pred(d_3) = \{p_1, p_1, p_1, b, c, c\}$
$Pred(d_4) = \{p_1, p_2\}$

$ND(\mathcal{G}) = 18 \quad \text{és} \quad \mathcal{C}(\mathcal{G}) = 27.$

A program bonyolultságát csökkenthetjük a program dekompozíciójával. A fenti programból kiemeljük az (a, p_4) szekvenciális kódblokkot, hogy egy e cimkéjű eljárást hozzunk létre belőle.



A dekompozíció akkor eredményez bonyolultság-csökkenést, ha egy olyan kódblokkal végezzük azt, amely *lokális* adatkapcsolati élekkel rendelkezik, azaz olyanokkal, amelyek csak ezen blokk vezérlési csomópontjaihoz kapcsolódik. Ekkor ezen adatok

kezelése csak a helyi eljárástól függ, és kiemeljük őket a hívó kód predikátumainak hatóköréből. Ez megfelel a programozási nyelvek lokális változóinak viselkedésével.

$Scope(p_1) = \{d_1, e, c, d_{3_1}, d_{3_2}, b, d_{3_3}\}$
$Scope(p_2) = \{a, d_2, p_2, d_4\}$
$Scope(a) = \{d_2\}$
$Scope(b) = \{d_{3_1}\}$
$Scope(c) = \{d_{3_2}, d_{3_3}\}$
$Scope(e) = 0$

$Pred(p_1) = 0$
$Pred(p_2) = \{p_2\}$
$Pred(a) = \{p_2\}$
$Pred(b) = \{p_1\}$
$Pred(c) = \{p_1\}$
$Pred(d_1) = \{p_1\}$
$Pred(d_2) = \{p_2, a\}$
$Pred(d_3) = \{p_1, p_1, p_1, b, c, c\}$
$Pred(d_4) = \{p_2\}$

$ND(\mathcal{G}) = 14 \quad \text{és} \quad \mathcal{C}(\mathcal{G}) = 24.$

Figyeljük meg, hogy metrikánk nem alkalmazza a *deklaráció* fogalmát, így nem beszélhetünk a *deklaráció helyéről* sem. Ahogyan azt korábban már elmondtuk, mértékünk nem a *deklaráció*, hanem az *aktuális adathasználat* helye alapján számítja az adatkezelés bonyolultságát.

Milyen módszereink vannak az adatkezelés bonyolultságának csökkentésére? Természetesen az egyik legegyszerűbb lehetőség az adatcsomópontok *számának* csökkentése. Ezt elérhetjük, ha az egyedi adatokból összetett adatstruktúrákat, típusokat alkotunk. A következő példában egy komplex számokat ábrázoló adattípust hozunk létre:

```
struct complex
{
    double re;
    double im;
};
```

Amennyiben egy kliens program komplex számokkal kíván műveletet végezni, két lehetősége van. Vagy létrehoz két, egymástól független változót (x_re és x_im), melyekben egy komplex érték valós és képzetes részét tárolja, vagy létrehoz egy *complex* típusú c változót. Az első esetben a komplex számon végzett műveletek két különböző utasítás (pl. értékadás) során történnek, két különálló adaton:

```
double x_re, x_im;
double y_re, y_im;

x_re = y_re;
x_im = y_im;
```

A második esetben a komplex számot egyetlen adatcsomópontként kezelhetjük, amely elemi műveletekkel (pl. értékadással) kezelhető:

```
struct complex cx, cy;

cx = cy;
```

Ennek természetesen kisebb a bonyolultsági értéke, mint az előző esetnek, viszont feltételezi, hogy aktuális programozási nyelvünk elemi műveleteket tud végezni összetett adatszerkezeteken.

Hogyan számolunk el a második eset *struct complex* típusának definíciójával? Ez egy olyan kódrészlet, amely nyilvánvalóan növeli a program bonyolultságát, ezért hozzá kellene számolnunk a kliens kódhoz. Miután az AV gráf komplexitása értelmezett, de nulla üres vezérlőszerkezet esetén, javítanunk kell ezen a tulajdonságon, hogy beszélhessünk az adattípusok bonyolultságáról.

5. Az osztály bonyolultsága

Az imperatív programokat Wirth szavaival élve [44] úgy jellemezhetjük, mint az algoritmusok és adatstruktúrák együttesét. A programkészítés módszeres útja Wirth szerint a program által elvégzendő funkcionalitás és az ehhez felhasználandó input és output adatok meghatározása. Ezek után lépésről-lépésre lebontjuk a funkcionalitást alprogramokra; szekvenciát, ciklust és elágazást használva. Minden lépésben kevésbé absztrakt függvényeket és eljárásokat definiálunk. Ez a folyamat addig folytatódik, amíg az alprogramokat már össze tudjuk állítani az implementációs nyelv által megengedett elemi utasításokból, és adatokból. Ilyen értelemben az imperatív nyelvek elsődleges absztrakciós mechanizmusa az eljárások, függvények létrehozása.

A programozási nyelvek fejlődésével az *adatabsztrakció* egyre jelentősebb szerepet kapott [37]. A modul alapú adatabsztrakció jelentősen lecsökkentette azt a kódmenynyiséget, melyet az egyes programegységek felhasználóinak ismernie kell ahhoz, hogy megbízható programkódot írjon egy konkrét feladat implementálására. A nyolcvanas évek során a programozási nyelvek absztrakciós mechanizmusait kezdték kiterjeszteni az adatszerkezetek és absztrakt adattípusok területére [41]. Az adatabsztrakciót az *adatelrejtés* (encapsulation) támogatja, mely segítségével az adatok hatókörét és láthatóságát leszűkíthetjük. Elsősorban a *modulok* (module) és *csomagok* (package) segítségével a Modula-2 és az ADA nyelv tett komoly előrelépést az adatabsztrakciónak a procedurális absztrakcióval egy szintbe emelésére.

Az *objektum-orientált* programozás tovább lép az adatabsztrakció útján, az *osztály* (class) fogalma segítségével. Az osztály, mely először a Simula 67 nyelvben jelenik meg [9], az *objektumok* létrehozására (példányosítására) szolgáló sablon. Az egyes objektumok közös adatszerkezetét az osztályban írjuk le, mint *adattagokat*, és itt definiáljuk a rajtuk végzett műveleteket; a *tagfüggvényeket*, vagy *metódusokat*[12].

Az osztály által leírt adatszerkezet, amely a példányosítás során keletkező objektumok lokális adatterülete – az egyes objektumok *állapotát* hivatott leírni. Ezen állapot kiolvasására vagy módosítására szolgálnak a tagfüggvények. A tagfüggvények egy halmaza – a *konstruktorok* – arra a célra szolgálnak, hogy a frissen példányosult objektum állapotát az osztály invariáns feltételének megfelelően inicializálják. Amennyiben egy osztályhoz definiáltunk konstruktort, akkor csak valamely konstruktor segítségével példányosíthatunk objektumokat. Egyes OOP nyelvekben definiálhatunk *desktruktor* metódust is. Ennek az objektum élettartamának végén van szerepe: feladata az objektum által lefoglalt erőforrások szabályszerű felszabadítása. Más nyelvekben az erőforrások felszabadítása automatikusan, például *szemétgyüjtő eljárással* (garbage collection) történik.

[12] A mai objektum-orientált nyelvekben az adattagokon és metódusokon kívül számos más összetevő is definiált, így pl. publikus és implementációs célú típusok, stb.

Azt a kódot, amely egy osztály definícióját felhasználja; így objektumokat példányosít az osztályból, annak adattagjaira vagy metódusaira hivatkozik, az adott osztály *kliens kódjának*, (vagy röviden: kliensének) nevezzük. Az osztály *adatelrejtési mechanizmusa* korlátozza a kliens kód hozzáférését a tagokhoz: szétválasztva a külvilág számára elérhetetlen *privát* tagokat (adatok és segédfüggvények) és az elérhető, felhasználható *publikus* adattagokat és metódusokat.

5.1. Az osztály bonyolultsági mérőszáma

A következőkben megmutatjuk, hogy az előző fejezetben definiált metrika természetes módon alkalmazható az objektum-orientált programokra, és bemutatjuk, hogy használata segítséget jelent számos, az objektum-orientált programozás természetére jellemző technika vizsgálatához. Külön szeretnénk kiemelni, hogy az AV gráfokra alkalmazott bonyolultsági mérték **nem** tartalmaz objektum-orientált specifikus eszközöket.

Definíció 5.1. Az $\mathcal{O} = (\mathcal{N}, \mathcal{E})$ **class-gráf** egy olyan AV gráf, melyben az osztály tagfüggvényei külön kezdő és végcsúccsal rendelkező AV gráfok, az osztály attribútumai pedig adatcsomópontok, melyek több tagfüggvényhez is kapcsolódhatnak.

