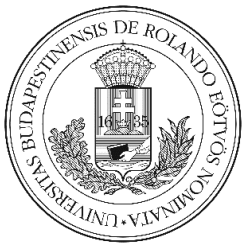


Eötvös Loránd Tudományegyetem

Informatikai Kar

Komputeralgebra Tanszék

Tesztttervezési technikák

Kovács Attila

Egyetemi docens

Mórocz Eszter Ilona

nappali, programtervező

matematikus szak

Budapest, 2018

Tartalomjegyzék

Bevezetés

Mai világunkban a különböző szoftver termékek életünk minden területén megjelennek: az orvostudományban, a pénzintézeteknél, a közlekedésben, a gyárakban, a szórakoztató iparban, stb... Egyes programok hibás működése csupán bosszantó, másikakon viszont életek múlhatnak. Ezért nagyon fontos a tesztelés.

Különböző szoftvereknél különböző céljai lehetnek a tesztelésnek, mint például kritikus hibák kiküszöbölése, a rendszer minőségének biztosítása, felhasználói élmény javítása. Mint ahogy a programok készítői, úgy a tesztelők sem tökéletesek, ráadásul egy összetettebb szoftverben a felhasználói esetek száma hatalmas, ezért többnyire esélytelen az összes lehetséges variációt kipróbálni, így a tesztelés sem tudja 100%-ban garantálni a termék hibátlanságát. Viszont a hibák előfordulása minimalizálható, és a megfelelő tesztesetek kiválasztásával a tesztelés ideje is a szükséges minimumra csökkenthető.

Ezért fontos, hogy ismerjük a különböző teszttervezési technikákat, hogy ki tudjuk választani a tesztelni kívánt szoftverhez a leghatékonyabbat, amivel a leggyorsabban a legtöbb hibát fel tudjuk fedni. Természetesen minden program különböző, így nem feltétlenül lehet egy az egyben ráhúzni egy tesztelési sablont az épp aktuális projektre, ezért a tesztelés megtervezése általában - a magas fokú szakértelmen kívül - sok kreativitást is igényel. A teszt tervezést célszerű már a szoftver tervezési fázisában elkezdeni. Boris Bezier szavait idézve:

> *„A tesztek végrehajtásán felül a teszt megtervezése az egyik legjobb megelőzése a hibák előfordulásának. Ugyanis mialatt azon gondolkodunk, hogy milyen használható teszteseteket hozzunk létre, rengeteg hibaforrást fedezhetünk fel, még mielőtt egy sort is kódoltunk volna. Éppen ezért a tesztelés (tervezés és/vagy végrehajtás) a szoftver termék létrehozásának minden egyes fázisában - kezdve a program megtervezésétől egészen a kódolásig - képes lecsökkenteni a hibák előfordulását."*
>
> − Boris Beizer, Software Testing Techniques [01]

Vagyis egy program fejlesztésén és tesztelésén dolgozó embereknek érdemes folyamatosan kommunikálni, nem csak akkor, amikor elkészült a termék fejlesztése.

A teszt tervezés első fázisában el kell dönteni (és leírni) néhány általános dolgot, hogy mit is tesztelünk (egy teljes szoftvert vagy annak csak egy kis részét), milyen

megközelítésből teszteljük az adott programrészt, mennyi erőforrást fordítunk a tesztelésre, és mik a határidők. Ekkor írjuk le, hogy milyen tesztfeladatokat kell majd végrehajtani, mik a szükséges feltételek az egyes tesztek elkezdéséhez, és mikor fejezhetjük be őket. A teszt tervezésének megközelítései nagyon különbözőek lehetnek, például specifikáción (black-box) vagy kódon alapuló (white-box), experimentális, azaz a tesztelő tapasztalatára támaszkodó, nagymértékben intuitív tesztelés vagy kimondottan a hibákat kereső.

Amint ezek a döntések megszülettek, elkezdhetjük kicsit közelebbről vizsgálni a készülő (vagy már elkészült) szoftver terméket. Ennek első lépéseként a megkapott írásos dokumentumokat (program specifikáció, felhasználói kézikönyv stb..) tanulmányozzuk. Ezután tudjuk meghatározni a különböző tesztelendő részeit a programnak, valamint a hozzájuk tartozó feltételeket. Például egy navigációs szoftver egyik része a cím beírás, ahol meg kell vizsgálni mondjuk, hogy az irányítószám megfelel-e a helyi szabályoknak (számjegyek száma, formátum) vagy hogy a települést, utcát házszámot milyen sorrendben kell megadni (mert Angliában például pont fordítva szokás, mint Magyarországon). Ha ez elkészült, akkor a következő lépésben szükséges megvizsgálni, hogy a meghatározott részekben milyen valószínűséggel fordulhatnak elő hibák, és ezek mekkora károkat okozhatnak. A navigációs szoftverek hibái általában inkább csak bosszúságot okoznak – bár néha olvasni olyan esetekről, hogy egy autós a tóba hajtott, mert a navigáció arra irányította – viszont ha majd az önvezető autókat fogják irányítani a szoftverek, akkor már komolyabb károkat is okozhat a nem megfelelő működés. Ezekből egy átfogó képet kapunk arról, hogy a tesztek futtatása során mely részekre kell jobban koncentrálni, és mik azok, amiket esetleg elhagyhatunk abban az esetben, ha nincs elegendő idő a részletes tesztelésre.

Miután elvégeztük ezeket az elemzéseket, jöhet a tesztesetek megtervezése. Itt következik a megfelelő teszttervezési technika kiválasztása, amiket ebben a munkában igyekszem szemléletesen és érthetően leírni. Célom, hogy a szakmabelieken túl azok is megértsék ezeket a technikákat, akik nem az informatika területéről érkeztek. Azért érzem ezt fontosnak, mert egy szoftver kidolgozásában rengetegen vesznek részt, akik nem feltétlenül jártasak a programozásban: a megrendelőtől elkezdve a befektetőkön keresztül a projekt menedzserekig. Fontos, hogy ők is értsék és elfogadják azt, hogy az adott tesztek hogyan és mennyire biztosítják a készülő program biztonságát, helyességét. Itt fontos megemlíteni azt is, hogy ezek a technikák adnak ugyan egy támpontot, de valós

rendszerek tesztelésének tervezésénél általában nem tisztán önmagukban fordulnak elő, hanem több technika keverve vagy kiegészítve egyéb ötletekkel, hiszen a mai szoftverek többnyire olyan összetettek, hogy egyetlen technikával nehezen lehetne lefedni minden részét.

Miután meghatároztuk, hogy milyen irányok mentén szeretnénk tesztelni, el kell kezdeni megtervezni a teszteseteket. Először általánosan írjuk le, hogy hol milyen működést várunk, például milyen típusú paraméterekre hogyan reagáljon a program (irányítószám megadásnál betűket ne fogadjon el, csak számokat), ezt absztrakt szintnek hívjuk. Ez után következik a konkrét tesztesetek meghatározása, amikor leírjuk, hogy pontosan milyen bemenő paraméterekkel milyen feladatokat hajtunk végre, és mi az elvárt működés vagy eredmény, amit látni szeretnénk (maradva az irányítószámos példánál adjuk meg, hogy 1211 és a program ebből ismerje fel, hogy Budapest 21.-ik kerületéről van szó). Egy teszteset lehet pozitív – mikor azt teszteljük, hogy egy-egy funkció működik-e – vagy negatív – amikor szándékosan a program hibás használatát szimuláljuk (például ha megadunk olyan irányítószámot, amely nem létezik, vagy hogy be tudunk-e írni betűket, és ha igen, arra hogy reagál a rendszer). Összességében elmondhatjuk, hogy egy jó teszteset az alábbi tulajdonságokkal bír:

