azon, hogy a kiemelkedő kutatók támogatása nemzetközi szinten nem korlátozódik a természettudományokra, hanem kiterjed a bölcsészet- és a társadalomtudományokra is.

Kulcsszavak: *bölcsészettudományi alapkutatás, minőség, minőségbiztosítás, tudománymetria, tudományos kiválóság, értékelés, forráselosztás, kutatásszerkezet, akadémiai intézethálózat*

IRODALOM

Beiner, Marcus (2009): *Humanities. Was Geisteswissenschaft macht. Und was sie ausmacht.* Berlin University Press, Berlin

Braun Tibor – Bujdosó E. – Ruff I. (1981): *A tudomány mint a mérés tárgya. Tudománymetriai kutatás Magyarországon. (Informatika és Tudományelemzés 1)* MTA Könyvtára, Budapest

Braun Tibor (szerk.) (2012): *Scientometrics*. Special Discussion Issue on Journal Impact Factors. 92, 2,

Braun Tibor – Bujdosó Ernő (szerk.) (1984): *A tudományos kutatás minősége*. (*Informatika és Tudományelemzés* 4) MTA Könyvtára, Budapest

Engels, Tim C. E. – Ossenblok, T. L. B. – Spruyt, E. H. J. (2012): Changing Publication Patterns in the Social Sciences and Humanities, 2000–2009. *Scientometrics*. 93, 373–390. • DOI 10.1007/s11192-012-0680-2 • http://www.ecoom.be/sites/ecoom.be/files/downloads/Engels%20et%20al%20changing%20pub%20patterns%20SSH%20Scientometrics%202012.pdf

ESF (2002): Building a European Citation Index for the Humanities. *ESF Communications*. 44 (Spring), 12–13.

Fischer, Roland A. (2006): Die Chemie stimmt – auch wenn's mal kracht / The Chemistry's Right – Even If Things Go Bang Sometimes. *Humboldt Kosmos*. Dezember, 88, 22–25. • http://www.humboldt-foundation.de/pls/web/docs/F24677/2006_kosmos88.pdf

Frühwald, Wolfgang (2008): *Die Autorität des Zweifels. Verantwortung, Messzahlen und Qualitätsurteile in der Wissenschaft*. Wallstein Verlag, Göttingen.

Glänzel, Wolfgang (2009): A tudománymetria hét mítosza – költészet és valóság. *Magyar Tudomány*. 170, 8, 954–964. • http://www.matud.iif.hu/09aug/09.htm

Kaube, Jürgen (2006): Forscher, was ist dein wert? *Frankfurter Allgemeine Sonntagszeitung*. 1, Oktober. 82,

Lack, Elisabeth – Markschies, Christoph (eds.) (2008): *What the Hell Is Quality?* Campus Verlag, Frankfurt–New York

Martin, Ben – Tang, P. – Morgan, M. – Glänzel, W. – Hornbostel, S. – Lauer, G. et al (2010): *Towards a Bibliometric Database for the Social Sciences and Humanities—A European Scoping Project: A Report Produced for DFG, ESRC, AHRC, NWO, ANR and ESF.* Science and Technology Policy Research Unit, Sussex • http://www.dfg.de/download/pdf/foerderung/grundlagen_dfg_foerderung/informationen_fachwissenschaften/geisteswissenschaften/esf_report_final_100309.pdf

Marton János – Pap Kornélia (2010): Mit tud az impaktfaktor? *Magyar Tudomány*. 171, 7, 811–815. • http://www.matud.iif.hu/2010/07/04.htm

Nederhof, Anton J. (2006): Bibliometric Monitoring of Research Performance in the Social Sciences and the Humanities: A Review. *Scientometrics*. 66, 81–100.

Pálinkás József – Csépe V. – Németh T. (2011): Kiválóság, fenntarthatóság, versenyképesség. *Magyar Tudomány*. 172, 11, 1282–1296. • http://www.matud.iif.hu/2011/11/01.htm

Palló Gábor (2012): Az akadémiai *tenure*. *Magyar Tudomány*. 173, 3, 322–333. • http://www.matud.iif.hu/2012/03/09.htm

Papp Zoltán (2011): A tudományos tevékenység értékelésének igazságosabbá tételét a saját rész elkülönítésével kell kezdeni. Esettanulmány. *Magyar Tudomány*. 172, 3, 347–353. • http://www.matud.iif.hu/2011/03/14.htm

Schollwöck, Ulrich (2006): Der Sinn des Lebens: *h* / The meaning of life: *h*. *Humboldt Kosmos*. Dezember, 88, 14–15. • http://www.humboldt-foundation.de/pls/web/docs/F24677/2006_kosmos88.pdf

Vinkler, Péter (2009): *The Evaluation of Research by Scientometric Indicators*. Chandos Publishing, Oxford–Cambridge–New Delhi

SZOFTVEREK MINŐSÉGELLENŐRZÉSE
A SZOFTVEREK IS ÖREGSZENEK?

