A programozás gondolkodási eszköztára – Algoritmikus absztrakció, dekompozíció-szuperpozíció

Szlávi Péter[1], Zsakó László[2]

{[1]szlavip, [2]zsako}@caesar.elte.hu
ELTE IK

Absztrakt. Miközben a programozó a feladat megoldásán fáradozik, tudatosan vagy sem, számos **gondolkodási művelet**et használ. Ahogy a feladatból kiindul, többszörösen szelídíti azt a saját, meglévő, tapasztalatain alapuló sémáihoz, körkörösen haladva egyre pontosabban fogalmazza újra a feladatot. A programozó legfontosabb gondolkodási műveletei a finomodó modellek felállításánál: a nyelvi absztrakció, az analógia, az algoritmikus absztrakció, a dekompozíció-szuperpozíció, a konverzió, az intuíció és a variáció.

Előadásban az **algoritmikus absztrakció**t helyezzük új megvilágításba, továbbá kifejtjük a **dekompozíció-szuperpozíció** gondolkodási eszközpár lényegét.

Kulcsszavak: programozás, didaktika, szisztematikus gondolkodás, gondolkodási eszközök, algoritmikus absztrakció, dekompozíció, szuperpozíció

1. A programozás gondolkodási eszköztára

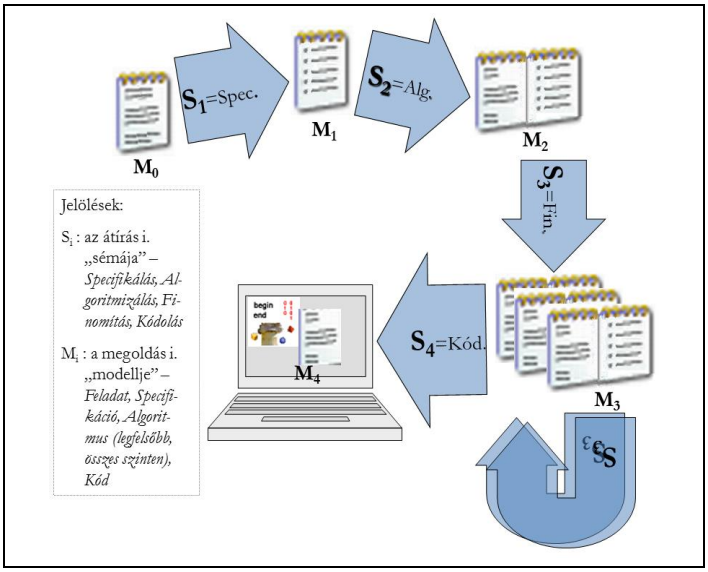
Miközben a programozó a feladat megoldásán fáradozik, tudatosan vagy sem, számos gondolkodási műveletet használ. Ahogy a feladatból kiindul, többszörösen szelídíti azt a saját, meglévő, tapasztalatain alapuló sémáihoz, körkörösen haladva egyre pontosabban fogalmazza újra a feladatot. Úgy is mondhatjuk, hogy a programozás a feladat egyre finomabb (sémákon alapuló) modelljeinek sorozata, amiben addig kell eljutni, ahol már a választott programozási nyelv szókincse (utasítás-sémáinak halmaza) jelenti a modell alapját. (1. ábra.)

A programozó legfontosabb gondolkodási műveletei a finomodó modellek felállításánál az alábbiak:

- a nyelvi absztrakció [1],
- az analógia [1],
- az algoritmikus absztrakció,
- a dekompozíció-szuperpozíció,
- a konverzió,
- az intuíció,
- a variáció…

Az egyes eszközöket sokszor, sokféle céllal veti be a programozó. Ez indokolja, hogy kénytelenek leszünk ugyanazt az eszközt többször is szóba hozni.

Előadásunkban a jól ismert **algoritmikus absztrakció**t helyezzük új megvilágításba, továbbá kifejtjük a **dekompozíció-szuperpozíció** gondolkodási eszközpár lényegét. Anyagunkkal szorosan kapcsolódunk az [1]-ben elkezdett gondolatsorhoz.



1. ábra: A programozás absztrakt modellje.