Az $\mathcal{N} = N \cup D$ a csomópontok halmaza, ahol $N = N_1 \cup ... \cup N_k$ a vezérlési csomópontok halmaza, melyeket egyértelműen felbonthatunk az egyes műveletek diszjunkt vezérlési szerkezetére. A D halmaz az adatcsomópontok halmaza, ahol $D = A \cup D_1 \cup ... \cup D_k$, ahol A az attribútumok halmaza, $D_1, ..., D_k$ az egyes metódusok (diszjunkt) lokális adatcsomópontjai.

Ez a természetes modell az osztály fogalmára. Kifejezi azt a tényt, hogy az osztály objektumai adatcsomópontok koherens halmazából állnak (az attribútumok), melyeken az osztály tagfüggvényei (metódusai) végeznek műveleteket. Az osztály attributumait a metódusok megosztva használják, Minden egyes tagfüggvény saját vezérlőszerkezettel rendelkezik, amely diszjunkt a többi tagfüggvénnyel. Az egyes tagfüggvények saját kezdő- és végcsúccsal rendelkeznek, ami jelképezi, hogy ezeket a metódusokat egymástól függetlenül végrehajthatjuk.

Az egyes tagfüggvények természetesen deklarálnak és használnak lokális változókat. Miután a lokális változókat – a programozási nyelv definíciója alapján – csak az őket deklaráló eljárások láthatják, és használhatják, az azokat reprezentáló adatcsomópontokat csak egyetlen tagfüggvény vezérlési csomópontjaihoz kapcsolhatjuk. Modellünkben egy tagfüggvény lokális változója ugyanolyan bonyolultsági értékű, mintha az osztály attribútumaként deklaráltuk volna. Ez megfelel nemcsak az elméleti megfontolásoknak, de az általános programozói gyakorlatnak is.

Látni kell, hogy annak garanciája, hogy egy attribútumot *csak* az osztályon belül használjuk, az láthatóságának korlátozása, pl. *private* deklaráció segítségével.

Hasonlóan, annak garanciája, hogy egy adatcsomópontot egyetlen tagfüggvényből használjuk, az adat lokális deklarációja az eljáráson belül.

Definíció 5.2. Az **osztály bonyolultsági mérőszáma**

$$\mathcal{C}(\mathcal{O}) = \mid \mathcal{N}' \mid + \sum_{\mathcal{G} \in \mathcal{O}} ND(\mathcal{G})$$

A definícióban szereplő $\mathcal{N} = N \cup D$ az AV gráf definíciója szerint egyaránt tartalmazza az egyes tagfüggvények vezérlési csomópontjait, és az adatcsomópontokat. Az adatcsomópontok lehetnek az osztály attribútumai, vagy az egyes tagfüggvények lokális változói.

Az osztály bonyolultsága egyaránt függ az attribútumoktól (az osztály adatszerkezetétől) és a metódusok kódjától (a tagfüggvények a vezérlési szerkezetétől, és a tagfüggvényekben felhasznált lokális adatoktól). A definíció ugyanakkor tükrözi azt az általános tapasztalatot, hogy a jó objektum-orientált programokban igen erős kapcsolatot találunk az osztályon belüli attributumok között, míg gyenge kapcsolatokat tapasztalunk az osztályok között.

Fontosnak tartjuk, hogy rámutassunk a mérték azon jelentésére, miszerint az objektum-orientált programokban az adatszerkezet struktúrális bonyolultsága és a metódusok bonyolultsága *együttesen* határozza meg az osztály bonyolultságát. Értelmetlen e két tulajdonságot ketté választani és külön tárgyalni. Hasonlóan félrevezetőnek tartjuk azokat a mértékeket, melyek a metódusok számát veszi tekintetbe, elhanyagolva az egyes metódusok – nagyon is eltérő – belső bonyolultságát.

5.2. A metrika kiszámítása

Egy osztály bonyolultsági mérőszámának kiszámítását megkönnyítendő vezessük be a tagfüggvények lokális adatszerkezetének fogalmát, azoknak az adatoknak azonosítására, melyek az egyes tagfüggvénynek lokális változóit jelölik.

Jelölés 5.3. Legyen az $\mathcal{O} = (\mathcal{N}, \mathcal{E})$ osztály egy diszjunkt felbontása

$$\mathcal{O} = \mathcal{A} \cup (\bigcup_{\mathcal{G}_i \in \mathcal{O}} \mathcal{G}_i)$$

ahol $\mathcal{A}$ az attribútumok halmaza, $\mathcal{G}_i = (\mathcal{N}_i, \mathcal{E}_i, s_i, t_i)$ pedig az osztály tagfüggvényei, úgy, hogy $\mathcal{N} = \mathcal{A} \cup (\bigcup \mathcal{N}_i)$ és $\mathcal{N}_i \cap \mathcal{A} = 0$. Jelölje a $\mathcal{G}_i$ csomópontjait $L_{\mathcal{G}_i}$.

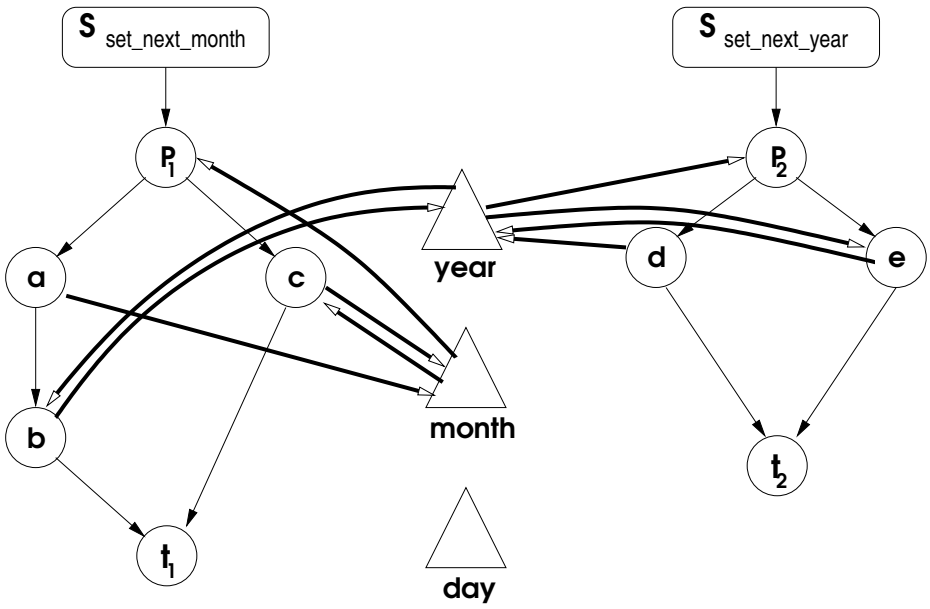
Lemma 5.4. Az osztály bonyolultsági mérőszáma egyenlő az attribútumok száma és a tagfüggvények külön-külön számított bonyolultságának összegével

$$\mathcal{C}(\mathcal{O}) = \mid \mathcal{A} \mid + \sum_{\mathcal{G}_i \in \mathcal{O}} (ND(\mathcal{G}_i) + \mid L_{\mathcal{G}_i} \mid)$$

Az alábbi kódrészlet egy dátum-típust reprezentáló osztály részlete. Az osztály állapotát három attributummal ábrázoljuk: az év, hó és nap mezőkkel. A member-függvények közül kettőt mutatunk be: a *set_next_month*() és a *set_next_year*() eljárásokat. (Vélhetően további metódusok is léteznek, mint a *set_next_day*() és *getter* függvények, az attributumok lekérdezésére.)

```
class date
{
public:
    void set_next_month()
    {
        if ( month == 12 )  { // p1
            month = 1;        // a
            ++year;           // b
        }
        else  {
            ++month;          // c
        }
    }
    void set_next_year()
    {
        if ( year == -1 )  {  // p2
            year = 1;         // d
        }
        else  {
            ++year;           // e
        }
    }
    void set_next_day() // ...
    // ...
private:
    int year;
    int month;
    int day;
};
```

Az osztályt a már ismert módon ábrázoljuk AV gráffal. Figyeljük meg a *year* adat-csomópontot, melyet két metódus is használ.



A bonyolultságot a hatókör és predikáló halmazok felírásával számítjuk ki, teljesen megegyező módon az AV gráfoknál használtakkal. Az egyes hatókörökben fel kell sorolnunk az adott csomópont által predikált csomópontok halmazát. Ez a predikátumcsomópontok esetében vezérlési- és adatcsomópontokból, a nem predikátum csomópontok esetében csak adatcsomópontokból áll. Külön kell számolnunk minden adatkapcsolati élt, de a jelölésben – egyszerűsítésként – ugyanazt a cimkét használjuk. További jelölésbeli egyszerűsítést alkalmazhatunk: a címkék ismételt felsorolása helyett a (n) együttható jelölje az adott címke előfordulási számát az adott hatókörben illetve predikáló halmazban.