Gyimóthy Tibor

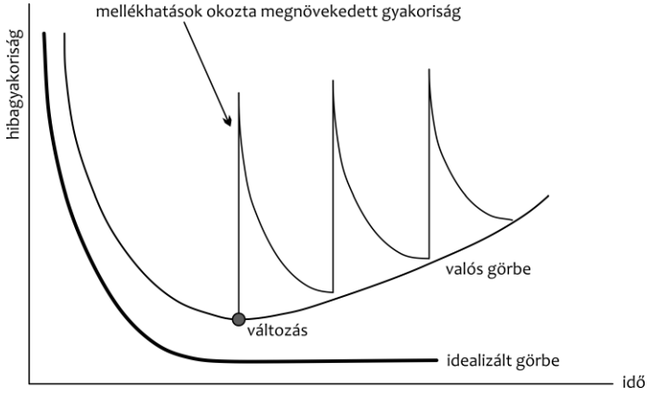
az MTA doktora, tanszékvezető egyetemi tanár,
Szegedi Tudományegyetem Szoftverfejlesztés Tanszék
gyimothy@inf.u-szeged.hu

Szoftvererózió

Az informatikai rendszerek terjedése egyre gyorsabb, szinte már nincs olyan iparág, ahol az informatikának ne lenne jelentős szerepe. Mindennapi munkánk, társadalmi életünk elképzelhetetlen olyan eszközök nélkül, amelyekben nagyon bonyolult programok működnek: gondoljunk csak például autókra, telefonokra, orvosi eszközökre vagy banki rendszerekre. Egy átlagos felhasználó számára talán nem is annyira nyilvánvaló, hogy ezen eszközök, szolgáltatások biztonságos működésének garantálása óriási erőfeszítéseket igényel az informatikai rendszerek fejlesztőitől és üzemeltetőitől. Ennek egyik oka az, hogy a rohamosan bővülő műszaki lehetőségek, valamint a folyamatosan növekvő felhasználói igények hatására a szoftverek mérete és bonyolultsága is robbanásszerűen növekszik. Egy több millió programsort tartalmazó, összetett kórházi vagy banki rendszer bonyolultsága hasonlítható az ember szervezetéhez. Ez annyit jelent, hogy az informatikai rendszerek elemei között nagyon bonyolult kapcsolatok léteznek. Ha egy helyen megváltoztatunk valamit a szoftverben, annak közvetlen vagy közvetett hatása lehet a szoftver más elemeire is.

Az emberi szervezettel való hasonlóságot az is mutatja, hogy az üzemelő szoftverrendszerek is öregszenek, bármennyire furcsán hangzik is. Úgy is szoktuk mondani, hogy fellép a *szoftvererózió* folyamata. Az öregedés fő oka, hogy az üzemelő informatikai rendszereket folyamatosan változtatni kell. Ezek a változások elkerülhetetlenek, gondoljunk csak arra, hogy új követelmények keletkeznek, változnak a szabályozások, esetleg új hardverkörnyezetbe telepítjük az adott rendszert. A gyakorlat azt mutatja, hogy ezeket a változtatásokat általában nagyon szűk határidőre kell megvalósítani, nincs idő a szoftverek átgondolt újratervezésére. Ennek az lesz a hatása, hogy néhány év üzemelés után még a kezdetben nagyon gondosan megtervezett és megvalósított szoftverek esetében is fellépnek a szoftvererózió jelei. Azt vesszük észre, hogy még kisebb méretű módosítás esetén is nagyon sok következmény-hiba jelentkezik, egyre költségesebb a módosítás utáni tesztelés. Kiszámíthatatlan lesz a rendszer fejlesztésének költsége, tipikus probléma, hogy a rendszer működése lelassul, megbízhatatlanná válik.

Az *1. ábra* szemlélteti ezt a folyamatot. Az X tengely egy szoftver fejlesztésének és üzemeltetésének idejét mutatja, az Y tengely pedig a rendszerben észlelt hibák számát (ez repre-



1. ábra

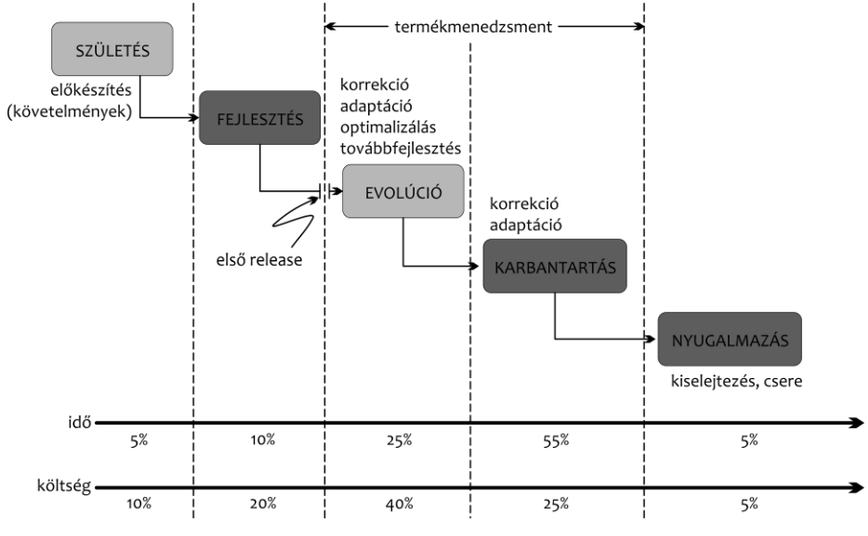
zentálja a rendszer minőségét). Azt gondolhatnánk, hogy az ábrán szereplő „idealizált görbe" jól jellemzi egy informatikai rendszer viselkedését. Vagyis a fejlesztés fázisában sok hiba lép fel, azonban ezeket egy intenzív tesztelési folyamattal sikerül kiszűrnünk, és ezután már alacsony hibaszázalékkal tudjuk üzemeltetni a rendszert. Azonban a valóságban sajnos nem ez a helyzet. Amint azt korábban említettük, az üzemelő informatikai rendszerek folyamatosan változnak. A változások miatt hibák keletkeznek, ezeket valamilyen százalékban kiszűrjük, de általában soha nincs arra idő, hogy a változások hatását végigvezessük a rendszeren. Ennek az lesz az eredménye, hogy elromlik a szoftver szerkezete, a módosításoknál egyre több következmény-hiba keletkezik, vagyis romlik a szoftver minősége. Ezt a folyamatot reprezentálja az ábrán a „valós görbe".

