Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Az AV-metrika kiterjesztése kivételkezelésre

Témavezető:

Porkoláb Zoltán
egyetemi docens

Szerző:

Nagy Gergely Attila
Programtervező matematikus
nappali tagozat

Budapest, 2012.

Tartalomjegyzék

1. fejezet

Bevezetés

Ebben a dolgozatban a modern programozási nyelvekre jellemző kivételkezelési technikák a programok komplexitására gyakorolt hatását vizsgáljuk. Ehhez felhasználjuk a tudományos életben és az iparban egyaránt elterjedt metrikákat, melyek általában nincsenek kivételkezelésre definiálva: az A-V-t [6], a McCabe-et [5], valamint a Lines Of Code-ot.

A kivételek céljukat tekintve a hibakezelés, a kivételes eseményekre való reagálás egyszerűsítése, a programkód átláthatóbbá, érthetőbbé tétele. Megjelenésük a programozási nyelvekben nem újkeletű: az 1960-as években megjelent PL/I már definiálta őket, azonban az egy évtizeddel későbbi CLU fejlesztette őket tovább jelentősen [4]. Mivel ezek mellett olyan más programozási nyelvek váltak széles körben igazán népszerűvé, amelyek nem tartalmaztak kivételkezelést, ezért kialakultak praktikák az ilyen események kezelésére. A legegyszerűbbek közé tartozik a hibák jelzésére használt függvényekből való visszatérési érték, a kissé jobban átgondolt, statikus hibakezelési „tárban” használt `errno`-n keresztül, a már-már modern hibakezelésre jellemző vezérlésátadási technikát szimuláló `setjmp`/`longjmp`-ig.

Mivel a programok bonyolultságának számszerű mérése inkább az ipari, mintsem a tudományos felhasználás számára fontos, valamint ebben a közegben főként az utóbbi csoportba tartozó nyelvek terjedtek el, ezért a komplexitás mérésére használható metrikák általában nem foglalkoznak a kivételekkel. A legnépszerűbb nyelvek közé tartozó C++ volt az, amelyik széles

körben elterjedtté tette a kivételkezelést. Ennek hatására a C++-szal foglalkozó szakemberek voltak az elsők, akik érdeklődni kezdtek a kivételkezelés komplexitásra gyakorolt hatásával.

A Guru of the week című, a C++-hoz kapcsolódó programozási feladatokkal foglalkozó cikkgyűjtemény egyik bejegyzése egy érdekes problémát vet fel: egy csupán négysoros programkódban kell megszámolni a lehetséges végrehajtási utakat [7]. Három kategóriát állítanak fel: aki maximum hármat talál meg, az átlagos hozzáértésű programozó, aki maximum tizennégyet, az kivételkezelés-tudatos, aki mind a huszonhármat, az kivételes tudással rendelkezik. Ez jól jellemzi, hogy jelenleg megbízható sejtések nélkül állunk a kivételkezelés és a kód komplexitásának kapcsolatához. További kérdést vet fel, hogy ha vajon egy mindössze négysoros programban háromról huszonháromra nőtt a lehetséges végrehajtások száma, akkor a kivételkezelés vajon valóban egyszerűsíti a kódot, vagy netán bonyolultabbá teszi? Dolgozatomban megpróbálok választ találni ezekre a kérdésekre.

1.1. A kivételkezelés vizsgálata metrikákkal

A dolgozatot a kivételkezelést eddig figyelembe nem vevő metrikák kibővítésével kezdjük. Megpróbálunk olyan adekvát számítási módszert találni, ami illeszkedik az adott metrika többi részébe, viszont megfelelően kiemeli a vizsgálódáshoz a kivételeket.

1.2. Az eredmények értékelése

Az új definíciók validálására kétféle technikát alkalmazunk. Először összehasonlítjuk a korábban elterjedt hibakezelési technikákat a kivételekkel példaprogramokon keresztül, majd egy nagyobb, élő kódbázison számítjuk ki a kód komplexitását először a kivételeket figyelembe véve, majd azokat mellőzve. Ezzel a módszerrel választ kaphatunk, hogy szintetikus kódrészleteken túl is, az iparban alkalmazott módszerekkel írt kódon is megállják-e a helyüket feltevéseink.

1.3. A dolgozat felépítése

Az elmúlt évtizedekben számos eredmény jelent meg programkód komplexitásának mérésére. Mivel ezek megfelelő módon mérik a bonyolultságot azokban az esetekben, amelyekben definiálva vannak, ezért ezeket felhasználva, kibővítve végezzük vizsgálódásainkat.

A dolgozatot a metrikákat bemutató fejezettel kezdem. Ebben megpróbálok egy általános képet adni róluk, rávilágítani hasznosságukra, közös jellemzőikre.

A kivételkezelés című fejezetben a kivételkezelés bevezetése mögötti gondolatokat, magyarázatokat ismertetem, bemutatok néhány általános módszert használatukra, néhány népszerű nyelv eltéréseire is rávilágítva.

A metrikák jelenlegi definíciója nem alkalmas a kivételek hatásainak megfelelő vizsgálatára. A Metrikák ajánlott kiterjesztése című részben megpróbálok ajánlást tenni ennek a kiküszöbölésére, magyarázatot adva ennek szükségességére.

A példák, eredmények című fejezet foglalja össze, hogy az előzőekben definiált kiterjesztés hogyan viszonyul korábbi elvárásainkhoz, megfelel-e annak, amit empirikus módon vártunk a kivételek hatásától, netán ellenkezik-e velük.

Az eredmények alátámasztását mindenképpen hasznos elvégezni az elméletet implementáló, azt felhasználó valós példákkal. Ennek elvégzésére készítettem egy könyvtárat, amit fel lehet használni különböző metrikák alkalmazására tetszőleges Java programok bonyolultságának mérésére. Ennek bemutatása történik A metrikák implementálása című fejezetben.

A dolgozatot az eredményeket összefoglaló fejezet zárja, mely megpróbál hasznos következtetést levonni, rávilágítani a jövőbeni feladatokra.

2. fejezet

Metrikák bemutatása

Ebben a fejezetben először általánosan foglalkozunk a szoftvermetrikákkal, mint tetszőleges programkód komplexitásának mérőszámaival. A bevezető után rátérünk a dolgozatban felhasznált három metrika rövid bemutatására.

2.1. Általános bevezető

A szoftvermetrikák egy program vagy programrészlet tulajdonságait valamilyen abszolút mérőszámmal jellemző mértékek. Alkalmazásukkal jobban kifejezhetőek egy szoftvertermék tulajdonságai, jobban tervezhetővé válik a termék életciklusa, az elkészítéséhez, karbantartásához szükséges idő és erőforrások, továbbá egzakt alapot adnak minőségbiztosítási, tesztelési kérdésekhez és a projekt irányításához.

Első megjelenésük egybeesik azokkal a törekvésekkel, melyek a szoftvertermékek előállítását mérnöki módon, mérnöki módszerekkel sürgették. Az 1970-es évek közepéig a programozók jellemzően nem használtak semmilyen módszertant programjaik előállításakor, mely karbantarthatatlan, bonyolult kódhoz vezetett. 1976-ban Thomas J. McCabe cikkében definiálta a Ciklomatikus Komplexitást [5], és ezzel ajánlást próbált tenni Fortran programok moduljainak hosszára, bonyolultságára. A McCabe-metrika mellett természetes úton jelentek meg sokkal egyszerűbb és naivabb metrikák is, mint például a sorok száma (Lines of Code - LOC), vagy a függvények (procedúrák)

számával jellemzett metrika.

A későbbiekben nyilvánvalóvá vált, hogy nem ismerünk olyan metrikát, jellemzési módszert, mely akár csak egy szoftver minden tulajdonságát megfelelő, jól összehasonlítható módon reprezentálja. Ez vezetett a tudományos és ipari körökben is olyan meglátásokhoz, hogy túl egyszerű metrikák alkalmazása inkább árt, mint használ egy szoftver előállításakor [1]. A nem ennyire radikális nezőpontok legalábbis megkérdőjelezik, hogy egy adott metrika értéke egy adott szoftveren kiszámolva pontosan mit is jelent, bármilyen változása a szoftver életciklusában mennyire használható információt biztosít. Az általános vélekedés szerint azonban „nem befolyásolhatunk valamit, amit nem tudunk mérni" [2]. Ennek következtében ma is folynak kutatások a területen.

