https://doi.org/10.63684/dk.2026.01.01

HADARICSNÉ DUDÁS NÓRA ERZSÉBET*

A Vanilla JavaScript szerepe a programozói gondolkodás fejlesztésében

Összefoglalás: A cikk azt vizsgálja, hogy a keretrendszerek nélküli (Vanilla) JavaScript oktatása hogyan járul hozzá az egyetemi hallgatók programozói gondolkodásának fejlődéséhez. A modellképzéshez olyan gyakorlatokat és feladatkört alkalmaztam, amelyek az alapalgoritmusok kézzel történő implementálására, a JavaScript beépített függvényeire, az eseménykezelésre, a DOM-kezelésre és a JavaScriptben való rajzolásra koncentrálnak. A kutatás kvalitatív jellegű, a hallgatók gyakorlati munkájának és a forráskód-elemzésnek tapasztalati vizsgálatára épül. Nem tartalmaz mérhető pontszámokat vagy statisztikai elemzést, célja a Vanilla JavaScript-oktatás hatásainak minőségi feltárása. Eredményeim szerint a Vanilla JavaScript alapú oktatás növeli a nyelvi tudatosságot, javítja a hibakeresési készségeket és erősíti az algoritmikus gondolkodás kialakulását, ugyanakkor nagyobb kezdeti kognitív terheléssel jár.
Kulcsszavak: Vanilla JavaScript, programozói gondolkodás, algoritmikus gondolkodás, hibakeresés, DOM-manipuláció, eseménykezelés, vizuális visszacsatolás, kognitív terhelés.

Abstract: This article explores how teaching framework-free (Vanilla) JavaScript contributes to the development of programming thinking among university students. The instructional approach involved exercises and tasks designed to emphasize manual implementation of fundamental algorithms, utilization of JavaScript's built-in functions, event handling, DOM manipulation, and graphical programming in JavaScript. The study is qualitative in nature, relying on empirical examination of students' practical work and source code analysis. It does not employ measurable scores or statistical analysis; rather, its aim is to qualitatively investigate the effects of teaching Vanilla JavaScript. The findings suggest that this approach enhances language awareness, improves debugging skills, and fosters algorithmic thinking, though it is accompanied by a higher initial cognitive load.

* *Dunaújvárosi Egyetem, Informatikai Intézet, Szoftverfejlesztési és Alkalmazási Tanszék*
Email: dudasn@uniduna.hu
ORCID: 0009-0001-5825-4965

Keywords: Vanilla JavaScript, programming thinking, algorithmic thinking, debugging, DOM manipulation, event handling, visual feedback, cognitive load.

Bevezetés

A webfejlesztés az elmúlt évtizedben jelentős átalakuláson ment keresztül. A modern JavaScript-keretrendszerek, mint a React vagy Angular, a fejlesztési folyamatot magas szintű absztrakciók bevezetésével tették hatékonyabbá, és lehetővé teszik, hogy a fejlesztők gyorsan hozzanak létre összetett webalkalmazásokat. Ugyanakkor ezek az eszközök sok esetben elfedik a JavaScript nyelv valódi működését, így a kezdő fejlesztők gyakran anélkül alkalmazzák a keretrendszereket, hogy teljes mértékben megértenék a nyelv alapvető mechanizmusait. Ennek következtében a hallgatók a webprogramozás tanulása során sokszor „kész rendszerek" logikáját követik anélkül, hogy átlátnák a háttérben zajló folyamatokat, valamint a kód és a böngésző közötti kapcsolatot. Ez a helyzet vezette a fejlesztői közösséget is a Vanilla JavaScript kifejezés kialakításához, amely a nyelv natív, keretrendszerektől mentes használatát jelenti.