Ha egy kicsit mélyebben meg szeretnénk vizsgálni, hogy konkrétan milyen programozási megoldások vezetnek ehhez a minőségi romláshoz, akkor az ún. *copy-paste* programozási gyakorlatra mindenképpen fel kell hívni a figyelmet. Ennek lényege az, hogy a programozók nagyon sokszor egy adott rendszerben egy új funkció megvalósításához nem készítenek teljesen új kódot, hanem vesznek egy korábban megírt kódrészletet, azt lemásolják *(copy-paste),* és ezt alakítják át az új követelménynek megfelelően. Ez nem is okozna problémát, ha az átalakítást körültekintően hajtanák végre. Azonban a gyakorlat azt mutatja, hogy a programozók (sokszor a határidő szorítása miatt) általában csak arra figyelnek, hogy az átalakított kód megvalósítsa az új követelményt, de arra már nem ügyelnek, hogy ezt hogyan teszi. Tipikus hiba például, hogy felesleges kódrészletek maradnak az átalakított kódban, amelyek vagy végre sem hajtódnak, vagy pedig az eredményüket senki sem használja (az ilyen kódot szokás *halott* kódnak nevezni). Ezek a halott kódrészletek nagyon veszélyesek, hiszen rengeteg felesleges kapcsolatot teremtenek, megnehezítik a programok megértését, és rejtett aknaként nem várt futási hibákat vagy jelentős futási lassulásokat is tudnak okozni. A probléma komolyságát jelzi, hogy vizsgálataink szerint több jelenleg is üzemelő nagy informatikai (pl. banki, telekommunikációs) rendszerben a *copy-paste* programozással készült kódrészek aránya meghaladja a 30%-ot.

Az ilyen rendszerek üzemeltetése nagyon nehéz, új funkciókkal való bővítése egyre nagyobb költséggel jár. Ezt úgy is szoktuk mondani, hogy a rendszerek egy bizonyos idő után az evolúciós fázisból átkerülnek a karbantarthatósági fázisba.

A szoftverfejlesztés folyamata

A *2. ábrán* szemléltetjük a szoftverfejlesztési folyamat egyes fázisait. A folyamatból kiemeljük az *evolúciós fázist*. Ez a szakasz egy informatikai rendszer első változatának átadása után indul. Ebben a fázisban még kellő pontossággal becsülhető egy új funkció bevezetésének fejlesztési határideje és költsége, kordában tartható a tesztelési ráfordítás. A korábbiakban ismertetett szoftvererőzió folyamat hatására azonban egy üzemelő rendszer bizonyos idő után átkerül a *karbantarthatósági fázisba*. A szoftverfejlesztés ezen szakaszában már nehezen becsülhetők a fejlesztéshez szükséges erőforrások, és a tesztelés óriási ráfordítást igényel. Ilyenkor már nem célszerű nagyobb fejlesztést végezni, csak egyszerű módosítások, illetve hibajavítási tevékenységek folytathatók. Ez a szakasz lezárul az ún. *kivezetési (nyugalmazási) fázissal*, amikor megszületik a döntés egy teljesen új rendszer létrehozására. Meg kell jegyezni, hogy ez a döntés nagyon fájdalmas lehet egy cég életében. Alapos elemzések megmutatták, hogy egy több évtizeden át működő, nagyobb méretű informatikai rendszer cserélése rendkívül kockázatos, hiszen legtöbb esetben a vállalatban az évek alatt felgyülemlett üzleti tudást ez a rendszer tartalmazza. Ezt a tudást szinte lehetetlen teljes mértékben átültetni egy új rendszerbe. Tehát arra kell törekednünk, hogy az üzemelő informatikai rendszert minél hosszabb ideig az evolúciós fázisban tartsuk, vagyis lassítanunk kell a szoftvererózió folyamatát. A tapasztalatok azt mutatják, hogy a mindennapi életünkben üzemelő szoftverek egyetlen biztos „specifikációja"



