Architektúrális gondolkodás fejlesztése valós idejű rendszerekkel

Korom Szilárd

korom.szilard@gmail.com
ELTE IK

Absztrakt. A középiskolai programozás oktatásban az algoritmikusan nehéz feladatok megoldása jóval hangsúlyosabb, mint az architektúrális gondolkodás fejlesztése. Az előadásom célja ezen probléma bemutatása, részletezése, valamint egy olyan tananyagba építhető modul bemutatása, mely a valós idejű rendszereken alapszik, s a fent említett kompetencia fejlesztésére szolgál.

Kulcsszavak: programozás, informatikai kompetenciák, rendszerszintű gondolkozás, architektúra, architektúrális gondolkozás, valósidejűség, real-time, beágyazott rendszer

1. Informatikai kompetenciák

A programozás középiskolai oktatásának fontosságával már számtalan cikk és előadás foglalkozott [1]. Ezt tekinthetjük tehát axiómának még akkor is, ha a Nemzeti alaptantervben ez csak csekély mértékben van jelen. [2] Ha viszont ezt fontosnak tartjuk, meg kell vizsgálnunk mit értünk programozás oktatásán, pontosan mit- és miért akarunk fejleszteni. A Nemzeti alaptanterv alapján az informatika oktatást érintő kulcskompetenciákat megfogalmazta dr. Szlávi Péter és dr. Zsakó László egy korábbi cikkben. A következőket sorolták fel:

1. Algoritmikus gondolkodás
2. Adatmodellezés
3. A valós világ modellezése
4. Problémamegoldás
5. Kommunikációs képesség
6. Alkalmazói képesség
7. Csoportmunka, együttműködő-képesség
8. Alkotó képesség
9. Információs tájékozódási és tájékoztatási képesség
10. Rendszerszintű gondolkodás

Ezek közül nyilván az *Alkalmazói képesség* kerül elő leggyakrabban a jelen kor oktatásában, s a nemzeti alaptanterv is erre helyezi a hangsúlyt. [2] Ez a cikk azonban csak azokkal a kompetenciákkal kíván foglalkozni, melyek szorosan kapcsolódnak a programozáshoz. Így például bár a programozás a valós életben igen gyakran összefonódik a *csoportmunkával* (pl.: agilis szoftverfejlesztés), mégsem tekinthetjük a programozás részének. A cikk nem foglalkozik a *problémamegoldással* sem, hiszen ez egy olyan alapvető kompetencia, mely bármelyik másikkal összevonható, lényegében egy ős képességnek nevezhetjük, melyből tetszőlegest leszármaztathatjuk (akár más tantárgyakon belül is). Ezek alapján a következő kompetenciák maradtak:

1. Algoritmikus gondolkodás

EFOP-3.6.3-VEKOP-16-2017-00001: Tehetséggondozás és kutatói utánpótlás fejlesztése autonóm járműirányítási technológiák területén – A projekt a Magyar Állam és az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

2. Adatmodellezés
3. A valós világ modellezése
4. Rendszerszintű gondolkodás

A kiválasztás fő szempontja tehát azon a kérdésen alapszik, hogy **milyen jellegű feladatokkal** érdemes tanítani a programozást. Ebből a nézőpontból a *problémamegoldás*, *kommunikációs képesség*, *csoportmunka*, *alkotó képesség*, *információs tájékozódás* kompetenciák egy feladat jelleg részét képezik, egy módszertani szempontot egy konkrét projekt tanítása során.

Jelen cikkben az *adatmodellezés*, valamint a *valós világ modellezése* kompetenciákkal nem kívánok foglalkozni. Ezen téma kidolgozása egy teljes dolgozatot igényelne. Másfelől, már született magyar nyelvű cikk a témakörben, melyeket a források között meg is említek. [3]

2. Rendszerszintű gondolkodás

A *rendszerszintű gondolkodás* meghatározására számtalan tanulmány készült. [4, 5, 6, 7, 8, 9] Ezeket kívánta összefogni, s egységesíteni Arnold és Wade A *rendszerszintű gondolkodás* olyan szinergikus analitikus készségek csoportja, melyek javítják egy rendszer azonosításának és megértésének képességét, megjósolják viselkedésüket és módosítják azokat, a kívánt hatások elérése érdekében. Ezek a készségek rendszerként működnek együtt. [10]

Röviden tehát a fenti szerzők a *rendszerszintű gondolkodást* úgy azonosították, mint egy rendszert, a rendszerekről való gondolkodásról. A probléma tehát az, hogy a komponensek működése, a részproblémák megoldása nem okozza a rendszer működését. Az *algoritmikus gondolkodás* egy komplex rendszer esetében nem mindig vezet megoldásra (vagy nem kellően gyorsan), hiszen ilyenkor a jelenségek kimenetelét sokszor a rendszer struktúrája, nem pedig az egyedi összetevők állapotai határozzák meg. [11] Például, a szükséges algoritmusok megírása önmagában nem elegendő ahhoz, hogy egy program működjön, sőt algoritmikusan nem is biztos, hogy azonosítható a struktúra. A feladatra, mint rendszerre kell tekinteni, akként kell azonosítani és elhelyezni, szükség esetén pedig a részegységeken kell módosításokat végrehajtani. Arnold és Wade a számos definíció és tanulmány alapján a következő készségelemeket fogalmazták meg (Veres Gábor fordításában): [12/227. o.]