A *set_next_month* metódus bonyolultsága:

$Scope(p_1) = \{a, b, c, (4)month, (2)year\}$
$Scope(a) = \{month\}$
$Scope(b) = \{(2)year\}$
$Scope(c) = \{(2)month\}$

$Pred(p_1) = 0$
$Pred(a) = \{p_1\}$
$Pred(b) = \{p_1\}$
$Pred(c) = \{p_1\}$
$Pred(year) = \{(2)p_1, (2)b, \}$
$Pred(month) = \{(4)p_1, a, (2)c\}$

$\mathrm{ND}(set\_next\_month) = 14$

$| N' | = 4$

A *set_ next_ year* metódus bonyolultsága:

$Scope(p_2) = \{d, e, (4)year\}$
$Scope(d) = \{year\}$
$Scope(e) = \{(2)year\}$

$Pred(p_2) = 0$
$Pred(d) = \{p_2\}$
$Pred(e) = \{p_2\}$
$Pred(year) = \{(4)p_2, d, (2)e\}$
$Pred(month) = 0$

$\mathrm{ND}(set\_next\_year) = 9$
$| N' | = 3$

Az attribútumok száma az osztályban:

$| A | = 3$

Az osztály teljes bonyolultsági száma:

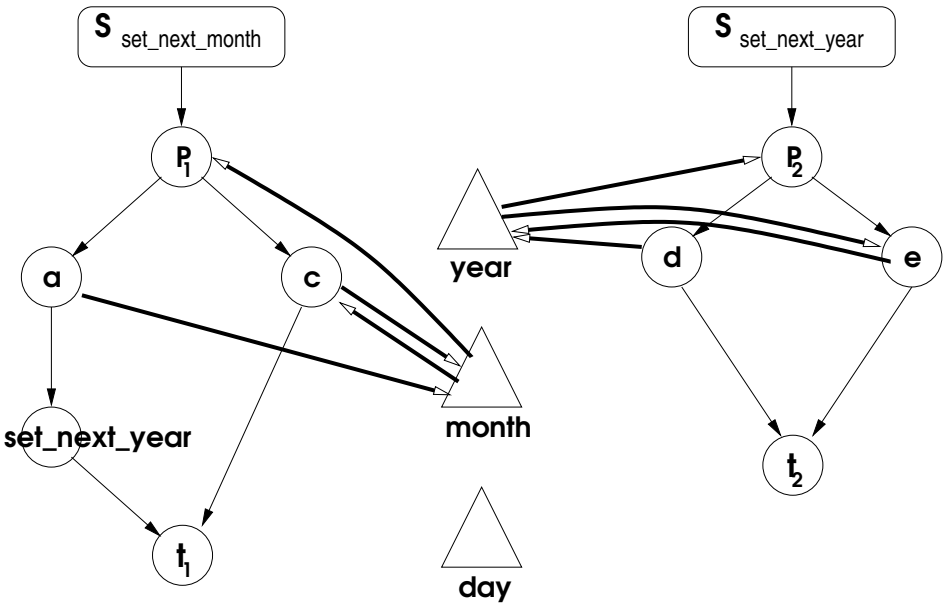
$\mathcal{C}(date) = 18 + 12 + 3 = 33$

Az 5.4 lemma segítségével az osztály bonyolultságát úgy számoltuk ki, hogy összegeztük az attribútumokat és az egyes metódusok, mint hagyományos függvények bonyolultságát. Ez utóbbi számításakor figyelembe vettük a metódus vezérlési szerkezetét, esetleges lokális változóit, és az adatkezelés komplexitását (ide számolva a metódus attribútumokon végzett adatkezelését is).

5.3. A metrika jellemzése

Tekintsük a *date* osztály előző megvalósítását. A fenti példában a *++year* utasítás a *set_ next_ month()* eljáráson belül szükségtelenül „ismeri" az év adatreprezentációját. Ezt az utasítást helyettesíthetjük a *set_ next_ year()* eljárás hívásával. Ez komoly mértékben csökkenti a kódrészlet bonyolultságát. (Arról nem beszélve, hogy az első verzió ignorálja a -1 és +1 év közötti átmenet problémáját.) Természetesen a bonyolultság csökkenésében az is szerepet játszik, hogy egy már *létező* metódust *set_ next_ year()* hívunk, így nem kell új metódust bevezetnünk, annak komplexitást növelő hatásaival.

Általában is igaz, hogy miközben nagy léptékben objektum-orientált technikákat alkalmazunk a kód komplexitásának csökkentésére, az egyes osztályok implementációjakor a hagyományos procedurális technikákat alkalmazhatjuk.



$\mathcal{C}(date) = 14 + 12 + 3 = 29$

Ez általános jelenség. Az objektum-orientált programozás egyik legfontosabb célkitűzése a kód újrafelhasználásának elősegítése. Akkor járunk el helyesen, ha osztályunkon olyan memberfüggvényeket definiálunk, melyek lefedik az osztályon végzett leggyakoribb műveleteket. Amennyiben egy művelet megvalósítható más műveletek kombinációjaként, felmerül annak a lehetősége, hogy ne közvetlenül az adatreprezentáción végezzük el az operációt, hanem már megvalósított eljárásokat hívjunk.

Objektum-orientált programozási nyelvekben ismert a *sovány* (thin) és a *kövér* (fat) interfész fogalma. Az első esetben az osztály metódusai csak azt a *minimális* tagfüggvény készletet tartalmazzák, melyek segítségével az osztály specifikációjának megfelelő összes műveletet végre lehet hajtani. A második esetben a kövér interfész az osztályon végezhető műveletek egy bővebb halmazát tartalmazza.

Gyakran előfordul, hogy ezt a bővebb eljáráshalmazt nem az adott osztály tagfüggvényeiként implementáljuk. A C++ programozási nyelvet alkalmazva szokás például globális műveleteket deklarálni az osztály definícióját tartalmazó headerfájlban. Ezek a műveletek nem tagfüggvények, szigorú *programnyelvi* értelemben nem

részei az osztálynak. Ugyanakkor *logikai* értelemben az osztály interfészét képezik, melyeket gyakran technikai okokból nem helyezünk az osztályba[13].

Külső eljárások nem férhetnek hozzá osztályunk privát attributumaihoz, így a kövér interfész eljárásai rendszerint a sovány interfész metódusait hívják, anélkül, hogy közvetlen kapcsolatuk lenne az osztály adatreprezentációjához. Ilyen metódus lehet a *date* osztályon az aktuális dátumot n nappal növelő *add(n)* művelet:

```
class date
{
public:
    void set_next_day()  //...
    // ....
};
void add( date d, int n )
{
    for ( int i = 0; i < n; ++i )  {
        d.set_next_day();
    }
}
```

Az ilyen eljárásokat ugyanúgy az osztály bonyolultságához számíthatjuk, mint a tagfüggvényeket.

5.4. Néhány szélső eset vizsgálata

Vegyük észre, hogy AV gráf definíciónk megengedi, hogy egy osztály vezérlési szerkezete teljesen üres legyen. Ebben az esetben „klasszikus" adatszerkezetet kapunk:

```
struct complex
{
    double re;
    double im;
};
```

Az ilyen típus bonyolultsága az adattagok bonyolultságának összege; ha az adattagok a konkrét programozási nyelvünk elemi típusai, akkor a tagok száma.

[13]Ilyen technikai okok vezetnek a C++ nyelvben arra, hogy a beolvasó (>>) és kiíró (<<) műveleteket szintaktikájuk miatt (bal oldali operandusuk egy standard könyvtári osztály) globális függvényként írjuk meg. Konverziós okok miatt ugyancsak globális műveletként hozzuk létre a legtöbb kommutatív operátort is.

Az ilyen adatstruktúrák is csökkenthetik a teljes program bonyolultságát, de csak abban az esetben, ha az alkalmazott nyelv képes elemi műveleteket végezni rajtuk.

Ellentétes eset is elképzelhető: amennyiben egy „osztály" olyan metódusokból áll, amelyek egyáltalán nem osztoznak közös adatokon, azaz osztályunknak nincsenek attributumai, az osztály bonyolultsága az egyes metódusok külön-külön számolt bonyolultságának összege lesz. Vegyük észre, hogy ez a klasszikus eljárásgyűjtemény, függvénykönyvtár esete, amely már McCabe vizsgálatainak is tárgyát képezte (lásd 12. oldal).

A fenti példák azt igazolják, hogy metrikánk *nem paradigmafüggő*: egyaránt képes mérni a klasszikus procedurális elven és az objektum-orientált módon készített programokat. Ez volt kitűzött célunk, mely segítségével lehetővé válik a multiparadigmás programok egységes elvek alapján történő bonyolultsági mérése.