A Vanilla JavaScript kifejezés a JavaScript nyelv natív, keretrendszerektől és külső könyvtáraktól mentes változatát jelöli. A „vanilla" szó az egyszerű, alapformát jelenti, szemben a magas szintű, előre elkészített komponenseket kínáló keretrendszerekkel. A terminológia a fejlesztői közösségben alakult ki, és mára széles körben elfogadott a szakirodalomban és az oktatási gyakorlatban is. A Vanilla JavaScript lehetővé teszi, hogy a fejlesztők közvetlenül a böngésző által biztosított API-kon keresztül, a nyelv alapvető elemeit használva valósítsanak meg funkciókat, anélkül, hogy előre elkészített komponensekre vagy magasabb szintű absztrakciókra támaszkodnának. A Vanilla JavaScript csupán a natív eszközökre támaszkodik, és minden lépést a fejlesztőnek kell explicit módon megvalósítania.

A natív JavaScript oktatása különösen fontos a kezdő programozók számára, mivel így jobban megérthetik a kód és a böngésző viselkedése közötti kapcsolatot, valamint átfogó, transzparens tudást szerezhetnek a webes működés alapelveiről. Ez a tudás nem csupán a szintaktikai elemek elsajátítására korlátozódik, hanem a problémák strukturált, algoritmikus megközelítésének megértésére is kiterjed, elősegítve a kognitív modellek kialakulását. A hallgatók így képesek megérteni, hogy az általuk írt kód miként eredményez konkrét viselkedést a böngészőben, és hogyan épülnek fel a programozott interakciók, vizuális változások és adatkezelési folyamatok.

Az utóbbi években a Vanilla JavaScript újra a figyelem középpontjába került, mivel a minimalista megközelítés lehetőséget nyújt arra, hogy az oktatásban a hallgatók közvetlen kapcsolatba kerüljenek a nyelv működésével és a böngésző natív API-jaival. Ezáltal a tanulók nemcsak a konkrét kódolási technikákat sajátítják el, hanem a webes működés mögöttes logikáját is, ami hosszú távon stabil alapot biztosít a későbbi keretrendszeres fejlesztéshez.

A tanulmány célja annak feltárása, hogy a Vanilla JavaScript oktatása milyen módon járulhat hozzá a programozói gondolkodás fejlődéséhez, különös tekintettel az algoritmikus gondolkodás, a hibakeresési készségek és az önálló problémamegoldás támogatására. A kutatás arra is kiterjed, hogy feltárja a kezdő hallgatók kognitív terhelését, valamint azt, hogy a natív JavaScript gyakorlati alkalmazása milyen előnyökkel és kihívásokkal jár az oktatásban.

Elméleti háttér

A programozói gondolkodás (computational thinking) a problémák strukturált, algoritmikus megközelítését jelenti. A szoftverfejlesztés oktatásában nem csupán a nyelvi szintaktika elsajátítása a cél, hanem a gondolkodási minták, az absztrakciók és a logikai összefüggések megértése is központi szerepet játszanak. A modern JavaScript-keretrendszerek előnye, hogy csökkentik a fejlesztői terhelést és gyorsabb eredményeket tesznek lehetővé. Ugyanakkor az oktatási környezetben gyakran „fekete doboz-hatást" idéznek elő: a hallgatók a működést nem értik, csak alkalmazzák a kész komponenseket és függvényhívásokat.

Ezzel szemben a Vanilla JavaScript explicit módon tárja a tanuló elé az eseménykezelés, a DOM (Document Object Model) manipuláció, az aszinkron folyamatokat és az állapotkezelés mechanizmusait. A kód transzparenciája elősegíti a kognitív modellek kialakulását, vagyis azt, hogy a hallgató megértse: a kódja miként eredményez konkrét viselkedést a böngészőben.

Ennek következtében a Vanilla JavaScript-alapú oktatás elősegítheti a tartós, fogalmi tudás kialakulását, amelyre később akár keretrendszeres fejlesztés már biztonságosan építhető.

Kutatási célok

A fenti bevezető és az elméleti háttér alapján világossá válik, hogy a Vanilla JavaScript oktatása nem csupán a programozási nyelv szintaktikájának elsajátítását teszi lehetővé, hanem a hallgatók algoritmikus gondolkodásának, hibakeresési képességeinek és problémamegoldó stratégiáinak fejlesztésére is alkalmas. A tanulmány célja annak vizsgálata, hogy a keretrendszerektől mentes, natív JavaScript alkalmazása az oktatás során milyen mértékben járul hozzá a hallgatók programozói gondolkodásának fejlődéséhez, és milyen kognitív folyamatokat stimulál a tanulási környezetben.