Szoftvermetrikák definiálásakor mindenképpen elérendő cél a matematikai minőségű pontosság. Nem válik jól használhatóvá egy olyan jellemzés, amely a kód minimális módosításakor is túl sokat perturbálódik. Ennek ellenére hamar kiderült, hogy ha a szoftvermetrikákat megpróbáljuk beilleszteni a matematikai metrika-fogalomba (nemnegativitás, egyenlőség, szimmetria, háromszög-egyenlőtlenség), akkor a kódot nem intuitívan jellemző mértékeket kapunk. Ebből kifolyólag a különböző metrikák készítői saját példakódokkal, esetekkel próbálták bizonyítani metrikájuk helyességét, hasznosságát, mely nyilvánvalóan nem elég egzakt. Ennek kiküszöbölésére alkotta meg Weyuker 1988-ban kilenc axiómáját [8], melyek matematikai igényességgel jellemzik a metrikákat, ahelyett, hogy a metrikát definiálók által készített példákkal próbálnánk bizonyítani annak hasznosságát. Később ugyan alkottak olyan metrikát, amely megfelelt Weyuker összes feltételének, mégsem nyújtott semmilyen hasznos információt [3]; Weyuker úgy érvelt, hogy ennek oka, hogy a legtöbb esetben nem tudjuk pontosan, mit várunk a metrikától, hogy milyen módon jellemzi a kódot [6].

2.2. Egy triviális metrika: LOC

A kód komplexitását a hosszával jellemezni természetes módszer. Minél hosszabb, annál több konstrukciót kell megérteni, annál bonyolultabb. Azon-

ban ez az egyszerű megközelítés rengeteg kérdést vet fel. Amíg a magasabb szintű nyelvek sem határolódtak el jelentősen az alacsonyabb szintűektől, addig a sorok száma valóban nem rossz indikátora volt a bonyolultságnak. Például a RISC processzorok által támogatott kevés utasítás lehetséges kombinációi egyértelműen a kód növekedésével járnak. Ennek ellenére itt is hozható olyan eset, melynek bonyolultságát nem a hossz adja: például több egymásba ágyazott feltételes ugró utasítás, vagy ciklusok és elágazások keverése, ágyazása.

A magasabb szintű nyelvek fejlődésével azonban egyre nőtt a kód hosszát nem feltétlenül növelő absztrakció mértéke. Ennek köszönhetően ma már nem feltétlenül tudunk egyenes arányt vonni a kód hossza és bonyolultsága között. Ezt jól jellemzik azok a nyelvek, amelyeknek kifejező ereje jelentősen túlmutat a régebbi programozási nyelvekéin. Vegyük például a 2.1. ábrán látható, Scala-ban írt kódrészletet. Annak ellenére, hogy csupán három

```
1 trait C[->[_, _], ->>[_, _], F[_]] {
2   def fmap[A, B](f: A -> B): F[A] ->> F[B]
3 }
```

2.1. ábra. Komplex Scala kód

sorból áll, valószínűleg nagyjából bármely programozó kiegyezne azzal, hogy nem egyszerű megérteni (ebben az esetben ez nyilván felvet a stílusával kapcsolatban is kérdéseket). Ehhez hasonló megfontolások közé tartozik, hogy a LOC (implementációtól függően) számolhatja a kódban szereplő kommenteket is, az olvashatóságot növelő újsorokkal, tabulált nyitó, záró elemekkel (pl. C-stílusú nyelvekben `{`, `}`) együtt, amelyek pont azért kerülnek így a program szövegébe, hogy a megértést, átláthatóságot könnyítsék.

2.3. Ciklomatikus komplexitás

A ciklomatikus komplexitás (szokás még feltételes komplexitásnak, illetve McCabe-metrikának is nevezni) egy Thomas McCabe által 1976-ban definiált metrika [5]. Elsődleges célja a szoftvert a benne szereplő, különböző

végrehajtási utak számával jellemezni. McCabe eredetileg Fortran programokat vizsgált, ezeknek optimális modularizációjához szeretett volna ajánlásokat tenni, amelyeket valamilyen matematikai háttérrel rendelkező elmélettel kívánt bizonyítani. Véleménye szerint egy szoftver elkészítéséhez használt erőforrások akár felét a tesztelésnél használják fel, míg legnagyobb részét az életciklus leghosszabb részét felölelő karbantartásnál; így kritikus, hogy az adott szoftver mennyire jól tesztelhető, karbantartható.

Metrikájának alapja a programot jellemző végrehajtási utak gráfja. Ennek a gráfnak a jellemzéséhez a lehetséges elágazásokat meghatározó predikátumok számát használja. Meggondolása szerint ugyanis ez befolyásolja, hogy az adott függvényt mennyire teljesen, illetve mekkora költséggel lehet tesztelni. Az akkori vélekedéssel ellentétben nem az adott modul ajánlott hosszát kívánta korlátozni, hanem a ciklomatikus komplexitásáét: attól a kivételes esettől eltekintve, amikor az adott modul sokágú értékkiválasztást tartalmaz, a modul bonyolultsága ne haladja meg a 10-et.

A metrika kiszámításához felhasználja a gráf ciklomatikus számát, melyet a következőképpen definiálhatunk: a $G = (v, e)$ gráf $V(G)$ ciklomatikus száma $V(G) = e - v + 2p$, ahol p a gráf komponenseinek számát jelöli. Egy tétellel bizonyítható, hogy ez megegyezik az erősen összefüggő gráfban található lineárisan független körök számával [6]. Ennek alkalmazásához a program vezérlési gráfját az alábbiaknak megfelelően definiálja: a gráf egy olyan irányított gráf, amelynek pontjai szekvenciális blokkok (azaz nem tartalmaznak ugró utasítást: ciklust, elágazást vagy tényleges ugró utasítást), élei pedig ezek az ugró, vezérlést átadó utasítások.

A fentiekből jól látható a McCabe-metrika több hátránya is: nem veszi figyelembe a predikátumok egymásba ágyazottságát –ami pedig nyilvánvalóan tovább növeli a komplexitást–, illetve nem törődik a programban felhasznált adatok szerepével. Utóbbi nagy jelentőséggel bír az objektumorientált programozás megjelenése és elterjedése óta.

2.4. Az A-V metrika

A Porkoláb Zoltán által definiált A-V metrika A McCabe-i úton halad tovább, azonban megpróbál annak hiányosságaira választ adni, miközben a metrika paradigmafüggetlenségét garantálja [6]. Ennek jelentős szerepe van a ma népszerű kevert- vagy multiparadigmás nyelvek körében: egy jellemzési módszer sem kielégítő, ha a program meghatározott részeire működik csupán.

Az A-V első javító megoldása a beágyazottság figyelembe vétele. A korábban definiált vezérlési gráfot megtartva definiálja a predikátumok beágyazottsági számát (nesting deepness, ND). A függvény belépési pontja után ez a szám minden egyes predikátum hatáskörében eggyel nő. A 2.2. ábrán

```
1  void function()
2  {
3    int x;
4    if( /*predikátum1*/ )
5    {
6      while( /*predikátum2*/ )
7      {
8        if( /*predikátum3*/ )
9        {
10         x += 2;
11       }
12       else
13       {
14         x = 2;
15       }
16     }
17   }
18   else
19   {
20     x = 3;
21   }
22 }
```

2.2. ábra. Predikátumok egymásba ágyazottsága

látható kódban például az `x` változó deklarálásakor a beágyazottsági mérték

1, ahol kettővel növeljük, illetve amikor kettőt adunk értékül neki 4, valamint amikor hárommal definiáljuk, 2. Ez a megközelítés direkt módon jellemzi, hogy az adott kód megértése mennyi erőfeszítést igényel: minél több lehetséges elágazásra, ciklusra ható feltételt kell megértenünk, annál több odafigyelésre, annál több változó fejben tartására, megértésére van szükség.