- **pontos**: egyértelmű, hogy mit csinálunk, és mit várunk el
- **hatékony**: határozottan bizonyítja a program helyességét vagy hibáját
- **gazdaságos**: nincsenek benne felesleges műveletek, a lehető leggyorsabban végre lehet hajtani
- **fejleszthető**: ahogy a program fejlődik úgy a tesztesetet is hozzá kell tudni alakítani, nem jó, ha a program minden változtatásánál teljesen új teszteseteket kell kitalálni
- **reprezentatív**: fedjünk le minél több lehetséges esetet minél kevesebb példával
- **nyomonkövethető**: meg tudjuk mondani, hogy melyik előzetesen megfogalmazott elvárásokat fedi le
- **megismételhető**: vélhetően egy-egy tesztesetet többször is végre kell majd hajtani, így fontos, hogy meg tudjuk ismételni a lépéseit
- **újrahasznosítható**: különböző programok/programrészek hasonló viselkedését egyaránt le tudjuk vele tesztelni

A tesztesetek pontos meghatározásához figyelembe kell vennünk az úgynevezett tesztlefedettséget. Azaz, hogy a tesztjeink a programmal szemben támasztott elvárásokat milyen mértékben igazolják. Minél nagyobb tesztlefedettséget érünk el, annál jobban tudjuk a tesztekkel igazolni a program helyességét valamint felfedni a hibákat. Fontos megjegyezni, hogy ennek nincs köze a programkód lefedettségéhez, azaz nem arról beszélünk, hogy a tesztek lefuttatása során a kód mekkora részét érintjük. Különleges esetekben előfordulhat olyan is, hogy egy kódrészletet a program hétköznapi használata során sosem hajtunk végre, így ehhez a részhez nem tartozik elvárás, ezért azt nem is teszteljük. Ezek lehetnek véletlenül bent maradt kódrészek, vagy egy túl buzgó programozó munkája, aki minden lehetőséget lekódol, még akkor is, ha egyébként egy ágba sosem juthatna el a program egy átlagos felhasználás során. Ezek azért veszélyesek, mert itt lehet a legkönnyebben megtámadni a szoftvert, pontosan azért, mert az ilyen kódrészek általában nincsenek tesztelve. Ezért lehet olyan elvárás a teszteléssel szemben, hogy a programkód bizonyos százaléka le legyen fedve tesztesetekkel. Ezek olyan programoknál különösen fontosak, amelyek működése során egy meghibásodás komoly károkat okozhat. Például egy orvostechnikai alkalmazásnál nem fordulhat elő, hogy egy véletlen elgépelésen vagy félrekattintáson, vagy akár a szoftver elleni szándékos támadáson emberéletek múljanak!

Mikor mindezeket végig gondoltuk le kell generálni a konkrét paramétereket, majd lefixálni a teszteseteket, amik tartalmazzák az összes szükséges (fent részletezett) információt, mint például a futtatás körülményeit, a teszt végrehajtásának lépéseit, a pontos bemenő adatokat, és az elvárt viselkedést, beleértve a végeredményt és a program futás utáni állapotát is valamint a futtatás várható idejét. Minden tesztesetnek kell, hogy legyen egy egyértelmű azonosítója és neve is, amivel hivatkozni tudunk rá. Ha mindezeket leírtuk, célszerű elfogadtatni a szoftver elkészítésében érdekelt összes résztvevővel, ideértve a megrendelőt és a fejlesztőket is a későbbi vitás kérdések elkerülése végett.

Mivel jó esetben a teszt tervezése a fejlesztéssel párhuzamosan folyik, ezért mire elérkezünk oda, hogy meg vannak a leírt és elfogadott teszteseteink, addigra vélhetően lesz már egy elkészült programunk vagy programrészünk, amin elkezdhetjük futtatni ezeket a teszteket. Elengedhetetlenül fontos, hogy lejegyezzük (logoljuk) az eredményeket és körülményeket, ugyanis ha nem várt működéssel találkozunk, ezekből a logokból kiindulva tudják a fejlesztők javítani a hibát vagy megfejteni, hogy miért azt

az eredményt kaptuk. Az eredmények elemzése a vezetőség számára is fontos, mert ebből tudják eldönteni, hogy mik legyenek a további lépések. Kerüljenek ki bizonyos részek a programból, vagy nem kerül sok erőforrásba a hiba javítása, így érdemes vele foglalkozni, esetleg egy új funkcióval bővüljön a program. Extrém esetekben előfordulhat az is, hogy a tesztek tervezésében történt hiba, és újra kell tervezni a tesztet. Ilyenkor is sokat segítenek az elkészült logok, és az addigi futtatásokkal kapcsolatos tapasztalatok. Újra tervezés esetén természetesen szükséges az új teszteseteket ismét elfogadtatni a résztvevőkkel, hogy mindenki tudja, hogy onnantól már máshoz kell igazodni.

Érdekes és fontos kérdéskör még az automatizálás témaköre. Amikor egy folyamatban az automatizáció felmerül, akkor általában valami unalmas, nem kreatív, ismétlődő tevékenységet szeretnénk kiváltani, hiszen ezeket géppel is végre lehet hajtatni. A másik potenciális ok pedig, hogy a gépek pontosabban és gyorsabban tudnak végrehajtani folyamatokat, mint az emberek. Ahhoz azonban, hogy automatizáljuk a teszt tervezést, ismernünk kell, hogy milyen lehetőségeink és eszközeink vannak. A dolgozatom második felében ezzel foglalkozom.

Funkcionális tervezési technikák

A funkcionális tesztelés vagy specifikáció alapú tesztelés alatt azt értjük, amikor egy szoftver termék vizsgálatát az alapján végezzük, hogy milyen működést várunk el tőle. Általában van valami féle dokumentum arról, hogy hogyan szeretnénk, ha működne a program. Ez lehet az üzleti vagy a felhasználói elvárások leírása, a program funkcionális terve vagy akár egy felhasználói kézikönyv. Ezek elolvasásával és tanulmányozásával képet kaphatunk arról, hogy hogyan kéne a programnak működni, és ez alapján fel tudjuk írni, hogy milyen teszteket végezzünk. Funkcionális teszt tervezés esetén nem foglalkozunk a program belső felépítésével (többnyire nem is ismerjük), csak azzal, hogy milyen használatra milyen reakciót várunk. Ez arra hasonlít, mintha egy fekete dobozt nyomkodnánk, amiben nem tudjuk, hogy mi történik, csak azt, hogy mit nyomtunk meg, és arra mi lett a reakció, éppen ezért ezt sokszor hívjuk black-box tesztelésnek is. Funkcionális teszteket sok féle képpen tervezhetünk meg, az alábbiakban ezekből a technikákból fogok bemutatni jónéhányat. A végén pedig igyekszem egy rövid, de átfogó összehasonlítást adni, ami alapján könnyeben el lehet majd dönteni, hogy egy adott specifikációjú programhoz melyik technikát célszerű választani.

Ekvivalencia osztályozás

Az ekvivalencia osztályozás ötlete abból indul ki, hogy a program egy-egy paraméterének lehetséges értékeit feloszthatjuk hasonlóan viselkedő csoportokra. Ezek lesznek az ekvivalencia osztályok. A hasonló működésből adódóan minden osztályból elég egy elemre (reprezentánsra) tesztelnünk a programot, és ha erre a reprezentánsra helyes működést kapunk, akkor feltételezzük, hogy az adott osztály többi tagjával is jól fog működni a programunk. Vagyis ebben az esetben annyi tesztesetünk lesz, amennyi ekvivalencia osztályunk.