A kutatás különös figyelmet fordít arra, hogy a hallgatók hogyan sajátítják el az alapalgoritmusok manuális implementálását, a DOM-manipulációt, az eseménykezelést, az adatvalidációt és a Canvas API használatát.

A cél az, hogy feltárjam, milyen módon segítik ezek a feladatok a kód és a böngésző közti kapcsolat megértését, az önálló problémamegoldást, valamint a strukturált gondolkodási minták kialakulását.

Hipotézis:
A Vanilla JavaScript oktatása elősegíti a JavaScript nyelv működésének mélyebb megértését, mivel a hallgatók közvetlenül láthatják a kód és a böngésző viselkedése közti kapcsolatot. Ennek következtében javul az algoritmikus gondolkodásuk, fejlődik a hibakeresési képességük, és erősödik az önálló problémamegoldó készségük. Ugyanakkor a natív JavaScript használata kezdetben nagyobb kognitív terhelést jelent, mivel minden funkciót és mechanizmust a nyelv és a böngésző natív eszközein keresztül kell megérteniük, külső keretrendszerek vagy könyvtárak nélkül.

Ez a kutatási megközelítés lehetővé teszi, hogy a vizsgálat eredményei a Vanilla JavaScript szerepét ne csupán a gyakorlati kódolási képességek fejlesztésében, hanem a mélyebb programozói gondolkodás és a problémamegoldó stratégiák kialakulásában is mérhetően bemutassa.

Módszertan

A vizsgálat során egyetemi hallgatókkal dolgoztam, akik egy féléves *Internet technológiák* című kurzus keretében először a HTML5 és CSS3, majd a Vanilla JavaScript alapjaival ismerkedtek. A kurzus első felében a hangsúly az oldalak szerkezetének felépítésén és a vizuális megjelenítés alapjainak elsajátításán volt. Világossá vált mindenki számára, hogy a HTML adja a weboldal szerkezetét, a CSS a megjelenést, míg a JavaScript a működést és a logikát biztosítja.

A félév második felében a JavaScript került előtérbe, és a feladatokat úgy alakítottam ki, hogy fejlesszék a hallgatók programozói gondolkodását, a nyelvi tudatosságot és a hibakeresési készségeket. A hallgatók gyakorolták az alapalgoritmusok megvalósítását, az egyszerű logikai feladatok és számítási műveletek implementálását, valamint a JavaScript beépített függvényeinek használatát. Emellett a DOM-manipuláció és az eseménykezelés révén megtanulták, hogyan érhetik el az oldal elemeit, módosíthatják, eltávolíthatják vagy új elemekkel bővíthetik a weboldalt, és hogyan reagálhat a felhasználói interakciókra.

A grafikus megjelenítés és vizualizáció terén a hallgatók a <canvas> HTML elemen belül a JavaScript Canvas API-t használták rajzolásra, ami lehetővé tette a program logikájának és az eseménykezelés vizuális szemléltetését. A formok és adatvalidációs feladatok során a hallgatók a felhasználói bevitel ellenőrzését és a hibakezelést gyakorolták. Továbbá az idő- és dátumkezelés révén megtanulták az aktuális idő és dátum lekérését, azok formázását, valamint a programozott események és vizuális változások (például háttérszín váltása) kezelését.

A vizsgálat kizárólag kvalitatív megközelítést alkalmazott: a hallgatók kódját, a problémamegoldási stratégiáikat és a gyakorlati órák során szerzett megfigyeléseket elemeztem. A cél nem a mennyiségi mérés, hanem a programozói gondolkodás fejlődésének minőségi feltárása. Minden feladatot külön értékeltem, figyelve az algoritmusok szerkezetére, a hibakezelési stratégiák alkalmazására, a DOM-manipuláció és eseménykezelés helyességére, valamint a Canvas API használatára. Részletesen vizsgáltam, hogy a hallgatók miként közelítik meg a problémákat, milyen logikai lépéseket követnek, és milyen módszerekkel próbálnak hibákat kijavítani. A gyakorlati órák során folyamatosan jegyzeteltem a hallgatók problémamegoldási viselkedését és a kognitív terhelés jeleit, például a bizonytalanságot, segítségkéréseket vagy a kód iteratív módosításait.