2. ábra

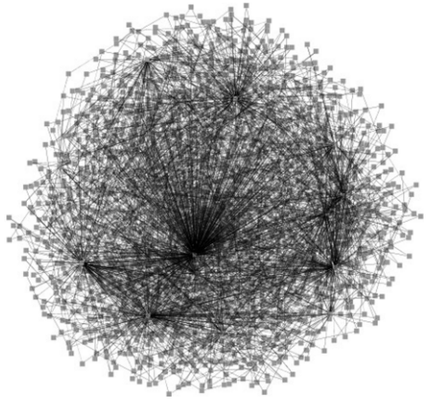
a forráskód. Ez még olyan szoftverek estében is igaz, amelyeknél a fejlesztés során készültek részletes specifikációs és tervezési dokumentumok. A folyamatos változtatások hatására ezek a dokumentumok elszakadnak a tényleges implementációtól. Általános, hogy nincs idő és elegendő erőforrás a rendszerspecifikáció és forráskód konzisztenciájának fenntartására. Ezért a forráskód minőségének javítására, illetve romlásának megakadályozására kell törekednünk.

Forráskódalapú minőségbiztosítás

A *3. ábrán* láthatjuk a Columbus forráskódalapú minőségbiztosító módszertan főbb elemeit (Ferenc et al., 2002). Mivel a minőségbiztosítás a forráskód alapján történik, ezért szükség van olyan hatékony elemző módszerek és eszközök kidolgozására, amelyek képesek a több millió soros programok gyors és pontos analízisére. Ezek a forráskódelemző eszközök elolvassák a programokat, és olyan belső reprezentációt állítanak elő, amelyből már könnyen tudunk szerkezeti és mennyiségi adatokat előállítani. Ez a belső reprezentáció lényegében egy nagyméretű, bonyolult gráf, amelyet szokás absztrakt szemantikus gráfnak (ASG) nevezni. Az ASG csomópontjai a program elemei (utasítások, eljárások, osztályok), a gráf élei pedig az elemek közötti függőségeket reprezentálják. A függőségek lehetnek szerkezeti, vezérlési, illetve adatfüggőségek. A szerkezeti függőségek a program felépítéséből származó kapcsolatokat reprezentálják (pl. osztályokon belüli eljárások). A gráf egy vezérlési éle azt mutatja meg, hogy az egyik programelem végrehajtása függ egy másik elem végrehajtási módjától. Egy adatfüggőségi él azt reprezentálja, hogy a program egyik változójának értéke függ a program másik változójának értékétől. A *4. ábrán* a Mozilla nyílt forrású webböngésző egy kis programrészletének (kb. 500 programsor) ASG-jét ábrázoltuk. A teljes Mozilla rendszer több mint egymillió programsorból áll. Már ezen a kicsi ASG-n is látszik, hogy vannak programelemek, amelyeknek nagyon sok kapcsolatuk van, így érzékenyek a változtatásokra.

3. ábra

Az ASG felépítése után a Columbus módszertan következő lépése, hogy gráf alapján numerikus értékeket állítson elő *(mérés és auditálás).* Ez a lépés nagyon hasonlít ahhoz, mint amikor egy orvosi vizsgálat során laborértékeket számítanak. Ezeket a numerikus értékeket *szoftvermetrikáknak* nevezzük. A szakirodalom több száz metrikát ismer, ezek közül legjelentősebbek a *méret, bonyolultság* illetve *csatolási metrikák* (Chidamber et al., 1994). Méretalapú metrika például, hogy egy adott rendszer hány utasítást vagy eljárást tartalmaz. Természetesen az eljárásokra külön is érdekes, hogy hány sorból állnak, hiszen egy több ezer sorból álló eljárás karbantartása nagyon nehéz, ami a szoftverminőség szempontjából magas kockázatot jelent. A bonyolultsági metrikák a program vezérlési szerkezetét vizsgálják, például azt, hogy egy eljárás esetében hány különböző végrehajtási mód lehetséges. Nyilvánvaló, ha nagy egy eljárás bonyolultsági metrikája, akkor ez problémát jelenthet. Ugyanis, ha megváltoztatjuk az eljárást, akkor nagyon sok ráfordítást igényel a különböző végrehajtási utak tesztelése. A csatolási metrikák a programelemek közötti kapcsolatok számát reprezentálják. Ilyen metrika lehet például, hogy egy eljárás hány másik eljárást hív, vagy az, hogy menynyi kapcsolata van adatfüggőségek alapján. A csatolási metrika magas értéke is komoly veszélyre figyelmeztethet. Azt mutatja, hogy az adott programelem sérülékeny, mivel a program változásai nagyobb valószínűséggel hatnak erre az elemre, mint a kisebb csatolási metrika értékkel jellemzett programelemekre.



4. ábra

A szoftvermetrikák számítása mellett az ASG-gráf alapján lehetőség van a program nagyobb szerkezeti egységeinek (komponensek) feltárására és az ezek közötti kapcsolatok azonosítására. A *visszatervezési (Reverse Engineering)* folyamat fő célja, hogy egy magasabb szintű tervezési dokumentumként megpróbáljuk előállítani az adott rendszer szoftverarchitektúráját. Az orvosi vizsgálat analógiánál maradva ez a folyamat hasonlítható ahhoz, mint amikor kétdimenziós metszetekből megpróbálunk háromdimenziós alakzatokat rekonstruálni.