2.1. A részek és kapcsolatok azonosítása

A rendszerszemlélet alapvető készsége, de megfelelő gyakorlat nélkül nehezen fejleszthető, illetve könnyen elvesztheto.

Egy komplex rendszer részeinek azonosításához rendelkezni kell sémákkal, amikre rá lehet illeszteni a komponenseket. Ilyen módon egy rendszer megtervezése, módosítása lehetetlen, ha nem vagyunk tisztában azzal, mik az előfordulható lehetőségek. Ehhez számtalan potenciális tudáselemre lehet szükség a definíciókon, tételeken, lexikális tudáson át egészen a gyakorlati alkalmazásig.

Mivel a *rendszerszintű gondolkodás* legalapvetőbb eleme is megkíván előismereteket, ezért kijelenthető, hogy a kompetencia fejlesztését meg kell előznie valamilyen ismeretátadás (pl: algoritmusok). Ez azonban nem jelenti azt, hogy „csak akkor lehet bevinni a tanórára, ha mindennel végeztünk". Módszertanként is elhelyezhető például úgy, hogy egy-egy új ismeret megszerzése után időt fordítunk annak rendszerben való működésére. Másik példa lehet a probléma-alapú oktatás projektjei. [13]

2.2. Visszacsatolások azonosítása és megértése

Bizonyos kölcsönhatások ok-okozati visszacsatolásokat alakíthatnak ki, amelyek alapvetően befolyásolják az adott rendszer viselkedését.

Ez az elem tehát nem magukról a kapcsolatokról szól, hanem azok egymáshoz való viszonyairól, a kapcsolatok hatásairól és működési mechanizmusairól. Ennél a szintnél beszélhetünk arról a tézisről (amit korábban említettünk), hogy a komponensek működése nem (feltétlenül) okozza a rend-

szer megfelelő viselkedését. A rendszer struktúráját tekintve ettől lesz egységes egész, s a megfelelő működéshez rendszerint a részelemek módosítása is szükséges.

Az egyszerűbb programozói feladatok esetében a megoldás általában automatikusan, a tényleges feladat realizálása nélkül megtörténik, hiszen természetesen adódik az elemek megfelelő összekapcsolása, az ok-okozati visszacsatolások elkészítése. Módszertanilag erre könnyen lehet építeni, hiszen ha a diák valóban képes erre, akkor pár feladat után levonható egy egységes tanulság.

2.3. A rendszer szerkezetének megértése

Látszólag az 1. és 2. ponttal megegyező készségeket igényel, azonban a kutatások szerint inkább azok speciális kombinációját jelenti.

Úgy is tekinthetünk erre, mint az első 2 pont egyfajta ötvözetére. Bonyolultsága, összetettsége a konkrét rendszertől függ, ezért tanításmódszertani szempontból ennek tanítása spirálisan képzelhető el egyre komplexebb rendszerekkel.

Ide tartozhatnak a kész architektúrák is, melyek sémákat biztosítanak az egyén számára a rendszer működésével kapcsolatban. A komplex rendszerek gyakorlati előállítása, kezelése szorosan összefügg ezzel, hiszen biztosra kell mennünk, hogy a szerkezeten mindenki ugyan azt érti. A sémák kialakítása kiválóan működhet a gyakorlatban *csoportmunkában*, hiszen ekkor a tagok rá vannak kényszerítve az egységességre.

2.4. Állandók és változók, folyamatok azonosítása

Állandó lehet például valamely fizikai erőforrás készlete (pl. tartalék tápanyag), vagy akár érzelmi, mint a bizalmi tőke egy kapcsolatban. A változók módosíthatják a készletek értékeit, így folyamatokat hívhatnak életre. Ez a rendszerszemlélet magasabb fokú készségeleme.

A szerző a listát egy hierarchiában képzelte el, ahol ezt az elemet mindenképpen meg kell előznie az előző pontoknak. Az informatikában azonban ez véleményem szerint nem ilyen szigorú. A folyamatok (vagy angol nyelven: *flow*) végig követése sokszor igen hasznos az egész rendszer megértésében. Különös tekintettel összetett alkalmazások esetében. Az azonban bizonyos, hogy ezt el kell különítenünk az előző 3-tól, hiszen azok a szerkezetre vonatkoznak, míg ez egyfajta dinamikus, konkrét folyamat esetében válik érdekessé.

Ennél a pontnál külön szeretném hangsúlyozni a programozásban betöltött szerepet, ugyanis egy alkalmazás esetében nagyon meghatározó a *data flow*. Gondoljuk például az egyszerű konzolos alkalmazásokra (amikkel általában bevezetik a programozást). Ezek rendre az input, feldolgozás, output szekvenciát követik. Azonban nem csak ilyen alkalmazásoknál, de komplex informatikai rendszereknél, hálózatoknál is nagyon meghatározó az adat, annak végigfolyása, változók értékeinek megváltozása, hiszen gyakran az utóbbi esemény a „belépési pont", azaz erre reagál számos másik folyamat.

