Tesztelési módszerek webes tárgyak tanításában

Horváth Győző[1], Visnovitz Márton[2]

{[1]horvath.gyozo, [2]visnovitz.marton}@inf.elte.hu
ELTE IK

Absztrakt. A tesztelés fontos részét képezi a módszeres feladatmegoldás lépéseinek. Ahogy az iparban is, úgy a programozásoktatásban is egyre hangsúlyosabban jelenik meg ez a témakör kezdve már a bevezető kurzusoktól. Az ELTE Informatikai Karának új képzési tervének keretében megújuló webes tárgyak vonatkozásában is szeretnénk a tesztelést markánsabban megjeleníteni. Ebben a cikkben azt járjuk körül, hogy az iparban ezen a területen használatos eszközök és módszerek közül mi és hogyan jelenhet meg az oktatásban úgy, hogy minél kisebb energiabefektetéssel minél jobban lássák a hallgatók a tesztelés előnyeit.

Kulcsszavak: programozásoktatás, tesztelés, webfejlesztés

1. Bevezetés

Az egyetemi programozásoktatásban a feladatmegoldást módszeres lépésekre bontjuk annak érdekében, hogy a tanulók segítő eszközt kapjanak tetszőleges számítógépes probléma megértéséhez, feldolgozásához és sikeres teljesítéséhez. Ezeknek a módszeres lépéseknek egyik fontos eleme a tesztelés. A feladat meghatározása (specifikáció, követelményspecifikáció), tervezése (algoritmusok, modellek) után implementált program helyességéről a tesztelés révén győződhetnek meg a hallgatók. Többek között azért lehet fontos ez számukra, hiszen elkészült munkáik helyessége általában jelentősen befolyásolhatja érdemjegyeiket. Hosszabb távlatokban gondolkodva az egész informatikai iparág számára fontos, hogy olyan szakemberek kerüljenek ki a felsőoktatási intézményekből, akik egy alkalmazás teljes életciklusát képesek végig követni, és minőségi munkát tudnak végezni a határidők betartásával. Ennek részét képezi – a megfelelő tervezési lépések és az implementációs technológia magabiztos ismerete mellett – a tesztelés is.

Az ELTE programtervező informatikus képzésében a tesztelés a kezdő, bevezető programozási kurzusoktól jelen van különböző formákban. Az első féléves programozás (korábban programozási alapismeretek) tárgyban a tesztelés témakör különböző formákban és eltérő hangsúlyokkal van jelen. Az előadásban külön hetet szánnak a tesztelés, hibakeresés, hibajavítás témakörének, ahol a hallgatók megismerik az elméleti tudnivalókat [1]. Gyakorlatokon azonban a módszeres tesztelés sokszor kisebb hangsúlyt kap a többi ismeret mellett. A hallgatóknak ebben a fázisban sokféle új ismerettel kell megküzdeniük: a specifikációs nyelvvel, az algoritmusleíró nyelvvel, a programozási nyelvvel. Mire a feladatmegoldásban idáig érnek, a módszeres tesztelésre sokszor se idő, se energia nem marad. Ha a tesztesetek átbeszélésre kerülnek is, a tesztelés manuális volta akadályozza ezt a lépést. Ezek mellett a hallgatók hangsúlyosan a számonkérésben találkoznak a teszteléssel, hiszen ebben az esetben egy értékelő rendszer az elkészült programjaikat input-output tesztelésnek veti alá, azaz az egyes bemenetekhez tartozó helyes kimeneteket veti össze a hallgató programja által generált kimenetekkel. Ez a megközelítés a hallgató programjának egészét tekinti egy inputról outputra képező tiszta függvénynek. A kezdő tárgyra épülő objektumelvű programozás (korábban programozás) tárgyban megjelenik a függvények egység- és integrációs tesztelése is [2]. A hallgatók további elméleti instrukciókat kapnak a tesztesetek előállítására, és egy egyszerű eszközön keresztül az automatikus tesztelést is elvárják tőlük.

Ilyen alapokkal érkeznek a hallgatók az Informatikai Kar megújult BSc képzésének egyik szakirányában kötelezően helyet kapó webprogramozás tárgyakra. Ezek közül az egyik a webprogramozás alapjaival foglalkozik, bemutatva a kliens- és szerveroldali dinamikus weboldalak

készítéséhez szükséges technológiák alapjait. A későbbi tárgyak erre az alapra építve ismertetik meg a hallgatókkal a modern kliens- és szerveroldali alkalmazások elveit, eszközkészletét, technológiáit, ami végül egy integráló tárgy keretében zárul le, ahol egy full-stack alkalmazás fejlesztésével tehetnek tanúbizonyságot a hallgatók mindkét oldalon szerzett ismereteikről.

Ugyan a képzés során a hallgatók dedikált tárgy keretében találkoznak a modern szoftverfejlesztési eszközökkel és módszertanokkal, azonban ahogy a fentebb felsorolt programozási tárgyak esetében is megjelentek ezek az ismeretek megfelelő mértékben, úgy szeretnénk, ha az új webes tárgyak oktatásának is szerves részét képezné a tesztelés témaköre. Ennek a cikknek az a célja, hogy bemutassa, milyen megoldásokat használnak az iparágban webes alkalmazások teszteléséhez, és ezek hogyan jelenhetnek meg az egyes tárgyak oktatásában.