Például vizsgáljuk meg egy navigációs szoftvernél a különböző zoom szinteket. Nyilvánvaló, hogy nem jeleníthetünk meg minden apró utat, amikor messziről nézzük a térképet, viszont ahogy egyre közelítünk, egyre részletesebben kell mutatnunk az információkat. Tehát tegyük fel, hogy

- az autópályák az 1-es,
- az autóutak a 2-es,
- a főútvonalak a 3-as,

- a mellékútvonalak a 4-es,
- a földutak, kisebb utcák pedig az 5-ös

kategóriába tartoznak, és minél nagyobb a térkép zoom szintje (azaz minél közelebbről nézzük), annál több utat láthatunk rajta. Mondjuk ha a zoom szint 0-21 közötti egész szám lehet, akkor

- 4-nél vagy az alatt még ne látszódjanak utak
- 5 és 7 között az 1-es
- 8 és 9 esetén már a 2-es is
- 10 és 12 között a 3-as is
- 13-nál jelenjen meg a 4-es is
- 14-nél vagy a fölött pedig már az összes út látszódjon.

Az előző felsorolásból látszik, hogy 6 különböző ekvivalencia osztályunk van. Ha mindegyikből kiválasztunk egy zoom szintet, akkor pontosan fel tudjuk sorolni, hogy mely utaknak kell látszódni és mely utaknak nem szabad. Tehát a teszteseteink például a következők lehetnek:

teszteset sorszáma	**zoom szint**	**elvárt eredmény (utak típusaival megadva)**	
		látszódó utak	**nem látszódó utak**
1.	3		1, 2, 3, 4, 5
2.	6	1	2, 3, 4, 5
3.	8	1, 2	3, 4, 5
4.	11	1, 2, 3	4, 5
5.	13	1, 2, 3, 4	5
6.	18	1, 2, 3, 4, 5	

Fontos megjegyezni, hogy az ekvivalencia osztályoknak le kell fedni a teljes paraméterteret, azaz nem lehet olyan előforduló paraméter, amely nem tartozik egyetlen osztályba sem (hiszen akkor ennek a paraméternek a működését nem teszteltük). Valamint minden paraméterről egyértelműen el kell tudnunk dönteni, hogy melyik osztályba tartozik. Az ekvivalencia osztályok között nem lehet átfedés, hiszen akkor honnan tudjuk, hogy az adott paraméterünk épp melyik osztálynak megfelelően viselkedik. Például ha olyan navigációt fejlesztünk, ami máshogy viselkedik személyautós, motoros, kerékpáros vagy teherautós használat esetén, és a jogosítvány

száma alapján ajánljuk fel, hogy ki melyiket választhatja (tegyük fel, hogy a jogosítvány számából meg tudjuk állapítani, hogy milyen kategóriákat vezethet az illető), akkor nem lesz jó ekvivalencia osztályozása a felhasználóknak, hogy

- A kategóriás jogosítvánnyal rendelkezők
- B kategóriás jogosítvánnyal rendelkezők
- C kategóriás jogosítvánnyal rendelkezők

hiszen akinek nincsen jogosítványa, az egyik osztályba sem tartozik bele, valamint előfordulhat, hogy valaki például rendelkezik A és B kategóriás jogosítvánnyal is. Tehát itt tovább kell bontani a felsorolt 3 osztályt és hozzá venni a kimaradt felhasználókat:

- jogosítvány nélküliek
- csak A kategóriával rendelkezők
- A és B kategóriával is rendelkezők
- csak B kategóriával rendelkezők
- …

és így tovább, míg eljutunk egy olyan osztályozáshoz, ahol mindenkiről meg tudjuk állapítani, hogy pontosan melyik az az egyetlen osztály, amibe beletartozik.

Ez a módszer akkor nagyon hasznos és hatékony, ha kevés ekvivalencia osztállyal le tudjuk fedni az állapotteret. Azaz találunk olyan egyértelmű felosztást, amely szerint a program bemenő paramétereit úgy tudjuk csoportosítani, hogy egy-egy csoportba a lehető legtöbb olyan elem legyen, amelyek a program futása során ugyanúgy viselkednek. Minél kevesebb ekvivalencia osztályunk van, annál kevesebb tesztet kell futtatni. Viszont figyelni kell arra is, hogy nehogy túl nagyra vegyük az osztályokat, és a különböző elemei esetleg eltérően viselkedjenek. Például a zoom szintes példában van olyan ekvivalencia osztály, amelynek csak egy eleme van (13-as zoom szinten fordul egyedül elő, hogy a 4-es szintű utak látszanak, de az 5-ösek még nem), viszont nem lehet egyetlen másik osztályba sem beolvasztani.

Eddig a pozitív eseteket néztük, azaz amikre a programnak valamilyen valós eredményt kell produkálni, azonban az ekvivalencia osztályzás kiterjeszthető a negatív más néven invalid esetekre is, vagyis azokra az esetekre, amik az elvárásainkon kívül esnek. Például ha azt vesszük, hogy bármely nem negatív egész számot meg lehet adni zoom szintnek, akkor a 21-nél nagyobb számokra már nincsen meghatározva, hogy mi történjen.

Előfordulhat azonban, hogy a felhasználó ilyen számot ad meg (mert például nem tudja, hogy mi a maximális érték). Ezért ezt az esetet is tesztelnünk kell. Így lesz még egy ekvivalencia osztályunk, amivel a hibás paraméter megadást tudjuk tesztelni. Így gyakorlatilag 7 tesztesettel le tudjuk ellenőrizni a programunk működését, és kiszűrni a hibák nagy részét. A jogosítvány szám megadásánál pedig azok az esetek tartoznak az invalid kategóriába, amikor egy nem létező jogosítvány számot adunk meg. Ez könnyen előfordulhat elgépelés esetén, így mindenképpen érdemes ellenőrizni, hogy ilyenkor mit csinál a programunk.

Határérték-elemzés

A határérték-elemzés (vagy határérték analízis) szorosan kapcsolódik az ekvivalencia osztályozáshoz. Az alap feltevése az, hogy a hibák legtöbbször az ekvivalencia osztályok határain vagy azok közelében fordulnak elő. Ez azért van általában így, mert lehetséges, hogy a határ közelében lévő elemeket tévedésből a másik osztályba soroljuk, és ez okozhat hibás működést. Ez azoknál az ekvivalencia osztályozásoknál fordulhat elő, ahol egy-egy osztályt valamilyen tól-ig értékkel, többnyire intervallumokkal adunk meg. Amennyiben valamilyen tulajdonság alapján egyértelműen eldönthető osztályozásról beszélünk, ott természetesen nincs értelme a határérték-elemzésnek.

Például a jogosítványszám alapján történő osztályozásnál nem tudunk ilyen határokat mondani. Ezzel szemben, ha a másik példánkban megengedjük, hogy a zoom szintek törtek legyenek, és ennek megfelelően a feltételeket kicsit máshogy fogalmazzuk meg, mégpedig a következő képpen:

- 5-nél kisebb zoom szinten nem látszanak az utak
- 5 és 8 közötti zoom szinten látszanak az autópályák
- 8 és 10 közötti zoom szinten már az autóutak is látszanak
- 10 és 13 között megjelennek a főútvonalak
- 13 és 14 között már a mellékútvonalakat is kirajzoljuk
- 14 fölött pedig a teljes úthálózat (beleértve a legkisebb földutakat is) látszódjon

akkor már rögtön nem olyan egyértelmű, hogy mi történik a határokon. Azaz, hogy például 8 esetén látsszanak-e a 2-es szintű utak, vagy még nem. Ha ez nincs pontosan meghatározva, vagy a program készítése során rosszul lesz lekódolva, akkor máris nem egyértelmű, hogy mi történik a határokon, és előfordulhat, hogy máshogy működik a programunk, mint ahogy elvárnánk.

Egy határ lehet nyitott vagy zárt. Azaz ha pontosan tudjuk, hogy melyik az osztály utolsó eleme, ami a másik osztályhoz a legközelebb esik, akkor a határ zárt (matematikai eszközökkel kifejezve <=, >= esetén). Viszont ha csak azt tudjuk, hogy egy elem nincs már benne az osztályban, de a hozzá közel esők igen, viszont ezek közül nem tudjuk megmondani a legközelebbit, akkor ez nyitott (ezt írjuk le általában a <, > relációs jelekkel).