2.5. Nem lineáris folyamatok azonosítása

Ez a készségelem a fontossága és a félreértelmezhetőség elkerülése miatt került külön pontba az előzőtől.

A párhuzamosság számtalan problémát felvet, hiszen ha egy folyamat „működik" egy lineáris rendszerben, korán sem biztos, hogy egy másikba is fog. A párhuzamos folyamatok egymás viselkedésére is hatással lehetnek, így a rendszer alapvető működési mechanizmusa változik meg.

Az informatikában különösen nem szabad megfeledkezni erről a pontról. Gondoljuk például a *Scratch*-re, mely pont a párhuzamos eseménykezelése miatt válik olyan érdekessé, dinamikussá és sokrétűvé.

2.6. A dinamikus viselkedés megértése

A kölcsönhatások és visszacsatolások befolyásolják a készleteket és a változókat, az időbeli folyamatok a rendszer dinamikus viselkedését alakítják ki. Ez a készségelem az előzőeket is feltételezi, és megfelelő gyakorlattal fejleszthető.

Ehhez a szinthez már kiemelten szükség van a gyakorlat, sőt a rendszerek működésével kapcsolatos tapasztalatokra is. Ezen a szinten az egyén már képes a struktúra átlátására, részegységeinek azonosítására, egy lehetséges folyamat végig követésére, az általa beindított egyéb viselkedésmegváltozások azonosítására. Ezen elem megértése után a rendszer struktúrájáról egy absztraktabb képet kaphatunk.

Véleményem és tapasztalataim szerint a programozás oktatásában ez a legnehezebb szint. Meglévő struktúra és a megfelelő elemek ismeretével sem biztos, hogy a diák meg tud oldani egy feladatot, vagy képes a dinamikus viselkedés megmagyarázására még akkor is, ha ismeri a forráskódot. Még ha tervezni tud is, sőt érzi, hogy milyen algoritmuselemet kellene használni, nem tudja ráilleszteni a konkrét folyamatra. Ennek elsajátításához nagyon sok gyakorlat és tapasztalat szükséges.

2.7. A komplexitás csökkentése a rendszermodell megfelelő tervezésével

A komplexitás különféle intuitív technikákkal (pl. redukció, transzformáció, absztrakció és homogenizáció) csökkenthető a modellekben. Lényegében az adott cél szerint felesleges rendszerelemek kizárásának képességét jelenti.

Ez a pont a gyakorlatban már nem mindig jelenik meg (főleg a középiskolai oktatásban), de a *rendszerszintű gondolkodás* feltétlen része. A kérdés, hogy a komplexitás csökkentése, mint meglévő rendszer módosítása, vagy mint a tervezésnél fontos szempont jelenik meg. Az utóbbira feltétlenül szükség van mind a diák, mint a tanár szempontjából, hiszen a lehető legegyszerűbben szeretnénk egy problémát megoldani.

2.8. Egymásba épülő rendszerszintek megértése

Ez a készségelem a rendszerek egymásba épülésével kialakuló hierarchia, a rendszer-alrendszer összefüggés megértése, az anyagi világ szerveződési szintjeinek átlátását foglalja magában;"

Ez a képesség már igen bonyolult. Nagyon sok sémára, gyakorlatra és elméleti tudásra van szükség ahhoz, hogy a rendszert, mint egységes egészet tudjuk látni, hiszen Mér László már megfogalmazta, hogy a rövid távú memóriában egy időben tárolható kognitív sémák maximális száma. 7 ± 2 [10/12. o.] Ami tehát fontos az a lényeglátás, tehát a feladat szempontjából érdektelen komponensek kiszűrése.

További érdekes kérdés, hogy hogyan tudjuk megkülönböztetni az egyszerű komponens egy alrendszertől. Ha ugyanis a rendszer alrendszerekből épül fel, akkor a felsorolt hierarchia rekurzívan alkalmazható mindegyikre.

Az informatika középiskolai tanításában meglátásom szerint nem szükséges ilyen komplex rendszerekkel találkoznia a diáknak. A NAT-ban [2] sincsen rá utalás, hogy erre szükség lenne, s az érettségi feladatokat, vagy a versenyeket tekintve sem tűnik relevánsnak.

3. Rendszerszintű gondolkodás a NAT-ban

A rendszerszintű gondolkodás, mint elvárt kompetenciát az informatikai műveltségi területnél a Nemzeti alaptanterv [2] explicit nem fogalmazza meg. Utalás azonban több helyen is van rá. Például az adatmodellezés, algoritmizálás, információ tárolás, hálózati ismeretek s ezek komplex alkalmazásai elvárt készségek, amik együttes használata, kapcsolatuk megértése már igényli a rendszerszintű gondolkodást.

4. Rendszerszintű gondolkodás programfejlesztési szempontból

Az algoritmikus gondolkodás fejlesztésének fontosságáról és szerepéről számtalan tanulmány született. Hogy pontosan miről van szó, a következő idézettel magyarázható:

„Az algoritmizálás először nem számítógépes megvalósításról szól. Csak egy klasszikus, több ezer éves algoritmusra, a két egész szám legnagyobb közös osztóját meghatározó Euklideszi algoritmusra kell gondolnunk! Az algoritmus végrehajtója – a processzor – sok esetben lehet maga az algoritmust megalkotó, azt értelmező ember. (Sőt egy új probléma megoldásánál a rutinos programozó is magát „képzeli" a számítógép helyébe, s így próbálja ki a megoldás működőképességét.) Algoritmusokat mindenki hajt végre nap, mint nap, sőt az emberek többsége alkot is algoritmusokat saját maga és mások számára." [1/4. o.]