2. Professzionális tesztelési megoldások

Szoftverek tesztelésének rengeteg megközelítése, típusa és technikája van, ezeknek a részletes ismertetésére ez a cikk nem vállalkozhat. A tesztelést legtöbbször az *alkalmazás tesztelendő szintje* szerint szokták kategorizálni. Ezeket is figyelembe véve a helyes alkalmazás készítését a következő tesztek biztosítják:

1. *Statikus kódelemezés* használata: a megfelelő eszközök segítségével kiszűrhetők a szintaktikus hibák, az elírások és a rossz típusokból fakadó hibák.
2. *Egységtesztelés*: az alkalmazás önálló, izolált részei helyes működésének szeparált ellenőrzése. A tesztelés legelterjedtebb, leggyakrabban használt formája. Egységtesztből nagyon sok lehet, ezért nagyon fontos, hogy gyorsan lefussanak. A tesztelendő kis egységek általában függvények vagy osztályok. Az elkülönült viselkedés ellenőrzése érdekében az a jó, ha a függvények tiszta függvények és az osztályoknak nincsen külső függősége. Ha mégis megjelenik ilyen függőség, akkor azokat helyettesíteni kell. Bevált gyakorlat osztályok esetében, hogy direkt függőségek kialakítása (pl. öröklés vagy adattagként megjelenő példány) helyett, a függőségeket paraméterként tudjuk az osztály számára átadni (kompozíció).
3. *Integrációs tesztelés*: annak ellenőrzése, hogy az előzőekben tesztelt kis egységek jól működnek együtt. Itt már nem cél a függőségek helyettesítése, sőt éppen ezek együttes eredményét szeretnénk vizsgálni. Az integrációs tesztek általában szintén gyorsak, ha azonban az integrációba hálózati forgalmat, adatbázis működést is tesztelnek, akkor ezek a tesztek már lassabbak lehetnek. Az integrációs tesztek írásához sokféle megközelítést lehet alkalmazni, ezek közül emeljünk ki kettőt, amely az oktatásban is megjelenhet:
   a) *lentről felfele tesztelés*: itt először a legalsó szinteken lévő komponensek helyes működéséről győződnek meg, majd felhasználják ezeket a magasabb szinten lévő komponensek tesztelésére;
   b) *fentről lefele tesztelés*: az előző fordítottja, először a legmagasabb szinten lévő komponens helyes működéséről győződnek meg, majd azokat a komponenseket ellenőrzik, amelyektől az előző komponens függ. Hasonló megközelítést alkalmaztak a programozási tételek tesztelésénél [2].
4. *Funkcionális tesztelés* (end-to-end tesztelés): felhasználói tevékenységeket ellenőrző folyamat, azt vizsgálja, hogy az alkalmazás helyesen működik-e a felhasználó szempontjából. Tekinthetjük a legmagasabb szintű integrációs tesztnek, hiszen ebben az esetben az alkalmazás egésze kerül vizsgálat alá. Jellegéből fakadóan nagyon lassú a többi teszthez képest, ugyanakkor felderíthet olyan hibákat, amelyeket az előzőek nem képesek. Míg az egységteszteket fejlesztés közben folyamatosan futtatják, addig a funkcionális tesztek általában a publikálás előtt futnak.
5. *Hatékonysági tesztek*: mennyire hibatűrő az alkalmazás, hogyan viselkedik terhelés alatt.

Sokféle *tesztelési módszertan* terjedt el az utóbbi időben. A klasszikus *vízesés modell* szerint a tesztelést a fejlesztés után végzik el. Az agilis módszertanok legtöbbje a *tesztvezérelt fejlesztést*

propagálja több ok miatt is: egyrészt jobban megfogalmazható a tesztelendő funkció publikus interfésze használat oldalról, másrészt csak az kerül lefejlesztésre, amire szükség is van, végül sokkal nagyobb tesztlefedettség érhető el ezáltal, sok teszt pedig az alkalmazás helyes működését biztosítja.

Az *eszközök*et illetően már konkrétan a webes megoldásokat kell vizsgálnunk. Két nagy területe van a webprogramozásnak: a kliensoldali és a szerveroldali programozás. Ezt a két területet érdemes külön vizsgálni, mert nagyon eltérő megközelítésűek lehetnek. A *szerveroldali technológiák* és programtervezési minták történeti okok miatt sokkal érettebbek, így tesztelésük is kiforrottabb. Nyelvtől függően léteznek statikus kódelemzők és xUnit alapú könyvtárak. Terheléses tesztek és HTTP kérések tesztelése is könnyen lehetséges.

A *kliensoldal* mindig is gyorsabban változó és így ingoványosabb terület volt. Tesztelés szempontjából ez nemcsak a technológia miatt alakult így, hanem azért is, mert a felhasználóifelület-tesztelés eleve bonyolultabb egy backend tesztelésnél. A JavaScript nyelvhez léteznek *statikus kódelemző eszközök* (pl. ESLint), dinamikusan típusos nyelv lévén azonban a típushibákat ez felderíteni nem tudja. Éppen ezért sokan ajánlják a JavaScript nyelv típusos kiegészítéseit, ilyen a TypeScript és a Flow. Az *egységtesztelés* régebben kizárólag böngészőben történt, az egyetlen olyan platformként, ahol JavaScript fut. A Node.js projekt megjelenésével azonban a kód tesztelése parancssorban is elérhetővé vált. Sokat lendített a JavaScript egységtesztelésén az is, hogy a nyelv ma már támogatja a modulokat, így a kódot funkcionális egységekre bonthatjuk. Sajnos a modulkezelés egyelőre nem egységes a böngészőben és a parancssorban, így egy köztes átalakítási fázis szükséges, amely bonyolítja a használt eszközkészletet.