5.5. Láthatóság

Az osztály bonyolultságára bevezetett mértékünk nem különbözteti meg az eltérő láthatósági szintekkel deklarált attribútumokat. Az objektum orientált nyelvek az osztály tagjait (adattagokat és tagfüggvényeket) különböző láthatósági szinteken deklarálják, aszerint, hogy az illető nevek elérhetőek a külvilág számára (publikus, public), az (egy lépésben) származtatott osztályok tagjai számára láthatóak (védett, protected), vagy csak a saját osztályon belül használhatók (privát, private)[14].

A „klasszikus" enkapszuláció teljes figyelmen kívül hagyása meglepő lehet az objektum-orientált programok használóinak. Ugyanakkor vegyük figyelembe, hogy a tagok láthatósági szintje egyáltalában nem befolyásolja az adott osztály bonyolultságát. Tekintsük az alábbi két C++ osztályt:

```
class date
{
public:
    void set_next_year();
    void set_next_month();
    void set_next_day();
    // ...
private:
    int year;
    int month;
    int day;
};
```

[14] A fenti láthatósági szintek meghatározása a legjellemzőbb. Ugyanakkor a Java négy, a C# pedig öt eltérő láthatósági szintet definiál.

Ebben az osztályban a „helyes” módon elrejtettük az attribútumokat. A következő példában azonban „helytelenül” minden attribútumunk publikus, azaz elérhető a külvilág felől:

```
class date
{
public:
    void set_next_year();
    void set_next_month();
    void set_next_day();
    // ...
    int year;
    int month;
    int day;
};
```

Láthatunk e két kódrészlet között bármiféle bonyolultsági különbséget? Aligha válaszolhatunk igennel. A *date* osztály azon két implementációja, amely csak a tagok láthatóságában különbözik egyenlő bonyolultságú.

Természetesen az adatok elrejtése helyes és követendő programozási technika. A két fenti megoldás közötti különbség azonban nem az implementált osztály – hanem az azt használó klienskód bonyolultságában jelentkezik. Amennyiben a kliens kód növelni akarja a hónap attribútum értékét, ezt a második esetben megteheti a *set_next_month* metódus hívásával, vagy az attribútumok közvetlen elérésével. Az első eset kisebb bonyolultsági értéket eredményez. Abban a példában, ahol a privát attribútumokat „elrejtettük” a külvilág elől nincsen választási lehetőség; a kliens kénytelen a metódushívást – és ezzel a kisebb bonyolultságú megoldást választani.

5.6. Osztályok közötti kapcsolatok

Egy C osztály adattagját egyetlen adatcsomópontnak tekintjük, függetlenül az általa reprezentált típus belső bonyolultságától. Amennyiben ez az adattag egy összetett T típus, esetleg osztály saját attribútumokkal és tagfüggvényekkel, akkor a C és T típusokat ábrázoló AV gráfok uniója reprezentálja a teljes osztályrelációt.

Ez összhangban van mind programozási nyelveink gyakorlatával, mind az elméleti megfontolásokkal. Az alábbi határidőnapló bejegyzéseket reprezentáló osztály tartalmaz egy dátum és egy string típusú attribútumot: a bejegyzett esemény idejét és leírását. Az f eljárás összehasonlít két időpontot és ennek eredményétől függően felülírja az eseményt.

```
class diary
{
public:
    void f( diary other) {
        if ( time > other.time )
            event = other.event;
        // ...
    }
    // ...
private:
    date   time;
    string event;
    // ...
};
```

Vajon a *date* típus az aktuális programozási nyelv beépített típusa, és az összehasonlítás elemi művelet ezen a típuson? Esetleg a *date* a felhasználó által megírt típus (osztály), és $x > y$ hatására meghívódik az ezen osztályon definiált *operator* $>$ művelet, $x.operator > (y)$? A *diary* osztály implementálója nem lát különbséget a két eset között, számára a két változat bonyolultsága ugyanakkora. Egy jól megírt osztály pontosan így viselkedik, használata nem ró többletterhet a kliens kódra. Ugyanígy *string* lehet „beépített típus" vagy a standard osztálykönyvtár része, nem láthatunk bonyolultságbeli különbséget a két megoldás között.

Amennyiben *date* felhasználó által implementált osztály, természetesen a *teljes* programrendszer bonyolultsága, amely tartalmazza a szabványos osztálykönyvtárakat, eljárásgyűjteményeket és a felhasználók által korábban megvalósított osztályokat, megnövekszik *date* bonyolultságával. Ez azonban kifizetődik, amennyiben a típus bonyolultsága egyszer növeli mértékünket, az objektumok egyszerűbb használata viszont minden egyes alkalommal csökkenti a rendszer AV gráfjának bonyolultságát.

Ez az objektum-orientált programozás egyik legfontosabb motivációja. Amikor pl. egy C++ osztályhoz konstruktorokat, másoló konstruktort, destruktort, értékadó operátort, és egyéb műveleteket írunk, komoly bonyolultságú feladatokat oldunk meg abból a célból, hogy az osztály használata a kliens kód számára leegyszerűsödjön. Konstans bonyolultságnövelő tevékenységünk a teljes rendszer bonyolultságát az osztály használatával lineáris mértékben csökkenti.

Az *öröklődést* hasonló módon magyarázhatjuk. A származtatott osztály számos esetben (de nem szükségszerűen) hivatkozik a bázisosztály publikus vagy protected adattagjaira vagy tagfüggvényeire. Ezeket a hivatkozásokat ugyanúgy kezeljük,

mint a procedurális programok adathivatkozásait. Motivációnk most is a speciális objektum-orientált konstrukció paradigma-független elemzése.

```
class diary : private date
{
public:
    void f( diary other) {
        if ( getTime() > other.getTime() )
            event = other.event;
        // ...
    }
    // ...
private:
    string event;
    // ...
};
```

Az előző határidőnapló *privát* *örökléssel* történő megvalósítása[15] nem tér el lényegileg az aggregáció alkalmazásától. Ugyanakkor a *time* adathivatkozást a $getTime()$ metódusra cseréltük. Egyes programozási nyelvekben külön kulcsszóval hivatkozhatunk a bázisosztályra, más nyelvekben azonban szükségünk van ilyen „getter" függvényre.

Megváltozna a rendszer bonyolultsága, ha a *diary* osztály közvetlenül a *date* osztály attribútumaira ($year$,$month$,day) hivatkozna. Ekkor adathivatkozási élek kapcsolnák össze az f metódus vezérlési csúcsait és a *date* adatcsomópontjait. Ez a bonyolultság növekedésével járna.

Ez a jelenség annak az objektum-orientált programozási tapasztalatnak felel meg, hogy a jól kialakított osztályhierarchiában erős vezérlési- és adatkapcsolatok vannak egy osztályon belül, és gyenge kapcsolatok léteznek az osztályok között.

Az öröklés jóval gyakoribb változata, amikor a származtatott osztály *publikus* módon örököl a bázisosztályból. Ebben az esetben a bázisosztály teljes interfésze elérhető lesz a származtatott objektumok használói számára. A származtatott osztály ilyenkor „kiegészíti" a bázisosztály funkcionalitását. Szemlélhetjük úgy ezt a konstrukciót, mint az osztályok gráfjainak unióját.

[15]Ebben az esetben a külvilág nem érheti el a *diary* típusú objektumok *date* részét.

6. A Weyuker axiómák

Programok szintaktikai alapokon nyugvó bonyolultságának vizsgálatára az 1980-as évektől nagy számban definiáltak mértékeket. Ezen mértékek a program szerkezetének egy vagy több elemét kiemelve nemegyszer egyoldalúan vizsgálták a program bonyolultságát.

Elaine J, Weyuker, aki akkor a New York University Courant Institute of Mathematical Sciences intézetében dolgozott, 1988-ban megjelentetett egy tanulmányt a szoftver bonyolultsági mértékek értékelésének elveiről [43]. Ebben a tanulmányban Weyuker avval érvel, hogy ahelyett, hogy az egyes mértékeket a szerzők által felvázolt példákon és esettanulmányokon keresztül próbálnánk megítélni, állítsunk fel néhány általános programjellemzőből álló állítást, melyek teljesülését elvárjuk bármely szoftver metrikától.

Ezek az állítások, melyeket – kissé megtévesztő módon – a **Weyuker axiómákként** szokás idézni. A név félrevezető, hiszen nyilvánvalóan nem matematikai értelemben vett axiómákról van szó, melyből egy konzisztens rendszer állításait vezetnénk le. Éppen ellenkezőleg; az aktuálisan vizsgált metrika definíciójából próbálunk arra következtetni, hogy a Weyuker által felállított egyes állítások teljesülnek-e az adott metrikára. Hasonló feltételeket állítottak fel Iannino és társai szoftver megbízhatósági modellekre [24]. Prather három axiómát állított fel szoftver metrikák vizsgálatára [35]. Ronald Prather axiómáira alapozva ajánlott új megközelítést szoftvertermékek definiálásához Daróczy és Varga [11].