Az adatok elérési módjának meghatározása a ma használt legtöbb programban központi kérdés. Az objektumorientált programozási módszertan is ennek az elvnek a mentén alakult ki. Állapottal rendelkező entitásokat vezet be, melyeken az objektum leírásakor definiált függvények végeznek műveleteket. Ennek a megfontolásnak középpontba kerülése miatt vált szükségessé, hogy a kód bonyolultságának kifejezésekor is figyelembe vegyük. Az A-V metrika ezt a következőképpen teszi: kibővíti a vezérlési gráfot az adatelérési csomópontokkal (írás illetve olvasás), melyeknek súlyát az adott pont beágyazottsági mértéke adja meg. Így például a 2.2. ábrán szereplő `x` változó rendre 1, 4, 4, 2 súllyal szerepel.

A fenti két megfontolás eredményeképpen alakul ki egy függvény A-V-bonyolultsága: a beágyazottsági mélység és az összes változó írásának illetve olvasásának összege adja meg. Porkoláb ajánlást tesz még az osztály komplexitására: ezt az osztályban definiált mezők, illetve a metódusok bonyolultságának összegével írja le.

A fentiekből jól látható, hogy az A-V természetes válasz a McCabe által definiált metrika hiányosságaira. Alkalmazásával a mai programok jellemzőit jobban figyelembe vevő jellemzést kaphatunk.

3. fejezet

A kivételkezelés

A következő részben összefoglaljuk a kivételkezelés bevezetése illetve széleskörű elterjedése mögötti gondolatokat, magyarázatokat. Bemutatjuk a kivételkezelés előtti hibakezelési technikákat, majd rávilágítunk a kivételkezelés definiálásának szükségességére, végül röviden kitérünk a kivételbiztos (exception-safe) kódolásra, valamint az exception-ök hiányosságaira is.

3.1. Hibakezelés kivételkezelés nélkül

A kód futása közben előfordult váratlan esetek jelzésére az első programoktól fogva szükség volt, főleg azoknál, amelyeket nem csak az írójuk használt. Például egy szubrutin, ami valamilyen módszer szerint összehasonlít két stringet, esetleg nem működik, ha az egyik üres. Előfordulhatnak olyan esetek is, amikor előre nem látható módon, az algoritmus egyik részeredményeként derül fény arra, hogy a kapott bemenetek nem megfelelőek. Ezek az esetek az algoritmusok bonyolódásával párhuzamosan egyre gyakoribbak lehetnek. Szükség van tehát egy olyan megoldásra, ami a hívó fél számára jelzi, ha a futás közben valamilyen hibát tapasztaltunk.

Természetes megoldás - ha az adott programnyelv támogatja -, hogy a függvény visszatérési értékét használjuk a hibajelzésre. Ennek tipikusan két változata adott: az eredmények lehetséges halmazának egy extremális elemével jelezni, ha hiba történt, vagy a visszatérési értéket pusztán hibajelzésre

használni, és az eredményt valamilyen bemenő paraméteren keresztül visszajuttatni a hívóhoz. A második az általánosabb megoldás, ugyanis működik a visszatérési érték típusától függetlenül, így gyakorlatilag egy „hibatípust" képezve. Az elsőre találhatunk egy egyszerű példát a 3.1. ábrán. A hívó

```
int gcd(int a, int b)
{
  // Csak 0-kra nem működik az algoritmusunk
  if (a == 0 && b == 0)
  {
    return -1;
  }
    while(true)
    {
        if (a == 0) return b;
        b %= a;
        if (b == 0) return a;
        a %= b;
    }
}
```

3.1. ábra. Legnagyobb közös osztó keresése

```
int res = gcd(userInput1, userInput2);
if (res == -1)
{
  printf("Hibás bemenet")
}
else
{
  printf("Az eredmény:%d", res)
}
```

3.2. ábra. Hibajelző függvény használata

fél ilyenkor ellenőrizheti a függvény visszatérési értékét, hogy történt-e hiba. Ez jellemzően a 3.2. ábrán látható módon történik. Jól látható, hogy a módszer használata jelentősen növeli a kliens programban használt predikátumok számát, amely a kód bonyolultságának növekedéséhez vezet.

A fenti módszer egy kifinomultabb változata többek között a GNU libc implementációjában található `errno` alkalmazása. Ennek lényege, hogy a függvények hibás esetben egy globális változót állítanak be, amiben jelezhetik a hiba típusát is. Ennek a módszernek az előnye, hogy egyetlen helyre fogja össze a hibakezelést. Ugyan a predikátumok számát nem csökkenti, azonban egy gyakorlott programozó számára ezek könnyebben érthetővé válhatnak.

Az eddigi módszerek minden esetben összekeverték a hibakezelést a kód „természetes" futásával, ami jóval nehezebben értelmezhetővé tette azt. A harmadik módszer ezen próbál segíteni: a távoli ugrásokat biztosító `setjmp`, `longjmp` hívások, amelyek ugyan egy `goto`-hoz hasonlítanak a legjobban, mégis elősegítik a felelősség szétválasztását: a hibakezelő részeket megírhatjuk a normális futástól szeparálva, így egyszerűsítve a kódot. Ennek a módszernek nagy hátránya, hogy a klienskód felé nehezen alkalmazható: nem várhatjuk el, hogy a kliens a saját kódjában helyezzen el címkéket, amelyek a hibakezelést végzik.

Az ismertetett technikák jól láthatóan messze vannak az optimálistól, így szükségessé vált egy általános, jól átgondolt módszer, ami megoldja ezek problémáit.

3.2. A kivételkezelés

A kivételkezelés (exception handling) módszere a modern programozási nyelvek válasza a korábbi hibakezelési technikák hiányosságaira. Ebben biztosítottak a felelősség szétválasztásán túl egy olyan megoldást, amely nem befolyásolja a függvények bemenetét, nem használja fel a visszatérési értékeiket, tehát lényegében a kód eredeti céljának megváltoztatása nélkül kezelhetjük a hibás eseteket.

Ez egy olyan általános módszert jelent, amely a kódban előforduló hibás, vagy kivételes esetek kezelésére szolgál vagy az adott kódban, vagy a hívó félnél. Három fő meggondolást tartalmaznak:

- a hibát kezelő és a normális futást végző kód szétválasztása
- a hibák jelzésére külön típusok bevezetése

- a lekezeletlen hibák továbbgyűrűzése.

A fenti megoldások olyan kifinomult hibakezelő kódot eredményeznek, melyet a korábbi módszerekkel nagyon nehéz, esetleg lehetetlen elérni.

A hibát kezelő és a normális futást végző kód szétválasztása lényegében lehetővé teszi, hogy a használó kódot megírva, ettől függetlenül kezeljük a hibákat. Nincs többé szükség elágazásokkal teletűzdelni a kliens kódot, elég a potencionálisan hibás hívásokat körbevenni a kivételkezelő kóddal (a 3.3. ábrán a C++-stílusú `try` - `catch`-et szemléltettem).

```
1  try
2  {
3    MyClass c = possiblyFaultyFunction();
4    // Folytatódhat a hiba nélküli kód
5    // ...
6  }
7  catch (MyException e)
8  {
9    std::cout << "Az általam definiált hiba történt: "
10     << e.MyMessage << std::endl;
11 }
12 catch(...)
13 {
14   std::cout << "Egyéb hiba történt." << std::endl;
15 }
```

3.3. ábra. C++-stílusú `try` - `catch` használata

A hibák típusai jellemzően semmilyen kapcsolatban nincsenek a függvény által használt egyéb típusokkal. Ezt az `errno` készítésekor már figyelembe vették, azonban egy egész számmal nehéz reprezentálni a hibák különböző fajtáit. Egy állandó referenciát fenntartani a hiba értéke és jelentése között költséges, és jellemzően a hibakódok hibás értelmezéséhez vezet. Gyakran előfordul például, hogy a kód módosítása után már a kommenteket nem javítják, amik zavaróvá válnak - hasonló történhet egy táblázattal is, ami tartalmazza a hiba kódját és jelentését. Ennek elkerülésére találták ki a kivételek típusait. Ez a megfontolás vezetett oda, hogy - ha a legtöbb nyelvvel

ellentétben nem is teljesen általános típusú objektumot, de - tetszőleges bonyolultságú típust használhatunk hibák jelzésére. Ezzel a módszerrel egy egész hibatípus fát építhetünk fel, az összetartozó hibákat egymásból származtatva, kibővítve, illetve tetszőleges, a hibát jól jellemző értéket eltárolva bennük. A legtöbb modern nyelv (pl. Java, C#, Scala) ezt elő is írja: egy előre definiált típusból (jellemzően a nyelvhez köthető standard könyvtárban definiált `Exception` osztály) kell származtatnunk a saját kivételeinket is. A többrétű típusosság használata látható a 3.3. ábrán is: az első `catch`-blokk csak a `MyExcpetion` típusú kivételeket kapja el, a többit az utolsó `catch` ág feldata feldolgozni. A legtöbb nyelvben pontos szabálya van a a különböző típusok elkapásának: fentről lefelé haladva az első illeszkedő `catch` fogja el. Ezt a fordító vagy ellenőrzi (Java), vagy nem (C++). Ezen a módszeren

```
1 catch {
2       case ioe: IOException => ...
3       case e: Exception => ...
4 }
```

3.4. ábra. Mintaillesztés, mint kivételek elkapásának módszere

finomít tovább a Scala a 3.4. ábrán látható módon, mely a sokkal általánosabb pattern matching-et használja a kivételkezelő ágak kiválasztására - azonban itt is a legspecifikusabb ágnak kell először szerepelnie.