Az utóbbi időben a felhasználói felület kialakítására az ún. komponensalapú fejlesztés terjedt el. Ennek használatával a *felhasználói felület tesztelése* is egyszerűbbé vált. A komponensek ugyanis úgy viselkednek, mint a tiszta függvények: a kapott adathoz HTML-t rendelnek hozzá. Tesztelésükhöz böngésző sem kell, elég csak a generált struktúrát valamilyen könnyen kezelhető adatszerkezetben tárolni, és csak ezt kell ellenőrizni. Az utóbbi időben a felületek ellenőrzésére elterjedt az ún. pillanatkép tesztelés, ahol korábban generált struktúrát vetnek össze az újabb futásokkor.

Funkcionális tesztelés során az alkalmazást különböző operációs rendszerek különböző böngészőiben kell futtatni. Ennek automatizálásához sokszor egy bonyolult szerver-kliens rendszer kiépítésére van szükség, ahol a szerver az egyes klienseken elinduló böngészőket távolról vezérli. Ez korábban szinte kizárólagosan a Selenium Webdriver segítségével történt. Az utóbbi időben ezt több alternatíva váltotta fel. A gyors funkcionális tesztelés érdekében egyes megoldások szimulált böngészőkörnyezettel dolgoznak (jsdom), mások meglévő böngészők grafikus felület nélküli (headless) változatát használják tesztelésre (puppeteer), megint mások a tesztelendő kódot a vizsgálandó oldal kontextusába injektálják bele és ott futtatják (TestCafe).

3. Tesztelési lehetőségek a kliensoldali technológiák oktatásakor

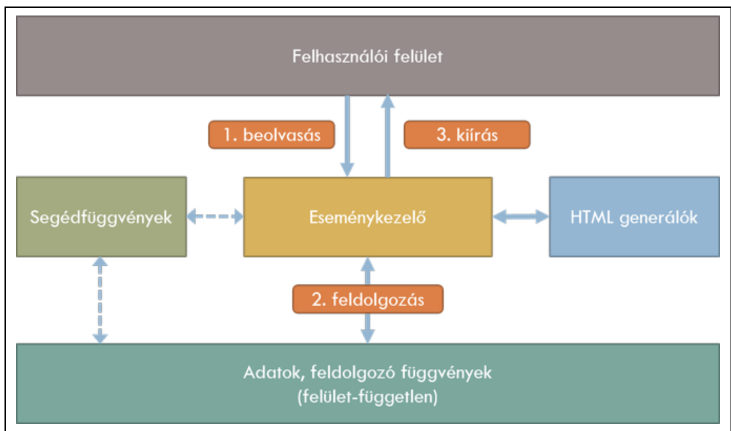
Az alábbiakban áttekintjük, hogy a webes tárgyak oktatása során milyen eszközöket milyen szinten vagy milyen előismerettel használhatunk. A vezérelvet az alapozó tárgy tematikája szolgáltatja, de látni fogjuk, hogy az ottani tapasztalatok a komplexebb tárgyakra is tanulságul fognak szolgálni.

Vizsgálataink során számos módszertani vezérlőelvet alkalmazunk a tesztelésre [3]:

- a tanulók értsék meg, hogyan működik egy tesztrendszer, mielőtt használnának egyet;
- próbálják meg az elérhető nyelvi eszközökkel megoldani a tesztelést a tesztkeretrendszerek használata előtt;
- eleinte ne ők maguk írják a teszteket (oktató tesztek), hanem tanulják meg, hogyan kell használni őket és dolgozni velük;
- fokozatosan vezessünk be új eszközöket.

A bevezető kurzus egymásra épülő ismeretkörökből áll. A pusztán nyelvi elemektől fokozatosan jutunk el komplex kliensoldali alkalmazások fejlesztéséig [4]. Egy ilyen összetettebb alkalmazás a következő elemekből állhat (ld. 1. ábra) [5]:

- *eseménykezelő függvények*: mini-programokként tekintünk rájuk; általában a felhasználói tevékenység következtében futnak le, bennük a *beolvasás-feldolgozás-kiírás* általános struktúrája figyelhető meg;
- *adatok és feldolgozó függvények*: megközelítéstől függően ezek egyben is lehetnek (OOP), vagy a feldolgozó függvényeket tekinthetjük állapottér-leképezéseknek is (funkcionális);
- *segédfüggvények*: az alkalmazás központi logikájától független kisegítő függvények;
- *HTML generáló függvények*: speciális segédfüggvények, amelyek az adatokból HTML szöveget állítanak elő, általában felhasználói felület generálására használjuk.



1. ábra: Kódszervezés JavaScriptben.