Összességében tehát ez a kompetencia sem kapcsolódik feltétlenül a programfejlesztéshez sőt, még az informatikához sem.[1]Persze ez nem annyira meglepő, ami azonban ebből feltétlenül következik, hogy az algoritmikus gondolkodás egyáltalán nem fedi le a valós programozói problémák megoldásához szükséges képességeket. A felvetés tehát az, hogy egy diák hiába tud algoritmikusan viszonylag nehéz problémákat megoldani (alapvető algoritmusok, rendezési algoritmusok, algoritmikus stratégiák), egy valós életben (például az iparban) használt program megírása igen nagy gondot okozhat neki még akkor is, ha eltekintünk a technológiai ismeretek hiányától. Tanárként már számtalanszor találkoztam azzal az anomáliával, hogy tehetséges, programozni tudó, algoritmikus gondolkodásukat tekintve fejlett diákok valójában nincsenek tisztában saját képességeikkel. Nincsenek tisztában azzal sem, hogy amit képesek megalkotni hogyan kapcsolódik a valós élet problémáihoz. Például hiába ismernek több rendezési algoritmust is, nincsenek tisztában azzal, hogy ebből hogyan lesz egy épkézláb alkalmazás. Valójában ez nem a diák hibája, hiszen mint tanár és programozó tudom, hogy az algoritmikus gondolkodása elég fejlett ahhoz, hogy egy átlagos programozói állásban előkerülő problémákat megoldjon (legalábbis nagy részüket), mégis távol áll még ennek megvalósításától. Természetesen az, hogy még nem alkalmas programozónak, számtalan aspektusa van a lexikális tudástól a pszichológiai érettségen át a kompetenciákig. Annak érdekében, hogy csak az informatikához kapcsolódó képességekre szűkítsük a kört, nem érdemes az iparban szükséges kompetenciákat vizsgálni, elegendő kellően összetett programozói feladatokkal foglalkozni. A cél tehát az, hogy kellő rálátást adjuk a diákoknak, mi is az a programozás, valamint, hogy tapasztalatokat szerezzenek arról, hogyan készül el egy összetett szoftveres rendszer. Ennek érdekében a *rendszerszintű gondolkodás* kompetenciát kell vizsgálnunk.

A programozás és a *rendszerszintű gondolkodás* egyvelegét **architektúrális gondolkodásnak** nevezem. Ennek az az oka, hogy az informatikában általánosan és a szoftverfejlesztésben is egy rendszert gyakran architektúrának neveznek, sőt az ezzel foglalkozó szakember beosztása is „architect". Fontos azonban visszautalni arra, hogy ennek fejlesztésével is egy informatikától független kompetenciát fejlesztünk.

Programozási szempontból tehát az az állítás, hogy az *algoritmikus-* és *architektúrális gondolkodás* együttes fejlesztésével a diák teljesebb képet kap szoftverfejlesztésről, valamint tapasztalatot szerez annak gyakorlati hasznáról, hiszen komplex szoftverrendszerek megalkotására is képes lesz. Ez persze motivációként is szolgál, hiszen az egyszerűbb, kevés funkcionalitással bíró alkalmazások és játékok gyártása egy idő után unalmassá, céltalanná válik még akkor is, ha olyan kiváló környezeteket használunk, mint a Logo, vagy a Scratch. Az *architektúrális gondolkodás* fejlesztése tehát meg kell előz-

[1] Természetesen létezik megközelítés, mely az algoritmikus gondolkodás fejlesztésénél a programfejlesztést helyezi középpontba. Erre az alábbi tanulmány szolgál példaként:
Fluent With Information Technology by the National Research Council. National Academy Press. June 1999. http://www.nap.edu/html/beingfluent/

nie az *algoritmikus gondolkodás* fejlesztésének. A diákoknak gyakorlati tapasztalatokat kell szerezniük apróbb alkalmazások, algoritmusok elkészítése terén[2] (ahogyan azt a részképességelemek első pontjában is láttuk).

5. Architektúrális gondolkodás fejlesztése jelenleg

A NAT-ban az informatikai műveltségi területnél az alábbi témakörök kerültek felsorolásra:

1. Az informatikai eszközök használata
2. Alkalmazói ismeretek
3. Problémamegoldás informatikai eszközökkel és módszerekkel
4. Infokommunikáció
5. Az információs társadalom
6. Könyvtári informatika [2]