Egy programkönyvtár írásakor a lehetséges hibák jó részével nem tudunk érdemben mit kezdeni. A legjobb megoldás jelezni őket, majd a kliens kódra bízni, hogy mi történjen: újra próbálkozik, jelzi a felhasználó felé, esetleg méltóságteljesen vet véget a programnak. Ennek megvalósítása majdnem lehetetlen feladat kivételek alkalmazása nélkül, azonban használatukkal egyszerű : a 3.5. ábrán szemléltetett kód egyszerűen dob egy saját típusú kivételt, majd a hívó félre bízza annak feldolgozását. Ha egy kivételt a hívó fél nem kap el, a függvényhívás helyét lényegében a `throw` veszi át, és az őt hívó felelősségévé válik a kivétel elkapása. A legtöbb futtatókörnyezet biztosít egy legfelsőbb szintű „elkapó” függvényt, mely minden el nem kapott kivételt lekezel: a legtöbb esetben egy stacktrace illetve egyéb hasznos információk biztosítása mellett. Ennek a módszernek a sarokköve a kupac visszagörgetése

```
1  void MyFunc()
2  {
3    throw MyException("A specifikus üzenetem");
4  }
```

3.5. ábra. C++-stílusú `throw`

(stack unwinding), azonban ennek ismertetése túlmutat a dolgozat keretein.

A kivételkezelés itt megfontolt tulajdonságai jól jellemzik, hogy egy általános, jól használható módszert adnak hibák jelzésére a programozók kezébe. Azonban ez bonyolult kérdéseket is felvet, mint például mi történik olyan helyzetben, ha fellép egy kivétel, amit nem kaptak el, de a program futásán életek múlnak.

3.3. Exception safety, egyéb megfontolások

Képzeljük el az alábbi helyzetet: egy repülőgép vezérlő szoftverének egyik modulja dob egy kivételt, amire azonban a hívó fél nem volt felkészülve, és így a program futásának befejezését eredményezi. Ez nyilván elfogadhatatlan ebben a helyzetben. Ennek jellemzésére vezették be a kivételbiztosság (exception safety) fogalmát: jellemezhetjük, hogy az adott kód mennyire biztonságosan fut. Öt szintjét különböztetjük meg:

1. No-throw garancia: minden kivételes esetet az eset előfordulásának helyén kezelünk

2. Commit-rollback vagy változtatás-mentesség: ha egy művelet közben kivétel dobódik, az eddigi hibás eredményt eldobjuk, és visszaállunk az eredeti állapotra

3. Alapvető biztonság: Ugyan lehetnek mellékhatások kivételekkor, azonban a meghatározott invariánsok teljesülnek

4. Minimális biztonság: Lehetnek mellékhatások, azonban a program nem fog véget érni, és minden erőforrást karban tartunk (no-leak)

5. Nincs biztonság: Nem tudunk semmit az adott kód kivételek esetén végbemenő biztosságosságáról.

A fenti lista a biztonságosság – és így a megvalósítás nehézségének – sorrendjében áll. Elég belegondolnunk a bevezetőben ismertett Guru of the week [7] cikkre, amelyben egy párszoros program a kivételek miatt huszonhárom különböző végrehajtási utat járhat be. Ennek fényében látszik, hogy a legbiztosabb garanciát közel nem triviális betartani.

A kivételkezelés felvet további egyéb kérdéseket is: vajon a biztosításához szükséges kódtöbblet hogyan viszonyul a program teljesítményéhez, illetve az elkapás közbeni stack unwinding mennyire költséges művelet; azonban ezeknek a kérdéseknek a vizsgálatához ez a dolgozat nem biztosít keretet.

Az előzőekből jól látszik, hogy legjobb esetben is csak sejtéseink vannak a kivételek bonyolultságra gyakorolt hatásáról, pedig fontos kérdéseket kell tudnunk megválaszolni ezzel kapcsolatban. A dolgozat további részében megpróbálunk ezekre választ találni.

4. fejezet

Metrikák ajánlott kiterjesztése

Az előzőekben láthattuk, hogy a metrikák hogyan fejezik ki szoftverek tulajdonságait, az itt tárgyalt esetekben a kód bonyolultságát. Mindhárom bemutatott metrika megpróbál egy szám és a komplexitás közé direkt párhuzamot vonni, azonban –a LOC-ot kivéve– nem foglalkoznak a kivételkezelés kérdésével. Az előző fejezetben bemutattuk, hogy a kivételkezelés milyen megoldásokat kínál a hibakezelésre, mely magyarázatot adhat elterjedésének okára, valamint hogy miért szükséges foglalkoznunk vele. Ebben a fejezetben bemutatom ajánlásomat a metrikák kibővítésére, melyekkel mérhetjük a bonyolultságra gyakorolt hatásukat, valamint bemutatom az ezek mögött meghúzódó érveket is. Természetesen a LOC-ot nem befolyásolják a kivételek, így azzal itt nem foglalkozom.

4.1. A ciklomatikus komplexitás

A McCabe által definiált metrika alapja a predikátumok szerepe a kódban, melyek meghatározzák az adott programrészlet végrehajtási útját. Ebbe a modellbe kell beillesztenünk a kivételek dobásának és elkapásának vizsgálatát.

Természetes döntés, hogy egy adott `try` blokk után az összes `catch` ágat mint egy-egy predikátumot fogunk fel: ezek gyakorlatilag ezen a módon viselkednek. Ha az adott `try` blokkban történt kivétel, akkor az azt követő

`catch` ág mind-mind egy feltétel a kivétel típusára. Ennek megfelelően a kiterjesztett metrikában minden egyes `catch` ág eggyel növeli a függvény bonyolultságát.

A kivételek dobásának értelmezése már nem ennyire triviális. A függvényben szereplő `throw`-k előtti feltételellenőrzéseket természetesen figyelembe vesszük, azonban ezeknek a függvényeknek hívásakor figyelmen kívül hagyjuk őket. Elsőre ez furcsának tűnhet, főleg a bevezetőben ismertett Guru of the Week cikk [7] kapcsán, hiszen az ott tapasztalható végrehajtási ágak számának robbanásszerű növekedése látszólag szoros összefüggésben áll McCabe vizsgálódásaival. Ennek ellenére mégsem lenne helyes számolnunk ezeket: a kliens kódban ugyanis nem növelik a predikátumok számát. Ha a mi kódbázisunkban szereplő függvényekről van szó, akkor azoknak a vizsgálatakor számoljuk a predikátumok számát, így növelve a teljes bonyolultságot. Azonban ha nem általunk fejlesztett kódról van szó, az a McCabe-i értelemben nem növeli a mi kódunk bonyolultságát.

4.2. Az A-V metrika

Az A-V metrika a McCabe által kijelölt útvonalon továbbhaladva, a predikátumok figyelembe vételének továbbfejlesztésén túl a kódban szereplő adatéléréseket is számításba veszi. Ennek megfelelően az ajánlott kiterjesztés is felhasználja a beágyazottsági mélység fogalmát, valamint figyelembe veszi a változóhasználatot is kivételkezelés közben. A következőkben felhasznált beágyazottsági mélység, illetve ennek definiálása során használt egyéb fogalmak matematikai pontosságú definíciói megtalálhatóak Porkoláb dolgozatában [6]; itt ezekre nem térünk ki részletesen, azonban az eredményeket felhasználjuk.