3.1. Függvények tesztelése

Az előző meggondolásokból látható, hogy alkalmazásunkban sok függvény kap helyet. Ezek jelentős részét megírhatjuk tiszta függvényként. A segédfüggvények és HTML generáló függvények szinte biztosan ilyenek, egyszerűbb alkalmazások esetén az üzleti logika is ilyen függvényekből áll, az állapottér módosításához használt függvények esetén pedig a funkcionális megközelítést alkalmazva is ilyen függvényeket írunk.

```
// segédfüggvény
function range(n) { return Array.from({length: n}, (e, i) => i+1); }
// HTML generáló függvény
function genList(list) {
    return `<ul>${list.map(e => `<li>${e}</li>`).join('')}</ul>`
}
// üzleti logikai függvény
function factorial(n) {
    let f = 1;
    for (let i = 1; i <= n; i++) { f *= i; }
    return f;
}
// állapottér módosítás
function novel(state) {
    const { db, ...maradek } = state;
```

```
    return { db: db+1, ...maradek };
}
```

Az órákon ezeknek a függvényeknek a tesztelését kézzel elvégezhetjük a fejlesztői konzolban. A függvények ezekben az egyszerűbb programokban a globális névtérben helyezkednek el, így a konzolból elérhetjük őket, meghívhatjuk különböző bemenetekkel, a kimenet pedig azonnal látszik. A következő lépés ennek a folyamatnak az automatizálása lenne: a függvénydefiníciók után egy elágazásban vizsgáljuk meg, hogy a függvény helyes eredményt adott-e vissza. A logika egy `assert` függvényben általánosítható. Végül be lehet mutatni, hogy az `assert` függvényt a konzol is támogatja a `console.assert()` híváson keresztül.

```
// konzol használata
> factorial(5)
< 120

// saját ellenőrzés
console.log('0! === 1');
if (factorial(0) === 1) { console.log('OK')          }
                   else { throw new Error('failed') }

// saját assert függvény
function assert(name, actual, expected) {
    console.log(name);
    if (expected === actual) { console.log('OK') }
                        else { console.warn(`failed! ${actual} !==
${expected}`) }
}
assert('1! should be equal 1',   factorial(1), 1);
assert('5! should be equal 120', factorial(5), 120);

// console.assert
console.assert(factorial(1) === 1,   '1! should be equal 1');
console.assert(factorial(5) === 120, '5! should be equal 120');
```

Ennek használatával mindenféle plusz eszköz bevezetése nélkül helyben ellenőrizhetővé válik a kód jelentős része [8]. Az ellenőrző kódot írhatja akár a hallgató is, de tanárként elő is készíthetjük, így építve az utat a tesztvezérelt fejlesztés felé.

Miért érdemes mégis tesztfuttató környezetet és tesztkeretrendszert használni? A keretrendszerek általában jól bevált gyakorlatokat alkalmaznak a gyakran előforduló problémákra, szabályok közé szorítják a csapongó lehetőségeket, és sok kényelmi szolgáltatást nyújtanak. Következő lehetséges lépésként egy böngészőben futó tesztrendszert lehet alkalmazni. A rengeteg lehetőség közül egy, az iparban is elterjedt eszközt, a *Jasmine*-t választottuk nemcsak az érettsége és a jó dokumentációja miatt, hanem amiatt is, mert nem kell további könyvtárakat alkalmazni az ellenőrzésre vagy a helyettesítések készítésére. Így is az előkészítéshez pár forrást be kell tölteni az oldal elején. Először azt a megközelítést követhetjük, hogy az alkalmazással egy dokumentumba töltjük be a Jasmine-t az oldal `<head>` részében. Az oldal törzsében pedig először a tesztelendő alkalmazáskódot emeljük be (*app.js*), majd a tesztelő kódot (*app.test.js*).

```
<!-- keretrendszer betöltése -->
<link rel="stylesheet" type="text/css" href="jasmine.min.css">
<script src="jasmine.min.js"></script>
<script src="jasmine-html.min.js"></script>
<script src="boot.min.js"></script>
<!-- az alkalmazás és teszt betöltése -->
<script src="app.js"></script>
<script src="app.test.js"></script>
```

A Jasmine a behaviour-driven development (BDD) által használatos kulcsszavakat használja a tesztesetek leírásához. Egy tesztesetet az `it` függvénnyel vezet be, a vizsgálatot az `expect` függvény oldja meg, a tesztesetek csoportosítását pedig a `describe` függvényre bízza. Gondot a névtelen függvény jelentheti kezdetben a hallgatók számára, de ezt helyettesíthetjük lambda függvényekkel, ami ismerős lehet a funkcionális programozás világából.

```
// app.test.js
describe('factorial with test array', () => {
    it('0! should be 1', () => {
        expect(factorial(0)).toBe(1);
    })
})
```

Az alapozó tárgyakon túllépve szükséges ismertetni a modul fogalmát JavaScriptben. Ekkor az üzleti logikai függvényünket külön modulba szervezhetjük ki, és önállóan, az alkalmazáson kívül tesztelhetjük. Megtehetjük ezt a böngészőben egy külön tesztfuttató HTML állományban:

```
<!-- testrunner.html -->
<script type="module" src="app.test.js"></script>
```

```
// app.js
export function factorial(n) { /* ... */}

// app.test.js
import { factorial } from './app.js';
it('0! should be 1', function () {
    expect(factorial(0)).toBe(1);
});
```

Másik lehetőség, hogy a tesztet parancssorban futtatjuk, de ekkor egy átalakítási fázist kell közbeiktatnunk a *babel* csomag segítségével. Parancssori futtatáshoz a legkényelmesebb eszköz a *Jest*, ezt kell felkészíteni az JavaScript modulok értelmezésére (*babel-jest* és *babel-core* csomagok). Ezek után az `npx jest app.test.js` futtatásával lehet a kódot tesztelni. A parancssor egyik előnye, hogy figyelő módban is elindítható a Jest, így a kód változásával a tesztek automatikusan lefutnak.

