Hogyan tanítsunk tesztelni?

Gregorics Tibor[1], Mózsi Krisztián[2], Szendrei Rudolf[3]

{[1]gt, [2]mozsik, [3]swap}@inf.elte.hu
ELTE IK

Absztrakt. A tesztelés módszertanának elsajátítása fontos része az informatikus képzésnek. Kívánatos, hogy ezzel már a kezdő programozó hallgatók is találkozzanak úgy, hogy viszonylag kis munka árán szembesüljenek a tesztelés előnyeivel. Ebben a cikkben azt mutatjuk meg, hogyan lehet a tesztelést a programkészítés szerves részévé tenni ott, ahol az egyszerű programozási feladatok megoldásai analóg programozási technikával, felsorolókra támaszkodó programozási tételek mentén születnek. Kidolgoztuk a felsorolók és a programozási tételek alkalmazásához kapcsolódó tesztelési szempontokat, olyan szürke doboz teszteseteket, amelyeket a programozási tételekkel együtt lehet megtanítani, és amelyek szabványos tesztelési eljárást adnak a programozók kezébe. Ennek automatikus végrehajtására javasolunk egy konkrét tesztelő eszközt, amelynek használatát egy rövidebb esettanulmány keretében mutatjuk be.

Kulcsszavak: analóg programozás, programozási tétel, felsoroló, tesztelés, szürke doboz tesztelés

1. Bevezetés

Egy programtervező informatikus szak diplomájának megszerzéséhez elengedhetetlen, hogy a hallgatók tisztában legyenek a szoftverfejlesztés módszertanával, és azt képesek legyenek alkalmazni. A módszertannak szerves részét képezi a megoldandó feladathoz készítendő szoftver teljes életciklusának végigkísérése. Ez magába foglalja a feladat követelmény-elemzését, a megfelelő architektúra megválasztását, a szoftver megtervezését, a mérföldkövek kijelölését, az implementációt, valamint a tesztelést és karbantartást, és nem utolsó sorban a dokumentálást.

Mivel a szoftverre, mint termékre kell tekintenünk, ezért szem előtt kell tartanunk az elkészítésével kapcsolatban a határidőt, a költséget, és a minőséget egyaránt. A minőség-biztosítás egy szoftver kifejlesztésének valamennyi fázisánál fontos szempont, és ezekhez megfelelő eszközök is társulnak. Ennek egyik legfontosabb eszköze a tesztelés, amely egyben a fejlesztés egyik konkrét fázisaként is megjelenik.

Manapság a tesztelés a szoftverek méretének és komplexitásának növekedése mellett egyre fontosabbá válik. Ipari környezetben ehhez különféle tesztelő eszközöket alkalmaznak, amelyek biztosítják a szoftvernek az egység, illetve az integrációs tesztjét. Ezen felül a fejlesztői csapatok megfelelő verziókezelő és tesztkörnyezetekkel biztosítják a fejlesztés nyomon követhetőségét. A fejlesztési módszerek közül egyre több helyen alkalmazzák az agilis, illetve a tesztvezérelt szoftverfejlesztést. Éppen ezért elengedhetetlenül fontos, hogy az egyetemi hallgatók ezen módszereket megismerhessék és elsajátíthassák tanulmányaik alatt.

Az ELTE programtervező informatikus képzésben ehhez külön kurzus kapcsolódik, melyet a hallgatók a tanulmányaik vége felé közeledve végezhetnek el. Fontos volna azonban, hogy a tesztelés szükségességét, a hozzá kapcsolódó szemléletet a diákok már tanulmányaik korai szakaszában megtapasztalhassák, hogy arra a későbbiekben ne csak egy szükséges, hanem egy hasznos eszközként tekinthessenek.

A következő fejezetekben bemutatunk egy, a tesztelés elsajátításához használható lehetséges módszertant. A módszertan alapját az egyszerű programozási feladatok megoldásánál használt programozási tételek szolgáltatják, amelyek nemcsak egy analóg programozási technika alapjául

szolgálnak, hanem viszonylag könnyedén összegyűjthetőek az alkalmazásuk esetén ellenőrizendő tesztesetek. Erre mutatunk rá a második fejezetben.

A harmadik fejezetben a programozási tételek alkalmazásánál vizsgálandó szürke doboz teszteseteket mutatjuk be, amelyek egyesítik a fekete és fehér doboz tesztelés sajátosságait. Ezek ugyanis abból a szempontból fekete doboz tesztesetek, hogy a feladat specifikációja alapján születnek. De mivel a specifikáció a programozási tételek segítségével írja le a feladatot, ezáltal már a megoldási tervet is tartalmazza, így implicit módon utal arra a megoldó algoritmusra, amelyből a kód születik. Ennél fogva a specifikációból származó tesztesetek fehér doboz tesztesetek is.

A negyedik fejezetben egy példán keresztül mutatjuk be, hogy az analóg programozás módszertanát alkalmazva, a specifikáció alapján hogyan tervezhető meg egy program tesztelése.

Az ötödik fejezetben összegyűjtjük azokat a C++ programozási nyelvhez köthető tesztelő eszközöket, melyek használata könnyű, és minimális időráfordítással integrálhatók a már meglévő programokba. Ezen eszközök közül egyben ki is választjuk a szempontjainkhoz számunkra legjobban illőt.