Érdekes, hogy később Cherniavsky és Smith megmutatták hogy konstruálható olyan triviális metrika, amely teljesíti a Weyuker által felsorolt összes állítást, mégis teljességgel haszontalan [4]. A Weyuker axiómák tehát nem képeznek *elégséges* feltételt a használható metrikákra. Ugyanakkor a legtöbb általánosan használt szoftver bonyolultsági mérték az állítások közül többet nem teljesít. A McCabe-féle ciklomatikus bonyolultsági szám [30] például 4 axiómát is megsért. (Prather három axiómáját viszont mind teljesíti.) Ilyen értelemben az axiómák nem is *szükséges* feltételek. Mégis, Weyuker „axiómái" váltak a szoftver metrikák elméleti vizsgálatának általánosan elfogadott alapjául, és mi is őket használjuk fel bonyolultsági mértékünk axiomatikus vizsgálatára.

Weyuker azt az ellentmondást, amire a fentiekben utaltunk, többek között avval magyarázza, hogy a szoftver mértékek célja nincsen minden esetben pontosan definiálva. Nem világos, hogy pontosan mit mérünk: az implementálás nehézségét, a tesztelés problémáit, a kód megértésének vagy módosításának bonyolultságát? Ezek a fogalmak maguk is gyakran homályosak. Weyuker azt reméli a kritériumok formalizálásától, hogy megkönnyíti a metrikák összehasonlítását a felhasználó sajátos igényei szempontjából.

6.1. Definíciók

Weyuker az axiómák megfogalmazásához egy nyelvet definiál, aminek segítségével írja le állításait. Ez a nyelv programozási nyelv-független, és véges számú azonosítóból áll. Az azonosítókból (VAR), konstansokból és a $+$, $-$, $*$ és $/$ műveletekből *aritmetikai kifejezéseket* (EXP) állíthatunk össze.

Az *értékadó utasítás* az alábbi formátumú:

$$VAR \longleftarrow EXP$$

Predikátum-nak nevezzük azokat a logikai kifejezéseket, melyek az alábbi formák valamelyikéből állnak:

$$B1 = B2,\ B1 \neq B2,\ B1 < B2,\ B1 \leq B2$$

ahol $B1$ és $B2$ egy konstans vagy azonosító. A *programtörzs* fogalmát rekurzívan definiálja Weyuker:

1. Az értékadás programtörzs.
2. Ha **PRED** egy predikátum és **P** és **Q** programtörzsek, akkor
   IF PRED **THEN** P **ELSE** Q **END** programtörzs.
3. **IF** PRED **THEN** P **END** programtörzs.
4. **WHILE** PRED **DO** P **ENDWHILE** programtörzs.
5. Ha **P** és **Q** programtörzsek, akkor a szekvencia:
   P ; Q is programtörzs.

A (2), (3) és (4) szabályokra, mint feltételes utasításokra fogunk hivatkozni. Definiáljuk a **PROGRAM**(vars) *program-utasítás* és a **OUTPUT**(vars) *kiviteli utasítás* fogalmát is, ahol *vars* a bemeneti változók listája. Végül a *program* egy program-utasításból áll, melyet egy *programtörzs*, majd egy kiviteli utasítás követ. Gyakran fogunk hivatkozni a programtörzsre, mint a „program”-ra.

Egy adott P program esetén $P(c) = b$ jelentése az, hogy a P program a c input adatokon végrehajtva megáll, és b outputot produkálja. Két program, P és Q ekvivalenciája $P \equiv Q$ azt jelenti, hogy a két program ugyanazokon az input adatokon áll meg, és ezekre a c adatokra: $P(c) = Q(c)$.

Miután a programjainkat elemi programtörzsekből fogjuk felépíteni, ésszerű a bonyolultsági mértékeket a programtörzsre definiálni. Evvel ugyan nem fogjuk számolni a program-utasítást és a kiviteli utasítást, de ez elenyésző torzítást okoz a metrikákon.

A következő jelölésekben P, Q és R programtörzseket jelöl. $\mid P \mid$ jelöli a P programtörzs bonyolultsági számát. Feltételezzük, hogy tetszőleges P programtörzs esetében kiszámolhatjuk a programtörzs $\mid P \mid$ bonyolultságát, és $\mid P \mid$ nemnegatív. Ebből az is következik, hogy tetszőleges P és Q programtörzsek esetében:

$$\mid P \mid \leq \mid Q \mid$$

vagy

$$\mid Q \mid \leq \mid P \mid$$

azaz tetszőleges két programtörzs összehasonlítható és rendezhető[16].

6.2. Az állítások ellenőrzése

A következőkben sorba vesszük a Weyuker „axiómákat", és megvizsgáljuk, hogyan teljesülnek az egyes állítások javasolt metrikánkra.

W1: $(\exists P)(\exists Q)\ (\mid P \mid \neq \mid Q \mid)$

Az állítás szerint létezik két különböző program, mely a vizsgált metrika szerint más bonyolultsági értékkel rendelkezik. Ez a feltétel a triviális, semmitmondó metrikák kizárására szolgál.

Jelen dolgozatban javasolt metrikánk triviálisan teljesíti ezt a feltételt, és teljesíti azt a dolgozatban bemutatott valamennyi más bonyolultsági mérőszám is.

W2: Legyen c egy nemnegatív szám. Ekkor legfeljebb végesszámú olyan program létezik, melynek bonyolultsága c.

Ez az axióma a **W1** szigorítása. Nemcsak azt várjuk el a bonyolultsági metrikától, hogy legyenek olyan programok, amelyekhez különböző értékeket rendelünk; hanem azt is, hogy kellően sok osztályba sorolja őket, azaz „jól szórjon". Természetesen tekintettel kell lennünk arra a tényre, hogy a Weyuker által definiált nyelv – akárcsak a ténylegesen vizsgálandó nyelvek – végesszámú azonosítóval rendelkeznek. Ezért feltételezhetjük, hogy minden esetben adható felső korlát az utasítások ábrázolásához szükséges tárolóhelyre.

Érdekes, hogy amíg a különböző méret metrikák, illetve a Halstead metrikák megfelelnek a 2. axiómának, a McCabe-féle ciklometrikus- és a Henry és Kafura-féle adatáramlási bonyolultság már nem felel meg ennek a feltételnek. Ennek oka pl. a McCabe bonyolultság esetében abban keresendő, hogy tetszőleges hosszúságú, predikátumot nem tartalmazó utasítássorozatok „nem mérhetőek". Két program

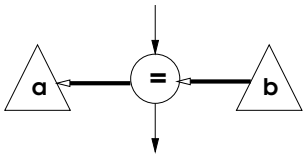
[16]Egyes metrikák vektor-értékűek. Ezek esetében vagy a felhasználó dönti el, hogy mely dimenziót kiválasztva végzi el az összehasonlítást, vagy a vektor normáját használjuk.

mindig ugyanolyan bonyolultságú, ha vezérlési szerkezetük megegyezik, függetlenül attól, hogy az egyik triviális utasításokat, a másik bonyolult számításokat végez.

A dolgozatunkban javasolt metrika vezérlési szerkezete szintén alkalmazza a „szekvenciális blokk" fogalmát, de az alprogramok szerepe c. fejezetben bevezettük az egyes eljárások hívását reprezentáló csomópontokat, az adatok szerepének vizsgálatakor pedig külön vezérlési csomópontokra bontottuk az adatokat olvasó vagy író utasításokat. Szekvenciális blokk-ként így csak azokat az utasításokat vonhatjuk össze, melyek nem predikátumok, nem végeznek adatolvasást vagy írást, és nem eljáráshívások. Ezek szerepét elhanyagolhatjuk. Ugyanakkor tekintsük a következő utasításokat:

```
a = b;
a = b + b;
a = b + b + b;
// ...
```

Amennyiben egy adatcsomópontot és egy vezérlési csomópontot egyetlen egy adatkapcsolati éllel kapcsoljuk össze, akkor az *összes* fenti utasítás az alábbi AV gráfként ábrázolható:



Ekkor metrikánk nem teljesíti a 2. axiómát. Ugyanakkor definiálhatjuk metrikánkat oly módon is, hogy a többszörös adathasználat az adatkapcsolati élek többszöröződését eredményezzék. Ekkor metrikánk már megfelel a második állításnak.

W3: Léteznek olyan $| P |$ és $| Q |$ különböző programok, melyekre $| P | = | Q |$.

A második axiómában kikötöttük, hogy elvárás metrikánktól, hogy kellően jól szórjon, azaz kellően sokféle értéket rendeljen a programokhoz. Ennek az ellentétes feltételét köti ki a harmadik axióma, amely szerint létezhet két különböző program, melynek bonyolultsága megegyezik. Ellenkező esetben előfordulhatna, hogy a metrika minden egyes programhoz más-más értéket rendel.