2. Algoritmikus absztrakció

A programozás során sokszor, sokféle fokon és szintéren használja a programozó az **általánosítás** gondolkodási eszközt. Ennek egyike az **analógiákra épülő gondolkozás**, amelyet [1,2]-ben részletesen kibontottunk a feladatsémákkal kapcsolatban. Kiderült, hogy az analógiás gondolkodást hatékonyabbá lehet tenni bizonyos, jól megválasztott sémák kidolgozásával és oktatásbéli bevezetésével. Mivel a sémák absztrahálás útján jönnek létre, ebből a szempontból is meg kell vizsgálnunk!

A kiinduló analógiakészletet memóriakímélő módon úgy készíthetjük el, ha számos *konkrét* rokonfeladatot (és megoldását) általánosítjuk: elkészítjük absztrakciójukat. Azaz a rokonságot kidomborítjuk, az egyedi különbözőségeket eliminálják az egy „közös ős”, *absztrakt* mag megtalálása kedvéért.[1]

Az analógiabázisnak egy-egy eleme: egy absztrakt feladat és egy hozzá illeszkedő, legcélszerűbbnek talált megoldás. E gondolat vezet el a **programozási tételek** fogalmához: absztrakt feladat + absztrakt algoritmus + helyességbizonyítás. A bizonyítást természetesen nem kell állandóan észben tartani, elegendő csak a tétel bevezetésekor elvégezni. Pontosan úgy, ahogy a természettudós tesz kutatása során az alkalmazott tételekkel: ellenőrzi a feltételeket, és mondanivalójának célirányosságát, majd elfogadja helyességét vizsgálat nélkül, hogy azután felhasználja. Nyilvánvaló, hogy a bizonyítás a közoktatásban minden fajta formalizmus nélkül, „józan paraszti ésszel” érvelve történik; de történjen, mert nagyban hozzájárul az algoritmus és a feladat kapcsolatának felderítésére. Az 1. táblázatban egy példát adunk egy programozási tétel definiálására. [2/22.o.]

[1] Vegyük észre, hogy ezzel a PISA-i kritika egyikére: a túlzásba vitt lexikális alapú oktatásra kínálunk gyógymódot.

Mi most az absztrakt feladattípusokon alapuló programozási tételekkel foglalkozunk. Más módszertanokban előfordulhat, hogy pl. absztrakt megoldástípusokra építenek hasonló, bár kevesebb programozási tételt [6].

A programozási tétel része	Példa
Absztrakt feladat – specifikáció:	**Eldöntés**(H^*,F(H,**L**)):**L** **Bemenet:** N∈**N**, X∈H^*, T:H→**L** **Kimenet:** VAN∈**L** **Előfeltétel:** N=Hossz(X) **Utófeltétel:** VAN = $\exists i \in [1..N] : T(x_i)$
Absztrakt megoldás:	**Konstans** MaxN:Egész(???) **Típus** THk=**Tömb**(1..MaxN:TH) **Eljárás** Eldöntés(**Konstans** N:Egész, X:THk, **Változó** VAN:Logikai): **Változó** i:Egész i:=1 **Ciklus amíg** i≤N **és nem** T(X(i)) i:+1 **Ciklus vége** Van:=i≤N **Eljárás vége.**
Bizonyítás:	Ezt most hosszadalmassága miatt elhagyjuk. [3/19-21. o.]

1. táblázat: Programozási tételek felépítése – az „Eldöntés" tétele.

Néhány megjegyzés a táblázathoz:

- A specifikáció **fejsor**át, mint egy **függvény**t adtuk meg. A zárójelek közt szereplő halmazok alkotják a függvény értelmezési tartományát. (E halmazok direktszorzata alkotja az értelmezési tartományát.) A kettőspont utáni szereplő halmaz az értékkészlete. E felírás haszna a tételek kombinálásánál válik nyilvánvalóvá, így ugyanis a kombinálható tételek egy körét eleve kizárhatjuk, mint „illeszthetetlent", s ezzel kevesebb „analógia" végiggondolását teszi lehetővé. Végső fokon a gondolkodás hatékonyságát segíti.
- A specifikáció apró finomságokat tartalmaz, amelyek a továbblépésnél kamatoztathatók. [2/25-27. o.) Ilyen például a sorozat „X∈**H***" megadása helyett írhattuk volna: „X∈$\mathbf{H^N}$", amely ugyan világos kapcsolatot teremt a két paraméter (N és X) között, de a progranozás következő fázisában az adatok leírásánál a helytelen „**Típus** THk=**Tömb**(1..N:TH)" definícióra csábít. Ami az így hiányzó N-X kapcsolatot illeti: helyre hozható az előfeltételben, ahogy meg is tettük.
- A programozási tételek jól ismertek a programozás módszertanban. Sajátos **formalizmus**sal definiálva a 17 legfontosabbat (és néhány speciális változatát) megtalálhatjuk a következő irodalmakban: [4, 5]. A feladatleíró nyelv – amint a korábbi példából kiderült – **halmazelmélet** és **elsőrendű predikátumkalkulus** elemein alapulnak. A formalizmushoz tegyük azonban mindjárt hozzá, hogy a közoktatásban ezt a nyelvet csak „beszélni" kell tudni, nem „írni". Azaz a formalizáltság kezdetben egyáltalán nem lényeges; csak *a használat tudatossága*.
- A haladó programozás oktatásbéli specifikálás középpontjában a **sorozat** absztrakt matematikai fogalma áll. Számos más nyelvi változatot is megalkottak a specifikációkészítéshez. Például egy, a sorozatok helyett **halmaz**okon nyugvót alkalmaznak az [7] irodalomban. Didaktikai

okok miatt döntöttünk a sorozat központi szerepe mellett: a **sorozat** fogalma **közelebb áll a kezdő programozók által előszeretettel használt algoritmikus tömb** fogalmához.

A programozási tételek alkalmazásakor a programozó egyrészt **absztrahál**: elnevezi és körülírja a részfeladatot jelentő finomítást, majd egy **tételanalógiát keres**, amely ráhúzható a körvonalazott részproblémára. A programozási „hétköznapokban" sokszor a tételeknél „elemibb" absztrakciókat használ a programozó. Nézzük, mik ezek!

Akkor, amikor a **felülről lefelé tervezés** stratégiai elvet alkalmazza a programozó, s ezt tükrözi az algoritmikus nyelv egy újonnan bevezetett eljárásával vagy függvényével (*finomítás*), lényegében a nyelvet, magát bővíti az új fogalommal (vö. a nyelvi absztrakcióval [1]). A nyelv szókincséhez egy elem hozzávétele két dolgot jelent. Egyrészt a fogalom „alakjának", **szintaxisának** és az elvárásának, **specifikációjának** összekapcsolását; ez maga az absztrakció, másrészt a „mögöttes" tartalom, a precíz **szemantika** definiálását, a fogalom reprezentálását-implementálását jelenti. A programozásban ezt a fajta építkezést nevezik hagyományosan **algoritmikus absztrakció**nak, tehát némileg szűkebben értelmezik nálunk. Ez az építkezés azonban – többnyire – sok lépésben halad előre, mondják úgy is: a megoldást **lépésenként finomítva**, a programozási nyelvhez közeli szintig kell folytatni (l. dekomponálást). Így hozunk létre – Dijkstra-t idézve – a feladathoz illeszkedő absztrakt **fogalomhierarchiá**t.

Tapasztalataink szerint a fogalom megismerése két szinten zajlik: az első szinten a kezdő programozó megérti, hogy ezzel egyrészt egy *algoritmusrövidítő* eszköz birtokába jut, másrészt ennek alkalmazása során fogalmazódig meg benne a *paraméterezhetőség* igénye. Érdekes, hogy az algoritmusrövidítés előnyét hamarabb konstatálják, mint a fő értékét: a szellemi energiájuk optimálisabb beosztását lehetővé tevő „oszd meg és uralkodj" elv kivitelezhetőségét. A paraméteres finomítás fogalmában további nehézséget a formális és az aktuális paraméterek elválasztása és helyes értelmezése jelenti. Ezért írtunk a megértés két szintjéről, amelynek 2. szintjét jelenti a paraméterek okozta absztrakciós gát átugrása.

Még egy, absztrakcióval összefüggő algoritmikus gondolatot érdemes megemlíteni. A „feladat általánosítása" módszertani elvünk [3/97] betartása is egy absztrakciós lépés a tervezésben. Lényege: nemcsak a konkrét feladatra, hanem egy bővebb feladatkörre alkalmazható programot érdemes alkotni úgy, hogy a feladatban szereplő fogalmakat (elsősorban konstansokat) általánosítjuk.