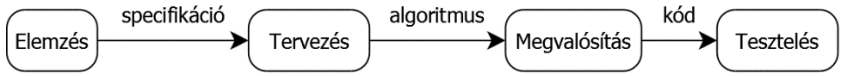
A hatodik fejezetben a már kiválasztott tesztelő eszközt felhasználva megvizsgáljuk az általunk javasolt módszertannak a gyakorlatba történő átültetési lehetőségeit. Ennek keretében egyúttal mélyebb betekintést is nyújtunk a konkrét tesztelő eszköz használatába.

Cikkünk végén összefoglaljuk tapasztalatainkat a javasolt módszertannal, illetve a tesztelő eszközzel kapcsolatban.

2. Programozási tételek és a tesztelés

Az egyetemi kezdő programozás tanításnak egy jó alternatívája az algoritmikus gondolkodás kialakítása mellett az analóg programozás [4], amelyiknek egyik változata a visszavezetés. Ennek során egy feladat megoldását úgy állítjuk elő, hogy felismerve annak hasonlóságát egy korábban megoldott feladattal, a korábban megoldott feladathoz előállított algoritmust alakítjuk át úgy, hogy az új feladatot megoldja. Ehhez nincs szükség algoritmikus gondolkodásra, nem kell az algoritmust működésében látni, elég az új és a korábbi feladat közötti eltéréseket pontosan feltárni, és azokat „rávezetni" az algoritmusra. [3]

A programkészítésnek ez a technikája nemcsak az algoritmus tervezést segíti, de áthatja a program készítés teljes életciklusát. Ezt a ciklust egyszerű programozási feladatok megoldásakor négy lépésre lehet felbontani (lásd 1. ábra).



1. ábra: A programkészítés fázisai: a) Elemzés, melynek eredménye a specifikáció, b) Tervezés, aminek eredményeképpen megkapjuk az algoritmust, c) Megvalósítás, ami az algoritmus kódolásán túl az input és output tevékenység programozásából, és a megfelelő programszerkezet kialakításából is áll, d) Tesztelés

Egy megoldandó feladat elemzése során ugyanis már eleve azon gondolkodhatunk, hogy gondosan kiválogatott mintamegoldások – programozási tételek [3] – melyikének kombinálásával tudnánk az új feladatunkat megoldani. Az elemzés eredményeként létrejövő specifikáció ilyenkor nemcsak a feladatot fogalmazza meg informatikus igényességgel, hanem arra is utal, hogy miként lehet azt megoldani. Ez a végrehajtható specifikáció a nagy szoftverek követelményelemzésénél ismert technika. [5]

A tervezéskor ezek után csak két dologra kell ügyelni. Precízen kell adaptálni a programozási tételeket az adott részfeladatokra [4], és az így nyert algoritmusokból össze kell állítani a teljes absztrakt programot. [2]

A megvalósítás is ehhez a gondolatmenethez illeszkedik. Egy tradicionális procedurális megoldást készítve, az egyes részalgoritmusok önálló alprogramokba (eljárás, függvény) kerülnek.

A tesztelés is sajátos színezetet kaphat az analóg programozás során. Lényegében azáltal, hogy az elemzés során egy végrehajtható specifikációt kapunk, a specifikáció alapján készülő tesztesetek nem mind számítanak az úgynevezett fekete doboz tesztesetek közé, hiszen azok, amelyek tekintettel vannak az alkalmazandó programozási tételre már belelátnak a megoldó programba is: ezek tehát úgynevezett szürkedoboz tesztesetek lesznek. Ahogy az analóg programozás szabványosítja a specifikációt és az algoritmus tervezést, úgy a tesztesetek kialakítására is kihat. Különféle, de ugyanazon programozási tételre visszavezethető feladatok esetén is ugyanis a vizsgálandó tesztesetek közül is számos ugyanaz lesz.

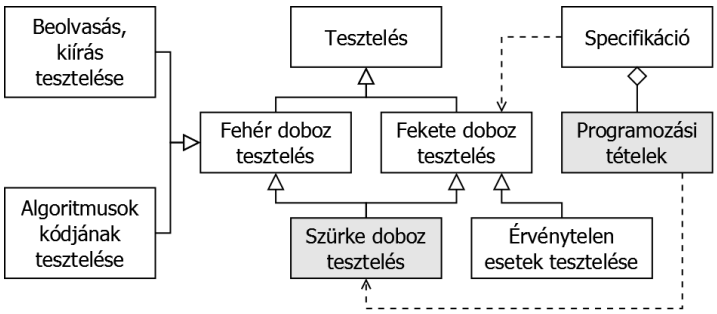
Ha egy-egy programozási tétel kapcsán összegyűjtjük a javasolt vizsgálandó teszteseteket, akkor ezzel olyan szabványt adhatunk a programozó kezébe, hogy ne kelljen egyedileg kitalálnia a vizsgálandó teszteseteket. Aki oktatóként megküzdött már azzal a problémával, hogy hogyan tudná a viszonylag egyszerű feladatok megoldásánál rábírni a hallgatókat egy precíz tesztelési terv elkészítésére, az nyilván tudja, hogy kezdő programozók (de a gyakorlottabbak is) milyen nehezen tudnak a programfejlesztői szerepből kilépve saját programjuk bírálójává lenni, és annak működését megfelelő tesztesetekkel lefedni.