Ahogyan a korábbiakban ismertettük, a legtöbb programozási nyelv valamilyen típussal reprezentálja a dobott kivételeket. Vizsgálódásunk szempontjából elhanyagolható, hogy ezeknek egy konkrét ősosztályból kell-e származniuk, vagy tetszőleges típus dobható-e. Ennek megfelelően nem teszünk kivételt az excpetion-ök létrehozásakor, illetve elkapásakor sem az A-V által definiált adatírási- és olvasási definícióktól. A kivétel létrehozása egy írási, valamint a meghívott konstruktortól függően $1..n$ olvasási művelet, persze fi-

gyelembe véve az adott hívás beágyazottsági mértéket. Annak ellenére, hogy nem tartják jó programozási gyakorlatnak, ha egy kivételt az adott osztály mezőként tartalmaz, azonban a teljesség kedvéért ezt is figyelembe vesszük, és egy adatelérésként tartjuk nyilván.

Az A-V metrika nyilvántartja az adott művelethez vezető döntéseket jellemző predikátumok számát. Ennek megfelelően jobban jellemezhetjük a kivételek dobását is, amelyet az adott beágyazottsági mélységgel reprezentáljuk. Az explicit `throw` utasításokon túl azonban –a McCabe-metrikával ellentétben– itt figyelembe vesszük a meghívott függvények lehetséges kivételdobásait is: az adott `throw` utasítást illetve függvényhívást regisztráljuk azoknak beágyazottsági számával az összes kivételre. Ez `throw`-nként természetesen csupán egy kivételt jelenthet, azonban ha egy függvény több fajta kivételt is dobhat, mindegyiket figyelembe vesszük, ez ugyanis több lehetséges hibát jelent, ami természetesen növeli a tesztelendő esetek számát és a megértéshez szükséges erőfeszítést is.

A kivételek elkapása kliens kódban gyakran történik a kivétel lehetséges dobásával megegyező függvényben, azonban ezek között jelentős bonyolultságú kód szerepelhet. Könnyen elképzelhető az is, hogy a dobások és az elkapások között több feltételt kell kiértékelni, ami jelentősen bonyolítja a kódot. Ezen érvek miatt a kivételek dobásának helyére és elkapására jellemző beágyazottsági mélységek közötti különbséget vesszük figyelembe. Egy `try-catch` páros között egy soklépcsős ellenőrzés is történhet, amit figyelembe kell vennünk. Természetesen ennek számításakor szem előtt tartjuk a programozási nyelv szemantikáját: a helyes zárójelezéshez hasonlóan megkeressük a függvényben található explicit vagy implicit dobásokhoz tartozó elkapó ágakat (ezek egyértelműek minden esetben), és ezeknek számítjuk a különbségét. Erre találhatunk a 4.1. ábrán egy példát: a `NotMyException` dobása és elkapása közötti beágyazottsági különbség 2, így ezt a típusú kivételt ebben a függvényben ezzel az értékkel tartjuk nyilván.

A kliens kódokkal ellentétben a könyvtárakra jellemző, hogy a kivétel előfordulásának helyén azzal nem tudunk kezdeni semmit, így a dobás helye távol kerül az elkapásétól. Ennek kifejezésére az adott függvényben el nem kapott kivételeket a dobásuknak megfelelő beágyazottsági számmal tartjuk

```
1  void f() throw MyException
2  {
3    try
4    {
5      if (/*predikátum1*/)
6      {
7          if (/*predikátum2*/)
8          {
9            throw new NotMyExcpetion();
10         }
11     }
12   }
13   catch (NotMyException) { /*...*/ }
14   if (/*predikátum3*/)
15   {
16     throw new MyException();
17   }
18 }
```

4.1. ábra. `throw-catch` utasítások beágyazottsága

nyilván, és a függvény bonyolultságához adjuk hozzá. Ennek oka, hogy a kivételes esemény bekövetkezéséhez jutó végrehajtási út megértése ugyanolyan bonyolult, viszont a kivétel a függvény hívásakor növeli a bonyolultságot.

A fenti megfontolások következtében a következőképpen alakul egy függvény bonyolultsága a kiterjesztett metrikában: vesszük az eredeti A-V által definiált komplexitást, ehhez hozzáadjuk a kivételek használatakor történt adatműveleteket, majd ezt növeljük a kiszámított `throw-catch` különbségekkel, végül az el nem kapott kivételek beágyazottsági mélységével.

A fejezetben ismertetett módszerekkel igyekszünk választ adni a kivételkezelés szoftvermetrikák által eddig elhanyagolt helyzetére, hogy jobb jellemzést kaphassunk egy adott szoftvertermék komplexitásáról.

5. fejezet

Példák, eredmények

Az alábbi fejezetben összefoglaljuk a metrikák eddig tárgyalt bővítésének eredményeit. A fejezet első részében egyszerű, rövid példákon mutatjuk be a kivételkezelés a számított bonyolultságra tett hatását. A második rész egy nyílt forráskódú program, az Apache alapítvány által készített Tomcat webszerver elemzésének bemutatásával foglalkozik. A harmadik részben röviden kitérünk a Weyuker axiómiái és a metrikák kibővítése közötti kapcsolatra.

5.1. Példák

Ebben az alfejezetben néhány egyszerű példán keresztül kívánom bemutatni a kivételek hatását a metrikák által képzett eredményekre, rávilágítva arra, hogy mennyire fontos figyelembe vennünk őket, amikor szoftverek bonyolultságával foglalkozunk.

Elsőként vegyük példaként a 5.1. ábrán látható, Java-ban készült osztályt. A kód 21 sorból áll, tehát a LOC metrika eredménye is ez. A McCabe metrika megszámolva a predikátumokat a kódban 2-t ad eredményül, figyelembe véve a 7. és 13. sorokban található elágazások feltételeit. Mivel ebben az egyszerű függvényben nem szerepel `catch`-blokk, a bővített metrika szintén 2-vel tér vissza.

Érdekesebb a helyzet az A-V esetében. Adatműveletek történnek a 7., 15. és 18. sorokban, rendre 1, 3, 1 beágyazottsági mélységgel (ND), vala-

```
1  package av.metrics.org.test;
2
3  public class TestClass {
4
5          int myMethod(int x) throws IllegalArgumentException
6          {
7                  if (x > 3)
8                  {
9                          throw new IllegalArgumentException();
10                 }
11                 else
12                 {
13                         if (true)
14                         {
15                                 myMethod(x--);
16                         }
17                 }
18                 return x;
19         }
20 }
```

5.1. ábra. Egy egyszerű Java-osztály

mint predikátumok 1 és 3 mélységgel. Így a függvény bonyolultsága az eredeti A-V szerint 8. Ez természetesen nem veszi figyelembe a 9. sorban található `throw`-t, és a 15. sorban található függvényhívást sem. Ha azonban a kibővített metrikát használjuk a számításra, eredményként 13-at kapunk, ugyanis az eredeti értékhez hozzá kell adnunk a két kivételkezeléshez kapcsolódó utasítást a 9. és 15. sorban, 2 és 3 ND-vel. Ez a példa azért is remekül példázza a kivételkezelés fontosságát, mert nincs benne felesleges adatkezelés. A kivételek figyelembe vételével a számított bonyolultság csaknem másfélszeresére nőtt, ami intuitívan is helyesnek tűnik: a kódrészlet bonyolultságát főként a kivételkezelés használata okozza.

Az 5.2. ábrán található, kissé bonyolultabb példán jól látható néhány kivételkezelési technika (a függvényeket ezúttal Java-szerűen formáztam, hogy a példa kevesebb helyet foglaljon). A kód ebben az esetben 32 sor hosszú. A kivételek nélküli ciklomatikus komplexitás 4, a 7., 10., 18. és 22. sorban található predikátumok következtében. Ezúttal szerepel a `calc` függvényben

```
1  package av.metrics.org.test;
2
3  public class TestClass {
4  int field = 0;
5
6  public int myMethod(int x) throws IllegalArgumentException {
7          if (x > 3) {
8                  throw new IllegalArgumentException();
9                  } else {
10                 if (true) {
11                         calc(x + field++);
12                 }
13         }
14         return x;
15 }
16
17 private void calc(int x) throws IllegalArgumentException {
18         if(x < 3) {
19                 throw new IllegalArgumentException();
20         } else {
21                 try {
22                         if (field > 5) {
23                                 throw new IllegalAccessError();
24                         } else {
25                                 calc(field);
26                         }
27                 } catch (Exception e) {
28                         field = x;
29                 }
30         }
31 }
32 }
```

5.2. ábra. Bonyolultabb Java-osztály

egy `catch`-ág is a 27. sorban, amely minden `Exception`-ből származó kivételt elkap, így ezt figyelembe véve a kibővített metrika eredménye 5.