Ez a kvalitatív megközelítés lehetővé tette, hogy részletesen értékeljem a nyelvi tudatosság, a hibakeresési készségek és az algoritmikus gondolkodás fejlődését, valamint azt, hogy a hallgatók a megszerzett tudást később biztonságosan alkalmazhassák bármilyen fejlettebb fejlesztési környezetben. A kvalitatív elemzést a hallgatók által írt forráskód és a gyakorlati órák során szerzett megfigyelések alapján végeztem.

Az alábbi táblázat összegzi a kvalitatív elemzés fő szempontjait, a vizsgált készségeket és az értékelés kritériumait, hogy áttekinthető módon látható legyen, mely feladatok hogyan járultak hozzá a hallgatók programozói gondolkodásának fejlődéséhez.

1. táblázat. Kvalitatív elemzés szempontjai

Elemzés szempontja	Vizsgált aspektus	Megfigyelt viselkedés/értékelési kritérium
Algoritmusok szerkezete	Hogyan építik fel a hallgatók a kódot, ciklusok, feltételes utasítások	Logikai sorrendiség, hatékony szerkezet, helyes lépések
Hibakeresés	Hogyan észlelik, majd javítják a kód hibáit	Kód-iteráció, hibák azonosítása, javítási stratégiák
DOM-manipuláció	HTML elemek elérése, módosítása, eltávolítása, új elem létrehozása	Megfelelő elemválasztás, események helyes kezelése
Eseménykezelés	Felhasználói interakciók kezelése (pl. gombnyomás, adatbevitel)	Helyes esemény-hozzárendelés, válasz a felhasználói műveletekre
Canvas API	Rajzolási feladatok, vizuális visszajelzés	Kód és megjelenítés kapcsolata, logikai összefüggések vizualizálása
Adatvalidáció	Űrlapok helyes kitöltésének ellenőrzése, reguláris kifejezések használata	Érvényes adatok ellenőrzése, hibajelzés implementálása
Idő- és dátumkezelés	Aktuális dátum/óra lekérése, formázás, vizuális változások	Feltételes logika, aszinkron események kezelése, kódmegjelenítés-kapcsolat

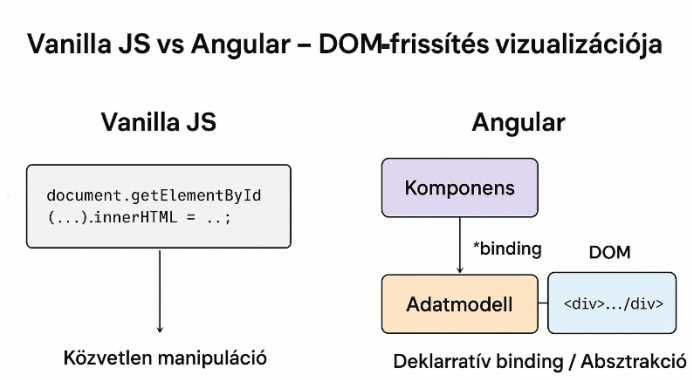
A táblázatban összegzett szempontok a hallgatók kognitív és gyakorlati készségeit tükrözik, és a kvalitatív elemzés fő irányvonalait szemléltetik. Az elemzési szempontok alapján világossá vált, hogy a hallgatók készségei és gondolkodási mintázatai jól nyomon követhetők voltak a feladatmegoldások során.

Az algoritmusok szerkezete és a hibakeresés vizsgálata az algoritmikus gondolkodás és a problémamegoldó képesség fejlődését, míg a DOM-manipuláció, az eseménykezelés és a Canvas API használata a kód–megjelenítés kapcsolatának megértését támogatta. Az algoritmusok manuális implementálása segít fejben modellezni a logikai lépéseket és a folyamatok sorrendjét. Különösen hatékony, ha ezt algoritmusleíró eszközökkel, például pszeudókóddal vagy folyamatábrával szemléltetjük.