3. Szabványos szürke doboz tesztesetek

A megvizsgálandó teszteseteket két nagy csoportba szokták sorolni.

A specifikáció alapján felállított fekete doboz tesztesetekre, amelyek között mi külön figyelmet szentelünk majd az analóg programozás során felhasznált programozási tételekhez köthető szürke doboz teszteseteknek. Ez utóbbiak mellett a fekete doboz tesztesetek között külön csoportot alkotnak az úgynevezett érvénytelen tesztesetek, amikor a feladat bemenő adatai nem elégítik ki a feladat előfeltételét. Ilyenkor a megoldó programnak programozott hibajelzést kell adnia. Az oktatási tapasztalatok alapján talán a legnehezebb azoknak az érvényes fekete doboz teszteseteknek a felderítése, amelyek nem tartoznak egyik előbb említett csoportba. Például két halmaz uniója esetén vizsgálni kellene az unió művelet algebrai tulajdonságait: kommutativitás, asszociativitás, neutrális elem. Érdekes, hogy ezek a vizsgálatok feleslegessé válnak, ha két halmaz unióját egy speciális összefuttató felsorolásra épített összegzés programozási tételével oldjuk meg, és ennek szabványos szürke doboz teszteseteit ellenőrizzük.

A tesztesetek másik nagy csoportját a fehér doboz tesztesetek alkotják. Ezek között érdemes külön választani a beolvasást és kiírást végző kódrészek tesztelését a megoldáshoz tervezett algoritmus kódjának tesztelésétől. Ez utóbbit ugyanis az analóg programozással tervezett egyszerű programozási feladatok esetén szinte teljesen kiváltják a szürke doboz tesztesetek, amelyek ellenőrzése már önmagában is meglepő bizonyosságot ad a program helyességéről.



2. ábra: Teszteset csoportok és kapcsolatuk a feladat specifikációjával, illetve az abban felhasznált programozási tételekkel.

Látjuk, hogy mindkét előbb részletezett teszteset csoport szempontjából izgalmas kérdés az ismert programozási tételekhez kapcsolódó szürke doboz tesztesetek feltárása és használata (lásd 2. ábra). Ezeket, akárcsak a visszavezetés során a megoldó algoritmust, a konkrét feladatra vetített adaptációval fogalmazhatjuk meg. Az informatikus közösség által elfogadott programozási tételek [1, 3] (összegzés, számlálás, maximum kiválasztás, keresés, kiválasztás, feltételes maximumkeresés) közösek abban, hogy mindegyik egy gyűjtemény felsorolására épül. Sokszor ez a gyűjtemény az egész számok egy intervalluma, vagy egy halmaz, sokszor egy tömb vagy sorozat, esetleg egy szekvenciális inputfájl. Ennél fogva számos teszteset a feldolgozott gyűjteményre vonatkozik, így ezek közösek a programozási tételeknél. Természetesen vannak speciálisan egy-egy adott programozási tételre illeszkedő tesztelési szempontok is.

Mindig vizsgálni kell, hogy a megoldás hogyan viselkedik különféle számosságú gyűjteményre (lásd 3. ábra). Szinte hihetetlen, hogy egy géptermi zárthelyin milyen sok beadott program bukik el azon, hogy üres gyűjteményre nem megfelelően működik. Érdekes eset a pontosan egy elemű, a kételemű gyűjtemény, de szükség van egy általános, jó néhány elemet tartalmazó gyűjteményre is. A teszteredmény értékelését a számlálásnál segíti, ha a vizsgált gyűjtemény minden eleme kielégíti a számlálás feltételét. Fontos megjegyezni, hogy a maximum kiválasztás programozási tételének előfeltétele a nem üres gyűjtemény. Ennek ellenére az ilyen esetben is meg kell nézni az üres gyűjteményre való működést, de ez nem a programozási tétel érvényes (azaz szabályos bemenetének) tesztelése, hanem egy érvénytelen teszteset, amely a megoldandó feladat előfeltételét vizsgálja.

Tesztesetek	
Gyűjtemény határainak vizsgálata	Első elem feldolgozásra kerül-e
	Utolsó elem feldolgozásra kerül-e
	Közbenső elem feldolgozásra kerül-e
Gyűjtemény mérete szerint	Üres gyűjtemény kezelése
	Egy elemű gyűjtemény kezelése
	Több elemű gyűjtemény kezelése

3. ábra: A gyűjtemények felsorolásával kapcsolatos szürke doboz tesztesetek

A gyűjtemények feldolgozásánál fontos megvizsgálni, hogy a program figyelembe veszi-e a felsorolás legelső és legutolsó elemét. Ez egy általános, a programozási tételek mindegyikét érintő szempont, de ellenőrzéséhez figyelembe kell venni az adott tétel specialitásait. Például egy összegzés

esetén egy kételemű, különböző elemekből álló gyűjtemény kell hozzá. Számlálás esetén két olyan elemet tartalmazó gyűjtemény, amely kielégíti a számlálás feltételét. Maximum kiválasztásnál két tesztadat kell: az egyikben a felsorolás legelső eleme legyen a legnagyobb, a másikban az utolsó. Keresésnél is két tesztadat kell: az egyikben a felsorolás legelső eleme a keresett tulajdonságú, a másikban az utolsó. Kiválasztásnál is két eset van: a legelső elem a keresett, illetve nem a legelső. A feltételes maximum keresésnél a keresés vagy maximum kiválasztás eseteit kell vizsgálni.