Legegyszerűbb esetben a feladatbéli *konstansok*at szimbólumokkal helyettesítve építjük be a megoldásba. A szimbolikus adatok válhatnak a feladat eredeti megfogalmazásához képest többlet bemenő paraméterekké, de lehetnek csak szimbolikus konstansok (hívhatjuk belső vagy látens paramétereknek). Már ez utóbbi esetben is számolhatunk azzal az előnnyel, hogy e konstans értékét érintő karbantartási változtatásokat egyetlen mozdulattal és teljes biztonsággal hajthatjuk végre.

Például a „soroljuk fel egy kosárlabda-csapat játékosai közül a 210 cm-nél magasabb játékosokat" feladatban, kézenfekvő általánosítás lehet

- a 210 cm helyett a `Mag` konstans (210 kezdőértékkel) vagy többlet bemenő paraméterrel, és
- bár kosárlabda-csapatról esik szó a feladatban, így kötött létszámot jelent, mégis a játékos létszámot vagy a `Létszám` konstanssal, vagy ugyanilyen nevű bekérendő változóval deklarálni.

Ehhez hasonlóan járhatunk el a feladat „indukálta" *típusok*kal is. Azaz ahol lehet, általánosabb alaphalmazokkal dolgozzon a majdani program. Természetesen az általánosítás itt jól kitapintható hatékonysági kérdéseket is fölvethet. Például a fenti feladat esetében a következő típusokat érintő általánosításokra gondolhatunk:

- az egész számokkal, cm-ben megadott magasság helyett valós számokkal is dolgozhatunk,
- tömb helyett más sorozat-típusokkal is ábrázolhatjuk a játékosokat megtestesítő adatokat.

A feladat általánosítás harmadik esete az előfeltétel-gyengítés. Az egyik legegyszerűbb példája talán az, amikor egy kiválasztási feladatot mégis keresési programozási tételre vezetünk vissza, megszüntetve ezzel azt az előfeltételt, hogy a keresett típusú elem biztosan szerepel a bemenő sorozatban.

A feladat most elemzett általánosítása tehát nem annyira absztrakciós nehézségeket okoz a programírónak, sokkal inkább az ideális arány megtalálása a **feladatszintű általánosság** és a **tervezés szintű hatékonyság** között okozhat fejtörést.

Ezt a fajta feladat-absztrakciót ötvözhetjük a finomításossal. Nézzük a fentebb említett feladatot! Az algoritmusbéli magassági korlátra vonatkozó reláció helyett egy általánosított tulajdonságot megtestesítő (azt általánosító) *logikai függvény* is szerepelhet. Ez által megoldásunk minden olyan feladatnak majdnem kész megoldása lesz, amelyben „csak" a játékos érintett tulajdonsága változik az itt említetthez képest.

Az általánosítás következő területeit érintettük ebben az alfejezetben:

- **feladatáltalánosítás** – feladatparaméterek, típusok
- **megoldási sémák absztrakciója** – programozási tételek
- a szorosabban vett **algoritmikus absztrakció** – finomítások és paraméterezésük

Az algoritmikus absztrakción keresztül felismerhetjük a gondolati absztrahálódás általános működését. [8/18. o.] nyomán az adat-algoritmus fogalomkettős absztrahálódási folyamata a következő:

- **konkrét** feladatokhoz **konkrét** programmegoldások →
- **utasítás-, adat-absztrahálódás** – absztrakt utasításfajták: elemi műveletek, szekvencia, elágazás, ciklus; adatfajták: elemi típusok; összetett típusok: rekord, tömb, fájl →
- „**rekonkretizálódás**" – ezek felhasználása konkrét feladatok megoldásában →
- **absztrahálódás** – **teljesebb típusfogalom**, **típuskonstrukciók**: típusdefiniálás, paraméteres típusok; **programsablonok**: programozási tételek →
- „**rekonkretizálódás**" – ezek felhasználása konkrét feladatok megoldásában →
- **absztrahálódás** – adatszinten: tömbből **asszociatív tömb**, **általánosított sorozat**, **halmaz**; algoritmus szinten: **paraméteres függvények** →
- „**rekonkretizálódás**" –
  adat szinten: konkrét paraméteres típusok széleskörű alkalmazása →
  algoritmus szinten: konkrét paraméteres tétel-függvények alkalmazása – **tételkombinálás**= **függvénykombinálás** →
- típus **absztrahálódás tervezési** szinten: **feladat és eszközkészlet elválása** – algebrai leírás + modul-szintű megvalósítás;
  típus **absztrahálódás kód** szinten: modul-szerű fogalmak: template, osztály-sablon, generic

Ugyanezt követhetjük nyomon az adatok fogalmának metamorfózisán az [9] alapján.

