Binzberger Viktor

A szoftverhiba jelensége hermeneutikai megközelítésből[1]

KIVONAT

Egy esettanulmányt fogok bemutatni, amellyel a szoftver és a szoftverhiba értelmezésének hermeneutikai modelljét kívánom illusztrálni. A modell felépítésében a jelentésesség heideggeri és gadameri elméletére támaszkodom, és kiépítek egy analógiát a megértés két köre: a hermeneutikai kör és az iteratív szoftverfejlesztési ciklusok között.

Közvetett célom, hogy az elmével kapcsolatban általánosan elterjedt komputácionalista nézet előfeltevéseként azonosított szoftver-redukcionizmus (egyfajta esszencializmus) kritikáját adjam, és rámutassak: amennyiben emberi megértésre törekszik, a komputácionalista kutatónak is el kell foglalnia a hermeneutikai perspektívát.

BEVEZETŐ

Egy olyan példát szeretnék bemutatni, amelyik a jelentés, a reprezentáció és a megértés problémájának metszetében fekszik. A megértés két körét szeretném ezzel illusztrálni, és amellett szeretnék érvelni, hogy a kettő közötti kapcsolat több mint metafora. A példa a szoftverhibáról szól. A megértés szóban forgó körei közül pedig az első a hermeneutikai kör.

Mi állhat egymástól távolabb, mint a számítógépes szimbólumok manipulációja és a hermeneutika? A hermeneutika az emberi megértésről szóló szellemtudomány. Ezzel éles kontrasztban áll Alan Turing modellje, mely szerint a számítógépek *absztrakt szimbólum-feldolgozó gépek*, melyeket a szimbólumstruktúrák és a szimbólummanipulációs szabályok jellemeznek. A Church–Turing tézis szerint minden „effektív módszerrel kiszámítható függvény" kiszámítható Turing-géppel is. Ez a gondolat, az „effektív módszerrel kiszámítható függvények" esetéről az emberi gondolkodás egészére általánosítva, a komputácionalizmus téziseként vonult be napjaink elmefilozófiájába. Eme tézis szerint az emberi elme voltaképpen ekvivalens egy Turing-géppel. Ez annyit tesz, hogy az elme megértéséhez az elmét mint számítógépet, mint szoftvert kell megértenünk: fel kell tárnunk a benne található szimbolikus struktúrákat és az ezek manipulá-

[1] *Köszönettel tartozom az MTA–BME Tudományfilozófia és Tudománytörténet Kutatócsoportnak, és az OTKA-nak (F046354) a támogatásért, amelyet ennek a tanulmánynak az elkészülése során nyújtottak.*

ciójára szolgáló szabályokat. Nem kell bizonygatnom, hogy Fodort és másokat követően a kognitív tudomány egyes területein sokan osztják ezt a nézetet.

Ha azonban az elmét mint szoftvert kell megértenünk, először is vessünk egy pillantást arra, *hogyan is értjük meg a szoftvert?*

Nézzünk egy példát. Mi kell ahhoz, hogy megértsük, miért következett be az alábbi, 1.87-es revíziós számú változtatás a GNU Standard C könyvtár legközönségesebb függvényében, a vfprintf.c fájlban:

```
/* Search for the end of the string, but don't search past
   the length specified by the precision.  */

len = __strnlen (string, prec);
```

le lett cserélve az alábbi kódra

```
/* Search for the end of the string, but don't search past
   the length (in bytes) specified by the precision.  Also
   don't use incomplete characters.  */

if (__NL_CURRENT_WORD (LC_CTYPE, __NL_CTYPE_MB_CUR_MAX) == 1)
    len = __strnlen (string, prec);
else
{
    /* In case we have a multibyte character set the
       situation is more complicated.  We must not copy
       bytes at the end which form an incomplete character. */
    wchar_t ignore[prec];
    const char *str2 = string;
    mbstate_t ps;

    memset (&ps, '\0', sizeof (ps));
    if (__mbsnrtowcs (ignore, &str2, prec, prec, &ps)
        == (size_t) -1)
    {
        done = -1;
        goto all_done;
    }
    if (str2 == NULL)
        len = strlen (string);
    else
        len = str2 - string - (ps.__count);
}
```

(A kódrészlet forrása http://sources.redhat.com/cgi-bin/cvsweb.cgi/libc/stdio-common/vfprintf.c)

A tézis, amelyet ennek a problémának a megvilágításával szeretnék alátámasztani, az, hogy a szoftver megértéséhez másra van szükség, mint a szimbólumstruktúrák és a szimbólummanipulációk ismerete. Szeretném megmutatni továbbá, hogy a jelentésesség heideggeri elmélete mennyiben illeszkedik a programozók meglévő megértési gyakorlatához.