Az összegzés programozási tételének alkalmazása esetén terheléses tesztet kell alkalmazni: ki kell mérni, mekkora gyűjteményekre lehet a programot hiba nélkül futtatni. Számlálás esetén olyan esetekre van szükség, amikor egy gyűjteményben nulla, egy, kettő vagy több adott tulajdonságú elem van. Maximum kiválasztás esetén a maximális elemnek a felsorolás közepén való megjelenését, illetve több azonos maximális elemet tartalmazó gyűjteményt kell megvizsgálni. Keresésnél két fontos eset van: amikor nincsen keresett tulajdonságú elem a gyűjteményben, illetve amikor van. Ez utóbbit érdemes olyan adatra kipróbálni, amikor az első keresett tulajdonságú elem a felsorolás közepén van (hiszen azt, amikor az elején vagy a végén, már néztük). A feltételes maximumkeresésnél a keresés és a maximum kiválasztás eseteit mind vizsgálni kell (lásd 4. ábra).

Programozási tételek	**Tesztesetek**
Összegzés	Két különböző elemet tartalmazó gyűjtemény
	Terheléses teszt
Számlálás	Kételemű gyűjtemény, ahol mindkét elem kielégíti a számlálás feltételét
	Az adott tulajdonságnak a gyűjteményben nulla, egy, kettő vagy több elem tesz eleget.
Kiválasztás	A keresett elem a gyűjtemény első eleme
	A keresett elem a gyűjteménynek nem az első eleme
Keresés	A keresett elem a gyűjtemény első eleme
	A keresett elem a gyűjtemény utolsó eleme
	Létezik a keresett tulajdonságnak megfelelő (közbenső) elem
	Nem létezik a keresett tulajdonságnak megfelelő elem
Maximum kiválasztás	Kételemű gyűjtemény, melynek első eleme a nagyobb
	Kételemű gyűjtemény, melynek második eleme a nagyobb
	Több elemű gyűjtemény közbenső eleme a legnagyobb
	Több elemű gyűjteményben több maximális elem fordul elő
Feltételes maximum keresés	A keresés vagy a maximum kiválasztás teszteseteinek vizsgálata

4. ábra: Programozási tételek speciális tesztesetei

4. Tesztelési terv előállítása

Tekintsük át egy konkrét példán keresztül, hogyan készíthetünk szabványos tesztesetekből álló tesztelési tervet egyszerű programok hibáinak felderítéséhez. Legyen a feladat egy egész számokat tartalmazó mátrix azon sorának a kiválasztása, melynek a szokásos rendezést használva a legnagyobb

a sorösszege. A visszavezetés technikájával a specifikáció fázisában a maximum kiválasztás és az összegzés programozási tételeket választjuk a feladat megoldásához.

Specifikáció:

```
Változók:     (data:ℤ^(n×m), maxSumIndex:ℕ, maxSum:ℤ)
Előfeltétel: (data = data' ∧ n > 0)
Utófeltétel: (Q ∧ maxSum = MAX_{i=1}^{n} rowSum(i) ∧
              maxSum = rowSum(maxSumIndex))

ahol rowSum: ℕ → ℤ
     rowSum(i) = Σ_{j=1}^{m} data[i,j]
```

A tervezés fázisban felírjuk a tételeknek megfelelő algoritmusokat, melyekre ráillesztjük a konkrét feladat specialitásait, ezután pedig már könnyedén implementálhatjuk a kapott megoldó programot. Tegyük fel, hogy a külső programozási tételt, azaz a maximum kiválasztást egy *maxRowSum* nevű eljárásban valósítottunk meg, mely paraméterként kapja azt a mátrixot, amiből meg kell határoznia az eredményt. Továbbá van két kimeneti szemantikájú, egész típusú paramétere is *maxSumIndex* és *maxSum* elnevezéssel, melyekben rendre előáll a legnagyobb összegű sor indexe és értéke. A belső összegzés tételt a *rowSum* függvényben implementáljuk, mely egy adott sor elemeit összegzi.

A definiált alprogramok fejlécei:

```
void maxRowSum(const std::vector<std::vector<int>> &data,
               int &maxSumIndex,
               int &maxSum);
int rowSum(const std::vector<int> &row);
```

A következő lépésben egy tesztelési tervet szeretnénk készíteni, amihez igen jelentős segítséget ad a program előállításához használt technika: írjuk fel a két megvalósított függvényben felhasznált tételek által determinált ellenőrzéseket. A következő javasolt vizsgálandó tesztesetek adódnak, melyeket ellenőrizni fogunk: a gyűjtemény határain megfelelő viselkedést vizsgáló esetek, a gyűjtemény mérete szerinti esetek, maximum kiválasztásnál általános eset a felsorolás közepén, továbbá több azonos maximális elem esete.

Foglalkozzunk először csak a külső tétellel. Az előfeltételnek nem megfelelő bemenetekre adott válasz ellenőrzéséhez adnunk kell egy érvénytelen tesztesetet. Jelen esetben ennek megválasztásánál nincs túl sok lehetőségünk: egy olyan mátrixot kell adnunk, aminek nincsen egyetlen sora sem.