Ez a gyakorlatban a tanmeneteknél általában úgy jelenik meg, hogy az irodai alkalmazások használata igen hangsúlyos, [15/6. o.][3] szerint 50%, a többi pedig meglehetősen fel van darabolva (többek között a kerettanterv [2] alapján). Ennek részletes elemzése most nem a cikk tárgya, ami azonban fontos, hogy a megszerzett tudás és készségek integrálására egy komplex rendszerben általában nem jut külön idő. Ennek hiányában pedig az *architektúrális gondolkodás* fejlesztése sem jelenik meg. A kompetencia fejlesztése tehát csak akkor kerül elő az informatika tantárgy keretében, ha a szaktanár az adott témakört egy olyan módszer szerint építi fel, melyben az összegzés, összekapcsolás, rendszerbe helyezés kérdése előfordul. Erre természetesen az érettségi feladatokon keresztüli tanítás teljes mértékben alkalmatlan, hiszen ott az irodai alkalmazásokkal foglalkozó feladatok külön egységet képeznek. Például, ha az adatbáziskezelést kizárólag az MS Access segítségével oktatjuk, a diák nem találkozik azzal, hogy az adatbázis rendszerek mikor- és miért hasznosak. Feladatokkal találkozik, melyet egy programmal kell megoldani, azaz „l'art pour l'art" módon tanulja meg a témakört.

Jó motivációt adhatnak a diákoknak a programozói versenyek. A Dusza Árpád Országos Programozói Emlékverseny [17] például egy kiváló lehetőség, hogy ne csak az algoritmikus gondolkodás kompetenciáját fejlesszük a diákoknak. A verseny lényege röviden annyi, hogy 3 fős csapatokban kell komplex alkalmazásokat[4] fejleszteni körülbelül 4 óra alatt (korosztálytól és fordulótól függően). A feladatok általánosságban úgy néznek ki, hogy MVC (Model-View_Controller) architektúrában felírhatók, azaz van egy vagy több adatforrás (jellemzően szöveges fájlokban), egy üzleti logikai, valamint valamilyen megjelenés a felhasználó számára. Érdekes, hogy bár a feldolgozandó egység algoritmikusan nehéznek mondható, a verseny kihívását mégsem ez a rész jelenti, hanem sokkal inkább az időhiány és a csapattagok közötti feladatmegosztás. Tapasztalataim szerint, mind a kettőt megkönnyíti, ha a diákoknak van valamilyen szoftverarchitektúra sémája. Ez azt jelenti, hogy ha meglátnak egy feladatot, a programra mint rendszerre tudnak nézni, s annak analizálásával a konkrét implementálási feladatok világosabbak és a megoldás gördülékenyebben születik meg.

[2] [17] tanulmány kiváló példákat hoz *Scratch* játékokra

[3] A cikk igen érdekes elemzést mutat az új kerettantervről úgy, hogy összehasonlítást ad a brit megfelelőjével.

[4] Ami érdekes továbbá, hogy nem csak asztali alkalmazásfejlesztő kategória van, hanem web programozás, mobil programozás kategóriák is léteznek.

6. Architektúrális gondolkodás fejlesztése valós idejű rendszerekkel

A továbbiakban azzal kívánok foglalkozni, hogyan érdemes az *architektúrális gondolkozást* fejleszteni a középiskolai informatika oktatásban. Ezzel kapcsolatban először egy vázlatos összefoglalót kívánok adni a témával kapcsolatban, amelyeket korábban a cikk keretein belül előkerültek:

1. A NAT [2] nem szán külön modult a képesség fejlesztésére az informatikai műveltségi területen belül, így a tanmenetben külön témakört általában nem képezhet.
2. Az előző probléma megoldása az lehet, ha a *rendszerszintű gondolkodást* a tanításmódszertan módosításával fejlesztjük.
3. Különböző hazai versenyek5 jó alapanyagot (feladatokat), motivációt szolgáltathatnak a kompetencia fejlesztésére.
4. Az *algoritmikus gondolkozás* és az *architektúrális gondolkozás* fejlesztésével a diák alkalmassá válik összetett programok megírására, ami sikerélményt, motivációt adhat számára, valamint tapasztalatot szerezhet a valós programozói tevékenységéről6.

A kompetencia fejlesztéséhez elsőrendűen olyan problémákra, projektekre van szükség, melyek sikeres megoldása a készség fejlődését vonják maguk után. Ehhez komplex, alrendszereket, alprojekteket, alkomponenseket tartalmazó feladatokra van szükség. A cikk a *programozásra* koncentrál, de nem tartom kizártnak, hogy például *alkalmazói rendszerekkel* is lehet ezt fejleszteni.[7] Azt persze érezzük, hogy az alapvető programozási tételeket, érettségi feladatokat, alapvető programozási stratégiákat gyakoroltató feladatok erre nem igen alkalmasak. Fontos szempont továbbá, hogy bár összetett projekteket keresünk, azok megoldása ne igényeljen túlságosan sok új lexikai tudást, megismerésükre ne menjen el sok időt.