A feltevés tehát az, hogy ahol két ekvivalencia osztály találkozik, ott érdemes részletesebben vizsgálni a programot. Ezért a határérték-elemzés technikája szerint minden egyes határhoz tartozik egy-egy teszteset a szomszédos ekvivalencia osztályokból, méghozzá úgy, hogy az egyik a határon legyen, a másik pedig ahhoz lehető legközelebb. Amennyiben a határ része a vizsgált ekvivalencia osztálynak, úgy abból kifele, azaz a másik szomszédos osztály fele kell lépni, ha viszont nem része, akkor befele, hogy az adott osztályból is legyen tesztesetünk. Ahhoz, hogy megmondjuk mit jelent a „lehető legközelebb”, mindenképpen definiálnunk kell egy lépésközt. Ha ez mind megvan, meg tudjuk határozni a teszteseteinket.

Vizsgáljuk meg ezzel a szemlélettel a példánkat. Most célszerű az elvárásainkat a következő formában megadni:

- 0 <= zoom szint < 5 : nem jelenik meg út
- 5 <= zoom szint < 8 : 1-es típusú utak
- 8 <= zoom szint < 10 : 1-es és 2-es típusú utak
- 10 <= zoom szint < 13 : 1-es, 2-es, 3-as típusú utak
- 13 <= zoom szint < 14 : 5-ös kivételével mindegyik típusú út
- 14 <= zoom szint <= 21 : mindegyik út

Ha még mindig feltételezzük, hogy negatív számot nem tudunk megadni, akkor egy olyan ekvivalencia osztályunk lesz, amivel az érvénytelen értékeket vizsgáljuk, méghozzá a 21-nél nagyobb zoom szintek. (Ha be lehetne adni negatív számot is, akkor a másik invalid osztályunk a 0-nál kisebb számokból állna.) Mivel most már nem csak egész számokról beszélünk, így a lépésköz nem egyértelmű. Ilyen esetben meg kell egyezni, egy megfelelő lépésközben, ez a zoomolásnál legyen mondjuk 0,1. Ezekből az információkból már fel tudjuk írni a teszteseteinket.

teszteset sorszáma	zoom szint	elvárt eredmény (utak típusaival megadva)	
		látszódó utak	nem látszódó utak
1.	0		1, 2, 3, 4, 5
2.	4,9		1, 2, 3, 4, 5
3.	5	1	2, 3, 4, 5
4.	7,9	1	2, 3, 4, 5
5.	8	1, 2	3, 4, 5
6.	9,9	1, 2	3, 4, 5
7.	10	1, 2, 3	4, 5
8.	12,9	1, 2, 3	4, 5
9.	13	1, 2, 3, 4	5
10.	13,9	1, 2, 3, 4	5
11.	14	1, 2, 3, 4, 5	
12.	21	1, 2, 3, 4, 5	
13.	21,1	hibás bemenő adat!	

Látszik, hogy itt kétszer annyi pozitív tesztesetünk van, mint az ekvivalencia osztályozásnál, illetve hogy gyakorlatilag minden ekvivalencia osztályból két elemet is vizsgálunk. Ez feleslegesnek tűnhet, viszont amennyiben a program jól működik a különböző osztályokra, csak a határok lettek hibásan leprogramozva, akkor ezt a határérték-elemzés során létre hozott tesztekkel gyorsan ki lehet deríteni.

Döntési táblák

A döntési táblák leginkább akkor nagyon hasznosak, ha több körülmény figyelembe vételével kell meghatároznunk, hogy mit várunk el a programtól. Általában különböző állítások formájában fogalmazzuk meg a körülményeket illetve bemenő paramétereket valamint külön a lehetséges eredményeket. Mindegyik állítás a táblázat egy-egy sora lesz, méghozzá úgy, hogy a felső részben lesznek a bemenő paraméterekre vonatkozók, amik alapján majd kiszámítjuk az alsó részben lévő eredmények értékét. A táblázat 2^n oszlopból fog állni, ahol n a bemenő adatok száma, ugyanis az oszlopokba a paraméterek minden lehetséges kombinációját fel kell írni.

Nézzünk egy olyan példát, amikor egy adott helyet keresünk a térképen. Ehhez meg kell adnunk egy létező várost, és abban egy címet (utca és házszám). Ha ez megvan, akkor

legyen a keresett hely a kijelzőnk (képernyőnk) közepén, a zoom szint pedig legyen 17 (ebben az esetben már elég közel vagyunk, hogy látszódjon minden utca, viszont még egy egész átlátható képet kapunk a környékről is). Csak városra is tudunk keresni, ekkor azonban nem nézzük olyan közelről a térképet, így a zoom szint most 11 legyen, és a város közepe legyen a kijelzőnk közepén. Ekkor a kisebb utcákat ugyan már nem látjuk, de a városról kaphatunk egy átfogó képet. Ha várost nem adtunk meg csak utcát, akkor nem egyértelmű a cím, hiszen több városban is lehet ugyanaz az utca. Ekkor egy „Ismeretlen cím" hibaüzenetet írjunk ki a felhasználónak. Az egyszerűség kedvéért, most vegyük egy esetnek, ha a felhasználó hibás adatot adott meg, vagy ha nem adott meg adatot. Hibaüzenet esetén természetesen sem zoom szintet sem koordinátákat (azaz térkép nézetet) nem állítunk.

Ebből a leírásból a következő bemenő feltételeket állapíthatjuk meg:

1. Megadtunk egy létező várost
2. Megadtunk egy létező címet (utca, házszám)

Valamint a következőket várhatjuk el:

1. Középen látjuk a keresett helyszínt
2. Zoom szint 17
3. Zoom szint 11
4. Hibaüzenet

A döntési táblánknak ezek lesznek a sorai, és mivel 2 bemenő állításunk volt, ezért 2^2 azaz 4 kitöltendő oszlopa lesz, tehát a következő képpen fog kinézni:

Megadtunk egy létező várost				
Megadtunk egy létező címet				
Középen látjuk a keresett helyszínt				
Zoom szint 17				
Zoom szint 11				
Hibaüzenet				

Ha felrajzoltuk a döntési táblánkat, akkor elkezdhetjük kitölteni. A táblának először a felső – bemenő paraméterekre vonatkozó – részével foglalkozunk. Ezt úgy célszerű kitölteni, hogy először az első sor feléig igazakat, utána hamisat írunk be, a következő

sorban ugyanezt csináljuk, csak itt a felette lévő azonos értékek feléig töltjük ki igazzal, utána hamissal, majd ismét igaz és hamis. És ezt a mintát folytatjuk, amíg a felső részt teljesen ki nem töltöttük. Miután ez megvan, minden oszlopban megkapjuk a bemenő paraméterek valamilyen kombinációját. (Ha jól töltöttük ki a táblát, akkor az összes különböző lehetőség előáll.) Ezek után megnézzük, hogy az adott kezdeti feltételek esetén mit várunk el a programtól, és ezeket beírjuk a táblázat aljába. Az előző példa kitöltött döntési táblája így fog kinézni:

Megadtunk egy létező várost	Igaz	Igaz	Hamis	Hamis
Megadtunk egy létező utcát	Igaz	Hamis	Igaz	Hamis
Középen látjuk a keresett helyszínt	Igaz	Igaz	Hamis	Hamis
Zoom szint 17	Igaz	Hamis	Hamis	Hamis
Zoom szint 11	Hamis	Igaz	Hamis	Hamis
Hibaüzenet	Hamis	Hamis	Igaz	Igaz

A táblázat mérete beláthatóan nagyon gyorsan képes nagyon nagyra nőni, ezért túl sok állítás esetén kezelhetetlenné tud válni. Ebben az esetben, ha lehet, akkor több részre érdemes bontani a tesztfeladatot. Ha ez nem lehetséges, akkor meg kell vizsgálni, hogy van-e lehetőség, hogy a döntési táblát egy kicsit „összecsukjuk”, azaz összevonjunk olyan oszlopokat, amiknek az eredménye nem függ az összes bemenő állítástól. Ha ilyen nem fordul elő, akkor a túlzottan nagy döntési táblák helyett, lehet, hogy érdemes másik tesztelési technikát választani.