Az egyszerű A-V figyelembe veszi az osztályban található adatmezőt, melyet a `field` névvel láttunk el, így az egész osztály bonyolultsága máris 1. Az előző példához képest a `myMethod`-ban található függvényhívást kicseréltük a `calc`-ra, valamint az ennek átadott paraméterek is változtak: ebben az

esetben itt szerepel a `field`, valamint a lokális változó, `x` is, mindkettő 3-as ND-vel, így a függvény adatkomplexitása 8, a predikátumok mélységeinek összege 3, azaz összesen 11.

A második, `calc` nevű függvény adathasználata a 18., 22., 25., 27., 28. sorokban történnek, rendre 1, 2, 3, 2, 2, 2 beágyazottsági mélységgel, ezek összege 12. Furcsának tűnhet a `catch`-ben található `e` változót számolnunk, azonban ez egy egyszerű adatolvasási művelet. Az előbbiekhez kell hozzáadnunk még a függvényben található predikátumok mélységeit, amik a 18. és 22. sorokban találhatóak, ND összegük ebben az esetben 3, így lesz a függvény 15, az osztály bonyolultsága pedig 27.

Az előzőeken jelentősen módosít, ha figyelembe vesszük a kivételeket is. Mivel a 11. sorban található `calc` hívás dobhat kivételt, ezért ezt regisztrálnunk kell 3-mal. A `calc` függvényben explicit dobások szerepelnek a 19. valamint a 23. sorokban, az előbbi 2, míg utóbbi a 27. sorban található `catch` miatt 1-es mélységgel. Található még egy rekurzív hívás a 25. sorban. Mivel a függvény `IllegalArgumentException`-t dobhat, ezért ezt figyelembe kell vennünk a hívás beágyazottsági mélységével, azonban nincs a potenciális dobáshoz tartozó `catch`-ág, ezért a dobás ND-jével számolunk, ami 3, így a kivételek használata 9-et ad a bonyolultsághoz, ami így 36-ra nő.

A hosszabb esetben a McCabe-féle metrika változása nem kiemelkedő: 4-ről 5-re nőtt, ami 25%-ot jelent. Az A-V azonban ebben az esetben is jelentősebben nőtt: 27-ről 36-ra, ami 33%-os változás. Ebből a példából jól látszik, hogy csekély adathasználat mellett az A-V és a McCabe bővítése hasonló eredményt szül, ami implicit módon bizonyítékot ad arra, hogy a kivételekre vonatkozó definíciók jól használható bővítését adják a metrikáknak.

A következő példában a korábbi hibakezelési módszereket hasonlítom össze a kivételekkel, így itt csak ezekkel a konstrukciókkal foglalkozom. A programokban szereplő egyéb utasítások az előzőekhez hasonló módon számolhatóak. A 5.3. ábrán látható példában a `calcWE` függvény két kivételt is dobhat, melyekkel átad egy, a felhasználók számára is hasznos üzenetet. Az itt szereplő két függvény bonyolulságának összege 17. Vizsgáljuk most meg a 5.4. ábrán látható osztályrészletet. Ezzel próbáltam szimulálni az előző

```
1  int calcWE(int x) throws
2          IllegalArgumentException, MyException {
3          if (x == 0) {
4                  throw new IllegalArgumentException("Nulla");
5          }
6          if (x == 1) {
7                  throw new MyException("Egy");
8          }
9          if (x == 0)
10         x--;
11         calcWE(x);
12         return x;
13 }
14
15 void clientWE() {
16         int number = 15;
17         try {
18                 calcWE(number);
19         }
20         catch(Exception e) {
21                 System.out.println(e.getMessage());
22         }
23 }
```

5.3. ábra. Java függvények kivételekkel

példában használt kivételeket. Ebben a méretben kicsit furának tűnhet ilyen megoldásokat használni, azonban nagyobb rendeszereknél ezek a hibakezelési módszerek nem példátlanok. Ebben az esetben az osztály bonyolulsága 19-re nőtt, ami nem tűnik jelentősnek, viszont a teljesen pontos szimulációhoz a 12. sorban található `calcNE` hívás eredményét is vizsgálnunk kellene, így még tovább növekedne ennek a verziónak a bonyolultsága – a másik függvényben használt ellenőrzések használatával 4-gyel, ami viszont már jelentős növekedést jelent.

A fentiekből jól látszik, hogy a kivételek használata pozitívan befolyásolja a kód bonyolultságát (azaz csökkenti azt), míg mellőzésük akár spagetti-szerű kódot, átláthatatlan hivatkozásokat is eredményezhet, ami nagyobb méretben a karbantartás költségeinek növekedésével jár.

```
1  private HashMap<Integer, String> errors =
2          new HashMap<Integer, String>();
3
4  int calcNE(int x) {
5          if (x == 0) {
6                  return 0;
7          }
8          if (x == 1) {
9                  return -1;
10         }
11         x--;
12         calcNE(x);
13         return x;
14 }
15
16 void clientNE() {
17         int number = 15;
18         int res = calcNE(number);
19         if (res == 0) {
20                 System.out.println(
21                 "IllegalArgumentException: " +
22                         errors.get(res));
23         }
24         if (res == -1) {
25                 System.out.println(
26                 "MyExcpetion: " + errors.get(res));
27         }
28 }
```

5.4. ábra. Java függvények kivételek nélkül

5.2. Egy szoftver elemzése: Apache Tomcat

A szintaktikus példák mellett érdemes az elméletet egy, az iparban fejlesztett szoftverrel is tesztelnünk, amely jobban példázza azokat a módszereket, amiket a programozók valóban használnak. Ehhez a vizsgálódáshoz egy nyílt forrású szoftvert, az Apache Tomcat-et választottam. Az elérhető verziók közül a 6-ossal végeztem a vizsgálódásaimat, a dolgozathoz készült programkönyvtár segítségével. Ennek a futtatásnak az eredményeit és azok elemzését tartalmazza ez a szakasz.

Metrika	Kivételek nélkül	Kivételekkel	Változás mértéke
LOC	279830	279830	0%
McCabe	14812	16801	13,428%
A-V	50438787	52723815	4,5303%

5.1. táblázat. A Tomcat elemézésének eredménye

A 5.1. táblázatban megtalálhatóak a számszerű adatok. Az általam használt verzió 279830 sorból állt, ami természetesen nem változott a kivételek figyelembe vételével.

A szoftver ciklomatiks komplexitásának változása 13%, ami jelentősnek mondható. Ez jól példázhatja, hogy a kivételkezelés mennyire elterjedt hibakezelési mód napjainkban. Ez a jelentős növekedés különösen érdekes, ha figyelembe vesszük, hogy a Tomcat jellemzően nem klienskód, sokkal inkább hasonlít egy programkönyvtárra, valamint hogy a McCabe-i kibővítés jellemzően a klienskóddal foglalkozott, hiszen csupán a `catch`-blokkokat vettük figyelembe mint lehetséges predikátumokat. Ez arra enged következtetni, hogy a Tomcat írói a saját kódjukban is használják hibajelzésre a kivételeket, és nagy valószínűséggel a különböző rétegek között jelzik velük a kivételes helyzeteket. Előfordulhat még az is, hogy a kivételeket nem teljesen eredeti céljuknak megfelelően használják: nemcsak hibás, kivételes esetek jelzésére, hanem az adott ponton akár elfogadható, azonban nem tökéletes helyzetek jelzésére is a hívó félnek.

Az A-V metrika növekedése nem szembeötlő a McCabe-hez képest: csupán 4% körüli. Ennek oka feltehetően az, hogy az adathasználat „elnyomta” a kivételek által generált növekedést. Ha jobban megfontoljuk, ez is teljesen helyes adatnak tűnik: a kódban a helyes, normális futáshoz tartozó részek sokkal jelentősebbek, mint a kivételes helyzeteket kezelő. Ez a gyakorlatban írt szoftvereket is jól jellemzi: amikor már nem tudunk megfelelően hibát kezelni, egy egyszerű hívással dobunk egy kivételt, amibe eselteg belefoglaljuk egy-két jellemző változó értékét, azonban ez nagyon csekély a többi művelethez képest is.