A cikk példaként a valós idejű, beágyazott rendszereket használja[8]. A *valósidejűség* jelentőségére a következő bekezdésben fogok részletesebben kitérni. A lényeg azonban az, hogy a közoktatásban is egyre gyakrabban alkalmaznak *beágyazott-*, *IoT rendszereket* (Raspberry Pi, Arduino, Micro:bit stb.) programozás oktatásra. Ennek nagyszerűsége és hasznossága számtalan cikkben és tananyagban előkerült már [19, 20, 21], én csak egy rövid összefoglalót kívánok adni:

1. Gyors, látványos, gyakorlati tapasztalattal járó élményt kínál a diákoknak
2. Egy tananyag modulárisan, spirálisan felépíthető az elérhető környezetek miatt
3. Számtalan programozási környezet elérhető a blokk alapú nyelvektől a kódolásig, pl.: Blockly, Microsoft MakeCode, Scratch, Logo, Python, C# stb.
4. Az általános iskola alsó tagozatától kezdve lehet oktatni, egészen a középiskola végéig és mindig új, izgalmas feladatokat lehet adni
5. A diákok többféle hardverrel, operációs rendszerrel, programozási környezettel ismerkednek meg tanulmányai végére, ami által teljesebb képet kapnak az informatikáról, az *informatikai rendszerekről*

---

[5] Zsakó László egyik előadásában [18] elkészített egy összefoglalót a hazai programozást érintő versenyekről

[6] Az [1] cikk részletesen megvizsgálta a kérdést, hogy a diákok menyire nincsenek tisztában azzal, mit csinál egy informatikus

[7] Ennek lehetőségére talán egy másik cikk majd válaszol.

[8] Bár nem tartom kizártnak, hogy más is legalább ennyire alkalmas lehet. Például a https://www.netsblox.org/ kollaborációra alkalmas környezetet biztosít még mindig blokkos elemekkel.

A cikk szempontjából talán az utolsó pont a legérdekesebb. Ez ugyanis azt jelenti, hogy ezeket az eszközöket használva egy olyan környezetet kapunk, mely a szokványos asztali-, vagy konzolos alkalmazások felépítésétől eltérő. Másrészt, mivel hardverekről is szó van, fizikailag létező komponensekről beszélhetünk, melyeket már csupán össze kell kötnünk valahogy, **rendszert kell csinálnunk belőlük**. Ez azért hasznos, mert ha pusztán szoftveresen szeretnénk összetett rendszert alkotni, ahhoz nagyon bonyolult feladatra lenne szükség (pl.: Dusza verseny feladatai). A valós idejű-, beágyazott rendszerek alkalmazásával kevés lexikai tudással, rövid forráskóddal, jelentős eredményeket érhetünk el. Mivel a fizikai komponensek + szoftveres környezet alkotja majd a rendszerünket a *csapatmunka* is hangsúlyossá válik, hiszen a megvalósítandó részelemek nem csak logikailag, de a valóságban is teljesen elkülönülnek.

7. Valós idejű rendszerek az oktatásban

A fentebb felsorolt rendszerek attól válnak *valósidejűvé*, hogy megszorításaink vannak arra, hogy egy-egy folyamatot mennyi idő alatt hajtsanak végre. Ezeket folyamatokat ezután rendszerint a valameddig megismételjük. Például egy szenzoradat begyűjtését valós időben szeretnénk továbbítani, vizualizálni, de egy motor vezérlése is valós időben kell, hogy megtörténjen (pl.: drón). Ha a *valósidejűség* szempontjából közelítjük meg a hardvereinket az osztályteremben, akkor lényegében okos-otthon projekteket kapunk. *„Az IoT (Internet of Things) elterjedésével, a smart otthonok, smart city program kiteljesedésével még inkább előtérbe kerül ez a terület, ahol az adatgyűjtés és azok azonnali feldolgozása szükségszerű."* [22] Amennyiben ezeket az interneten keresztül vezéreljük, például egy központi alkalmazásból, már is kaptunk egy IoT rendszert. Összességében tehát a projektünk a következő elemekből állnak:

1. A csoport létszámától függő a beágyazott rendszerek száma
2. A hardverek néhány kifejezett funkciót látnak el valós időben (pl.: szenzoradat begyűjtése, motor vezérlése stb.)
3. A hardverelemek az információt az interneten keresztül valós időben kapják vagy adják
4. Létezik egy központi alkalmazás, mely képes a modulok vezérlésére, adatok vizualizációjára

7.1 Az ötlet megvalósíthatósága

A programozás oktatása beágyazott rendszereken egyre elterjedtebb. Egyre több iskola engedheti meg magának, hogy Raspberry Pi vagy Arduino segítségével oktasson, ezzel ablakot nyitva az IoT világára. Az ötlet előfeltétele ezen eszközök elérhetősége. Ha van ilyen, akkor ez tulajdonképpen egy módszer, egy projekt annak alkalmazására.

Azért jól alkalmazható ez a rendszer, mert a programozási környezetben nagy mozgásterünk van. Például a Raspberry Pi-re készíthetünk Python-ban és C#-ban[9] is szoftvereket. Mivel egy szenzoradat, vagy egy motor vezérlése algoritmikusan nem nehéz feladat, ezért a plusz információ egy asztali alkalmazás elkészítéséhez képest (ha feltételezzük, hogy a diák a fenti nyelvek valamelyikében már tud programozni) csak pár utasítás, mely egy rövid dokumentációban összegyűjthető.