Vegyük mondjuk a példánkban külön, hogy megadtuk az adatokat és hogy a beírt hely létezik-e, akkor eggyel több sorunk lesz, amiből az következik, hogy kétszer annyi oszlopunk. A döntési táblánk így fog kinézni:

Megadtunk egy várost	I	I	I	I	H	H	H	H
Megadtunk egy címet	I	I	H	H	I	I	H	H
A megadott adatok helyesek	I	H	I	H	I	H	I	H
Középen látjuk a keresett helyszínt	I	H	I	H	H	H	H	H
Zoom szint 17	I	H	H	H	H	H	H	H
Zoom szint 11	H	H	I	H	H	H	H	H
Hibaüzenet	H	I	H	I	I	I	I	I

Láthatjuk, hogy a táblázat utolsó négy oszlopának alsó része (azaz, hogy milyen működést várunk a programtól) megegyezik. Ez logikus is, hiszen ha nem adunk meg várost, akkor mindegy, hogy megadtunk egy létező utcát vagy sem, nem tudjuk megtalálni a címet. Ebben az esetben fordulhat elő az, hogy ezeket az oszlopokat összevonjuk és a fenti 8 oszlop helyett már is csak 5 lesz. Ilyenkor a táblázatba azokat az értékeket írjuk csak be, amitől függ a végeredmény, a többit üresen hagyjuk:

Megadtunk egy várost	I	I	I	I	H
Megadtunk egy címet	I	I	H	H	
A megadott adatok helyesek	I	H	I	H	
Középen látjuk a keresett helyszínt	I	H	I	H	H
Zoom szint 17	I	H	H	H	H
Zoom szint 11	H	H	I	H	H
Hibaüzenet	H	I	H	I	I

Végül a táblázat különböző oszlopait figyelembe véve meg tudjuk alkotni az egyes teszteseteket. A fenti példánkhoz a következő teszteseteket írhatjuk:

Sorszám	**Bemenő paraméterek**	**Elvárt viselkedés**
1.	Város: Gödöllő Utca: Kossuth Lajos utca 12	Zoomszint: 17 A képernyő közepén látjuk a Kossuth Lajos utca 12 számú házat
2.	Város: Gödöllő Utca: Kossuth Alajos utca 12	A képernyőn az alábbi hibaüzenet jelenik meg: „Ismeretlen cím”
3.	Város: Gödöllő Utca: -	Zoomszint: 11 A képernyő közepén látjuk Gödöllő város térképét
4.	Város: G9döllő Utca: -	A képernyőn az alábbi hibaüzenet jelenik meg: „Ismeretlen cím”
5.	Város: - Utca: Kossuth Lajos utca 12	A képernyőn az alábbi hibaüzenet jelenik meg: „Ismeretlen cím”

Ok-okozati (vagy cause-effect) gráfok

A döntési táblához hasonlóan a cause-effect gráfokkal is azt írjuk fel, hogy a program különböző bemenő paraméterek kombinációjára hogyan reagáljon. Ez a módszer a Boole-algebrára épül, azaz logikai állításokat teszünk, és azokat kombináljuk, amiből megmondjuk, hogy milyen eredményeket várunk el. Tehát ezt a módszert jellemzően olyan esetekben használjuk, amikor a program működése egyszerre több bemenő paramétertől is függ. Akár csak a döntési táblánál itt is először felírjuk az okokat (kezdeti állítások), és az okozatokat, egy-egy logikai állítással. Ha egy állítás igaz, az azt jelenti, hogy a benne megfogalmazott feltétel teljesül. Ezek után mindegyik állításhoz hozzárendelünk egy azonosítót. Ezt célszerű úgy megtenni, hogy külön jelöljük az okokat, és külön az okozatokat, például minden ok azonosítója C-sorszám, az okozatoké pedig E-sorszám.

Ezek alapján a címkeresést az alábbi állításokkal írhatjuk fel:

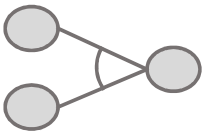
- C-1: Város megadva
- C-2: Utca, házszám megadva
- C-3: A beírt adatok helyesek
- E-1: Térkép koordinátáinak beállítása és zoom szint 17-re
- E-2: Térkép koordinátáinak beállítása és zoom szint 11-re
- E-3: Hibaüzenet: „Ismeretlen cím”

Amint ezek megvannak, elkezdhetjük felrajzolni a gráfot. Ehhez csupán néhány egyszerű jelölést elég ismerni, melyekre a legbonyolultabb feltételrendszereket is vissza tudjuk vezetni:

 A gráf csúcsai az egyes állítások lesznek. Az okokat és okozatokat ugyanúgy jelöljük, de a gráf minden csúcsát fel kell címkéznünk, valamint a könnyebb átláthatóság érdekében az okokat a baloldalon, az okozatokat pedig a jobb oldalon helyezzük el.

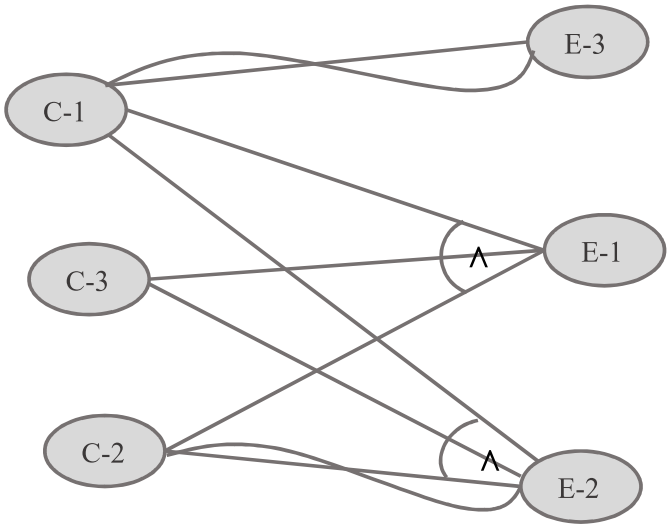
 A gráf élei jelölik a kapcsolatot az okok és okozatok között.

 Ha egy ok nem teljesüléséből következik egy okozat, akkor a kettő közti kapcsolatra egy hullámvonalat teszünk (negálás).



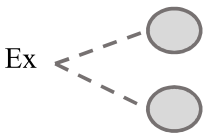
Valamint ha több okból következtetünk egy okozatra, akkor az okok között fennállhat „és” (∧) kapcsolat, azaz ha minden ok egyszerre teljesül, illetve „vagy” (∨) kapcsolat, azaz ha az okok közül legalább az egyik teljesül, akkor várjuk el az okozat teljesülését.

A fenti címkeresés példa ok-okozati gráfja így fog kinézni:

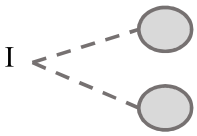


Látszik, hogy a csúcsok nem a számozás sorrendjében helyezkednek el. Ezt az átláthatóság kedvéért tetszőlegesen variálhatjuk, de arra figyeljünk, hogy az okok határozottan különüljenek el az okozatoktól.