A PROBLÉMA

Vessünk egy pillantást a példánkra.

```
/* Search for the end of the string, but don't search past
   the length specified by the precision.  */

   len = _ _ strnlen (string, prec);
```

Mi is számítana a fenti szövegrészlet megértésének? A megközelítésnek legalább két módja lehetséges. A felső sor, a kommentár közelebb áll a jelentésességek hétköznapi rendszeréhez, hiszen olyan fogalmakat említ, mint a „pontosság", a „hossz" és a „keresés", amelyek a hétköznapi tevékenységeink kontextusába tartoznak. A programozási nyelvek tanítása során a tankönyvek általában hétköznapi tapasztalatokon nyugvó metaforákba ágyazzák be a szimbolikus struktúrákat: a fenti szavak jelentésüket a memória doboz-metaforájának kontextusában nyerik el.

Az alsó sor, a C programnyelvű forráskód ezzel szemben abba a formálisan artikulált jelentés-összefüggésbe ágyazódik be, amelyet GNU Standard C függvénykönyvtárnak nevezünk. Az utóbbi egy kompozicionális, formális rendszer, szemben az előbbivel. Amikor azt mondjuk, hogy ez a program egy karakterlánc hosszát határozza meg azáltal, hogy megkeresi a „karakterlánc vége jelet", akkor az előbbi, hétköznapi jelentés-összefüggésben mozgunk. Hasonló leíráshoz az utóbbi, formális jelentés-összefüggésben az „__strnlen" függvény szimbolikus definícióját kellene felidéznünk.

Az egyik megközelítés szerint tehát a programot a szimbólumstruktúrák konstituálják. A másik megközelítés szerint azonban az az érdekes, ahogyan a szimbólumstruktúrák beágyazódnak a hétköznapi jelentés-összefüggésekbe, mert csak ennek fényében tudunk olyan leírásokat adni, mint hogy „a vfprintf függvény feladata karakterláncok képernyőre írása".

Az ehhez hasonló leírásokat funkcionális leírásoknak szokták nevezni. Az első megközelítés értelmében a programozó feladata nem más, mint ezen funkcionális leírások programkódra „redukálása", és a redukció kivitelezése, az „implementálás" befejezése után a funkcionális leírások feleslegessé válnak. Nevezzük ezt az álláspontot a továbbiakban szoftver-redukcionizmusnak – ez a szimbólumokkal kapcsolatban egyfajta esszencializmust takar. Figyeljük meg, hogy a komputácionalizmus ezt a megközelítést előfeltételezi, hiszen magát az *interpretálatlan* szimbólumrendszert tekinti az elmével ekvivalensnek!

Ha azonban a funkcionális leírások ilyenformán nélkülözhetőek volnának, miért használják ezeket a programozók lépten-nyomon? Miért van az, hogy a program jelentős részét általában a kommentár és a dokumentáció teszi ki? Feleslegesek-e az informális funkcionális leírások? A programozók gyakorlata arra látszik utalni, hogy nem. Hogyan tudnánk ezt a jelenséget közelebbről megérteni? A jelenség értelmezéséhez induljunk ki Heidegger jelentés- és rendeltetésfogalmából!

HEIDEGGER ELEMZÉSE A JELENTÉSESSÉGRŐL

Heidegger elemzése a jelentésességről határozottan nem egy „jelentéselmélet" abban az értelemben, ahogyan az analitikus tradícióban ezt a kifejezést használni szokták. Ez ugyanis az elemzés alapszintjének nem szimbólumok referenciális és kompozicionalitási viszonyait teszi meg, hanem a számunkra feltáruló fenomenális világ struktúráját és strukturálódási folyamatait.