Ennek a feltételnek minden eddig ismertetett metrika megfelelt, természetesen az általunk javasolt metrika is. Valójában az általunk definiált bonyolultsági mérték bizonyos értelemben ebben az axiómában „teljesít" a legjobban. Tegyük fel, hogy

egy feladatot megoldhatunk többféle módon is: bonyolultabb adatszerkezet és egyszerűbb algoritmusok, vagy éppen fordítva, egyszerű adatok és bonyolult algoritmusok alkalmazása segítségével. Mivel metrikánk egyenértékűen képes mérni a program adat és vezérlési szerkezetét, ezért a két ellentétes implementációban tapasztalt eltérés az adat és vezérlési szerkezet bonyolultságában „kiegyenlítheti" egymást.

W4: $(\exists P)(\exists Q)\ (\ P \equiv Q \wedge\ \mid P \mid \neq \mid Q \mid\ )$.

Az első három tulajdonság minden értelmes metrikára igaz, attól függetlenül, hogy szoftvereket mérünk-e vagy sem. A negyedik tulajdonság – ami szintén az első szigorítása – azt hivatott garantálni, hogy mértékünk szintaktikai elemekre épül, azaz létezhet két teljesen más implementációjú program, melyek ugyanazt az eredményt szolgáltatják, és ezek bonyolultsági mértéke eltérő. Amit mérünk tehát, az nem az elvégzendő feladat – a programfüggvény – bonyolultsága, hanem a konkrét megvalósítás bonyolultsága.

Minden mérték, ami megfelel a 4. axiómának, kielégíti az 1. axiómát is. Melyek lehetnek azok a mértékek, amelyek teljesítik az 1., de nem teljesítik a 4. axiómát? Mivel két program ekvivalenciája nem eldönthető kérdés, ezért nem tudunk definiálni olyan metrikát, amely a program viselkedése alapján osztályoz. Így *gyakorlati* szempontból az első és negyedik axiómák ekvivalensek.

W5: $(\forall P)(\forall Q)\ (\ \mid P \mid \leq \mid P;Q \mid\ \wedge\ \mid Q \mid \leq \mid P;Q \mid\ )$.

A programokat úgy tekintjük, mint kisebb alprogramok, modulok egységét. Természetes feltételezés, hogy egy programot alkotó összetevők bonyolultsága ne haladja meg a teljes program bonyolultságát. Ez a feltétel egyfajta *monotonitási* kritérium mértékünkre.

A ciklometrikus bonyolultság és a különböző méret metrikák megfelelnek ennek az elvárásnak, és a *dolgozatunkban javasolt mérték is teljesíti* ezt a kritériumot. Az információ-áramlás metrikák és a Halstead-féle metrikák azonban nem teljesítik azt. Ez utóbbira lehetséges pl. olyan példát konstruálni, amelyre $E(P;Q) < E(P)$.

W6: (a) $(\exists P)(\exists Q)(\exists R)\ (\ \mid P \mid = \mid Q \mid\ \mid P;R \mid \neq \mid Q;R \mid\ )$

(b) $(\exists P)(\exists Q)(\exists R)\ (\ \mid P \mid = \mid Q \mid\ \mid R;P \mid \neq \mid R;Q \mid\ )$

A hatodik axióma két állítása a mérték „környezetfüggőségére" vonatkozik. Az (a) állítás szerint van két olyan egyforma komplexitású P és Q program, melyhez létezik R utótag program, mely alkalmazása esetén eltérő módon hat P ill. Q programok bonyolultságára.

A ciklometrikus számon alapuló metrikák, így a McCabe metrika, valamint azok a mértékek, melyek az egyes szoftverkomponensekhez konstans számot rendelnek, nem képesek teljesíteni ezt a feltételt. Az információáramlás metrikák, ahol a modulok környezete jelentős szerepet játszik, teljesítik ezt a két kritériumot.

A program bonyolultságát befolyásoló prefix kód tipikus példája a deklaráció. Ugyanazt a programelemet eltérő módon deklarálva eltérő bonyolultsághoz jutunk. Erre láttunk példát egy függvény érték vagy referencia szerinti paraméterátadással történő deklarációjakor (41.-42. oldal). Figyeljük meg, hogy a Fóthi–Nyéky féle *data-flowgraph* eredeti definíciója nem teljesíti ezt az axiómát.

W7: Léteznek olyan P és Q programok, melyek csak az utasítások sorrendjében különböznek, és $\mid P \mid \neq \mid Q \mid$.

Az állítás megköveteli, hogy az utasítások sorrendje befolyásolja a program bonyolultsági értékét. Nem teljesítik a feltételt azok a mértékek, melyek az utasítások vagy a programsorok számát mérik. Nem felel meg a ciklometrikus szám és általában egyetlen olyan más mérték sem, amely nem veszi figyelembe az egymást követő utasítások közötti interakciókat.

Piwowarski mutatott rá [34] a McCabe komplexitás azon hibájára, mely ugyanolyan bonyolultsági értéket rendelt n darab ciklus szekvenciájához, mint egymásba ágyazásához. Piwowarski avval érvelt, hogy a beágyazási mélység fontos szerepet játszik a bonyolultsági érték meghatározásában.

A dolgozatunkban javasolt mérték a Piwowarski által javasolt beágyazottsági mértéket használja fel a vezérlési szerkezet definiálásához. Ezért teljesíti Weyuker hetedik feltételét.

W8: Ha P program a Q program átnevezése, akkor $\mid P \mid = \mid Q \mid$.

A programok megértésében, módosíthatóságában igen nagy szerepet játszanak a jól megválasztott nevek, azonosítók. Nem tartjuk azonban korrektnek azokat a mértékeket, amelyek más értéket produkálnak a program azonosítóinak átnevezése hatására. Természetesen javasolt mértékünk indiferens a vezérlési vagy adatcsomópontok átnevezésére, így teljesíti ezt a feltételt is.

W9: (a) $(\exists P)(\exists Q)$ $(\mid P \mid + \mid Q \mid < \mid P;Q \mid)$

(b) $(\forall P)(\forall Q)$ $(\mid P \mid + \mid Q \mid \leq \mid P;Q \mid)$

A kilencedik axióma (a) esete azt feltételezi, hogy legalább néhány esetben két programtörzs konkatenációja bonyolultabb programot eredményez, mint a programtörzsek bonyolultságainak összege. Az összefűzött programrészletek

tehát hatást gyakorolnak egymásra. A ciklometrikus és méret bonyolultságok környezetfüggetlen voltuk miatt ezt az állítást nem teljesítik. Az információáramlás metrikákat viszont éppen az ilyen tulajdonságok mérésére találták ki.

Sem az információáramlás, sem a Halstead-féle metrikák nem teljesítik az 5. axiómát, így nem teljesítik a 9.b. állítást sem. A méret metrikák megfelelnek a 9.b.-nek, a ciklometrikus bonyolultság viszont nem. McCabe definíciója alapján $\mid P;Q \mid = \mid P \mid + \mid Q \mid -1$. Ugyanakkor egyértelmű, hogy ez a konstans eltérés nem érinti a ciklometrikus metrika lényegét, ezért akár a metrikát javíthatjuk egy (negatív) konstans taggal, akár a 9.b. állítást fogalmazhatnánk át:

$$(\exists P)(\exists Q)(\exists c \geq 0)\ (\ \mid P \mid + \mid Q \mid < \mid P;Q \mid\ + c\ )$$

Metrikánk nem teljesíti a 9.a. állítást, de triviálisan teljesíti a 9.b. állítást.

6.3. Összefoglalás

A Weyuker „axiómák" szigorúan véve sem nem szükséges, sem nem elégséges feltételei egy használható metrikának. Ugyanakkor az egyes állítások teljesülése komoly érv lehet a metrika alkalmazhatósága mellett. Az alábbi táblázatban összefoglaljuk metrikánk viselkedését az axiómákra és összehasonlítjuk azt néhány nevesebb metrikával. (Az utasítások száma, mint egy tipikus méret-metrika szerepel.)

#	Utasításszám	McCabe	Halstead	Adatfolyam	AV gráf
1	+	+	+	+	+
2	+	-	+	-	javítással
3	+	+	+	+	+
4	+	+	+	+	+
5	+	+	-	-	+
6a	-	-	+	+	+
6b	-	-	+	+	-
7	-	-	-	+	+
8	+	+	+	+	+
9a	-	-	+	+	-
9b	-	-	+	+	+

7. Mérési eredmények

Az előzőekben elméleti úton kíséreltük meg struktúrális szoftvermértékünk igazolását: megmutattuk, hogy nagyobbrészt teljesíti a Weyuker által szoftvermetrikák számára felállított kritériumrendszert. (A legtöbb általánosan használt mérték *nem* teljesíti az összes feltételt.) Ebben a fejezetben gyakorlati működés közben vizsgáljuk meg a javasolt mértéket.