A valós idejű kommunikáció érdekében szükségünk lesz még egy *valósidejű* adatbázisra is. Ehhez a *Firebase* [24] rendszert ajánlom. Ez ugyanis egyéni felhasználásra, limitált kommunikációra ingyenes, használata pedig nagyon egyszerű. Az „adatbázis" szó talán ijesztően hangozhat, pedig egyáltalán nem az. Valójában nem is kommunikálunk közvetlenül adatbázissal, hiszen a szerviz API-án keresztül tudjuk elérni a platformot. A gyakorlatban ez pár új parancsot jelent, ahol szöveges adatokat lehet küldeni. Fontos megjegyezni, hogy a *Firebase* egy nem-relációs adatbázis, hanem egy JSON

[9] További nyelvek: C#, C++, Javascript, Visual Basic, Node.js[23]

fa. Amikor hozzáadunk egy új adatot, az egy csomópontként jelenik meg a JSON-ben egy kulccsal összekapcsolva. Ezek ismerete egyébként egyáltalán nem szükséges a diákok számára.

A következőkben 2 mintakódot mutatok be, demonstrálva a rendszer használatának egyszerűségét. A cikk formája nem alkalmas egy konkrét projekt részletes bemutatására, de számtalan példa elérhető a *Firebase* használatához. [25] Például az adatbázis Python kódból való inicializálása a következő sorokból adódik:

```
from firebase import firebase

firebase = firebase.
FirebaseApplication('https://adatbazisom.firebaseio.com/', None)
```

Egy adat pld. hőmérséklet lekérése a következő eljárással lehetséges. Ha az adatok folyamatosan frissülnek is, nekünk csak az elsőre van szükségünk. Az adatok a „message" csomópont alatt vannak.

```
def updateMessage():
    tempMessage = firebase.get('/message', None)
    if tempMessage is not None:
        tempMessage = tempMessage.values()[0].values()[0]
        print "Beerkezett uzenet: ", message
```

A *valósidejűség* azért jelentős, mert egy-egy adat frissülésének eseményére fel tudunk iratkozni egy eljárással, amely, az adatbázis adatainak megváltozásakor képes végrehajtani valamilyen műveletet.

7.2 A projekt és az architektúrális gondolkodás kapcsolata

Összességében tehát miért kapcsolódik ez a cikkben tárgyalt kompetenciához, miért alkalmas annak fejlesztésére? A fentebb felsorolt elemek önmagában egy komplex rendszert alkotnak. Persze az implementálás legalján az szerepel, hogy hogyan lehet például szenzoradatot beolvasni, de nyilván nem ez a feladat oroszlánrésze. Ha eltekintünk attól, hogyan lehet a konkrét technológiákat alkalmazni (ami egyáltalán nem nehéz, ráadásul az adatbázisok használata például elő kell, hogy kerüljön), a feladat a rendszer megtervezése (Ki/Mikor/Mit/Kivel kommunikál? Mik történnek, ha egy esemény bekövetkezik?).

A központi alkalmazás összességében adatok rendszerez, vezérel és ütemezési feladatokat lát el. Az architektúrája azonban egyértelműen megegyezik a leggyakrabban használt szoftveres architektúrával, az MVC-vel. Az érdekessége az, hogy ha lecsupaszítanánk a többi komponenstől, akkor egy igen egyszerű alkalmazást kapnánk. Képzeljük el például, hogy az adatokat egyszerű szöveges fájlban kapjuk, s a feladat ennek a fájlnak a menedzselése. Ha nem szeretnénk túl bonyolult vizuális megjelenítést, ez gyakorlatilag egy informatika érettségi programozói feladatával egyenértékű probléma.

8. Összegzés

A *rendszerszintű gondolkodás* kompetencia a jelenlegi középiskolai oktatásban nem igazán hangsúlyos. Ez azért probléma, mert a NAT-ban is előkerül közvetve ez a képesség, megléte pedig igen fontos a későbbi életben. A programozók esetében ez különösen igaz, hiszen bonyolult szoftver rendszerekkel is kell majd foglalkozniuk. Az *algoritmikus gondolkodás* fejlesztése pedig nem elegendő ahhoz, hogy valós képet adjunk erről a teendőről, hiszen rengeteg olyan probléma van, ami nem algoritmikusan nehéz. A programozás és a *rendszerszintű gondolkodás* ötvözetét *architektúrális gondolkodásnak* nevezem, ezzel is utalva a szoftverarchitektúrákra és az architect munkakörre. Fontos azonban kihangsúlyozni, hogy a *rendszerszintű gondolkodás* nem csupán az informatikus számára hasznos és fontos, hiszen számtalan más műveltségi területnél előkerül ez a fogalom (pl.: természettudományok, társadalomtudományok), ami igen jól definiálható és körül írható konkrét tantárgy vagy terület nélkül is.

Az oktatásba való integrálása ennek a kompetenciának egyáltalán nem egyszerű. A szakirodalomban a természettudománnyal kapcsolatban találtam csak cikket [12]. Jelen tanulmány az informatika oktatásban való megjelenítésére kíván ötletet adni. Véleményem szerint a *mikrokontrollerek*, *beágyazott rendszerek* elterjedésével, valamint az *IoT* térhódításával (oktatásban is) logikusan adódik ennek alkalmazása egy *valósidejű* megközelítésben.