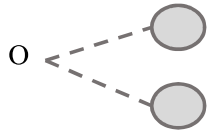
Az „és” kapcsolat egyértelmű, hiszen csak és kizárólag akkor lesz igaz, ha mindkét feltétel teljesül, viszont ha két ok között „vagy” kapcsolat áll fenn, azt már nem tudjuk ilyen egyszerűen meghatározni, hiszen ha a kettő közül bármelyik igaz, akkor már igazat kapunk. Olyan esetek is előfordulhatnak viszont, amikor a két állítás közül csak az egyik lehet igaz, vagy ha az egyik állítás igaz, akkor a másik is. Ezek miatt a további feltételek miatt találták ki az alábbi a jelöléseket:



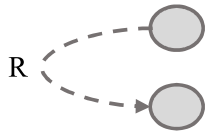
Exclusive, vagyis kizáró vagy: a két csúcs közül maximum az egyik lehet igaz, de lehet, hogy egyik sem az.



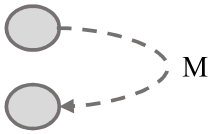
Inclusive: A két csúcs közül legalább az egyiknek mindig igaznak kell lenni.



Only: A két csúcs közül pontosan az egyiknek igaznak kell lenni



Require, azaz következmény: Ha az első csúcs igaz, akkor a második is az.



Mask: Ha az első csúcs igaz, akkor a másodiknak hamisnak kell lenni. Ezt a megkötést az effecteknél használhatjuk

Ezeknek a jelöléseknek a használatával már egészen komplex összefüggéseket is tudunk jelölni egy egyszerű ok-okozati gráfon. Bizonyos esetekben (bonyolultabb feltételrendszernél) előfordulhat, hogy köztes csúcsokat is beszúrunk, amikor több állítás meglétét mindenképpen együtt vizsgáljuk, és ebből és még más okokból következtetünk valamilyen okozatra. Ezeket a csúcsokat I-sorszám azonosítóval jelöljük.

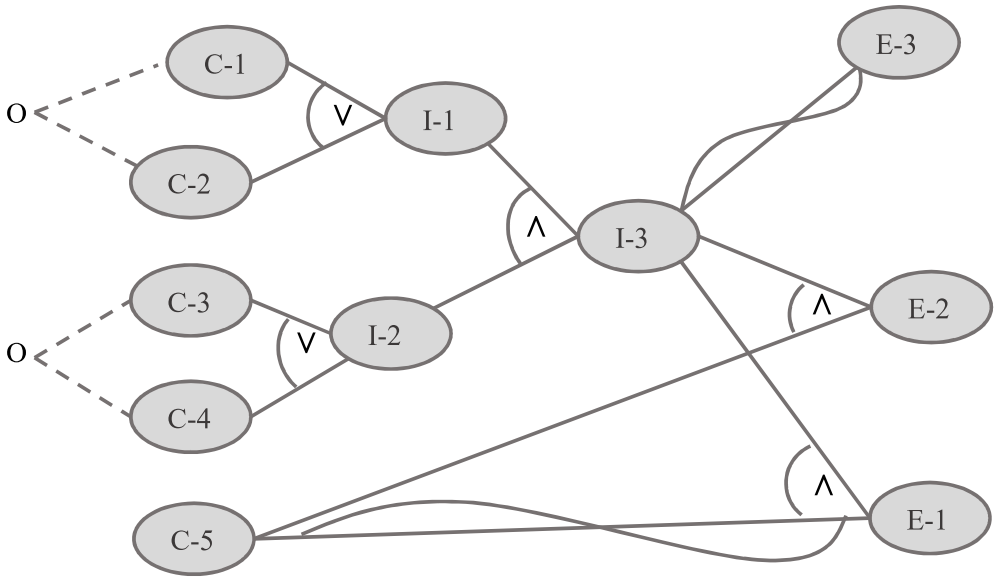
Pédául útvonaltervezésnél meg kell adnunk egy kiindulási pontot és egy érkezési pontot, mindkettőt megadhatjuk cím alapján (itt a cím lehet város, utca, házszám formában, de lehet egy ismert helyszín neve is) vagy koordinátákkal. Ha a két pont közti útvonalon díjat kell fizetni, akkor ezt jelezze a rendszer, ha nem, akkor rajzolja ki az útvonalat. Ha valamelyik pont nincs megadva, akkor írjon ki egy hibaüzenetet.

Ehhez a kis feladat leíráshoz a következő állítások tartoznak:

- C-1: Kiindulási cím megadva
- C-2: Kiindulási koordináták megadva
- C-3: Érkezési cím megadva
- C-4: Érkezési koordináták megadva
- C-5: Adott útvonalon fizetős útszakasz
- I-1: Kiindulási pont megadva
- I-2: Érkezési pont megadva
- I-3: Kezdő és végpont megadva
- E-1: Útvonal kirajzolása
- E-2: Üzenet: „Fizetős útszakasz”

- E-3: Üzenet: „Hiányzó helyszín”

Az ok-okozati gráf pedig a következő:



Miután felrajzoltuk a feladathoz tartozó ok-okozati gráfot, az ábra segítségével fel tudjuk írni a teszteseteket. Érdemes az egyes okozatokból kiindulva felírni az okok különböző variációit. Tehát ha például az E-1-et akarjuk elérni, akkor C-5 hamis, és az I-3 igaz. Ha I-3 igaz, akkor I-1-nek és I-2-nek is igaznak kell lenni, ami úgy érhető el, hogy C-1 és C-2 közül valamelyik, illetve C-3 és C-4 közül valamelyik igaz. Tehát a következő kombinációk valamelyikével: (C-1, C-3), (C-1, C-4), (C-2, C-3), (C-2, C-4). Azaz négy olyan tesztesetünk lesz, amiben E-1 eredményt kell, hogy kapjunk. Ugyanígy vezethető le E-2-re is négy teszteset. Az E-3-ra viszont öt esetünk lesz, méghozzá, ha csak C-1 igaz vagy csak C-2 vagy csak C-3 vagy csak C-4 vagy egyik sem, hiszen ez mind azt jelenti, hogy nincsen elég adatunk, ahhoz, hogy útvonalat tudjunk tervezni. Ezek alapján az alábbi teszteseteket írhatjuk a fenti példához:

Sorszám	**Bemenő paraméterek**	**Elvárt viselkedés**
1.	Start: Budapest, Blaha Lujza tér Cél: Esztergomi Bazilika	Útvonal mutatása
2.	Start: Budapest, Blaha Lujza tér Cél: 47.799080, 18.736792	Útvonal mutatása
3.	Start: 47.506446, 19.038669 Cél: Esztergomi Bazilika	Útvonal mutatása

4.	Start: 47.506446, 19.038669 Cél: 47.799080, 18.736792	Útvonal mutatása
5.	Start: Budapest, Batthyányi tér Cél: Fonyód, Süllő utca 1-3	Üzenet: „Fizetős útszakasz”
6.	Start: Budapest, Batthyányi tér Cél: 46.753923, 17.571166	Üzenet: „Fizetős útszakasz”
7.	Start: 47.506446, 19.038669 Cél: Fonyód, Süllő utca 1-3	Üzenet: „Fizetős útszakasz”
8.	Start: 47.506446, 19.038669 Cél: 46.753923, 17.571166	Üzenet: „Fizetős útszakasz”
9.	Start: Szépasszonyvölgy Cél: -	Üzenet: „Hiányzó helyszín”
10.	Start: 47.890428, 20.359053 Cél: -	Üzenet: „Hiányzó helyszín”
11.	Start: - Cél: Szépasszonyvölgy	Üzenet: „Hiányzó helyszín”
12.	Start: - Cél: 47.890428, 20.359053	Üzenet: „Hiányzó helyszín”
13.	Start: - Cél: -	Üzenet: „Hiányzó helyszín”