5.3. Weyuker állításai

Az alábbi részben összefoglaljuk, hogy a metrikák a dolgozatban szereplő ajánlott bővítése hogyan befolyásolja azok viselkedését a Weyuker által felállított szempontrendszerben. Korábban már megmutatták, hogy a McCabe-féle komplexitás négy axiómát is megsért [6], valamint hogy az A-V mind a kilencet teljesíti [6]. Az itt szereplő módosítások nem befolyásolják annyiban a McCabe számítási módszerét, hogy a weyukeri állításokat jobban kielégítse, azonban azt érdemes megvizsgálni, hogy az A-V-n esetleg ront-e. A Weyuker-axiómák pontos leírása megtalálható 1988-ban írt cikkében [8], valamint Porkoláb értekezésében adott egy jó összefoglalást róluk [6], így ezekre itt nem térek ki, csupán az ott használt eredményeket és fogalmakat használom fel.

1. Létezik két különböző program, amelyek más bonyolultsági mértékkel rendelkeznek. Ez az állítás triviálisan teljesül már az eredeti A-V-ban is, és ha kivételek nélküli kódon számítjuk ki, akkor a bővítés ugyanazt az eredményt adja. Ha csupán a kivételkezeléshez kapcsolódó konstrukciókat tartalmaz a program, akkor is teljesül ez az állítás: például az egyikben szerepeljen két `catch`-ág, míg a másikban csak egy.

2. Legyen c nemnegatív szám. Ekkor legfeljebb végesszámú olyan program létezik, melyeknek bonyolultsága c. Az eredeti A-V akkor teljesíti ezt az állítást, ha az adatkapcsolatok többszörös adathasználat esetén többszörös éllel van jelölve. Ha a bővítés közben is így járunk el, akkor nem rontjuk el az A-V ezen tulajdonságát.

3. Léteznek olyan különböző programok, amelyeknek értéke megegyezik. Ez szintén triviálisan teljesül: ha veszünk két olyan programot, amelyek két különböző függvényt hívnak meg, amelyek azonban ugyanazt az n kivételt dobják (n akár 0 is lehet), akkor ezek teljesítik a feltételt.

4. Létezik két olyan, más implementációjú program, amik ugyanazt az eredményt szolgáltatják. Erre a feltételre is könnyű példát találnunk:

amennyiben két olyan függvényt vizsgálunk, amelyek máshogyan kezelik a kivételeket (például az egyik a saját törzsében elkapja azokat, míg a másik nem dob kivételt, esetleg csak nem tartalmaz `catch`-blokkokat), akkor a két függvény különböző, azonban eredményük ugyanaz lesz.

5. Amennyiben egy adott programot bővítünk, a bővített program bonyolultsága nőjön. Mivel az eredeti és a kibővített A-V is minden konstrukciónál növeli a programot jellemző értéket, így ez a monotonitási feltétel is teljesül.

6. Léteznek olyan, ugyanolyan bonyolultsággal rendelkező programok, amelyeket ugyanazzal a programmal bővítve különböző értéket kapunk. Ahogyan Porkoláb is kifejti [6], mivel az A-V figyelembe veszi az adatáramlást is (így a környezettől is függ egy program bonyolultsága), és a bővítést is beillesztettük ebbe az adatáramlási definícióba, így a mi bővítésünk is teljesíti ezt a feltételt.

7. Léteznek olyan programok, melyek csak az utasításaik sorrendjében különböznek, azonban bonyolultságuk eltérő. Erre példát találnunk a beágyazottsági mélység felhasználásával a legegyszerűbb. Mivel a kódban szereplő `catch`-blokkok is befolyásolják az ND-t, így belátható, hogy ha ezeket felcseréljük, különböző bonyolultságot kaphatunk.

8. Egy program változóinak átnevezése nem változtatja meg a bonyolultságot. Mivel a definícióink során sehol nem használtuk fel a programok ezen elemeit, így ez triviálisan teljesül.

9. Bármely programok konkatenációja nem csökkenti a bonyolultságot. A feltétel ellenőrzéséhez hasonló állításokkal juthatunk, mint az előző, környezetekhez kapcsolódó feltételnél: mivel a metrika figyelembe veszi mind az adatáramlást, mind az utasítások beágyazottságát, így ez a feltétel is igaz rá.

6. fejezet

A metrikák kiszámításának implementálása

Ebben a fejezetben bemutatom az elmélet alátámasztásához készített programkönyvtár írása során használt eszközöket, megfontolásokat. Felhasználtam az Eclipse platform által nyújtott szolgáltatásokat, mint például a kódot objektumként reprezentáló `org.eclipse.jdt.core` csomagot, ilettve az Eclipse könnyű bővítését elősegítő plug-in API-kat.

6.1. Az Eclipse bemutatása

Az Eclipse projekt 2001-ben indult az IBM vezetésével, mely köré később egy IT-megoldásokat szállító cégekből álló konzorcium szerveződött. A projekt később túlnőtt az eredeti célokon, ezért 2004-ben megalakult az Eclipse Alapítvány, melynek elsődleges feladata az Eclipse, mint szoftvertermék nonprofit módon való fejlődésének irányítása.

A projekt célja egy olyan fejlesztési platform létrehozása volt, mely magába foglal könnyen bővíthető framework-öket, programkönyvtárakat; olyan általános célú, megvalósítású eszközökkel segíti a fejlesztőket, melyek elkészítése rengeteg erőforrást és időt igényelne. A projekt eredményeinek felhasználásával viszont bárki számára elérhető, open source, jó minőségű modulokkal mindenki jobban koncentrálhat a saját projektjének fő részeire. Az Eclipse

többbek között tartalmazza az SWT GUI-toolkit-et, Java integrált fejlesztői környezetek (IDE) létrehozásához szükséges parsereket, objektummodelleket (JDT), a C/C++-hoz köthető hasonló célú CDT-t, projektmanage-eléshez, tervezéshez szükséges UML-szerkesztőket és további tervező eszközöket. Az egész platform tervezésének egyik fő irányelve a könnyű bővíthetőség, épp ezért alapvető strukturális elem a plug-inezhetőség, minden alapértelmezett implementáció elérhető jól meghatározott API-kon keresztül. Az Eclipse-hez elérhető bővítmények száma virtuálisan végtelen, szinte minden új és régi programnyelvhez elérhető IDE-plugin. A legtöbbet használt ezek közül a Java bővítmény, mely a legnagyobb húzóereje a platform fejlesztéseinek, valamint az Eclipse-alapú Java IDE a legnépszerűbb a fejlesztők körében.

Az Eclipse legnagyobb része Java-ban készült, futtatása valamilyen JVM-en történik, épp ezért az API-k a legkönnyebben valamilyen bytecode-ra fordított, vagy ahhoz könnyen linkelhető nyelvből érhetőek el. Ezek közé tartozik természetesen a Java, de az egyre népszerűbb Scala, illetve Clojure is. Funkcionális aspektusai miatt én a Scala-t választottam a plugin elkészítéséhez.

6.2. Plugin-ek írása

Az Eclipse elékszítésének egyik fő mozgatórugója a közösség által készített bővítmények. Ezen okból kifolyólag Eclipse alól fejleszthetünk a legkönnyebben ilyen plugineket. Az IDE rögtön elérhető módon támogatja ezeknek a létrehozását, külön típusú Projecttel, mely tartalmaz minden szükséges beállítást, függőséget.

A dolgozathoz készült plugin is egy ilyen típusú Project, egyetlen Java osztállyal, amelyet az Eclipse indulásakor tölt be. A plugin beállításai a `plugin.xml`-ben történnek, ez a file tartalmazza az általános leírását minden Eclipse pluginnek.

A plugin két lényegi részre osztható. Az egyik, amely az Eclipse-be való integrálást végzi, a másik, ami a metrikák értékeink generálásáért felelős.