Ha egy függvény egy másik függvényt hív meg feladatának elvégzéséhez, akkor két út kínálkozik. Az egyik az, hogy a meghívott függvényt helyettesítjük egy saját implementációval. Ehhez több kód írása szükséges, és koncepcionálisan is nehezebb témakörről van szó. Példát az eseménykezelők tesztelésénél láthatunk. A másik út, hogy integrációs tesztként tekintünk az esetre, és például lentről felfele módszerrel előbb a belső függvény helyességéről győződünk meg, majd a külső függvényt teszteljük. Ez a megközelítés nem kíván újabb ismereteket a hallgatóktól.

3.2. Az állapottér tesztelése

Az alapozó webes kurzuson az alkalmazás állapotterét kezdetben globális változókban tároljuk és globális felületfüggetlen feldolgozó függvények végeznek el rajtuk módosításokat. Később az összetartozó adatokat és metódusokat objektumokba zárjuk. (Megjegyzés: az első megközelítés is

objektumba zárja az adatokat és a függvényeket, ugyanis a globális névtér a `window` objektum maga.) Ez az egységbe zárás utat nyit több funkcionális rész kialakítására. JavaScriptben lehetőség van a `class` kulcsszóval objektumkonstruktorokat kialakítani, amivel érdemes is élni, hiszen az új tárgy az objektumelvű alkalmazások után következik.

Privát adattagok hiányában ki nem mondott szabályként megfogalmazhatjuk, hogy az állapotteret csak dedikált függvényeken, metódusokon keresztül módosíthatjuk (üzenetek). Az állapottér lekérdezését viszont bárhonnan megtehetjük.

```
// globális változokkal
let szamlalo = 0;
let nev = 'Anonymous';
function novel(n = 1)    { this.szamlalo += n; }
function csokkent(n = 1) { this.szamlalo -= n; }
// egységbe zárva
class Allapotter {
    constructor(kezdetiAllapot) {
        this.szamlalo = 0;
        this.nev = 'Anonymous';
        Object.assign(this, kezdetiAllapot);
    }
    novel   (n = 1) { this.szamlalo += n; }
    csokkent(n = 1) { this.szamlalo -= n; }
}
const allapot = new Allapotter({nev: 'Valami'});
```

Tesztelés során előkészítjük az állapotteret, elvégezzük a tesztelendő műveletet, majd ellenőrizzük, hogy helyes változások történtek-e. A globális megközelítésnél a globális változók beállítása történik meg, osztályba zárás esetén tisztább a helyzet, hiszen itt tesztenként új példányt hozhatunk létre az alkalmazástól teljesen szeparált módon.

```
// globális esetben
it('a novel() paraméter nélkül eggyel növeli a darabszámot', () => {
    db = 10;             // előkészít
    novel();             // elvégez
    expect(db).toBe(11); // ellenőriz
})
// egységbezárás esetén
it('a novel() paraméter nélkül eggyel növeli a darabszámot', () => {
    const allapot = new Allapotter({db: 10}); // előkészít
    allapot.novel();                          // elvégez
    expect(allapot.db).toBe(11);              // ellenőriz
})
```

3.3. A felhasználói felület és az eseménykezelők tesztelése

Noha a felhasználói felület működésének a tesztelése a funkcionális tesztelés hatáskörébe esik, az eseménykezelő függvények tesztelése pedig az egység- vagy az integrációs tesztekébe, mégis együtt kezeljük őket a továbbiakban. Ennek oka az, hogy a két terület nagyon közel esik egymáshoz. A felhasználói felületi tevékenységei eseményeket váltanak ki, a böngésző pedig meghívja az ezekhez regisztrált eseménykezelőket. Az eseménykezelők ráadásul sok szállal kapcsolódnak a felület dokumentum objektum modelljéhez (DOM): működésükhöz szükséges adatokat onnan olvassák be, a feldolgozás eredményét pedig oda írják ki. Ezektől függetleníteni az eseménykezelőket igen nagy munka lenne, és az esetek többségében felesleges is.

A következőkben egy egyszerű példán keresztül mutatjuk be ezt a témakört: egy sugarával adott kör kerületét szeretnénk kiszámolni. Az ehhez tartozó HTML és JavaScript kód a következő:

```
<!-- index.html -->
<input  id="sugar"  >
<button id="gomb"   >Számol</button>
<span   id="kimenet"></span>
<script src="app.js"></script>
```

```
// app.js
// segédfüggvény
function $(sel) { return document.querySelector(sel); }
// feldolgozó függvény
function kerulet(r) { return 2 * r * Math.PI; }
// eseménykezelő függvény
function kattintas() {
    //beolvasás
    const r = parseFloat($('#sugar').value);
    if (isNaN(r)) { return; }
    //feldolgozás
    const ker = kerulet(r);
    //kiirás
    $('#kimenet').innerHTML = ker;
}
$("#gomb").addEventListener("click", kattintas);
```