3. Dekompozíció-szuperpozíció

A **dekomponálás** a komplex probléma elemibb problémák együttesére **bontás**át jelenti. Az elemi, illetve komplex probléma viszonyának megfordításakor **szuperponálás**ról beszélnek: ekkor a komplex probléma elemi feladatokból való **felépítés**ére gondolnak. A lényeg ugyanaz: a komplex probléma és elemibbek kapcsolatba hozása.

Az emberi elme jól ismert pszichológiai korlátjáról így ír – szabadon idézve – Mérő László: az emberi agy rövid távú memóriában egy időben tartható sémák, azaz gondolkodási struktúrák maximális száma egyénenként a 7±2 érték között mozog. [10/12. o.] Egyszerűen szólva ez azt jelenti, hogy az ember tervezéskor nem igen képes áttekinteni olyan „bonyolultságú" tervet, amelynek sé-

maszáma a fenti értéket meghaladja. Ezek szerint a strukturált programozás már többször emlegetett **felülről lefelé tervezés**, vagy hétköznapi szavakkal mondva: az **oszd meg és uralkodj** elve pontosan ezen emberi gyengeség beismerését, sőt elkerülésének módját fogalmazza meg, számszerűsítés nélkül. Tehát egy, programozásból kihagyhatatlan elvről van szó.

E gondolkodási művelet lényege, hogy a feladatot egyre finomodó műveletkészlet segítségével fogalmazzuk meg újra meg újra, amíg az algoritmikus nyelvünk eleve meglévő utasításáig el nem jutunk. Minden szint a feladatot teljesen megoldja, de a szint egy-egy összetett sémája (eddigi szóhasználatunk szerint: finomítása) csak kb. 5-9 még elemibb mikrosémára (azaz finomításra, vagy már definiált műveletre) bomlik. Ez tükröződik az 1. fejezetbéli 1. ábrán szereplő programozási modell 3. séma-transzformációjában.

Cormen és szerzőtársai [11/10. o.] könyvükben így közelítik meg a lényeget:

„*Az oszd meg és uralkodj paradigma a rekurzió minden szintjén három lépést vesz igénybe*[2]*:*
Felosztja a problémát több alproblémára.
Uralkodik az alproblémákon rekurzív módon való megoldásukkal. Ha az alproblémák mérete kicsi, akkor közvetlenül oldja meg az alproblémát.
Összevonja az alproblémák megoldásait az eredeti probléma megoldásává."

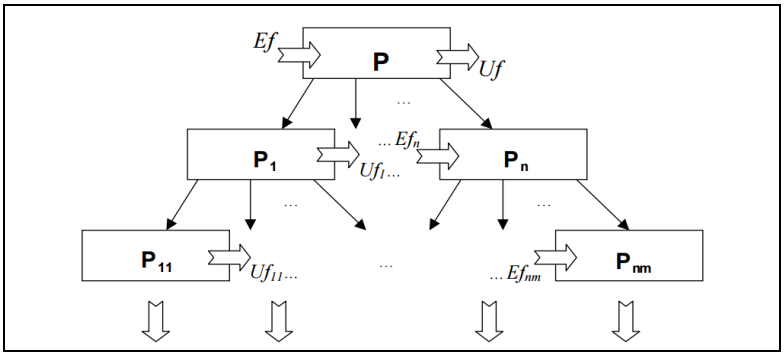
Ehhez két észrevételt fontos hozzátenni:

Egyrészt: az alproblémák (vagy a pszichológia szerint: sémák) összevonása **a megfelelő utasítás-szervezés kiválasztás**át jelenti. El kell rendezni az alproblémákat a rekurzió érintett szintjén. Dönteni kell egymáshoz való „viszonyuk" felől. Vagyis az alábbi lehetőségek – és esetleges kombinációi – közül választunk:

- **szekvencia** – több alprobléma adott sorrendben kerül egymással mellérendelő viszonyba, amelyek mindegyike végrehajtandó;
- **elágazás** – több, önálló feltételtől függő alprobléma, amelyek közül az igaz-feltételű hajtandó végre, az elágazás egészéhez képest az említett alproblémák alárendelt viszonyban állnak;
- **ciklus** – egy „feltételtől függő számú"-szor hajtódjék végre a ciklus törzsét alkotó alprobléma, a ciklus egészéhez képest az alprobléma alárendelt viszonyban áll.