Cause-effect gráfok segítségével sokszor könnyebben ki tudjuk tölteni a döntési táblát. Bizonyos esetekben pedig nem is érdemes döntési táblát használni, mert sok olyan állítás kombináció állhat elő benne, amivel nincs értelme foglalkozni, mert nem fordulhat elő a program futása közben. Például, ha a fenti ábrát vizsgáljuk, akkor a C-1 és C-2 csúcsok közül az egyiknek kell igaznak lenni, sőt ha a programba egy helyre írhatjuk be a kiindulási helyszínt – ami vagy egy szöveg, és akkor címként kezeljük vagy két szám, és akkor koordinátaként kezeljük – akkor a kettő közül maximum az egyik teljesülhet. Tehát a döntési táblának értelmetlen az az oszlopa amelyben C-1 és C-2 is igaz. Valamint a C-5 állítással csak akkor kell foglalkoznunk, ha I-3 igaz. Ez a program futása során is így lesz, hiszen amíg nincs kitöltve a kiindulási és az érkezési hely, addig nem tervezünk utat a kettő között, így nem is tudjuk, hogy azon van-e útdíj. Ebben az esetben fordulhat elő az, hogy összecsukjuk a döntési táblát, és C-5 értékét csak akkor írjuk bele, amikor értelmezhető lesz. Ezek alapján a feladathoz az alábbi döntési táblát rajzolhatjuk:

I-1	Igaz	Igaz	Igaz	Hamis	Hamis
I-2	Igaz	Igaz	Hamis	Igaz	Hamis
C-5	Igaz	Hamis	-	-	-
E-1		Igaz			
E-2	Igaz				
E-3			Igaz	Igaz	Igaz

Ahol az I-2 és I-2 az alábbiak szerint áll elő:

C-1	I	I	H	H
C-2	I	H	I	H
I-1	-	I	I	H

C-3	I	I	H	H
C-4	I	H	I	H
I-2	-	I	I	H

Az ok-okozati gráf felrajzolása különösen hasznos azoknak, akik vizuális gondolkodásúak, és így a gráf segítségével jobban át tudják látni, hogy miből mi következik, mintha egy táblázatból kéne kiolvasni. Amikor azonban nagyon sok éle van a gráfnak, mert minden okból következtetünk több dologra is attól függően, hogy mi a többi ok, akkor nagyon átláthatatlanná tud bonyolódni. Ebben az esetben célszerűbb a döntési táblát használni.

Véges automaták

Minden programnak vannak különböző jól meghatározható állapotai. Egy navigációs szoftver nagyon elnagyolt állapotai például: a cím megadására vár, megadott címet mutat, vagy útvonal opciókat ajánl. Amikor valamiféle külső behatás történik, például a felhasználó megnyom egy gombot, beír egy szöveget, vagy akár letér a navigáció által kijelölt útról, akkor kerül át a program az egyik állapotból a másikba. Azt, hogy pontosan melyik állapotból, milyen behatásra, melyik másik állapotba kerül a program, véges automatával írjuk le.

A véges automaták lehetnek determinisztikusak vagy nemdeterminisztikusak. A determinisztikus véges automatát gyakorlatilag egy olyan függvény határozza meg, amely egy állapotban egy adott bekövetkező eseményhez megmondja, hogy milyen állapotba kerülünk át. Ugyanaz az esemény bekövetkezhet különböző állapotokban is (például egy gomb lenyomása), és ekkor különböző állapotokba is kerülhetünk, viszont

ha egy adott állapotban bekövetkezik egy adott esemény, akkor az mindig ugyanabba az állapotba visz át.

A nemdeterminisztikus automatáknál ez nem mondható meg ilyen egyértelműen. Ott előfordulhat, hogy egy adott állapotból egy adott esemény hatására egyszer egyik, máskor másik esemény következik be. Például ha a navigációban van egy olyan funkció, hogy ajánljon egy éttermet a környéken, mondjuk 1 km-es körzeten belül, de ezen túl nem szabunk meg más feltételt, akkor ha több étterem is van az 1 km-es körzetünkben, akkor lehet, hogy egyszer az egyiket másszor a másikat ajánlja a program. A nemdeterminisztikus működés mindig valamilyen véletlen eseményt jelent, ezért a legtöbb programot, aminek a működése kiszámítható, determinisztikus automatával tudjuk leírni.

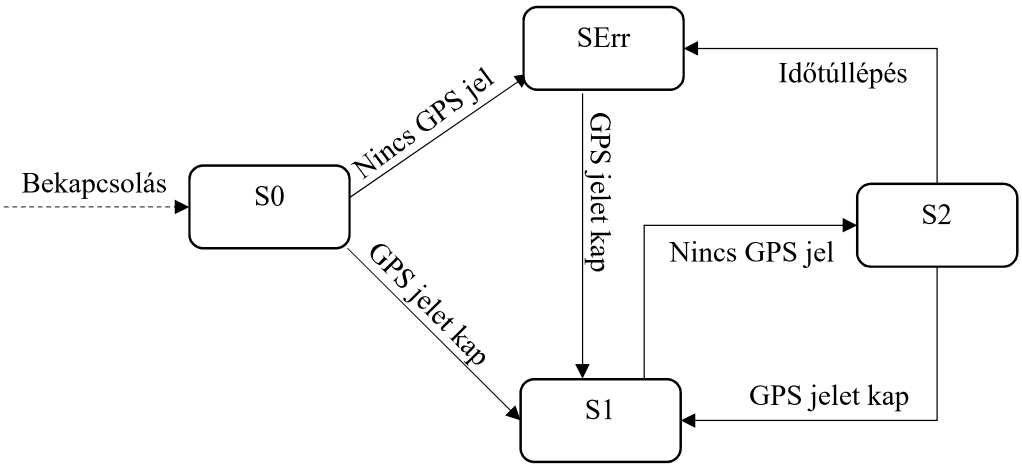
Vizsgáljuk például a navigációs eszköz állapotait, és állapot átmeneteit. Miután bekapcsoljuk az eszközt, vegyük úgy, hogy üzemkész állapotban várja, hogy kapjon GPS jelet, és meg tudja jeleníteni a pozíciónkat a térképen. Ha ez nem történik meg, akkor a kijelzőn megjelenik egy hibaüzenet, hogy „GPS jel nem található”. Bármikor, amikor kap GPS jelet (akár üzemkész állapotban van, akár a hibaüzenetet jelzi), akkor elkezdi mutatni, hogy hol vagyunk. Ha a GPS jel ezután szűnik meg, például bemegyünk egy alagútba, akkor egy ideig még az addigi sebesség alapján kiszámolja, hogy hozzávetőlegesen merre vagyunk, és ezt mutatja, majd amikor újra kap jelet, akkor már a GPS szerinti pozíciót jelzi újra. Ha viszont sokáig nem kap jelet, akkor ismét megjelenik a kijelzőn a hibaüzenet.

Ezeket az állapot átmeneteket szépen fel lehet rajzolni egy gráfba, aminek segítségével egész könnyen át tudjuk látni a program működését. Ez a determinisztikus és nemdeterminisztikus automatákra egyaránt igaz. Itt a gráf csúcsai a program különböző állapotai lesznek, az élek pedig a lehetséges átmenetek, amiken fel kell tüntetni, hogy mi az az esemény, aminek hatására az állapotváltozás megtörténik. Szerencsés esetben ezt az állapot átmenet gráfot megkapjuk a program specifikációjával együtt. Ha nem, akkor a meglévő dokumentumokból, leírásokból nekünk kell felrajzolni. Mivel ez egy elég jól szemlélteti a program működését érdemes megvitatni az ügyféllel az elkészült gráfot, mert ez alapján sok félreértés kiderülhet akár a teszteléssel akár a fejlesztéssel kapcsolatban.