A tesztelési lépések meghatározásánál itt is abból indulunk ki, hogy mi történik kézi tesztelés esetén: megnézzük, hogy az oldal tartalmazza-e a szükséges elemeket, majd különböző beviteli értékek mellett „megnyomkodjuk" a felületet, és ellenőrizzük a megjelenített értéket. Ezt a működést programozottan is elő tudjuk állítani. Első megközelítésben a tesztfájlokat az alkalmazás mellett helyezzük el és futtatjuk. Látható, hogy a teszteléshez új ismeretre nincsen szükség.

```
describe('felület működése', () => {
    // saját $ függvény
    function $(sel) { return document.querySelector(sel); }
    // minden teszt előtt készítsük elő a felületet
    beforeEach(() => {  $('#sugar')   .value     = '';
                        $('#kimenet').innerHTML = ''; });
    // tesztek
    it(`Minden megjelenik`, () => {
        expect($('input#sugar') ).not.toBeNull();
        expect($('button#gomb') ).not.toBeNull();
        expect($('span#kimenet')).not.toBeNull();
    });
    it(`Üresen hagyva nem jelenik meg semmi`, () => {
        $('#gomb').click();                         // elvégez
        expect($('#kimenet').textContent).toBe(''); // ellenőriz
    });
    it(`Számot írva be, jó eredményt ad`, () => {
        const n = 10, ker = kerulet(n);             // előkészít
        $('#sugar').value = n.toString();
        $('#gomb').click();                         // elvégez
        // ellenőriz
        expect($('#kimenet').textContent).toBe(ker.toString());
    });
    it(`Szöveget írva be nem jelenik meg semmi`, () => {
```

```
        $('#sugar').value = 'alma';                    // előkészít
        $('#gomb').click();                            // elvégez
        expect($('#kimenet').textContent).toBe(''); // ellenőriz
    });
});
```

A teszteléshez további függvénykönyvtárak vehetők igénybe (*jasmine_dom_matchers*, *dom-testing-library*), amelyekkel a felület kezelése és az ellenőrzés kényelmesebb lehet.

Az eseménykezelőket leválaszthatjuk a feldolgozó függvényekről, mégpedig úgy, hogy a feldolgozó függvényeket egy helyettesítő implementációval látjuk el, ami figyeli az adott függvényt. Jasmine-ban ezt a `spyOn` függvénnyel lehet megtenni. A következő példában megnézzük, hogy meghívódna-e a `kerulet` függvény:

```
it('eseménykezelő, kerulet vizsgálata', () => {
    $('#sugar').value = '10';                // előkészít
    spyOn(window, 'kerulet');
    $('#gomb').click();                      // elvégez
    expect(kerulet).toHaveBeenCalled();      // ellenőriz
    expect(kerulet).toHaveBeenCalledTimes(1);
    expect(kerulet).toHaveBeenCalledWith(n);
});
```

Későbbi tárgyakban jogosan merülhet fel az igény a felületi elemek és funkciók izolált vizsgálatára. Erre több lehetőség adódik:

- A *jasmine-fixture* könyvtár használatával egy külön tesztoldalon belül hozhatunk létre izolált HTML elemeket, és ezeken futtathatjuk meg a teszteket. Hátránya, hogy az eseménykezelőket külön kell alkalmazni a dinamikusan beszúrt elemekre, így az eredeti alkalmazást kicsit módosítani kell.
- Az izolált tesztelést elérhetjük úgy is, hogy külön osztályba zárjuk az adott felületi elemekhez tartozó logikát: az eseménykezelőket, ezek regisztrálását és felszabadítását. (Ez nagyon hasonlít a Backbone keretrendszer nézetére.)
- Továbbvíve az előző gondolatot, egy egységbe zárt osztály akár saját maga megjelenítéséért is felelős lehet. A kirajzolás hatékonysága érdekében valamilyen DOM diffing könyvtár használata javasolt.
- Hamar elérkezünk így a komponensekig, amiket a modern keretrendszerek használnak. A komponensek önmagukban megálló, az adott felületi elemért felelős funkcionális egységek.

Lehetőség van a felület parancssori tesztelésére is. A Jest tesztrendszer fel van készítve erre is a jsdom függvénykönyvtár segítségével. A tesztbe csak be kell tölteni az adott JavaScript állományt és a tesztek ugyanúgy működnek, mint a naiv esetben.

```
beforeAll(() => {
    document.body.innerHTML = `<input id="sugar">
                               <button id="gomb">Számol</button>
                               <span id="kimenet"></span>`;
    require('./app.js');
});
```

A felületi tesztelés egy másik alternatívája egy független Chrome böngésző programozott irányítása a *puppeteer* függvénykönyvtár segítségével. Lehet böngésző megjelenésével és anélkül is futtatni a teszteket. Sajnos a programozási interfésze teljesen aszinkron, így a benne való programozás kezdőknek nem ajánlott. Létezik hozzá egy *puppeteer-recorder* nevű felvevő eszköz, de az általa generált kód sem egyszerű. Parancssorból használható, a Jesttel jól működik együtt, akár saját

magunk által konfigurálva, akár a *jest-puppeteer* csomag segítségével. Elsősorban számonkérések automatikus ellenőrzéséhez ajánlott használni. Teszteléskor kétféle megközelítést alkalmaznak:

- az alkalmazás végig kattintható-e, az oldalon egyéb állapotának nincs ellenőrzése; ezzel nagyon egyszerűen írhatók tesztek;
- az alkalmazás jó adatokat jelenít-e meg, azaz nemcsak a felületi elemek megléte, hanem azok tartalma is számít.