Az eseménykezelés révén a hallgatók megértik, hogyan kezeli a program a felhasználói műveleteket és az aszinkron eseményeket. A DOM-manipuláció tanulása lehetővé teszi, hogy a hallgatók megértsék, hogyan jelennek meg a HTML-elemek, hogyan módosíthatók, és milyen hatással vannak a felhasználói interakciókra. Az adatvalidáció, valamint az idő- és dátumkezelési feladatok különösen a logikai gondolkodás, a feltételes utasítások és az aszinkron események kezelésének gyakorlását tették lehetővé, miközben azonnali visszajelzést adtak a kód működéséről.

A webfejlesztés során például a felhasználói felület manipulációja két alapvető paradigmát követhet: a közvetlen, imperatív DOM-manipulációt Vanilla JavaScript-ben, illetve a deklaratív, komponensalapú frissítést Angularban. Az *1. ábra* vizuálisan szemlélteti ezen két megközelítés közti különbséget, kiemelve, hogy a fejlesztői beavatkozás és a DOM-változás miként történik a gyakorlatban.

1.ábra. A DOM-manipuláció két paradigmája



Az ábra két megközelítést hasonlít össze a webes felületek frissítésében: a közvetlen DOM-manipulációt Vanilla JavaScript (JS) használatával, valamint az Angular-keretrendszer deklaratív adatbindingjét.

Bal oldal – Vanilla JavaScript

A bal oldalon a DOM-hoz való közvetlen hozzáférés látható, például document.getElementById(...).innerHTML = ... használatával. Ebben a megközelítésben a JavaScript azonnal módosítja a kiválasztott HTML elem tartalmát. A fejlesztő feladata, hogy minden változást explicit módon kezeljen a DOM-ban. Ez az imperatív stílus egyszerű, de nagyobb projektek esetén nehezen karbantartható, mivel a DOM frissítésének minden lépését kézzel kell végrehajtani.

Jobb oldal – Angular

A jobb oldalon az Angular deklaratív megközelítése látható. Itt a logikai egység a komponens, amely tartalmazza az adatmodellhez kapcsolódó adatokat és viselkedést. A DOM frissítése az Angular adatbinding mechanizmusán keresztül automatikusan történik: amikor a komponens adatmodellje megváltozik, a keretrendszer a megfelelő HTML elemeket frissíti, anélkül, hogy a fejlesztőnek közvetlenül kellene módosítania a DOM-ot. Ez a deklaratív stílus növeli a karbantarthatóságot és csökkenti a hibalehetőségeket.

Ezek az eredmények megalapozzák a következő fejezetben részletezett elemzést, amely bemutatja, milyen konkrét hatással volt a Vanilla JavaScript-alapú oktatás a hallgatók programozói képességeire és problémamegoldó stratégiáira.

Eredmények

A hallgatók kódjainak kvalitatív elemzése és a gyakorlati órákon szerzett megfigyelések alapján világossá vált, hogy a Vanilla JavaScript-alapú feladatok erősítették az algoritmikus gondolkodást. Különösen az alapalgoritmusok kézi implementálása során, amikor a hallgatóknak saját logikai és számítási lépéseiket kellett meghatározniuk. Ezt a korábbi kutatások is alátámasztják, amelyek szerint az algoritmusok manuális implementálása elősegíti a strukturált problémamegoldást és a programozói gondolkodás fejlődését [1] [2].