A Columbus módszertan szerint, miután egy informatikai rendszerre rendelkezésre állnak a szoftvermetrikák, kiegészülve az ar-

chitektúrára vonatkozó információkkal, lehetőség van a szoftverminőségi problémák feltárására (*diagnosztika*). Az orvosi vizsgálatokhoz hasonlóan ez a folyamat legnehezebb része. A diagnózis nagyon sok tényezőtől függ, ugyanazon numerikus értékeket fejlesztési platformonként másképpen kell interpretálni, a fejlesztési módszertan is lényegesen befolyásolja a kiértékelést. Bonyolítja a helyzetet, hogy egy nagyobb méretű rendszer esetén több millió metrikus érték alapján kell döntéseket hozni. Sokat segíthet a döntési folyamatban, ha lehetőség van rá, hogy egy adott rendszer változásait hosszabb ideig megfigyeljük, vagyis több egymást követő verzióra tudunk szoftvermetrikákat számolni. Ekkor vizsgálhatjuk a metrikus értékek változásának tendenciáját, ami sokat elárul a szoftverminőség alakulásáról. A minőségi problémák azonosításában kiemelkedő szerepe van a *szoftverminőség-modelleknek*. Ezek a modellek a metrikák alapján statisztikai, illetve gépi tanulási módszerek felhasználásával próbálják megbecsülni a szoftver problémás elemeit.

A minőségi problémák azonosítása után természetesen szükség van a feltárt hiányosságok javítására *(kezelés)*. Sokszor ezek a problémák nem közvetlenül fellépő hibát jelentenek, hanem olyan hiányosságokat, amelyek nagyon megnehezítik egy rendszer üzemeltetését (például a nagyméretű, komplex eljárások megnövelik a tesztelési költségeket). Sajnos az ilyen jellegű problémák javítása nem könnyű feladat, hiszen gyakran évekkel korábban megírt forráskódrészleteket kell átalakítani. Ezt az átalakítást úgy kell végrehajtani, hogy ne sérüljön a rendszer biztonságos működése, vagyis a felhasználó számára ne jelentsen észlelhető változást. Az ilyen típusú forráskód-változtatásokat, amelyeket nem egy konkrét hiba eltávolítására, hanem a program szerkezetének javítására használunk, *refaktoring* folyamatnak nevezzük. A szakirodalmi javaslat szerint célszerű lenne a fejlesztési költségek 10%-át folyamatosan refaktoring tevékenységre fordítani. Ezzel tudnánk azt biztosítani, hogy egy adott rendszer hosszabb ideig maradjon evolúciós fázisban. Sajnos a tapasztalatok azt mutatják, hogy a szoftverfejlesztő és -üzemeltető cégek a folyamatos piaci nyomás hatására nagyon kevés erőforrást szánnak a szoftverminőség karbantartására. Ezért van szükség időnként egy nagyon intenzív „kezelésre", vagyis a forráskód alapos vizsgálata után a problémás kódrészletek javítására. A refaktoring lépés után természetesen javasolt egy újabb mérés, vagyis ellenőrizni kell, hogy a változtatások hatására javultak-e a metrikus értékek (jobbak lettek-e a laboreredmények).

Már említettük, hogy egy nagy rendszer utólagos átalakítása nagyon költséges folyamat. Mivel általában a forráskódon kívül nincs más specifikációs dokumentum, ezért a forráskód alapján kell megérteni a rendszer szerkezetét ahhoz, hogy a szükséges javításokat el tudjuk végezni. Nyilvánvalóan sokkal jobb megoldás, ha az utólagos refaktoring helyett megelőzhető lenne a szoftver minőségének romlása *(megelőzés)*. Ez megvalósítható, ha a szoftverfejlesztési folyamatba beépítjük a szoftverminőség-ellenőrző eszközöket. Vagyis amint a programozó elkészít egy új kódrészletet, azt azonnal ellenőrizzük, és addig nem folytathatja a fejlesztést, amíg a feltárt hiányosságokat ki nem javítja. Nagy szoftverrendszereket üzemeltető cégek (például bankok, telekommunikációs cégek) is gondoskodhatnak arról, hogy elkerüljék kódbázisuk minőségének gyors romlását. Mivel az ilyen cégeknél általános, hogy az informatikai rendszereik fejlesztését külső beszállítókkal valósítják meg, ezért meg kell oldani a szállítók minőségi ellenőrzését. Vagyis a beszállítói szoftverek minden változatát nemcsak funkcionálisan kell tesztelni, hanem forráskódelemzéssel, a metrikák számításával meg kell győződni az átadott szoftver minőségéről is.