Másrészt: az **alá- és mellérendelő viszony**nak több szempontból is jelentősége van. Ezek egyike az *algoritmus leírási mód*ját befolyásolja. A másik fontos következménye a *bonyolultság*ot érinti. Míg utasítások mellérendelésével bővítve a programot, annak bonyolultságát (additíve) *lineárisan* növeljük, addig ugyanezen utasításokat alárendelten illesztve a programhoz, pl. egy összetett szerkezetben, (multiplikatíve) *hatványozottan*. Ezt tükrözi az egyik jól ismert bonyolultsági mérték: a *mélységi bonyolultság*, amelynek lényege, hogy a program egyes részstruktúrájához „*azt a kitevőjű kettőhatványt rendeljük, ahány magasabbrendű struktúra belsejében van*" [12/103. o.], s e részstruktúrák összege adja ki a program összbonyolultságát. Így **a finomítások alkalmazásával** numerikusan kimutatható módon is **csökken a program bonyolultsága** teljes összhangban az elvárásainkkal.

[2] Miután megfogalmaztuk a triviális esetet, amelynél már nem kell részproblémákra osztani.



2. ábra: A 'felülről lefelé' elvet követő és a specifikáció alapú programtervezés.

A fenti három probléma-összetételi mód közüli választáshoz is van kezdők számára kapaszkodó: a „*struktúra szerinti feldolgozás*" elve [13/46. o.]. Az elv szoros kapcsolatot állít föl a feldolgozás alatt álló adatszerkezet és a feldolgozást szervező algoritmus-szerkezet között.

Adatszerkezet		Algoritmus-szerkezet
skalár	→	függvény vagy eljárás
direkt-szorzat	→	szekvencia
unió	→	elágazás
sokaság	→	ciklus

2. táblázat: A 'struktúra szerinti feldolgozás' elve.

Az elv felhasználása abból áll, hogy

1. meghatározzuk a feladat vagy az alfeladat bemeneti és kimeneti adatait, az [13] irodalomtól kölcsönzött terminus technicus-szal élve: a feladathoz tartozó bemeneti és kimeneti *szuperstruktúrá*t,
2. egyiket kiválasztjuk „vezérlő" adatszerkezetnek[3],
3. megfeleltetjük az elv felkínálta adatszerkezetek egyikével, és
4. kiválasztjuk az elv szerint hozzátartozó algoritmus-szerkezetet.

Az elv használata látszólag csak az algoritmusvázlat összeállításában segít. A konkrét feladatok esetén meglévő további információ is sokszor beépíthető a hozzárendelt algoritmusba. Az elv gyümölcsöző voltát fejtegetjük a már idézett [13] irodalom II. fejezetében.

A **szuperpozíció** alatt gyakorta **alulról felfelé építkezés**t értenek. Ilyenszerű tevékenységet a programozók akkor végeznek, amikor egy új feladatcsaládhoz, egy nagyobb kaliberű problémavilághoz készítenek adekvát „programkörnyezetet". A cél egy olyan magasabb szintű „nyelv" definiálása, amelyen a speciális problémák könnyebben megfogalmazhatók. Tipikusan egy új *típuskonstrukció* definiálása (például gráfok, fák, speciális szerkezetű adathalmazok), vagy valamilyen *rutinkönyvtár* specifikálása, és reprezentálása-implementálása. Ekkor a cél nem egy adott feladat megoldása programmal, hanem egy program-eszközkészlet elkészítése. Ha a hagyományos (dekomponálásos) programozás célja egy programnak, mint **feladatmegoldó eszköz**nek az elkészítése, akkor a szuperpozíciós hozzáállás célja az előbbi feladatmegoldó eszközhöz szükséges **szerszámkészlet** legyártása.