Gondoljuk át az érvényes, szürke doboz teszteseteket. A gyűjtemény alsó határának vizsgálata azt jelenti a konkrét példában, hogy egy olyan mátrixot adunk meg tesztadatként, melyben az első sor elemeinek összege a legnagyobb, és azt szeretnénk megtudni, hogy vajon ilyen esetben hibázik-e a megvalósított program. Ennek megfelelően válasszunk egy két sorból álló mátrixot, melynek egy-egy eleme van, például 10 és 5. A felső határ vizsgálatához csak annyit kell tennünk, hogy a két sort felcseréljük, a másodikat várva maximálisnak.

Érdemes az egy, kettő, valamint több sort tartalmazó mátrixokra adott eredményeket is ellenőrizni. Első esetben értelemszerűen az elvárt eredmény a megadott elemek összege. Kettő és több soros tesztadatként soronként pontosan egy értéket tartalmazó mátrixokat adunk meg, a konkrét bemenetek és az ezekre adott elvárt válaszok az 1. táblázatban láthatóak.

Speciálisan a maximum kiválasztásnál meg kell vizsgálnunk azt is, hogy vajon hogyan viselkedik a megvalósított program akkor, ha a legnagyobb összegű sor a gyűjtemény közepén van, valamint ha több azonos maximális összeg is előfordul. Azonos legnagyobb elemek esetén azt várjuk, hogy az első előfordulást kapjuk eredményként. Az előző tesztesethez hasonlóan egyelemű sorokat tartalmazó mátrixokat adunk bemenetként.

Tesztelési szempont	Tesztadat	Elvárt eredmény
Érvénytelen eset	üres mátrix	programozott hibaüzenet
Gyűjtemény határainak vizsgálata	$\begin{bmatrix}10\\5\end{bmatrix}$	maxSum=10, maxSumIndex=1
	$\begin{bmatrix}5\\10\end{bmatrix}$	maxSum=10, maxSumIndex=2
Gyűjtemény mérete szerinti esetek	$[10 \quad 32]$	maxSum=42, maxSumIndex=1
	$\begin{bmatrix}10\\42\end{bmatrix}$	maxSum=42, maxSumIndex=2
	$\begin{bmatrix}2\\1\\10\\3\end{bmatrix}$	maxSum=10, maxSumIndex=3
Felsorolás közepén lévő maximális elem	$\begin{bmatrix}1\\10\\2\end{bmatrix}$	maxSum=10, maxSumIndex=2
Több azonos maximális elem	$\begin{bmatrix}2\\1\\12\\12\end{bmatrix}$	maxSum=12, maxSumIndex=3

1. táblázat: Tesztelési terv, maximum kiválasztás tétel tesztesetei.

Ezzel a tesztelési tervvel azonban még nem lehetünk elégedettek: egyértelműen látszik az előző példákból, hogy támaszkodunk arra, hogy a belső függvény is mentes a kódolási hibáktól. Eddig feltettük, hogy ez így van, most viszont adjuk hozzá a tesztelési tervhez a belső tételhez tartozó teszteseteket is annak reményében, hogy az esetleges hiányosságokat megtaláljuk. Összegzés esetén elég egy tesztadat a gyűjtemény mindkét szélének vizsgálatához: egy olyan kételemű vektort választunk meg, melynek különbözőek az elemei, és azt várjuk, hogy az összegüket kapjuk meg. Legyen ez az [1, 2] vektor.

Meg kell vizsgálnunk a vektor mérete szerinti eseteket is. Összegzésnél fontos elvárás, hogy ha üres vektorra hívjuk meg a függvényt, adja vissza eredményül a semleges elemet. Ha egy elem van a vektorban, azt az elemet kell visszakapnunk.

A gyűjtemény mérete szerinti vizsgálatokhoz hozzátartozik a két elemet tartalmazó vektor, de vegyük észre, hogy ilyen esetünk már volt, így nem kell újra felvenni.

Tesztelési szempont	Tesztadat	Elvárt eredmény
Gyűjtemény határainak vizsgálata	[1, 2]	3
Gyűjtemény mérete szerinti esetek	üres vektor	0
	[42]	42

2. táblázat: Tesztelési terv, összegzés tétel tesztesetei.

5. Tesztelő eszköz megválasztása

A kezdő programozók oktatása esetében célszerű olyan tesztelő eszközt választani, ami nem helyezi át a hangsúlyt a programozásról és a tesztelésről magára a tesztelő eszközre. Képzésünkön a C++ programozási nyelvet használjuk az analóg programozáshoz, ezért ehhez kerestünk megfelelő tesztelő eszközt. Ezek listája meglehetősen gazdag, jó néhány közülük kifejezetten programozási környezet függő, valamint többségük használata bonyolult előkészületeket igényel. Olyan eszközt kívántunk választani, amelyet a letöltés után nem kell lefordítani, telepíteni, vagy programozási környezetbe integrálni. Ezért azon megoldások közül választottunk, amelyek egyrészt platform függetlenek és egyetlen *.hpp* fejlécállományként vannak implementálva külső függőségek nélkül, másrészt használatuk kézenfekvő. Ezt szem előtt tartva a hallgatóknak a későbbiekben a tesztelést tekintve elegendő kizárólag a tesztesetek megfelelő megválasztására fókuszálniuk. Összegyűjtöttük azon eszközöket (lásd 1. táblázat), amelyek a mi koncepciónknak megfelelhetnek [7]. Ezek közül mi a CATCH eszközt választottuk, főként az egyszerű felhasználhatósága és a jól dokumentáltsága, valamint nem utolsó sorban a bővebb funkcionalitása miatt is.