Az integráció egy `LaunchConfigurationDelegate`-en keresztül történik. Ez lényegében egy olyan indítási konfiguráció létrehozását végzi, mely a kód

elkészülte után futtatható. Ebbe a kategóriába esik egy kész projekt fordítása és futtatása, de a kódot analizáló eszközök is ezen keresztül érik az Eclipse projekteket: természetes választás volt tehát ezt felhasználva indítani a metrikák kiszámítását. Az általam írt `LaunchConfigurationDelegate` ellenőrzi, hogy az adott Eclipse példányban található projekt-ek tartalmaznak-e fordítási hibát; ha nem, elkezdi a számítást. Erre azért volt szükség, mert a JDT által felépített reprezentáció csak abban az esetben felel meg tökéletesen a specifikációnak, ha a kód szintaktikai és szemantikai hibáktól mentes. Ennek kiküszöbölése jelentősen túlmutat a dolgozat keretein.

A pluginhez tartozik még a beállításokat tartalmazó grafikus felület is, amely azonban csak az közösség által ajánlott, előre elkészített modulokat (a felületen: füleket) tartalmazza, mivel a számítás nem tesz semmilyen felhasználói konfigurációt szükségessé.

6.3. A metrikák kiszámítása

A plugin lényegi munkáját végző kód Scala-ban készült. A Scala egy multiparadigmás nyelv, kevert módon objektumorientált és funkcionális. Ugyan a specifikációjában nem szerepel, hogy bytecode-ra kötelező fordítani, azonban a jelenleg elérhető legteljesebb implementáció így működik, és tökéletesen illeszkedik minden egyéb bytecode-ra forduló nyelvhez, így a Java-hoz is. Ezt az előnyét használtam ki, amikor az Eclipse API-kat felhasználó programot írtam benne. A Scala-nak rengeteg előnye van a Java-hoz képest: támogatja a funkcionális stílusú programozást, a mutálódó értékek helyett a konstansokat részesíti előnyben, valamint támogat minden, a funkcionális nyelvekből jól megszokott konstrukciót is, mint a függvényliterálok, listagenerátorok, pattern matching, tail-rekurzió, stb.

A könyvtár erősen támaszkodik a JDT-re. Ez a csomag az Eclipse-ben megtalálható Java-kódot transzformálja egy objektumfává, mely már programból könnyen elérhető módon támogat bejáró, módosító műveleteket. A csomag készít a kódból egy absztrakt szintaxisfát (AST), melynek csúcsai típusosan reprezentálják azt. Természetesen teljes mértékben követi a Java nyelvi specifikációját, minden esetet helyesen kezelve.

A bejárás a látogató-minta alapján történik. Ennek lényege, hogy egy osztálynak rendelkeznie kell a megfelelő függvényekkel, amelyeket rekurzív módon hív meg a program. Ennek a mintának egy speciális változatát használja a JDT is. Egy olyan osztályt kell készítenünk, mely leszármozottja az `ASTVisitor`-nak, és ebben túl kell terhelnünk (overload) azokat a függvényeket, amelyek az osztály érdekeltségébe tartozó, megfelelő típusú elemeken hívódnak meg. Ezek a függvények mind egy egyszerű szignatúrával rendelkeznek: `visit(node: NodeType): Boolean`, ahol a `NodeType` egy specifikus elemtípus. A függvények meghívása a túlterhelést feloldó szabályok következtében történik helyesen. A függvények visszatérési értéke azt reprezentálja, hogy az adott elem gyerekeit meg kell-e látogatni az adott osztállyal. Ennek megfelelően az `ASTVisitor` osztályban található implementációk mind egy utasításból állnak: `return true;`. Ez biztosítja, hogy ha egy adott elem után mi meg is szakítjuk a bejárást, a többi elemre tovább folytatódik az.

A fentiek mellett biztosít a környezet egyéb, a file-okat, mint dokumentumokat jellemző tulajdonságokat, például a bennük szereplő sorok számát, vagy hogy az adott file helyes-e. További érdekesség, hogy a Scala plugin is a JDT-éjével megegyező fát épít fel, így elméletileg a programom alkalmas Scala-kódok elemzésére is, azonban ennek teljes ellenőrzése nem fért bele a dolgozat kereteibe.

Mint látható, az Eclipse egy jól használható, a problémához mérten egyszerű megoldást kínál, amely nélkül igencsak fáradtságos lett volna a metrikát kiszámító programot elkészíteni, valamint a helyességét garantálni majdnem lehetetlen lenne.

7. fejezet

Összefoglalás

A McCabe- és A-V-metrikák bővítésével a kivételkezelés módszerének a komplexitásra gyakorolt hatását vizsgáltuk. Mindkét metrika esetében az őket meghatározó alapvető irányelveket követve adtunk ajánlást bővítésükre.

A ciklomatikus komplexitás esetében egyszerűen bővítettük a vizsgált predikátumok körét a kivételkezelés közben használt `catch`-blokkokkal, melyek a kivétel típusától függően hajtódnak végre. Ebben az esetben nem vizsgáltuk a kivételek kiváltódásait, ugyanis ez túl sokat torzított volna az eredeti, McCabe-i definíción.

Az A-V-metrikákat a rájuk jellemző módon egészítettük ki: először is figyelembe vettük a kivételkezelés közbeni adathasználatot –legyen az írás vagy olvasás–, beépítve azt az eredeti A-V adatáramlási gráfjába. Ezek után figyelembe vesszük a kivételek lehetséges dobásának következményeit is, legyen az explicit `throw` utasítás, vagy egy függvény hívása, mely tekinthető egy rejtett potenciális kivétel kiváltódásnak is. A kivételek elkapását az előző definícióhoz hasonlóan, a dobás és az elkapás közötti beágyazottsági mélység differenciájával mérjük, mekkora utat tesz meg a kivétel. Amennyiben a dobó függvény nem kapja el az adott kivételt, mindenképpen visszajut a hívás helyére, így olyan bonyolultsággal tekintünk rá, mint amilyen nehéz megérteni a kiváltódásának okát.

Egyszerű példákon keresztül megmutattuk, hogyan működik a metrikák új kiszámítása, valamint ezeken a példákon keresztül rávilágítottunk a de-

finíciók mögötti megfontolásokra. Összehasonlítottuk a korábbi hibakezelési technikák egyik elterjedt változatát a kivételkezeléssel, majd egy egész szoftver elemzését végeztük el. Az Apache Tomcat 6-os verziójának elemzése közben fény derült arra, hogy a McCabe-metrika érzékenyebb lett az új definícióinkra, mint az A-V, amit feltehetően a programban szereplő nagymértékű adathasználat magyaráz. Ezek után rátértünk a metrikákat jellemző Weyuker állítások elemzésére, és megutattuk, hogy a bővítések ebből a szempontból ugyan nem javítanak a metrikákon, de nem is rontják el azokat.

A mérésekhez használt programkönyvtár bemutatásánál kitértünk a használt eszközök fontos tulajdonságaira, melyek lehetővé tették, hogy Java programokon egyszerűen, mégis hatékonyan és helyesen tudjunk teszteket futtatni.

További vizsgálódási irányként elképzelhetőnek tartjuk az aktuális definíciókkal egy nagyobb szoftver két verziójának olyan összehasonlítását, melyek közül az egyik használ kivételeket, míg a másik nem. Ezzel nagyobb léptékben is vizsgálhtatnánk a kivételek használatának hatását. További fejlesztési lehetőség a dolgozatban használt programkönyvtárat nyelvagnosztikusan elkészíteni, így kibővíteni a lehetséges vizsgálódások tárgyát.

Ábrák jegyzéke

Irodalomjegyzék

[1] Cem Kaner, W. P. B. Software engineering metrics: What do they measure and how do we know? 10th International Software Metrics Symposium (Metrics 2004), 2004.

[2] DeMarco, T. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall, 1986.

[3] J.C. Cherniavksy, C. S. On weyuker's axioms for software complexity measures. *IEE Trans. Software Engineering, vol. 17, pp.1357-1365* (1991).

[4] Louden, K. C. *Programming Languages: Principles and Practice*. Course Technology, 2002.

[5] McCabe, T. J. A complexity measure. *IEE Trans. Software Engineering, SE-2(4), pp.308-320* (1976).

[6] Porkoláb, Z. *Programok Strukturális Bonyolultsági mérőszámai*. PhD thesis, Eötvös Loránd Tudományegyetem, 2002.

[7] Sutter, H. Code complexity - part i. http://www.gotw.ca/gotw/020.htm, September 1997.

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