[3] Szerencsés esetben bijektíve meg tudjuk feleltetni egymásnak a be- és kimeneti struktúrák elemei.

Tehát szó sincs arról, hogy egymás alternatívái lennének. Annál is kevésbé, mivel egy-egy bonyolultabb „szerszám" elkészítéséhez is a dekomponálás stratégiájával nyúlunk.

4. Összegzés

Értekezésünkben a 7 általunk definiált gondolkodási művelet (módszer) közül kettőt mutattunk be, az algoritmikus absztrakciót és egy egymást kiegészítő kettőst, a dekompozíció-szuperpozíciót. Az algoritmikus absztrakciót a hagyományosnál bővebben értelmeztük. A dekompozíció-szuperpozíció kettőse két stratégia, amelyek jól kiegészítik (s nem helyettesíthetik) egymást a komplexebb rendszerek tervezésénél.

A tárgyalt gondolkodási eszközök olyan fogalmak, amelyek nem rendelkeznek pontos határvonallal, kölcsönösen átnyúlnak egymás „érdekeltségi" körébe. Ennek ellenére megkülönböztetésük hasznos. A fogalmak körvonalazása **lehetővé teszi önálló vizsgálat**ukat, ami által feltérképezhetővé válik a **gondolkodás mechanizmusa**. Fontos következmény, hogy kidolgozhatóvá válnak **módszerek** az egyes **gondolkodási eszközök használatának fejlesztésére**. Ezek elsősorban a programozási eredményességben kamatoznak, de általában a **problémamegoldó gondolkodásra is pozitívan hatnak**.

Irodalom

1. P. Szlávi, L. Zsakó, G. Törley: *The Thinking Toolkit of Programming.* in Proceedings of XXIX. DidMatTech 2016, "New methods and technologies in education and practice" Conference, Budapest (2016) http://didmattech.inf.elte.hu/wp-content/uploads/2016/08/SzP_TG_ZsL_The-Thinking-Toolkit.pdf (utoljára megtekintve: 2016.11.06.)
2. Szlávi Péter: *A programkészítés didaktikai kérdései.* ELTE (2005) http://www.inf.elte.hu/karunkrol/szolgaltatasok/konyvtar/lists/doktori%20disszertcik%20adatbzisa/attachments/32/szlavi_peter_tezisek_hu.pdf (utoljára megtekintve: 2016.11.06.)
3. Szlávi Péter: *Programok, programspecifikációk.* in Informatika a felsőoktatásban'99, pp. 576-582 (1999) http://people.inf.elte.hu/szlavi/ProgModsz/Progspec.pdf (utoljára megtekintve: 2016.11.06.)
4. Szlávi Péter, Zsakó László: *Módszeres programozás: Programozási bevezető.* in Mikrológia sorozat, ELTE TTK Informatikai Tanszékcsoport (2002)
5. Szlávi Péter: *Programozási tételek – összefoglaló.* (2001) http://people.inf.elte.hu/szlavi/ProgModsz/Prtetel.pdf (utoljára megtekintve: 2016.11.06.)
6. Gregorics Tibor: *Programozás 1. kötet Tervezés.* ELTE Eötvös Kiadó (2013)
7. Fóthi Ákos: *Bevezetés a programozáshoz.* Tankönyvkiadó (1983)
8. Fényes Imre: *A fizika eredete.* Kossuth Kiadó (1980)
9. Szlávi Péter, Zsakó László: *Programozási nyelvek: Alapfogalmak.* in Mikrológia sorozat, ELTE TTK Informatikai Tanszékcsoport (2004)
10. Mérő László: *A mesterséges intelligencia és a kognitív pszichológia kapcsolata.* Tankönyvkiadó (1989)
11. T. Cormen, Ch. Leiserson, R. Rivest: *Algoritmusok.* Műszaki Könyvkiadó, 1. kiadás (1997)
12. Zsakó László: *Módszeres programozás: Hatékonyág,* in Mikrológia sorozat, ELTE TTK Informatikai Tanszékcsoport (2003)
13. Szlávi Péter, Zsakó László: *Módszeres programozás: Adatfeldolgozás.* in Mikrológia sorozat, ELTE TTK Informatikai Tanszékcsoport (2004)