Eszköz	xUnit alapú	Fixtures	Group fixtures	Generátorok	Mock	Kivételek kezelése	Makrók	Sablonok	Csoportosítás
Bandit		●	●			●	●		○
BugEye						●			●
CATCH		●	●	●		●	●	●	●
doctest		●	●			●	●	●	●
lest		●				●	●	●	○
liblittletest	●	●	●			●	●	●	●
tpunit++	●	●					●	●	
unit.hpp		●		●		●	●		
upp11	●	●				●	●	●	●

3. táblázat: Egyetlen fejléc állományból álló tesztelő eszközök és képességeik.

6. Catch tesztelő eszköz használata

A megvalósítás során az elkészített funkciók kipróbálásához készülhet egy főprogram, ami elvégzi az adatok beolvasását, meghívja az implementált alprogramokat és kiírja a kapott eredményeket. Mivel ez a módszer nem a legmegfelelőbb út egy könnyen használható és karbantartható, ellenőrzéseket automatizáltan végrehajtani képes tesztkörnyezet kialakításához, ezért a munkánkat lényegesen megkönnyítve egy keretrendszert veszünk igénybe. Az előző fejezetben a Catch eszközre esett a választásunk, melynek használatához csupán egy fejlécet [8] kell meghivatkoznunk a kialakítandó tesztkörnyezetből. A keretrendszer a környezet kialakításának megkönnyítése érdekében lehetőséget biztosít a fő belépési pont generálására: amennyiben egy CATCH_CONFIG_MAIN nevű üres makrót definiálunk, a háttérben kapunk egy szokásos *main* függvényt, ami futtatja a felhasználó által megírt összes tesztet. Fontos, hogy ha több fordítási egységet hozunk létre a teszteléshez, akkor csak pontosan egy helyen generáljuk a belépési pontot, praktikusan egy külön erre a célra létrehozott

egységben. A tesztek tesztesetek formájában fogalmazhatóak meg, melyek az eddigi, kipróbáláshoz használt főprogram szerepét hivatottak kiváltani, egy-egy adott bemenetre.

A tesztesetek leírásához egy előre definiált, TEST_CASE nevű makró áll rendelkezésünkre, melynek kötelezően meg kell adni egy egyedi, a tesztesetet jól leíró nevet. További paraméterként megadhatunk címkéket, a kategorizálást elősegítve. A törzsben a tesztelendő funkció meghívása után kell megfogalmaznunk az eredményekre vonatkozó elvárásainkat, melyekhez a Catch szintén beépített makrókat biztosít. Ezek közül a legegyszerűbb a REQUIRE és a CHECK, melyek paraméterként a kapott eredményre vonatkozó logikai állítást várnak. A különbség köztük annyi, hogy amennyiben az ellenőrzött feltétel nem teljesül, akkor előbbi esetben a tesztek futtatása abortál, míg a CHECK használatával folytatjuk a futtatást. Fontos megjegyezni, hogy a keretrendszer elvárja, hogy az állítások egyszerűek legyenek, logikai és/vagy operátorok használatának mellőzésével. Ezen alapvető elemek segítségével már hatékonyan használható az eszköz egyszerű automatizált tesztek írására. Ugyanakkor a haladóbbak kezébe is sok hasznos funkciót ad, de ezek nagy részét most nem fogjuk kihasználni.

1. teszteset (egy triviális teszteset leírása Catch segítségével):

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("One equals one", "[int_test]") {
    REQUIRE(1 == 1);
}
```

Kimenet (sikeres lefutás esetén):

```
All tests passed (1 assertion in 1 test case)
```

Miután az előző „Hello world”-szerű példa futtatásával megbizonyosodtunk arról, hogy a tesztkörnyezet megfelelően működik, elkezdhetjük fejleszteni az automatizált ellenőrzéseket, és kód formájában írhatjuk le az implementált megoldó programmal szemben támasztott elvárásainkat. Tekintsük át a 4. fejezetben ismertetett konkrét példa tesztelési tervén keresztül, hogyan alkalmazható a Catch visszavezetéssel előállított programok szabványos validációjára, és a kódolási hibák minél jobb feltárására.

Kezdjük a maximum kiválasztás ellenőrzésével. Minden teszteset leírása hasonlóan fog kinézni: inicializáljuk az adatokat, meghívjuk a vizsgált függvényt, majd leellenőrizzük a kapott eredményeket. Az inicializálás természetesen történhet fájlból is, de most a példákban az áttekinthetőség miatt a tesztkódban adjuk meg az adatokat tartalmazó mátrixot.

2. teszteset (maximum kiválasztás, gyűjtemény alsó határának vizsgálata):

```
TEST_CASE("Check lower bound", "[max]") {
  std::vector<std::vector<int>> data = {{10}, {5}};
  int maxSumIndex, maxSum;

  maxRowSum(data, maxSumIndex, maxSum);