Heidegger szerint az olyan eszközök, mint a számítógépes programok, annyiban érthetők és ragadhatók meg számunkra, amennyiben hozzánk viszonyított rendeltetéssel (Bewandtnis) rendelkeznek. Egy programrészletről alkotott megértés azokban a használatokban és célszerűségekben, a „mirevalóságokban" (Wozu) testesül meg, amelyet a különféle használati kontextusokban betölt. Ez a megértés eredendően nem fogalmilag artikulált, hanem gyakorlati: a szoftver megértésének elsődleges módja a szoftver használata, és csak bizonyos speciális esetekben kezdjük el a szoftvert fogalmi elemzés, avagy teoretikus reflexió tárgyává tenni. Akkor tesszük ezt például, ha a gyakorlati megértés, azaz a használat kudarcot vall.

A rendeltetés nem egyedi tárgyak szemantikai tulajdonsága. Egyrészt nem tulajdonság, mert nem a tárgyhoz, és különösen nem a tárgy fogalmához tartozik, hanem a tárggyal folytatott tevékenységünkben testesül meg. Nem léteznek „funkciók" használatmódok nélkül, nem léteznek rendeltetések anélkül, hogy ezek valakihez kapcsolódnának. Másrészt pedig a rendeltetés nem egyedi tárgyak sajátja. A vfprintf függvény magyarázatakor „karakterláncokra", „képernyőre", „memóriára" hivatkoztunk, és ezzel előfeltételeztük a hallgató részéről a számítástechnika ama összefüggésének ismeretét, amelybe a függvény beágyazódik. Ezt az összefüggést Heidegger rendeltetésegésznek (Bewandtnisganzheit) nevezi. Ha a dolgok rendeltetésegészben betöltött szerepét nyelvileg artikuláljuk, akkor egy „funkcionális leírást" kapunk, ez azonban nem önálló, hanem a gyakorlat és a hétköznapi nyelv hátterére támaszkodik.

A dolgok megértése Heideggernél tehát eredendően egy rendeltetésegész összefüggésébe való gyakorlati bevonásukat jelenti. Ahhoz, hogy ezt megtehessük, a világot a használatmódok és céljaink szerint kell felosztanunk: ki kell „vetítenünk" a világra a megközelítésünket. Ezzel egy teret konstruálunk, amelyben funkciókkal rendelkező tárgyakkal találkozhatunk: ezt nevezi Heidegger a jelentésesség struktúrájának.

A „funkciók" tehát nem pusztán a szoftverben keresendőek, hanem egy milliónyi számítógépes szakértő által kivetített, társasan megosztott, gyakorlatként megtestesült rendeltetési térben. Az, hogy ezek az emberek megértik egymást, annak köszönhető, hogy osztoznak egymás céljaiban, és részt vesznek a közös tevékenységekben, amelyek ezt a rendeltetési teret fenntartják.

Amikor a programozó a szoftvert a hétköznapi jelentésességre támaszkodó metaforák révén, funkcionális leírások által érti meg, akkor egy rendeltetésegészben helyezi

el a szoftvert. Megértése szempontjából ez az elsődleges, és csupán ebbe a megértésbe ágyazódik bele a Standard C függvénykönyvtár formális-kompozicionális szimbólumrendszere.

A programozó megértése, noha szimbólumok manipulációjával kapcsolatos, nagyrészt mégis gyakorlati, nem-szimbolikus. Egyes programozási metodológiák – mint például az eXtreme Programming – igen nagy hangsúlyt fektetnek erre, és a párokban történő, intenzív, közös programozás, az ügyféllel való szoros együttműködés, a jó „skillek" elsődlegessége mellett érvelnek, a dokumentálás és elemző tervezés eluralkodásával szemben.

HIBAKERESÉS

Itt az ideje, hogy visszatérjünk eredeti kérdésünkhöz: miért következett be az 1.87-es revízió?

A naiv programozó számára a válasz egyszerű: a program nyilván nem úgy működött, ahogyan várták. A revízióhoz tartozó kommentár szerint „a %-os formátumú, többbájtos karakterláncok [...] kiírása javításra került". Tehát nyilván gond volt a több-bájtos karakterláncok kiírásával.

Ha a szoftver redukcionista programját szó szerint vesszük, akkor számára ez a kérdés voltaképpen nem értelmezhető. Számára a vfprintf függvény nem több és nem kevesebb, mint jelen pillanatbeli formális definíciója. Nincsenek „jó" és „rossz" definíciók, és nincsenek hibák sem, hiszen az elvárásoknak nincsen helye az ontológiájában.