A szoftverminőség mérése

A szoftverminőség ellenőrzését csak megfelelő mérések alapján lehet elvégezni, de nem mindegy, hogy mit mérünk. Azt már a szoftverfejlesztés hőskorában felismerték, hogy a tesztelés önmagában nem elegendő a szoftverminőség biztosítására, hiszen még egy viszonylag kis program esetében is általában végtelen sok tesztesetet kellene futtatnunk ahhoz, hogy a program minden lehetséges végrehajtási módját ellenőrizzük. Ezt belátva előtérbe kerültek az olyan megoldások, amelyek a szoftverfejlesztés folyamatára koncentrálva kívánják elérni a szoftverminőség javulását. Ez a szemlélet a szoftverfejlesztést egy klasszikus értelemben vett gyártási folyamatnak tekinti, és az ilyen folyamatokra alkalmazott ISO-követelmények betartását tekinti a szoftverminőség elsődleges feltételének. A későbbiek során annyiban módosult ez a szemlélet, hogy miután világossá vált, hogy a szoftverfejlesztés folyamata nagyon különbözik a szokásos gyártási folyamatoktól, ezért létrejöttek a kifejezetten *szoftverfejlesztési folyamatra kidolgozott minősítő módszerek*. Ezek közül legismertebb a Carnegie Mellon egyetemen létrehozott *Capability Maturity Model Integration* (CMMI) módszer. A szoftverfejlesztési folyamatra kidolgozott speciális minősítő módszerek már valóban pontosabban mérték/mérik a szoftverfejlesztő cégek fejlesztési folyamatait, azonban sok esetben nem adnak megbízható következtetést a cégek által előállított szoftvertermékek minőségére. A tapasztalatok azt mutatták, hogy a szoftverfejlesztő cégek megtanulták azt, hogyan lehet úgy alakítani a fejlesztési folyamataikat, hogy például egy CMMI-minősítés során elérjék a legmagasabb szintet, ugyanakkor termékeik minőségében gyakran nem tükröződik ez a magas minősítés.

Szükség van tehát a folyamatok minősítésén túl a szoftvertermékek minőségének mérésére is. A folyamatok és a termékek ellenőrzése numerikus értékek, vagyis metrikák számításán alapul. Folyamatmetrika például egy hiba javításához szükséges átlagos idő vagy egy adott időszak alatt fejlesztett programsorok száma. Termékmetrikák például a korábbiakban már említett méret-, bonyolultsági vagy csatolási metrikák.

A metrikáknak fontos szerepük van a vállalati döntésekben. Segítenek a kockázatok elemzésében, az erőforrás- és költségigény becslésében, segítségükkel regressziós modellek építhetők a szoftvertermék minőségének meghatározására.

A továbbiakban szoftvertermék-metrikákkal foglalkozunk. Ezeket a metrikákat elsősorban a forráskód alapján számítjuk, habár érdemes megjegyezni, hogy az utóbbi években születtek eredmények a forráskódhoz fűzött természetes nyelvi megjegyzések (kommentek) elemzéséből származó metrikák használatával is. Hasznosnak bizonyult például az a csatolási metrika, amely két eljárás kapcsolatát a kommentjeikben szereplő közös szavak gyakoriságával számolja (Újházi et al., 2010)

A termékmetrikákat két nagy csoportra oszthatjuk. A *belső metrikák* közvetlenül számíthatók a forráskódból, ilyenek például a korábbiakban említett méret, bonyolultság és csatolási metrikák. A másik csoportot a

külső vagy másképpen *aggregált metrikák* alkotják. Ezek a belső metrikákból származtathatók, és az a céljuk, hogy elsősorban a menedzsment számára adjanak mérhető információt az adott szoftver minőségéről. Ilyen metrika pl. a karbantarthatósági, tesztelhetőségi, megbízhatósági metrika. Megjegyezzük, hogy a külső metrikák megbízható származtatása nagyon nehéz, általában valószínűségi modelleken (Bakota et al., 2011), ill. heurisztikus megfontolásokon alapuló (Heitlager et al., 2007) megoldásokat használnak.

A belső metrikák közül hosszú ideig a *méretalapú metrikáknak* meghatározó szerepük volt. Sokaknak az volt a véleményük, hogy a szoftver mérete (utasítások száma) önmagában elegendő arra, hogy megbecsüljük karbantarthatóságát vagy a teszteléshez szükséges ráfordítást. Méretmetrikákat számíthatunk a teljes szoftverre, illetve a program kisebb egységeire is (eljárások, osztályok). A legegyszerűbb és a gyakorlatban is leginkább használt méretmetrika az utasítások száma, az ún. LOC- (lines of code) metrika. Megjegyezzük, hogy még ezt a legegyszerűbb metrikát sem egyszerű pontosan meghatározni, nagyban függ például a programozói stílustól. Ezt a metrikát régebben használták arra is, hogy mérjék a szoftverfejlesztők teljesítményét, azonban később rájöttek, hogy az ilyen típusú mérésnek komoly negatív hatása is lehet. Arra ösztönzi a programozókat, hogy minél több programsort állítsanak elő, ami pedig elősegíti a korábbiakban említett veszélyes *copy-paste* típusú programfejlesztést.