4. Tesztelési lehetőségek a szerveroldali technológiák oktatásakor

Az alapozó tárgyban a szerveroldali webprogramozást PHP-ban tanítjuk. A tananyagot a kliensoldalhoz hasonlóan lépésről lépésre, a nyelvi elemektől a technológiai ismereteken keresztül a tervezési mintákig építjük fel [10]. A szerveroldali kód szervezésében is ügyelünk arra, hogy az adatokat és az üzleti logikát képviselő feldolgozó függvényeket elválasszuk az I/O-t jelentő HTTP kérésektől, valamint a fájl- és adatbázis-műveletektől [6].

4.1. Egységtesztek

A feldolgozó függvényeket a kliensoldalhoz hasonlóan egységteszteknek, illetve lentről felfelé építkező integrációs teszteknek vethetjük alá. A PHP-ban is van egy `assert` függvény, amely használható elemi tesztek írására. Tesztrendszer használatakor bár PHP-ban a legelterjedtebb az xUnit alapú PHPUnit, ennek formája eltér a kliensoldalon bemutatott BDD stílustól. Mivel fontosnak tartjuk, hogy minél kevesebb energiabefektetéssel tudjuk a tesztelést tanítani, így PHP-ban is kerestünk egy BDD stílusú könyvtárat, a *Kahlan*t. A tesztelés hasonlóan történik a JavaScripthez képest: vagy globálisan definiált változókat és függvényeket húzunk be a teszteseteket tartalmazó fájlba `include`-dal, vagy osztályokat példányosítunk:

```
include('./functions.php');
describe('Pozitív számok kiválogatása', function () {
    it('üres tömbre üreset ad vissza', function () {
        expect(kivalogatas([]))->toBe([]);
    });
    it('csupa pozitív tömbre az összeset visszaadja', function () {
        expect(kivalogatas([1, 3, 5, 7]))->toBe([1, 3, 5, 7]);
    });
});
```

4.2. Felületi tesztek

A szerveroldali alkalmazások célja HTML oldalak generálása. Ennek a tesztelése igazából funkcionális tesztelés: egy programmal megszólítjuk a HTTP végpontot, és megnézzük, milyen válasz jön vissza. Ehhez például a puppeteer projekt kiválóan használható. A teszteseteket ebben az esetben JavaScriptben írjuk, de léteznek PHP-ban megírt end-to-end tesztelést végrehajtó rendszerek (pl. Codeception).

5. Esettanulmányok

5.1. Tesztvezérelt fejlesztés

Az ELTÉn oktatott JavaScript technológiák nevű speciálkollégiumon egy hallgató gyakorlatvezető segítségével kipróbáltuk, milyen az, amikor a hallgatóknak előre megírt teszteket kell kielégíteniük. A hallgatók számára új volt a tesztelés, új volt a tananyag, sőt még az elvárt parancssori környezet is gondot okozott többségüknek, így az anyag ilyetén való oktatása nehézkessé vált. Tanulságként

levonható, hogy a hallgatóknak szükségük van a fokozatos építkezésre és az egyes eszközök átlátására.

5.2. Zárthelyi dolgozat értékelése

Az alapozó tárgyat évente kb. 200 hallgató veszi fel az egyik félévben. Korábban a félév végi zárthelyi gépes dolgozatot manuálisan javítottuk. Ez azonban sok energiát vett el, és szubjektivitásra adott lehetőséget. 2018 tavaszi félévében a tárgyra jelentkezett közel 200 hallgató vizsgáztatását automatikus tesztrendszerrel végeztük el. Mivel az implementációs részleteket nem vizsgálhattuk (hiszen vagy JavaScripttel vagy PHP-val oldott meg egy adott feladatot), a tesztelés funkcionálisan történt a *mocha* keretrendszer és a puppeteer használatával. A beadás egy webes felületen történt. A hallgatók év közben nem tudták a rendszert használni, a zh előtt kaptak egy tesztalkalmazást, ahol a tesztrendszer visszajelzéseivel megismerkedhettek. A zh-n főleg az ismeretlenségnek köszönhetően sok kérdés volt, de ettől függetlenül a teljesítés nem maradt el a manuális jegyektől. Az automatikus rendszer egyik előnye, hogy differenciáltabb eredményt lehet látni, hiszen a manuális ellenőrzés során adott 1-5 osztályzatok helyett itt egy nagyobb skálán mozgó pontszámot látunk, és az is kideríthető, hogy a hallgató mely ismeretkörrel nem boldogult.

A tesztek írásában érdekes kihívás, hogy az adatok tárolásának módját sem tudjuk előre meghatározni, hiszen a vizsgázó fájlban, adatbázisban egyaránt tárolhat adatot. Az ellenőrzést csak felületen keresztül tehetjük meg. Erre kétféle megoldást is kidolgoztunk: az egyik esetben véletlenszerű adatokat állítunk elő, azt mentjük el a felületen és keressük meg a megjelenítő oldalon; a másik esetben pedig előírjuk a hallgatóknak, hogy állítsanak be fix adatokat, amiket nem változtathatnak meg, mivel azoknak a helyes működését vizsgálja a program. Mivel a puppeteer API-ja az aszinkronitás miatt nehezen kezelhető, ezért egy felhasználóbarátabb teszt API kialakítását kezdtük el.