Ahhoz, hogy hibáról tudjunk beszélni, kell, hogy legyenek a definíciótól független elvárásaink, ezek pedig abból fakadnak, hogy a szoftvert funkciója szerint értjük meg. A heideggeriánus megközelítés értelmében a jó szoftver kézhez álló (Zuhandenes): beleilleszkedik a mindennapok gyakorlatának rendeltetésegészébe, eszközként szolgál tennivalóink elvégzéséhez.

A hiba nem más, mint ennek a kézhezállóságnak a megszűnése. A kivetített rendeltetésegész által konstituált megértésünk elégtelenné válik a szoftver használatához. Az a „funkció", amelynek fényében a szoftvert korábban megértettük, betöltetlen marad, és ahogyan a funkció nem a szoftverben keresendő, hanem a társasan megosztott rendeltetési térben, ugyanígy a programhiba sem a szoftverben van.

A hiba egyrészt jelentheti a megértés kudarcát a felhasználó részéről, amennyiben az ő hibája, ha tévesen úgy gondolja, hogy a programnak bizonyos módon kellene működnie. De a hiba másrészt jelentheti a megértés kudarcát a programozó részéről, amennyiben az ő hibája, ha tévesen úgy gondolja, hogy a programot *nem* kell felkészítenie bizonyos működésmódokra.

A hiba mindkét esetben a *kölcsönös megértés* kudarca. A programozó annyiban képes „jó" szoftvert írni, amennyiben meg tudja érteni a program szerepét a felhasználó rendeltetésegészében. Ehhez meg kell értenie a felhasználó céljait, tennivalóit, adminisztratív gyakorlatát, el kell gondolnia jövőbeli interakcióit a szoftverrel. Emellett azonban a felhasználónak is meg kell értenie azt a rendeltetésegészt, amelyikben a program a programozó számára megjelenik. Ezt a kölcsönös megértést azáltal képesek megtenni, hogy osztoznak egymás céljaiban, és részt vesznek a közös tevékenységben, amely a rendeltetési teret kifeszíti. *A programozó és a felhasználó a szoftver megalkotása és használata során megpróbálják összeolvasztani megértésük horizontját.*

Első pillantásra úgy tűnhet, mintha ez a definíció kizárná a hibák köréből azokat a szituációkat, amikor maga a programozó azonosítja a hibát. Ennek a szituációnak három alesetét különböztetem meg, amelyből kettőt visszavezetek a fenti szituációra. Először is a programozó egy adott szoftverrészlet megírásakor rengeteg más szoftverhez felhasználóként viszonyul. Amikor a saját szoftverében hibát tapasztal, az gyakran az ő és egy másik programozó közötti megértés sikertelenségéből fakad. Másrészt a programozó felhasználója saját, korábban írt programjainak is. Némi idő elteltével az a megértés, amely a szoftver megírásakor eredetileg fennállt, feledésbe merülhet. Jellegzetes például, hogy a programozó megértése fokozatosan követi a társas rendeltetési tér változásait, de a program eközben változatlan marad: észrevétlenül megváltozik a rendeltetése. A hiba tehát fakadhat az ő saját korábbi és későbbi megértésének konfliktusából is – ennek elkerülésére szolgálhat a kommentár és a dokumentáció. Harmadrészt pedig a hiba lehet szándékos – ez a hermeneutikai jóindulat elvének megsértésével lenne analóg, de ennek az esetnek az elemzése kivezetne jelen tanulmány keretei közül.

A szoftver csak az egyik médium, amelyen át a felhasználó és a programozó érintkezésbe lépnek. A szoftverhiba jelenségének vizsgálata elválaszthatatlan a gyakorlati, társas közegtől, az intézményi hierarchiáktól, a kommunikációs formáktól, oktatástól, hétköznapi nyelvbe ágyazott metaforáktól és a közös kognitív felépítéstől, amelyek a programozó és a felhasználó kölcsönös megértésének hátterét alkotják. A társadalmi létnek ez az állványzata kézenfekvő módon adott anélkül, hogy a benne résztvevők teoretikus reflexió tárgyává tennék. A hermeneutikai megközelítés célja ezen kölcsönös megértés lehetőségét megteremtő, vagy a megértés útját éppenséggel eltorlaszoló háttér tudatosítása.