A *bonyolultsági metrikák* között kiemelt szerepű a McCabe-féle bonyolultsági mérték. Programok karbantarthatósága szempontjából fontos kritérium, hogy mennyire bonyolult a vezérlési szerkezetük. McCabe-féle bonyolultsági mértékkel azt mérjük, hogy egy eljárás esetén elméletileg hány lényegileg különböző végrehajtási útvonal van az eljárás kezdő- és végpontja között. Könnyű belátni, hogy minél nagyobb ez a szám, annál problémásabb lehet egy ilyen eljárás karbantartása. Nehezebb a megértése, és változtatás esetén sok tesztesetet kell végrehajtani, ha minden végrehajtási útvonalat legalább egyszer érinteni szeretnénk. A szakirodalom szerint nem tanácsos tíz-tizenöt McCabe-bonyolultságot meghaladó eljárásokat írni. Ugyanakkor tapasztalataink azt mutatják, hogy a napi használatban üzemelő nagy informatikai rendszerek esetében nem ritka a több százas McCabe-bonyolultsági mértékű eljárás. Bizonyos szituációkban elfogadhatók a nagy bonyolultságú eljárások (pl. generált kód), azonban az esetek túlnyomó többségében komoly szoftverminőségi problémát jeleznek.

Napjaink vezető programozási paradigmája az *objektumvezérelt programfejlesztés*. Ennek lényege, hogy az informatikai rendszerek funkcióit és a hozzájuk tartozó adatokat megpróbáljuk egymástól jól elhatárolható egységekre (objektumosztályokra) osztani. Konkrétan egy osztály adatokat és ezeken műveleteket végrehajtó metódusokat tartalmaz. A rendszer teljes működését az osztályok belső működésével és az osztályok közötti kapcsolatokkal tudjuk megadni. A *csatolási metrikák* elsősorban az objektumvezérelt programozási nyelvek osztályai közötti kapcsolatok erősségének mérésére jöttek létre. Közülük legismertebb a CBO- *(Coupling Between Objects)* metrika. Ez a metrika azt méri, hogy egy adott osztály hány további osztállyal áll kapcsolatban. A kapcsolatokat a metódushívások, illetve adattagok közös használata jelenti. Amint azt már említettük, a magas CBO-értékű osztályok a karbantarthatóság szempontjából veszélyesek lehetnek, mivel a

program változása esetén a változások következménye nagyobb valószínűséggel érinti ezeket az osztályokat.

A belső metrikákra alapozva lehetőség nyílik szoftverminőség-ellenőrző modellek kidolgozására. Elsősorban olyan modellek készültek, amelyek a metrikák alapján próbálják felderíteni a szoftver azon elemeit (például osztályokat), amelyek leginkább érzékenyek a változtatásokra, vagyis a program módosításakor várhatóan hibák jelentkeznek bennük. Számos modell készült, amelyek közül kiemelhető a Victor Basili és munkatársai által publikált úttörő munka (Basili et al., 1996), azonban ipari méretű programra a Szegedi Tudományegyetem Szoftverfejlesztés Tanszék kutatóival dolgoztunk ki először ilyen modellt (Gyimóthy et al., 2005). A több mint egymillió programsort tartalmazó, nyílt forrású Mozilla webböngésző szoftvert elemeztük, és az osztályokra metrikákat számítottunk. A Mozilla folyamatosan fejlődő rendszer, és egy adatbázisban *(Bugzilla)* évekre visszamenőleg rendelkezésre álltak az észlelt hibák adatai. Az osztályokhoz hozzárendeltük ezeket a hibákat, így minden osztályra a metrikus értékek mellett rendelkezésre állt az is, hogy a múltban hány hiba jelentkezett az adott osztályban. Ezen adatok alapján statisztikai (lineáris és logisztikus regresszió), valamint gépi tanulási (döntési fa, neuronháló) módszerekkel modelleket készítettünk. Konkrétan azt vizsgáltuk, hogy melyik metrikák a legalkalmasabbak a hibára hajlamos osztályok azonosítására. Tapasztalataink azt mutatták, hogy a statisztikai és a gépi tanulási modellek esetén is a CBO-csatolási metrika alapján lehetett legpontosabban megbecsülni a hibára hajlamos osztályokat. Vagyis a méret és bonyolultság metrikáknál is kritikusabb arra figyelni, hogy csak az igazán szükséges függőségeket építsük ki a szoftverelemek között. Másképpen fogalmazva az olyan szoftverek, melyek „kapcsolati hálója” nagyon sűrű, nagyon sérülékenyek, változtatások esetén sok következmény-hiba jelentkezhet bennük. Ez az eredmény az igen költséges tesztelési folyamat optimalizálását is segítheti. Konkrétan: egy változtatás utáni teszteléskor érdemes nagyobb figyelmet fordítani a magas CBO-értékű osztályok tesztelésére.

Tapasztalataink alapján még számos kérdést kell megválaszolni ahhoz, hogy az ilyen metrikus modellek a mindennapi szoftverfejlesztésben és -üzemeltetésben rutinszerűen alkalmazhatók legyenek. Egyik legfontosabb megoldandó probléma a találati pontosság növelése, a hamis riasztások csökkentése. Ez annyit jelent, hogy ha a modell azonosít egy minőségi problémát, akkor az valódi probléma legyen. Nem várható, hogy olyan általános modellek készülnek, amelyek minden szoftverfejlesztési területre alkalmazhatók lesznek. A szakterület egyedi sajátosságait és a fejlesztési platformok sajátosságait egyaránt figyelembe vevő, metrikaalapú szoftverminőség-modellek elterjedésére azonban reális esély mutatkozik.