  CHECK(maxSumIndex == 0);
  CHECK(maxSum == 10);
}
```

Kimenet (sikeres lefutás esetén):

```
All tests passed (2 assertions in 1 test case)
```

A felső határ vizsgálata ugyanilyen módon történhet. Bővítsük az automatikus ellenőrzéseinket a tesztelési terv alapján, térjünk át a hossz szerinti vizsgálatokra: az egy, kettő, valamint több sort tartalmazó mátrixok teszteseteit kell leírnunk.

A logikailag összetartozó ellenőrzéseket összefoghatjuk egy külön tesztesetben, az őt alkotó részeket pedig a SECTION makró segítségével definiálhatjuk. Minden egyes SECTION futása a teszteset elejétől kezdődik, így elhelyezhetünk egy közös inicializáló részt, elkerülve ezzel a kódismétlést. Természetesen ez a konstrukció akkor használható igazán jól, ha az összefogott esetek inicializáló részeiben nagy átfedés van.

Habár maximum kiválasztás esetén az üres gyűjtemény érvénytelen bemenetnek számít, hiszen nem felel meg az előfeltételnek, mégis leírhatjuk akár ebben a tesztesetben is a logikai összetartozás miatt. Legkézenfekvőbb megoldás az előfeltételt sértő bemenet kezelésére egy kivétel dobása, de nyilvánvalóan más módon is reagálhatna a program ezekre az esetekre. A Catch segítségével az is ellenőrizhető, hogy az elvárt kivételtípust dobja-e a tesztelt függvény egy adott bemenetre.

3. teszteset (maximum kiválasztás, gyűjtemény mérete szerinti esetek):

```
TEST_CASE("Checks by collection size", "[max]") {
  std::vector<std::vector<int>> data;
  int maxSumIndex, maxSum;

  SECTION("Empty collection") {
    REQUIRE_THROWS_AS(maxRowSum(data, maxSumIndex, maxSum),
                      std::invalid_argument);
  }

  SECTION("Collection with 1 element") {
    data.push_back(std::vector<int> {10, 32});

    maxRowSum(data, maxSumIndex, maxSum);

    CHECK(maxSumIndex == 0);
    CHECK(maxSum == 42);
  }
}
```

A kettő és több sort tartalmazó mátrixokra vonatkozó ellenőrzések is nagyon hasonlóan fogalmazhatóak meg. A tesztelési terv szerint szükségesek még olyan teszteseteket, amik azokat az eseteket fedik le, amikor a legnagyobb összegű sor a gyűjtemény közepén van, továbbá több azonos maximális összeg is előfordul.

4. teszteset (maximum kiválasztás, gyűjtemény közepének vizsgálata):

```
TEST_CASE("Check the middle of the collection", "[max]") {
  std::vector<std::vector<int>> data = {{1}, {10}, {2}};
  int maxSumIndex, maxSum;

  maxRowSum(data, maxSumIndex, maxSum);

  CHECK(maxSumIndex == 1);
  CHECK(maxSum == 10);
}
```

5. teszteset (maximum kiválasztás, több azonos maximum):

```
TEST_CASE("Check multiple maximum values", "[max]") {
  std::vector<std::vector<int>> data = {{2}, {1}, {12}, {12}};
  int maxSumIndex, maxSum;

  maxRowSum(data, maxSumIndex, maxSum);
```

```
  CHECK(maxSumIndex == 2);
  CHECK(maxSum == 12);
}
```

Kimenet (sikeres lefutás esetén):

```
All tests passed (9 assertions in 4 test cases)
```

Implementáltuk a külső tételhez tartozó teszteseteket, azonban ezen a ponton a tesztlefedettség még nem elegendő. Egyelőre csak azt tudjuk kellő bizonyossággal mondani, hogy a *maxRowSum* eljárás kódolási hibáktól mentesnek látszik, de a *rowSum* függvényről még nincs ilyen információnk.

Az eddigiekhez hasonlóan bővítsük ki a belső függvényre vonatkozó fennmaradó tesztesetekkel az automatikus ellenőrzéseinket. Mivel a belső függvényt összegzés tételére vezettük vissza, elég volt egy bemenetet adnunk a gyűjtemény mindkét szélének vizsgálatához. A vektor mérete szerinti eseteket most is a SECTION makró segítségével írjuk le.

6. teszteset (összegzés, gyűjtemény határainak vizsgálata):

```
TEST_CASE("Check bounds", "[sum]") {
  std::vector<int> row = {1, 2};

  int sum = rowSum(row);

  CHECK(sum == 3);
}
```

7. teszteset (összegzés, gyűjtemény mérete szerinti esetek):

```
TEST_CASE("Checks by row size", "[sum]") {
  std::vector<int> row;

  SECTION("Empty collection") {
    CHECK(rowSum(row) == 0);
  }

  SECTION("Collection with 1 element") {
    row.push_back(42);

    CHECK(rowSum(row) == 42);
  }
}
```

Kimenet (az összes említett teszt sikeres lefutása után):

```
All tests passed (16 assertions in 7 test cases)
```

Rontsuk el egy pillanatra az egyik feltételezésünket. Így megbizonyosodhatunk egyrészt arról, hogy valóban megfelelően lefutnak az ellenőrzéseink, valamint arról, hogy értelmezhető hibaüzenetet kapunk, és az elbukó teszt pontos helyét is láthatjuk. Adjunk hamis feltételezést például az összegzés határellenőrzéséhez, módosítsuk az elvárt összeget 3-ról 10-re. Újra futtatva a teszteket megkapjuk, hogy melyik összehasonlításnál volt a probléma.