Ebből a célból elkészült a mérték szoftveres implementációja, amely Java programozási nyelvű forrásfájlokon működik. Az implementáció maga is Java nyelven készült, így számos operációs rendszeren futtatható, többek között különböző UNIX/Linux platformokon és Windows verziókon. Az implementáció felhasználja a Princeton egyetemen készült CUP nyílt forráskódú, szabad felhasználású Java elemzőt [45].

Az implementáció paraméterezhető, így képes számolni egy adott Java program hatókör számát is (azaz az AV gráf vezérlési komplexitását, az adatok kezelésének figyelmenkívül hagyásával).

A vizsgálat céljából negyed- és ötödéves programtervező matematikus hallgatókkal azonos specifikációjú programok megírását végeztettük el.

A diákok elé kitűzött feladat az volt, hogy tervezzenek meg, és Java nyelven implementáljanak egy elemi kifejezéseket kiértékelő „számológépet". A feladat részét képezte a nyelv elemi egységeit azonosító *lexer* és a szintaxis-fát felépítő *parser* megvalósítása is. Nem volt semmiféle előzetes ajánlás a tervezés és implementálás során használatos technikákra, bár a Java nyelv nyilvánvalóan erőteljesen hatott az objektum-orientált megoldások irányában.

A következő outputot a program szolgáltatta az egyik hallgató 5 Java osztályból álló megoldására. A program külön-külön kiszámolja az egyes metódusok bonyolultságát és a végén az attribútumok számát. (A konstruktor számozása a túlterhelés miatt szükséges.)

```
file NumericNode.java
Constructor1 8
getLeft 3
getRight 3
setLeft 4
setRight 4
getData 3
getValue 3
fields 1
```

```
file OperationNode.java
Constructor1 7
getLeft 3
getRight 3
setLeft 4
setRight 4
getData 3
getValue 79
fields 3

file Tokenizer.java
Constructor1 19
getState 3
getLastToken 3
errorMessage 5
nextChar 909
end 89
fields 17

file TreeBuilder.java
Constructor1 9
buildTree 362
precedence 45
getNextToken 36
errorMessage 5
fields 2

file TreeNode.java
Constructor1 2
getLeft 3
getRight 2
setLeft 3
setRight 3
getData 2
getValue 2
errorMessage 5
inOrder 15
getTreeDepth 14
fields 1
```

Figyeljünk fel a *Tokenizer* osztály *nextChar* metódusának kiugró bonyolultsági értékére. Ott valami tervezési gondot lehet sejteni. (A jelenséget egy igen sok kiválasztásból álló *switch* utasítás okozta. Már McCabe is figyelmeztetett arra, az ilyen konstrukciók esetében a mértékeink torzíthatnak.)

Az alábbi táblázat egy összefoglaló a mérési eredményeinkről. A táblázatban a megoldások szerepelnek a programsorok száma, az eLOC, a byte-ban számolt programméret, az osztályok száma, a beágyazási mélység, és az általunk javasolt metrika szerint. (A táblázat ? bejegyzései arra utalnak, hogy ott olyan Java konstrukcióval találkozott programunk, amellyel jelen állapotában nem tudott megbírkózni.)

#	# lines	eLOC	# bytes	# class	SN(G)	AV gráf
1	1010	971	31449	13	?	?
2	368	350	8259	5	494	565
3	736	481	24008	5	126	136
4	1659	1182	44477	41(30)	?	?
5	624	604	22965	6	1579	1691

Látszik, hogy a választott implementációs startégiák alapján igen nagy szórás mutatkozik az osztályok számában. Kisebb az eltérés a eLOC mérték szerint. Figyelemreméltó, hogy kb. ugyanannyi osztály és nem drasztikusan eltérő eLOC mérték mellett kiugróan magas az 5. sz. programozó SN(G) és AV bonyolultsági értéke. Ez utalhat helytelen programozási stílusra.

Meglepő, hogy az SN(G) és AV értékek mennyire közel vannak egymáshoz. Ez talán arra utal, hogy a vezérlési szerkezet csúcspontjainak száma „elnyomja" az adathivatkozások számát. Lehetséges, hogy ezen az adathivatkozások konstans súlyozásával lehetne módosítani.

Az eredmények további elemzést követelnek.

8. Összefoglalás

Az AV gráfon definiált mértékkel programok struktúrális bonyolultságát mérjük paradigmafüggetlen módon. Az elemi programozási fogalmakra építve a mérték egyaránt méri a programok vezérlési szerkezetének, adatstruktúrájának, és az adatok kezelésének tulajdonságait.

A programok vezérlőszerkezetét a Piwowarski által bevezett beágyazási mélység segítségével definiáltuk, J. Howatt és A. Baker szigorú matematikai apparátusát követve. Ezt a definíciót értelmeztük az alprogramok használatára is. Az adatok kezelésének módjánál Fóthi Ákos és Nyékyné Gaizler Judit *data-flowgraph* elvét módosítottuk oly módon, hogy megkülönböztettük az adatkezelés irányát. Megmutattuk, hogy így olyan bonyolultsági különbségeket is ki lehet mutatni, ami egyébként rejtve maradna.

Demonstráltuk, hogy ez az új metrika megfelelően működik a hagyományos, procedurális programok esetében. A mérték jól tükrözi a programok dekompozíciójánál gyakorlatban tapasztalt komplexitás-csökkenést is. A mérték természetes módon kiterjeszthető az adattípusok bonyolultságára. Definiáltuk az osztály bonyolultságát.

A javasolt mérték jól viselkedik speciális „objektumok" esetében is. Az AV gráf definíciónk megengedi, hogy egy osztály vezérlési szerkezete teljesen üres legyen. Ebben az esetben „klasszikus" adatszerkezetet kapunk. Az ilyen típus bonyolultsága az adattagok bonyolultságának összege; ha az adattagok a konkrét programozási nyelvünk elemi típusai, akkor a tagok száma. Az ilyen adatstruktúrák is csökkenthetik a teljes program bonyolultságát, de csak abban az esetben, ha az alkalmazott nyelv képes elemi műveleteket végezni rajtuk.

Ellentétes eset is elképzelhető: amennyiben egy „osztály" olyan metódusokból áll, amelyek egyáltalán nem osztoznak közös adatokon, azaz osztályunknak nincsenek attributumai, az osztály bonyolultsága az egyes metódusok külön-külön számolt bonyolultságának összege lesz. Vegyük észre, hogy ez a klasszikus eljárásgyűjtemény, függvénykönyvtár esete.

A fenti példák azt igazolják, hogy metrikánk *nem paradigmafüggő*: egyaránt képes mérni a klasszikus procedurális elven és az objektum-orientált módon készített programokat. Ez volt kitűzött célunk, mely segítségével lehetővé válik a multiparadigmás programok egységes elvek alapján történő bonyolultsági mérése.

Az osztály bonyolultságára bevezetett mértékünk nem különbözteti meg az eltérő láthatósági szintekkel (private, protected) deklarált attribútumokat. A „klasszikus" enkapszuláció teljes figyelmen kívül hagyása meglepő lehet az objektum-orientált programok használóinak. Megmutattuk, hogy a tagok láthatósági szintje egyáltalában nem befolyásolja az adott osztály bonyolultságát. Két osztály, amely csak

a tagok láthatóságában különbözik egymástól egyenlő bonyolultságú. Az osztály láthatósági szabályok alklmazása a kliens kód bonyolultságát befolyásolja.

Egy C osztály adattagját egyetlen adatcsomópontnak tekintjük, függetlenül az általa reprezentált típus belső bonyolultságától. Amennyiben ez az adattag egy összetett T típus, esetleg osztály saját attribútumokkal és tagfüggvényekkel, akkor a C és T típusokat ábrázoló AV gráfok uniója reprezentálja a teljes osztályrelációt. Amennyiben T a felhasználó által implementált osztály, a *teljes* programrendszer bonyolultsága, amely tartalmazza a szabványos osztálykönyvtárakat, eljárásgyűjteményeket és a felhasználók által korábban megvalósított osztályokat, megnövekszik T bonyolultságával. Konstans bonyolultságnövelő tevékenységünk a teljes rendszer bonyolultságát az osztály használatával lineáris mértékben csökkenti.

Az osztályok közötti öröklést is ábrázolhatjuk az AV gráfok kapcsolatával. Ez a jelenség annak az objektum-orientált programozási tapasztalatnak felel meg, hogy a jól kialakított osztályhierarchiában erős vezérlési- és adatkapcsolatok vannak egy osztályon belül, és gyenge kapcsolatok léteznek az osztályok között.

Megvizsgáltuk, hogy mértékünk mennyiben elégíti ki a Weyuker–féle logikai állításokat. Elkészült a mérték szoftveres implementációja, mely Java programnyelvi források elemzését végzi.