Miért érdemes ezt a megközelítésmódot választani a szoftverhiba jelenségének leírásához? Két fontos érvem szól emellett.

Az első érvem az, hogy a szoftverfejlesztéssel foglalkozók körében a megértés eme problémái különösen élesen exponálódnak és aktív reflexió tárgyát képezik, noha ők nem nevezik a problémát „hermeneutikainak". A mai szoftverfejlesztés-módszertanok egyik fontos célja – a hermeneutikai megközelítéshez hasonlóan – épp a megértés hátterének tudatosítása és átformálása annak érdekében, hogy elkerülhetőek legyenek a megértésbeli kudarcok.

Ezen módszertanok fontos témája a programozók, tervezők és leendő felhasználók közötti kommunikáció a fejlesztés során. Valahol a spektrum egyik szélén áll a formalizált diagramokat, standardizált eljárásokat alkalmazó Unified Modeling Language és a hozzá kapcsolódó Rational Unified Process, míg a másikon a közvetlen interperszonális kommunikáció jelentőségére, az innovatív, improvizatív zsenialitásra hangsúlyt fektető, „anarchisztikus" eXtreme Programming-ot találjuk. Szintén fontos téma a szoftver dokumentációja, a „funkciók" terének artikulált megosztása a jövőbeli felhasználóval – ennek elmaradása ugyanis a szoftvert gyakran használhatatlanná, karbantarthatatlanná és továbbfejleszthetetlenné teszi.

A hermeneutika a megértés körmozgását hangsúlyozza, míg a kortárs szoftverfejlesztési módszertanok a fejlesztést iteratív, ciklikus folyamatnak tekintik. Gadamer számára a szöveg megértése a szöveg eredeti világának a jelen felől való megközelítése, amelyet a jelentés-összefüggés egészlegessége miatt nem lehet egyetlen lépésben végrehajtani, csak visszalépések és ismételt olvasások során. A szoftver esetében a program megalkotása a felhasználó világának a programozó világa felől való megkö-

zelítése, amelyet a rendeltetésegész egészlegessége miatt hasonlóképpen nem lehet egyetlen lineáris folyamatban végrehajtani (mint azt valaha Barry Boehm klasszikus „vízesés-modellje" javasolta), hanem csak ismételt konzultációk, hibajavítások és újraprogramozás során. Ezek alapján gondolom, hogy a hermeneutikai kör és a fejlesztési ciklus a megértés egymással párhuzamba állítható körei.

A második érvem szerint pedig ez a megközelítés rámutat arra, hogy az általam szoftver-redukcionistának nevezett megközelítés, amely pusztán a szoftver szimbolikus struktúráit tartja szem előtt, nem rendelkezik fogalmakkal a szoftver rendeltetésének, a szimbolikus struktúrák emberi megértésbe való beágyazódásának elemzésére.

Ennek illusztrálása végett válaszoljuk meg kiinduló kérdésünket! A több-bájtos karakterláncok kiírásának igénye a megértés hátterében lezajló változások során jött létre: a globális piacon ugyanis szükségessé vált a csak több bájton reprezentálható, ékezetes és kínai karakterek képernyőre írása. Pusztán a szimbólumstruktúrákat szemlélve soha nem bukkanhatnánk rá erre a magyarázatra, hiszen ott nyoma sincs a „globális piac"-nak. Ezt és az összes ilyen jellegű magyarázatot az teszi lehetővé, hogy a program szimbólumai bele vannak szőve abba a társasan megosztott rendeltetési térbe, amelynek, kedves Olvasó, Ön is részese.

KONKLÚZIÓ

Általánosítva a tanulságokat: az érvelés azon fordul meg, hogy a szoftvernek rendeltetése van, ez a rendeltetés meghatározó szereppel bír a formális struktúrákra nézve, és a rendeltetés változását indukáló erők nem lelhetők fel magában a szoftverben. Ha pusztán a szoftver szimbolikus struktúráit tartjuk szem előtt, eltekintve attól a háttértől, amelyben a programozó és a felhasználó számára ezek a szimbólumok jelentésessé válnak, a megértésünk róla csak részleges lesz, csupán a szoftver jelenlegi állapotát, egy statikus pillanatképet ragad meg.