```
Check bounds
FAILED:
  CHECK(sum == 10)
with expansion:
  3 == 10
```

```
test cases:  7 |  6 passed | 1 failed
assertions: 16 | 15 passed | 1 failed
```

Sikerült szabványos automatizált tesztekkel lefednünk az előállított programot, amivel jó eséllyel kiszűrhettük a kódolásból adódó hibákat. Bonyolultabb programoknál fontos szerepet kaphat az így előálló kód, hiszen ez egyben egy jó dokumentációja annak, hogy melyik egységtől milyen bemenetre milyen választ várunk el.

7. Értékelés

A kezdő programozók oktatásakor kihívást jelent, hogy a hallgatók megfelelően működő programokat tudjanak előállítani. A mi esetünkben a feladatok megoldását az analóg programozás módszertanán keresztül mutatjuk be, azaz visszavezetjük a feladatokat ismert programozási tételekre. Ahhoz azonban, hogy a program egyúttal megfelelően is működjön, a tesztelés elengedhetetlen.

Képzésünk során korábban azt tapasztaltuk, hogy a hallgatók által készített programok jó része az ellenőrzéskor bizonyos teszteseteknél hibásan működik, ezért bevezettük a számonkérés részeként a tesztelést, illetve annak dokumentálását is. A tesztelés megkövetelése után nőtt a megfelelően működő programok aránya, így megalapozottnak látjuk, hogy a tesztelés már a programozó képzés kezdetén megjelenjen. A tesztelési jegyzőkönyveket olvasva ugyanakkor azt is tapasztaltuk, hogy a tesztesetek megválasztása leginkább intuíció alapján történt, így azok nem feltétlen voltak képesek felfedni a program bizonyos működési hibáit vagy hiányosságait.

Cikkünkben bemutattunk az analóg programozással kapcsolatban egy tesztelési módszertant, amit szürke doboz tesztelésnek neveztünk el, mivel egyesíti a programozási tételek ismerete alapján adódó fehér doboz tesztelést a különböző adatokkal végrehajtandó fekete doboz teszteléssel. Ennek segítségével az intuitív módon adott tesztesetek helyett előre meghatározott tesztelési terveket használhatunk. Egy-egy programozási tétel alkalmazása során megvizsgálandó teszteseteket két csoportra osztottuk: az egyikbe a felsorolással kapcsolatos, a másikba az adott programozási tételre jellemző esetek kerültek. Ezen tesztesetek szabványosíthatóak, azaz könnyen adaptálhatóak a programozási tételekkel megoldható feladatokra.

Kiegészítve a fenti szürke doboz eseteket néhány ismert fekete doboz tesztesettel (pl. érvénytelen bemeneti adatok vizsgálata), az egyszerű programozási feladatok megoldásainak tesztesetei már jó lefedettséget adnak a program helyességének vizsgálatához. Ennek a módszernek köszönhetően a mennyiségi helyett a minőségi tesztelés került a középpontba, aminek köszönhetően egyúttal a programok minősége is javul.

Felismertük, hogy az oktatásban azzal kényszeríthető ki leginkább a tesztelés, ha automatikus tesztkörnyezetet készíttetünk a hallgatókkal. Így a tesztelési terv megjelenik kód formájában is, melynek futtatásával biztosak lehetünk benne, hogy a tervben leírt tesztesetekre a program megfelelően működik. Ennek előnyeivel oktatóként is találkozhatunk: az automatizált tesztekkel ellátott programok programozási tételekre visszavezetett része viszonylag gyorsan ellenőrizhető, manuális bemenetekkel való futtatás nélkül is.

Az automatikus tesztkörnyezet kialakításához kerestünk egy egyszerű és praktikus eszközt, ami támogatja a tesztesetek leírását és futtatását. Ezt az eszközt választva a tapasztalatok alapján intuitív használata miatt nem okoz gondot a hallgatóknak megszerezni a tesztíráshoz szükséges magabiztosságot, és könnyen elsajátíthatják vele az automatikus tesztek készítéséhez kapcsolódó általános tudást is.

Irodalom

1. Gregorics, T.: *Programming theorems on enumerator*. Teaching Mathematics and Computer Science, Debrecen, 8/1 (2010), 89-108.
2. Gregorics, T.: *Abstract levels of programming theorems*. Acta Universitatis Sapientiae, Informatica, 4, 2 (2012), 247-259
3. Gregorics, T.*: Programozás* 1.kötet Tervezés. ELTE Eötvös Kiadó, 2013. (in Hungarian)
4. Gregorics, T., Sike, S.: *Generic algorithm patterns*, Proceedings of Formal Methods in Computer Science Education FORMED 2008, Satellite workshop of ETAPS 2008, Budapest March 29, 2008, p. 141-150.
5. Sommerville, I.: *Software Engineering*. Pearson (2011)
6. Gregorics, T.: *Force of Summation*. Teaching Mathematics and Computer Science, Debrecen, 10/1 (2014), 1-15.
7. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B
8. https://github.com/philsquared/Catch/blob/master/single_include/catch.hpp