További kutatási irányként két főbb vonalat tudunk kijelölni. Az egyik a speciális vezérlőszerkezetek, mint pl. a *kivételkezelés* vizsgálata. A kivételkezelés egyre szélesebbkörűen használatos a mai programozási nyelvekben [52], mint nemstruktúrált alternatív vezérlési konstrukció, de szinte teljesen hiányzik a programok bonyolultságára gyakrolt hatásainak elemzése.

Másik rendkívül érdekes kutatási irány a *generatív programozás*, mint új programozási paradigma bonyolultságának vizsgálata [51]. A generatív programozás segítségével (pl. a Turing-teljes C++ template mechanizmus használatával) fordítási időben végezhetünk el bonyolult algoritmusokat [50]. Ezáltal a futási időben végrehajtott kód bonyolultsága csökkenhet, de ennek árán egyre jobban előtérbe kerül a template *metaprogramok* bonyolultsága.

Hivatkozások

[1] Balla, K. *The Complex Quality World*,
Eindhoven University Press, Eindhoven (2001).

[2] Boehm, B. W. *Software and its impact: A quantitative assessment*, Datamation, vol.19, pp.48-59 (May 1973).

[3] Cammack, W. B., Rogers, H. J. *Improving the Programming Process*, IBM Tech. Rep., TR 00.2483 (Oct 1973).

[4] Cherniavsky J.C., Smith C.H. *On Weyuker's Axioms For Software Complexity Measures*, IEEE Trans. Software Engineering, vol.17, pp.1357-1365 (1991).

[5] Chidamber S.R., Kemerer, C.F. *A metrics suit for object oriented design*,
IEEE Trans. Software Engeneering, vol.20, pp.476-498, (1994).

[6] Coplien J.O. *Multi-Paradigm Design for C++*, Addison-Wesley, (1998).

[7] Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., Love, T. *Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics*, IEEE Trans. Software Engineering, vol.5 no.2, pp.96-107, (1979).

[8] Czarnecki K., Eisenecker U.W. *Generative Programming*, Addison-Wesley, (2000).

[9] Dahl, O., Myhrhaug, B., Nygaard K. *A Simula 67 Nyelv Definíciója*, BME Vegyészmérnökkar, (1985).

[10] Dunsmore, H. E., Gannon, J. D. *An Empirical Investigation*, Computer, 12(12), pp.50-59 (1979).

[11] Daróczy Z., Varga L. *A new approach to defining software complexity measures*, Acta Cybernetica, Tom. 8. pp.287-291. (1988).

[12] Davis J.S., LeBlanc R.J. *A Study of the Applicability of Complexity Measures*, IEEE Trans. Software Engineering, vol.14, pp.1366-1372 (1988).

[13] Dijkstra,E.W. *A Discipline of Programming*, Prentice-Hall, Engelewood Cliffs, N.Y., (1976).

[14] Fenton, N.E., Neil, M. *Software Metrics: Roadmap*, The Future of Software Engineering. ACM Press, New York, (2000).

[15] Fóthi Á., Nyéky-Gaizler J. *On the Complexity of Object-Oriented Programs* Proc. of the 3rd Symp. on Programming Languages and Software Tools Kaariku, Estonia, (1993).

[16] Grassberger, P. *Estimating the information content of symbol sequences and efficient codes* IEEE Transactions on Information Technology. Vol.35,669. (1989).

[17] Halstead, M. H. *Natural laws controlling algorithm structure*, SIGPLAN Notices, vol.7. pp.19-26 (1972).

[18] Harrison,W.A. and Magel,K.I. *A Complexity Measure Based on Nesting Level*, ACM Sigplan Notices,16(3), pp.63-74 (1981).

[19] Harrison,W.A. and Magel,K.I. *A Topological Analysis of the Complexity of Computer Programs with Less Than Three Binary Branches*, ACM Sigplan Notices,16(4), pp.51-63 (1981).

[20] Henderson-Sellers, B., *Object-oriented metrics: measures of complexity*, Prentice-Hall, pp.142-147, (1996).

[21] Henry S., Kafura D. *Software Sructure Metrics Based of Information Flow*, IEEE Trans. Software Engineering, vol.7, pp.510-518 (1981).

[22] Howatt,J.W. and Baker,A.L. *Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting*, The Journal of Systems and Sofware 10, pp.139-150 (1989).

[23] Humphrey, W.S. *Managing the Software Process*, Addison-Wesley, Reading, Massachusetts (1989).

[24] Iannino, A., Musa, J. D., Okumoto, K., Littlewood, B. *Criteria for software reliability model comparisons*, IEEE Trans. Software Engineering, vol.10, pp.687-691, (1984).

[25] Kiczales G. *Apect Oriented Programming*, AOP Computing surveys 28(es), 154-p (1996)

[26] Knuth, D. E. *Structured programming with GOTO statements*, Computing Surveys, vol.6, pp.261-301 (Dec 1974).

[27] Kokol, P., Podgorelec, V., Zorman, M. *Universality - A Need For A New Software Metric* International Workshop on Software Measurement (IWSM'99). Lac Supérieur, Canada, (1999).

[28] Lakshmanan K.B., Jayaprakash S., Sinha P.K. *Properties of Control-Flow Complexity Measures*, IEEE Trans. Software Engineering, vol.17, pp.1289-1295 (1991).

[29] Legard, H., Marcotty, M. *A generalogy of control structures*, Commun. Assoc. Comput. Mach., vol.18, pp.629-639 (Dec 1974).

[30] McCabe, T.J. *A Complexity Measure*, IEEE Trans. Software Engineering, SE-2(4), pp.308-320 (1976).

[31] Meyer,B. *Object-Oriented Softwar Construction*, Prentice Hall, New York, (1988).

[32] Morris, K.L. *Metrics for Object-Oriented Software Development Environments*, Master's Thesis, M. I. T. Sloan School of Management, (1989).

[33] Mills, H. D. *Mathematical foundations for structured programming*, Federal System Division, IBM Corp., Gaithesburg, MD, FSC 72-6012, (1972).

[34] Piwowarski,P. *A Nesting Level Complexity Measure*, ACM Sigplan Notices, 17(9), pp.44-50 (1982).

[35] Prather, R. E. *An axiomatic theory of software complexity*, Comput. J., vol. 27. no. 4. pp.340-346, (1984).

[36] Schenkel, A., Zhang, J., Zhang, Y. *Long range correlations in human writings* Fractals, Vol.1, no.1, pp.16-19. (1993).

[37] Scott, M. L. *Programming Language Pragmatics*, Morgan Kaufmann Publishers, (2000).

[38] Shannon, C.E. *Prediction and entropy of printed English* Bell System Technical Journal. Vol.30,50. (1951).

[39] Stroustrup B. *The C++ Programming Language*, Addison-Wesley, (1997).

[40] Tian J., Zelkowitz M.V. *Complexity Measure Evaluation and Selection*, IEEE Trans. Software Engineering, vol.21, 641-650 (1995).

[41] Tucker, A., Noonan R. *Programming Languages, Principles and Paradigms, 2.ed.*, Mc Graw Hill, (2002).

[42] Varga, L. *A new approach to defining software design complexity*, In: R.Mittermeier (ed.): Shifting Paradigms in Software Engineering. Springer Verlag, Wien, New York, pp.198-204.(1992)

[43] Weyuker, E.J. *Evaluating software complexity measures*, IEEE Trans. Software Engineering, vol.14, pp.1357-1365 (1988).

[44] Wirth, N. *Algorithms + Data Structures = Programs*, Prentice Hall, (1976).

[45] CUP Parser Generator for Java,
`http://www.cs.princeton.edu/~appel/modern/java/CUP`

[46] Eclipse.org formation,
`http://www.eclipse.org/org/index.html`

[47] Method Object,
`http://www.c2.com/cgi/wiki?MethodObject`

[48] Fóthi Á., Nyéky–Gaizler J., Porkoláb Z. *The Structured Complexity of Object-Oriented Programs*, Computers and Mathematics with Applications accepted for publication (2002).

[49] Fóthi, Á., Nyéky-Gaizler, J., Porkoláb, Z. *On the Complexity of Class*, Proc. of the FUSST'99, Tallin, Estonia, pp.221-231 (1999).

[50] Porkoláb Z., Kisteleki R. *Alternative Generic Libraries*, ICAI'99 4th International Conference on Applied Informatics, Eger-Noszvaj, pp.79-86. (1999).

[51] Frohner Á., Kozma L., Kozsik T., Porkoláb Z. *Beyond 2000, Beyond Object-Orientation*, ICAI'01 5th International Conference on Applied Informatics, Eger-Noszvaj, to appear. (2001).

[52] Csontos P., Porkoláb Z. *On the Complexity of Exception Handling*, ECOOP 2001 Ph.D. Workshop presentation, Budapest, (2001).