[1] Grover, S.–Pea, R. (2013): Computational thinking in K–12: A review of the state of the field. *Educational Researcher,* 42., (1.), pp. 38–43. (Teljes cikk elérhető PDF ben: https://multimedia.uoc.edu/carlos/chipro/wp-content/uploads/2013/10/38.full_.pdf)

[2] Taub, R.–Armoni, M.–Ben Ari, M. (2012): CS Unplugged and middle school students' views, attitudes, and intentions regarding CS. *ACM Transactions on Computing Education,* 12., (2.). (Teljes cikk elérhető PDF ben: https://www.researchgate.net/publication/241623893_CS_Unplugged_and_Middle-School_Students%27_Views_Attitudes_and_Intentions_Regarding_CS)

A beépített függvények és a DOM-manipuláció gyakorlása hozzájárult a nyelvi tudatosság fejlődéséhez. A hallgatók egyre biztosabban tudták alkalmazni a nyelv beépített eszközeit, és felismerni azok működését a böngészőben, ami a későbbi hibakeresést és a kód transzparenciáját is elősegítette. Az eseménykezelési feladatok révén a hallgatók megtanulták, hogyan reagáljon a program a felhasználói interakciókra, valamint hogyan kezeljék az aszinkron eseményeket és a vizuális visszajelzéseket.

A Canvas API használata különösen hatékony volt a program logikájának vizuális szemléltetésében. A hallgatók egyszerű rajzolási feladatok során (például négyszögek, karikák, vonalak, változó sugarú karikák, valamint az olimpiai ötkarika rajzolása) közvetlenül láthatták, hogyan jelennek meg a rajzelemek a kód hatására a képernyőn. Ez elősegítette a programozói gondolkodás és a kód–megjelenítés kapcsolatának megértését, valamint növelte a kognitív megértés mélységét.

Az űrlapok és adatvalidáció feladatok során a hallgatók megtanulták a bevitt adatok ellenőrzését és a hibakezelést, valamint a programozott logika alkalmazását a valós felhasználói interakciók kezelésére. A feladatok során a hallgatóknak különböző típusú mezőket kellett validálniuk: a névmezőben csak betűk engedélyezettek, az email mezőben a bevitt szövegnek érvényes email-formátumnak kellett megfelelnie, az irányítószámnak pontosan négy számjegyből kellett állnia, a telefonmezőben pedig a +36 XX XXX XXXX formátumot kellett ellenőrizniük. Ezekhez a validációkhoz reguláris kifejezéseket alkalmaztak, ami lehetővé tette a bevitt adatok formátumának pontos és automatizált ellenőrzését.

Emellett a hallgatóknak a valós idejű ellenőrzést is implementálniuk kellett, hogy a felhasználó ne tudjon érvénytelen karaktereket beírni, valamint a submit gombnál ellenőrizni, hogy minden mező helyesen legyen kitöltve, és szükség esetén hibaüzenetet jelenítsen meg. Ezek a feladatok különösen erősítették a problémamegoldó képességet, a logikai gondolkodást és a programozói tudatosságot, mivel a hallgatóknak figyelembe kellett venniük a felhasználói interakciókat és az adatfeldolgozás helyes sorrendjét a webes környezetben.

Az idő- és dátumkezelési feladatok többféle készséget és gondolkodási mintázatot fejlesztettek. A hallgatóknak meg kellett érteniük, hogyan épülnek fel a dátum- és időadatok (év, hónap, nap, óra, perc, másodperc), és figyelembe kellett venniük a programozott események sorrendjét és függőségeit, például amikor a háttérszín változott. A feladatok során ciklusokat, feltételes utasításokat és függvényeket kellett alkalmazniuk, például a hét napjának meghatározására vagy ismétlődő vizuális változások kezelésére, miközben a setTimeout használatával az aszinkron események működését is gyakorolták.

A hallgatók megtanulták, hogyan lehet a dátumot és az időt különböző formátumokban megjeleníteni, például rövid dátumként vagy a hét napját szövegesen, és hogyan biztosítható a vizuális visszajelzés a kód hatására, ami közvetlen kapcsolatot teremtett a programozói logika és a látvány között. A feladatok kombinálták a dátumkezelést, a feltételes logikát és a vizuális megjelenítést, így erősítve a problémamegoldó készséget, a kódolási tudatosságot és a hibakeresési képességeket, mivel minden apró logikai vagy szintaktikai hiba azonnal látható eredményeltérést okozott.