Kulcsszavak: *szoftverminőség, szoftverevolúció, metrikák, forráskódalapú szoftverminőség-modellek*

IRODALOM

Bakota Tibor – Hegedűs P. – Körtvélyesi P. – Ferenc R. – Gyimóthy T. (2011): A Probabilistic Software Quality Model. *Proceedings of the 27th IEEE International Conference on Software Maintenance.* 243–252. DOI:10.1109/ICSM.2011.6080791

Basili, Victor R. – Briand, L. C. – Melo, W. L. (1996): A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering.* 22, 10, 751–761. • http://doi.ieeecomputersociety.org/10.1109/32.544352

Chidamber, Shyam R. – Kemerer, Chris F. (1994): A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering.* 20, 6, 476-493. • http://faculty.salisbury.edu/~stlauterburg/COSC425/MetricForOOD_ChidamberKemerer94.pdf

Ferenc Rudolf – Beszédes Á. – Tarkiainen, M. – Gyimóthy T. (2002): Columbus – Reverse Engineering Tool and Schema for C++. *Proc. of the IEEE International Conference on Software Maintenance.* 172–181, *IEEE Computer Society,* Most Influential Paper Award. • http://www.inf.u-szeged.hu/~beszedes/research/tech27_ferenc_r.pdf

Gyimóthy Tibor – Ferenc R. – Siket I. (2005): Empirical Validation of Object-oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering.* 31, 10, 897–910. DOI:10.1109/TSE.2005.112

Heitlager, Ilja – Kuipers, T. – Visser, J. (2007): *A Practical Model for Measuring Maintainability. Proceedings of the 6th International Conference on the Quality of Information and Communications Technology. 30–39.* • http://portal.ou.nl/documents/informatica/snapshots/T66311_02.pdf

Újházi Béla – Ferenc R. – Poshyvanyk, D. – Gyimóthy T. (2010): New Conceptual Coupling and Cohesion Metrics for Object-Oriented Systems. *Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation.* Best paper award. DOI:10.1109/SCAM.2010.14

AZ ÚJ POLGÁRI TÖRVÉNYKÖNYV ÉS A KÖRNYEZETVÉDELEM

Julesz Máté

PhD, tudományos kutató,
Szegedi Tudományegyetem
mate.julesz@freemail.hu

Szomszédok az új Ptk.-ban

Az új Ptk. 5:23. §-a kimondja, hogy a tulajdonos a dolog használata során köteles tartózkodni minden olyan magatartástól, amellyel másokat – különösen a szomszédokat – szükségtelenül zavarná, vagy amellyel jogaik gyakorlását veszélyeztetné. Ehhez a polgári jogi diszpozícióhoz továbbra is a birtokvédelmi és a kártérítési szankciórendszer járul.

A két szankciórendszer közül megválasztható, hogy melyik vezetne inkább eredményre. A diszpozícióhoz így szankció is járul. Ezért környezetvédelmi magánjogi *lex perfectá*-ról beszélhetünk. A szabályozás többé-kevésbé egyezik a korábbival. Ekként a birtokvédelmi szankciórendszer vagy a kártérítési szankciórendszer felhívható. A károkozónak azonban előre kellett vagy kellett volna látnia a kárt. E nélkül hiányzik a kauzalitás. Az ok-okozati összefüggés ilyetén hiányára az új Ptk. gyakorlatában sokszor fognak hivatkozni a felek. Évek vagy évtizedek kellenek majd a kazuisztika kialakulásához. A jogegységi határozatok, illetve a joggyakorlat-elemzés végzi majd el a kazuisztika elsődleges szintű absztrakcióját.

A Kúria által végzett absztrakció már az elvi bírósági határozatok és az elvi bírósági döntések közzétételi aktusával megindul. A joggyakorlat-elemző csoport összefoglaló véleménye alapján a Kúria kollégiumvezetője jogegységi eljárást indítványozhat, vagy jogalkotás kezdeményezése érdekében a Kúria elnökén keresztül az Országos Bírósági Hivatal elnökéhez fordulhat. Az elsődleges szintű, a bíróság által elvégzett absztrakció után a jogtudomány és a jogalkotás is el fogja végezni a tudományos, illetve kodifikációs szintű, tehát a másodlagos absztrakciót.

A környezetvédelmi magánjogi felelősség alaptényállása, valamint a hozzá tartozó birtokvédelmi, továbbá kártérítési szankciórendszer kiállta a joghasználat próbáját. Némiképp új alapokra helyezve, de az eddigiek szerint folytatódhat az új magyar polgári törvénykönyv szabályrendszere szerint is. A környezetvédelmi magánjogi felelősség gyakorlati eredményeinek hasznosítása talán csak ott marad el, hogy a kodifikáció során a bírósági gyakorlatban kiforrott *lex perfecta* nem kap önálló fejezetet a polgári jogi kódexen belül. Akár a dologi jogi könyvben, akár a kötelmi jogi könyvben legalább egy §-ban érdemes volna kitérni a környezetvédelmi magánjogi diszpozíció és szankció rendszerére. Mindezt azért, hogy ne a bírósági gyakorlatból lehessen