Irodalom

1. Németh Tamás, Sárkány Rita, Tornai Henrietta, Wiandt Zsófia, Németh-Szabados Klára Viktória, Holló Csaba: A programozás oktatásának motivációi a közoktatásban In: Szlávi Péter, Zsakó László (szerk.) INFODIDACT 2016 ISBN: 978-615-80608-0-6
2. Nemzeti Alaptanterv. http://www.okm.gov.hu/
3. Szlávi Péter, Zsakó László: Informatikai kompetenciák - A valós világ modellezése In: Szlávi Péter, Zsakó László (szerk.) INFODIDACT 2013 ISBN:978-963-08-8387-0
4. Richmond, B. (1994). Systems Dynamics/Systems Thinking: Let's Just Get On With It. In International Systems Dynamics Conference. Sterling, Scotland
5. Benson, H., Borysenko, J., Comfort, A., Dossey, L., & Siegel, B. (1985). Economics, Work, and Human Values: New Philosophies of Productivity. The Journal of Consciousness and Change, 7(2), 198
6. Sweeney, L. B., & Sterman, J. D. (2000). Bathtub dynamics: initial results of a systems thinking inventory. System Dynamics Review, 16(4), 249–286. doi:10.1002/sdr.198
7. Stave, K. A., & Hopper, M. (2007). What Constitutes Systems Thinking? A Proposed Taxonomy. In 25th International Conference of the System Dynamics Society. Boston, MA
8. Kopainsky, B., Alessi, S. M., & Davidsen, P. I. (2011). Measuring Knowledge Acquisition in Dynamic Decision Making Tasks. In The 29th International Conference of the System Dynamics Society (pp. 1–31). Washington, DC.
9. Squires, A., Wade, J., Dominick, P., & Gelosh, D. (2011). Building a Competency Taxonomy to Guide Experience Acceleration of Lead Program Systems Engineers. In 9th Annual Conference on Systems Engineering Research (CSER) (pp. 1–10). Redondo beach, CA.
10. Arnold, R. D. és Wade, J. P. (2015): A Definition of Systems Thinking: A Systems Approach Procedia Computer Science, 44, 669–678.
11. Szlávi Péter, Zsakó László: Informatikai kompetenciák: Algoritmikus gondolkodás https://people.inf.elte.hu/szlavi/InfoDidact10/Manuscripts/ZsL_SzP.pdf (utoljára megtekintve: 2018.11.05)
12. Veres, G. (2017): Gondolkodási készségek azonosítása és fejlesztése a biológia tantárgyban tankönyvelemzés. In. Zsolnai Anikó és Kasik László (szerk.): A tanulás és nevelés interdiszciplináris megközelítése. Budapest, Magyar Tudományos Akadémia Pedagógiai Tudományos Bizottsága
13. Kovácsné Pusztai Kinga: A probléma–alapú oktatás az informatika órán In: Szlávi Péter, Zsakó László (szerk.) INFODIDACT 2017 ISBN: 978-615-80608-1-3
14. Mérő László: A mesterséges intelligencia és a kognitív pszichológia kapcsolata. Tankönyvkiadó (1989)
15. Mahler Attila: A magyar és a brit informatika kerettanterv 2012-es megújítása In: Szlávi Péter, Zsakó László (szerk.) INFODIDACT 2013 ISBN:978-963-08-8387-0
16. Dusza Árpás Országos Programozói Emlékverseny: https://isze.hu/dusza-arpad-orszagos-programozoi-emlekverseny/ (utoljára megtekintve: 2018. 11. 05.)
17. Bernát Péter: Feladattípusorientált játékfejlesztés a Scratchben In: Szlávi Péter, Zsakó László (szerk.) INFODIDACT 2017 ISBN: 978-615-80608-1-3
18. Zsakó László: Tehetséggondozás az informatikából: https://people.inf.elte.hu/szlavi/InfoOkt/Eloadasok/Tehetseggondozas.pdf (utoljára megtekintve: 2018. 11. 05.)
19. Módi József (2015): A Raspberry Pi számítógép a gyakorlati oktatásban

20. Michael Kölling (2016), Educational Programming on the Raspberry Pi (elérhető: https://www.researchgate.net/publication/304455824_Educational_Programming_on_the_Raspberry_Pi, utoljára megtekintve: 2018. 11. 05.)
21. Brian W. Evans írása alapján fordította, kiegészítette és frissítette Cseh Róbert (2011): Előszó, Arduino, Szerkezet – struktúra In: Arduino Programozási kézikönyv, TavIR (5-12. oldal) (elérhető: http://www.peschka.hu/userfiles/7/files/tavir_arduino_notebook.pdf, utoljára megtekintve: 2018. 11. 05.)
22. Heizlerné Bakonyi Viktória, Ifj. Illés Zoltán, Illés Zoltán: Valós időben, valós világban In: Szlávi Péter, Zsakó László (szerk.) INFODIDACT 2015 ISBN: 978-963-12-3892-1
23. Windows 10 IoT Core, Developing foreground applications: https://docs.microsoft.com/en-us/windows/iot-core/develop-your-app/buildingappsforiotcore (utoljára megtekintve: 2018. 11. 05.)
24. Firebase: https://firebase.google.com (utoljára megtekintve: 2018. 11. 05.)
25. Firebase samples: https://firebase.google.com/docs/samples/ (utoljára megtekimtve: 2018. 11. 05.)