Összességében a kutatás eredményei azt mutatják, hogy a Vanilla JavaScript-alapú oktatás jelentősen hozzájárul a hallgatók önálló problémamegoldó képességének fejlődéséhez, javítja a hibakeresési készségeket, és erősíti az algoritmikus gondolkodást. Ugyanakkor a tanulási folyamat kezdeti szakasza nagyobb kognitív terhelést jelent, mivel a hallgatóknak minden mechanizmust a nyelv és a böngésző natív eszközein keresztül kell megérteniük, külső keretrendszerek vagy könyvtárak nélkül.

Ezek a megfigyelések összhangban vannak azzal a feltételezéssel, hogy a keretrendszerek nélküli, natív JavaScript oktatás hosszú távon stabilabb alapot biztosít a hallgatóknak, mivel a megszerzett tudásra később bármilyen magasabb szintű fejlesztési környezet biztonságosan építhető.

A webfejlesztésben a felhasználói felület (UI) manipulációja két alapvető paradigma mentén történhet: közvetlen DOM-manipuláció (Vanilla JS) és reaktív, komponens-alapú architektúra (például Angular).

Vanilla JavaScript-paradigma

A hagyományos JavaScript-megközelítésben a fejlesztő közvetlenül a DOM objektumait módosítja. Ez imperatív jellegű paradigma, azaz a fejlesztő pontosan meghatározza a DOM-frissítés lépéseit. Az állapot- és eseménykezelés manuálisan történik, ami kis projektek esetén egyszerűen követhető, ugyanakkor bonyolultabb alkalmazásoknál a kód olvashatósága és karbantarthatósága csökkenhet.

Angular paradigma

Az Angular reaktív, komponensalapú modellt alkalmaz, ahol a DOM-frissítés közvetetten történik az adatmodellek és sablonok közötti binding mechanizmus által. A változásészlelés automatikusan szinkronizálja a felhasználói felületet az alkalmazás állapotával. Ez a deklaratív megközelítés lehetővé teszi komplex interakciók egyszerűbb kezelését, magasabb szintű absztrakciót biztosítva, miközben csökkenti a hibalehetőségeket.

2.táblázat. Összehasonlító megfigyelés

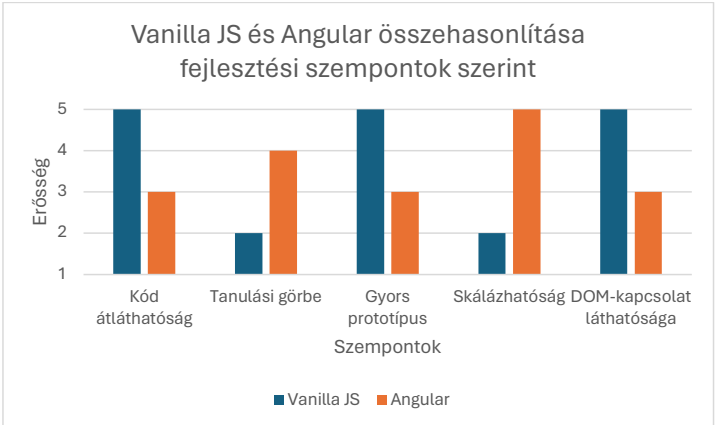
Szempont	Vanilla JS	Angular
Kód átláthatósága	Magas: minden lépés látszik	Közepes: absztrakció mögött a DOM kezelése
Tanulási görbe	Alacsony-moderált	Magas: CLI, TypeScript, modulok
Gyors prototípus	Könnyű kis projekteknél	Lassabb setup miatt kezdetben
Skálázhatóság	Kis projektekhez ideális	Nagy alkalmazásokhoz optimalizált
Kód- és DOM-kapcsolat	Közvetlen, vizuálisan látható	Absztrakt, Angular kezeli

A táblázat alapján a Vanilla JS magas átláthatóságot biztosít, mivel minden kódlépés közvetlenül látható, és a DOM-manipuláció explicit módon történik. Angular esetében a DOM kezelése absztrakció mögé van rejtve, ami csökkenti a közvetlen láthatóságot, ugyanakkor strukturáltabb és moduláris felépítést biztosít. A tanulási görbe szempontjából a Vanilla JS alacsony-moderált, kezdő vagy kisebb projektek esetén gyorsan alkalmazható, míg Angular komplexitása (CLI, TypeScript, moduláris architektúra) kezdetben meredekebb tanulást igényel, hosszú távon azonban előnyös a nagyobb, skálázható alkalmazások fejlesztéséhez.