Ha pedig visszaidézzük a komputácionalista tézisünket, mely szerint az emberi elmét mint szoftvert kell megértenünk, láthatjuk, hogy az elméről való megértés sem teljes, ha az embert pusztán mint izolált szimbólum-feldolgozó gépet tekintjük, azaz nem vesszük figyelembe azt a társadalmi-biológiai értelem-összefüggést, amelynek kontextusában az elmébe lokalizált „szimbólumok" egyrészt a kísérleti alany, másrészt pedig az elme kutatója számára jelentésessé válnak. Ekkor nem tudunk számot adni arról sem, hogy az elme magyarázatára alkotott funkcionális leírások milyen módon kerülnek megértésre, és milyen erők indukálják a változásaikat.

A rendeltetések és jelentésességek eme egészleges összefüggésének dinamikája tehát az a probléma, amelynek megragadásában a hermeneutika és a szoftverfejlesztési módszertanok célja közös. (Itt meg kell említenem Dewey nevét, aki némiképp hasonló utat követ a pszichológia evolúciós-biológiai kontextusba ágyazását illetően, de míg a hermeneutika a fenomenológiai, Dewey a naturalista perspektívát képviseli (Dewey 2004, 209–217). A „rendeltetés" és a „megértés" természetesen jellegzetesen antropocentrikus, szellemtudományi fogalmak – de ez a perspektíva a szoftverek értelmezése esetén épp olyan fontos, mint a szövegek és az életvilág értelmezésének klasszikus problémája esetében. Ez teszi a hermeneutikai kör és az iteratív fejlesztési ciklus között vont felszínes párhuzamot komollyá: ettől tekinthetjük mindkettőt a megértés körének.

IRODALOM

CAPURRO, Rafael 1992. Informatics and Hermeneutics. In Christiane Floyd–Heinz Züllighoven–Reinhard Budde–Reinhard Keil-Slawik (eds): *Software Development and Reality Construction*. Berlin–Heidelberg–New York: Springer-Verlag. pp. 363–375.

DEWEY, John 2004. A reflexív fogalma. In Pléh Csaba–Győri Miklós (szerk.): *Olvasmányok a kísérleti pszichológia történetéhez*. Budapest: Osiris. pp. 209–217.

DREYFUS, Hubert 1991. *Being-in-the-World*. Massachusetts Institute of Technology.

HEELAN, Patrick A.–SCHULKIN, Jay 2003. Hermeneutical Philosophy and Pragmatism: A Philosophy of Science. In Robert C. Scharff–Val Dusek (szerk.): *Philosophy of Technology: An Anthology*. Blackwell Publishing. pp. 138–154.

HEIDEGGER, Martin 2004. *Lét és idő*. Budapest: Osiris.

KISIEL, Theodore J. 2001. Heidegger és az új tudománykép. In Schwendtner Tibor–Ropolyi László–Kiss Olga (szerk.): *Hermeneutika és a természettudományok*. Budapest: Áron Kiadó.

KOCHAN, Jeff 2005. *A Poetics of Tool-use – Explorations in Heidegger and Science Studies*. Doktori disszertáció. Cambridge: Churchill College.

MARGITAY Tihamér 2003. A tapasztalat két fogalma. Empirizmus vs. hermeneutikai fenomenológia Quine és Heidegger nyomán. In *Tudomány megértő módban – Hermeneutika és tudományfilozófia*. Budapest: Harmattan Kiadó.

MÁRKUS György 2001. Miért nincs hermeneutikája a természettudománynak? In Schwendtner Tibor–Ropolyi László–Kiss Olga (szerk.): *Hermeneutika és a természettudományok*. Budapest: Áron Kiadó.

ROUSE, Joseph 1999. Understanding Scientific Practices: Cultural Studies of Science as a Philosophical Program. In *The Science Studies Reader*. New York–London: Routledge. pp. 95–109.

SHAPIRO, Gary–SICA, Alan 1984. *Hermeneutics: Questions and Prospects*. Amherst: University of Massachusetts Press.

WINOGRAD, Terry–FLORES, Fernando 1987. *Understanding Computers and Cognition*. Norwood, NJ: Ablex Corporation.