6. Módszertani kiértékelés és kitekintés

A fentiek alapján jól látható, hogy a webes tárgyak oktatása során a tanult ismeretekhez képest viszonylag kevés többlettel elérhető az alkalmazások automatikus tesztelése. Módszertani szempontból az alábbiakat érdemes megjegyezni:

- Ahogy a szoftverfejlesztő cégeknél igaz, úgy az oktatásban is vegyük figyelembe, hogy a tesztelés elsajátításához, megírásához időre van szükség. Ez kb. még egyszer annyi idő, mint az alkalmazás írása, az egyszerű programok esetében. Ha fontosnak tartjuk a tesztelés oktatását, akkor szánjunk rá kellő időt, különben a hallgatól azt fogják érezni, hogy ez a lépés kevésbé fontos az implementálásnál.
- A fent bemutatott módszer szerint a tesztelés fokozatosan is bevezethető a kézi teszteléstől a már ismert nyelvi elemek és vezérlési szerkezetek alkalmazásával kapott automatikus teszteken keresztül egészen a keretrendszerek használatáig. Ha szükséges, akkor a keretrendszereket akár el is hagyhatjuk.
- A tesztek is programok, használjuk ki a programozásoktatásban!
- A fenti eszközök alkalmasak arra, hogy különböző fejlesztési módszereket adjunk meg. A tanár által írt teszteknek való megfelelés már előre vetíti a tesztvezérelt fejlesztést. Ezt követheti a tesztek utólagos írása, végül egyszerűbb esetekben a tesztek írását is előre vehetjük [7]. Ez olyan tapasztalatot eredményez, amelyet manapság széles körben elvárnak a szoftverfejlesztésben.
- A tesztelés jótékonyan hathat a program minőségére is: tiszta függvények, függőségek megfelelő kezelése és egyéb hasznos programozási minta használatára világíthat rá.
- Az előre megírt tesztelést kiválóan lehet számonkérések objektív kiértékelésekor használni. Nagyon fontos azonban, hogy a visszajelzés a tesztekből egyértelmű és segítő legyen.

- A tesztvezérelt fejlesztést akár tutorialszerűen lehet alkalmazni online feladatmegoldó rendszerekben [8, 9]. A teszteseteket részekre bontva lépésről lépésre lehetne a tanulókkal a feladatot megoldani, ezzel irányítva és gyakoroltatva őket a feladatmegoldás lépéseiben.
- A tesztek képesek kódlefedettség információt is adni, ezt kihasználhatjuk a hallgatóktól elvárt tesztek értékelésében.
- Érdekességképpen meg lehet mutatni könnyen kezelhető hatékonysági tesztelő programokat, mint például a majomtesztelésre használt *gremlins.js* vagy a terheléses teszthez használt *ab*.

Ebben a cikkben elsősorban az alapozó tárgy tematikája volt fókuszban, az sem teljes egészében. Érdemes lenne megvizsgálni a *canvas* alapú alkalmazások tesztelését, illetve az alapozó tárgyra épülő további tanegységek esetében egyre inkább az iparban használatos eszközöket használni.

Köszönetnyilvánítás

EFOP-3.6.1-16-2016-00023: Kutatás-fejlesztési tevékenység megvalósítása az Eötvös Loránd Tudományegyetem szombathelyi kampuszán – A projekt a Magyar Állam és az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Irodalom

1. A Programozás tárgy előadásainak tematikája az ELTE IK-n, http://progalap.elte.hu/?El%C5%91ad%C3%A1s (utoljára megtekintve: 2018.11.20.)
2. Gregorics Tibor, Mózsi Krisztián, Szendrei Rudolf: *Hogyan tanítsunk tesztelni?*, INFODIDACT 2017, Zamárdi, 2017.
3. David S. Janzen, Hossein Saiedian: *Test-Driven Learning: Intrinsic Integration of Testing into the CS/SE Curriculum*, SIGCSE 2006, Houston, Texas, USA, 2006
4. Horváth Győző, Visnovitz Márton: *Egy bevezető webfejlesztési kurzus módszertani megfontolásai*, Informatika a Felsőoktatásban, Debrecen, 2017
5. Horváth Győző, Visnovitz Márton: *A böngésző mint alkalmazás-fejlesztési platform*, http://webprogramozas.inf.elte.hu/tananyag/kliens/ (utoljára megtekintve: 2018.11.20.)
6. Horváth Győző, Visnovitz Márton: Dinamikus weboldalak előállítása szerveroldali technológiákkal, http://webprogramozas.inf.elte.hu/tananyag/szerver/ (utoljára megtekintve: 2018.11.20.)
7. David S. Janzen, Hossein Saiedian: *Test-driven learning in early programming courses*, SIGCSE 2008, Portland, Oregon, USA, 2008
8. Horváth Győző: *A web-based programming environment for introductory programming courses in higher education*, ICAI2017, Eger, 2017
9. Horváth Győző, Menyhárt László: *Webböngészőben futó programozási környezet megvalósíthatósági vizsgálata*, INFODIDACT 2016, Zamárdi, 2016.