A prototípus-készítés során a Vanilla JS gyors implementációt tesz lehetővé kis projektekben, míg Angularban a kezdeti setup lassíthatja a fejlesztést. Skálázhatóság tekintetében a Vanilla JS kis projektekhez ideális, Angular viszont nagy alkalmazásoknál nyújt optimalizált és strukturált megoldást. A kód és a DOM közötti kapcsolat is eltérő: Vanilla JS-ben a változtatások közvetlenül, vizuálisan követhetők, Angularban a DOM frissítése absztrakt módon történik, a keretrendszer automatikus frissítései által.

A következő diagram a Vanilla JavaScript és az Angular főbb jellemzőit hasonlítja össze az oktatás és fejlesztés szempontjából. Minden tengely egy-egy kritikus aspektust reprezentál a *2. ábra* alapján: kód átláthatóság, tanulási görbe, gyors prototípus-készítés, skálázhatóság és a kód–DOM-kapcsolat, így vizuálisan érzékelteti az erősségek és korlátok különbségeit.

2. ábra. Vanilla JS és Angular összehasonlítása kulcsfontosságú szempontok alapján



Összességében az összehasonlítás rávilágít arra, hogy a Vanilla JS gyors, közvetlen és könnyen átlátható, ezért különösen alkalmas oktatási környezetben történő kezdeti tanulásra. Az oktatás során a hallgatók nem komplex projektek fejlesztésére fókuszálnak, és a tanórai keretek szűkös ideje nem teszi lehetővé az Angular mélyreható elsajátítását. Ezért a programozás alapjainak megtanításánál és a kódolási készségek fejlesztésénél a Vanilla JS használata célszerű, hiszen lehetővé teszi a logikai gondolkodás, a DOM-kezelés és az eseménykezelés alapjainak hatékony elsajátítását.

Következtetés

A kutatás eredményei azt mutatják, hogy a Vanilla JavaScript-alapú oktatás jelentősen hozzájárul a hallgatók programozói gondolkodásának és problémamegoldó képességeinek fejlődéséhez. A tanulmány során tapasztalt kezdeti kognitív terhelés arra utal, hogy a hallgatók számára kihívást jelent a nyelv és a böngésző natív mechanizmusainak elsajátítása. Ugyanakkor hosszú távon ez a megközelítés stabil, mélyebb tudásbázist biztosít, amelyre később akár keretrendszerek is biztonságosan építhetők.

A Vanilla JavaScript oktatásának konkrét előnyei a következők:
– Javuló nyelvi tudatosság és kódtranszparencia.
– Erősödő önálló problémamegoldó képesség.
– Strukturált algoritmikus gondolkodás kialakulása.
– Valós idejű vizuális visszacsatolás a kód működéséről.

Összességében a Vanilla JavaScript-oktatás nem csupán a technikai készségeket fejleszti, hanem alapvetően támogatja a programozói gondolkodás és a strukturált problémamegoldás kialakulását. Ez a megközelítés hosszú távú előnyt biztosít a hallgatók számára a modern webfejlesztési keretrendszerek használatában, mivel a natív szintű tudás lehetővé teszi a komplex rendszerek biztonságos és hatékony kezelését.

A kutatás korlátozásai között szerepel, hogy a vizsgálat kizárólag tanórai keretek között, kezdő hallgatók körében zajlott, és a vizsgált feladatok kis projektekre korlátozódtak. Emiatt az eredmények általánosíthatósága nagyobb, komplex fejlesztési környezetekre korlátozott lehet.

Jövőbeli kutatások során érdemes vizsgálni a Vanilla JavaScript-alapú oktatás hosszú távú hatását, illetve összehasonlítani a hallgatók teljesítményét különböző keretrendszerek (pl. Angular, React, Vue) bevezetésével, hogy feltárható legyen, hogyan befolyásolja az alapok elsajátítása a későbbi fejlesztési képességeket.