Vizsgáljuk meg az előző példát. Az automatának a következő állapotai lesznek:

- S0: Üzemkész
- S1: Pozíciót mutat GPS jel alapján
- S2: Pozíciót mutat becslés alapján
- SErr: „GPS jel nem található” Hibaüzenetet mutat

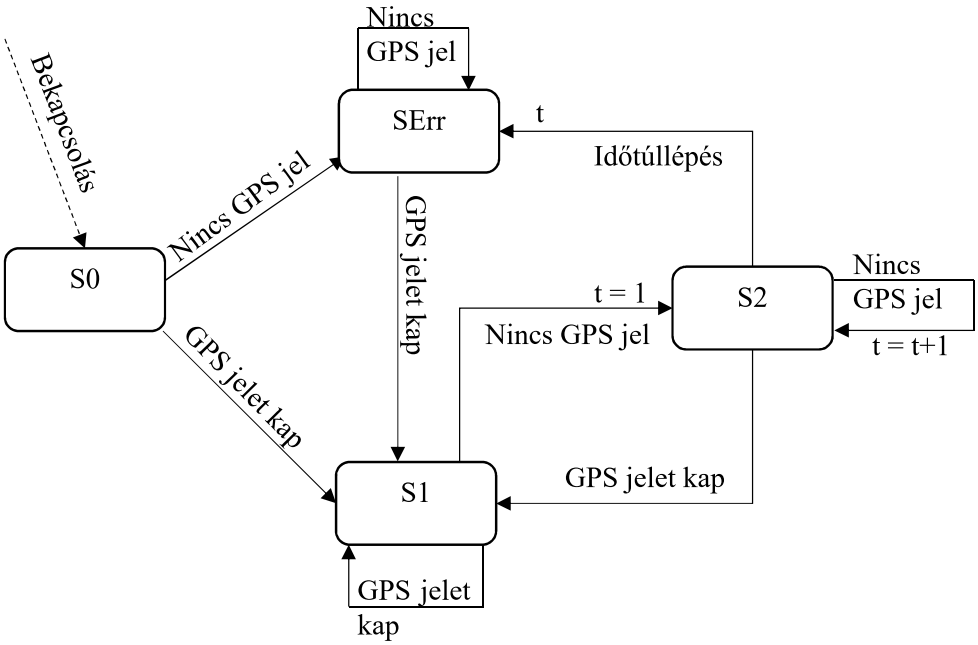
Az állapot átmeneteket pedig a GPS jel eltűnése vagy megléte okozza. Ez alapján a következő állapot átmeneti gráfot rajzolhatjuk fel:



Látszik az ábrán, hogy a „Nincs GPS jel” esemény különböző állapotokban különböző eredményre vezet. Viszont minden egyes állapotból egy adott esemény mindig ugyanabba az állapotba visz. Tehát ha az S1 állapotban az eszköz nem kap GPS jelet, akkor mindig az S2 állapotba fog kerülni, és nem fordul elő olyan, hogy az S0-ba vagy az SErr-be.

A „Bekapcsolás”-t szaggatott vonal jelöli, mert az gyakorlatilag a teszteléshez szükséges kezdő feltételt adja meg. Ezt az ’’átmenetet’’ nem vizsgáljuk, mert ha ez a feltétel nincs meg, akkor az egész tesztelést el sem tudjuk kezdeni.

Ha úgy tekintünk az automatára, hogy minden időpillanatban vagy kap GPS jelet vagy nem, és az S2 állapotból az időtúllépést úgy definiáljuk, hogy 10 időegységen keresztül nem kap jelet, az S2 állapotban töltött időt pedig egy paraméterben számoljuk, akkor a fenti ábrát kiegészíthetjük még néhány állapotátmenettel, valamint az éleken a kimenő paraméterek értékét is jelöljük.



Amikor véges automata alapján írunk teszteseteket, akkor különböző hosszúságú szekvenciákat vizsgálunk. A legrövidebb szekvencia, mikor egyetlen átmenetet hajtunk végre. Ekkor egy adott állapothoz végrehajtunk egy eseményt, és megvizsgáljuk, hogy a programunk abba az állapotba került-e, amit az automata alapján elvárunk. Tehát egy ilyen tesztesethez le kell írnunk, hogy:

- mi volt a kezdeti állapota a programnak
- milyen esemény történik, mik a bemenő adatok
- milyen eredményt várunk
- és milyen állapotban kell lenni az automatának az esemény után

A példánkban nem számolunk vagy dolgozunk fel adatot, így nincsenek külön eredmények, hanem igazából az átmenetek eredményei maguk az állapotok, amibe érkezünk. Az egyes teszteseteink a következők lesznek:

Teszteset azonosító	**T01**	**T02**	**T03**	**T04**
Kezdő állapot	S0	S0	S1	S1
Esemény	GPS jel	nincs GPS	GPS jel	nincs GPS
Eredmény	-	-	-	1
Végállapot	S1	SErr	S1	S2

Teszteset azonosító	**T05**	**T06**	**T07**	**T08**
Kezdő állapot	S2	S2	SErr	SErr
Esemény	GPS jel	nincs GPS	GPS jel	nincs GPS
Eredmény	-	2	-	-
Végállapot	S1	S2	S1	SErr

Ezekkel az egyszerű átmenetekkel az alapvető hibákat tudjuk egy programban kiküszöbölni. Írhatunk azonban bonyolultabb teszteseteket is, amikor kettő vagy több állapot átmenetet hajtunk végre. Ilyenkor le kell írni minden egyes köztes állapotot és az azokhoz tartozó műveleteket és eredményeket. Azaz egy két eseményt tartalmazó teszteset leírása a következő képpen fog kinézni:

Teszteset azonosítója	**T11**	**T12**	**T13**
Kezdeti állapot	S0	S0	S0
Bekövetkezett esemény	GPS jel	GPS jel	nincs GPS
Elvárt eredmény	-	-	-
Elvárt köztes állapot	S1	S1	SErr
Újabb esemény	GPS jel	nincs GPS	GPS jel
Következő eredmény	-	1	-
Elvárt végső állapot	S1	S2	S1

T14	**T15**	**T16**	**T17**	**T18**
S0	S1	S1	S1	S1
nincs GPS	GPS jel	GPS jel	nincs GPS	nincs GPS
-	-	-	1	1
SErr	S1	S1	S2	S2
nincs GPS	GPS jel	nincs GPS	GPS jel	nincs GPS
-	-	1	-	2
SErr	S1	S2	S1	S2

Ugyanígy ki lehet tölteni S2 és SErr kezdőállapotból is az átmeneteket. Az ilyen tesztesetek már rá tudnak világítani olyan hibákra is, amiket az egyszerű átmeneteknél nem vettünk észre. Előfordulhat például, hogy egy segédváltozó értékét megváltoztatja egy esemény, amit a végállapotnál nem veszünk figyelembe, így az egyszerű

átmeneteknél nem derül ki, viszont amikor két átmenetet hajtunk végre egymás után, akkor már hibás működést eredményez. A szekvencia hosszát, azaz hogy egymás után hány állapotátmenetet vizsgálunk, tetszőlegesen megnövelhetjük. Nagyobb bonyolultságú rendszereknél érdemes egy nagyobb lépésszámig vizsgálódni. A jelen példánkban további teszteseteket nem írunk fel.

Miután a valós teszteseteket felírtuk, ennél a technikánál is tudunk készíteni egy negatív esetekből álló tesztet, azaz a szándékosan rossz működtetést is vizsgálhatjuk. Ehhez meg kell néznünk, hogy az egyes állapotokban, mely események nem következhetnek be. Ezt a legegyszerűbben úgy tudjuk elvégezni, ha a táblázat első sorába beírjuk a lehetséges eseményeket, az első oszlopba pedig az állapotokat, a táblázat belsejét pedig kitöltjük a célállapotokkal és várható eredményekkel, ahol pedig nincs ilyen, azt jelezzük.

A fenti példára ez a következő képpen fog kinézni:

	GPS jelet kap	**nincs GPS jel**	**Időtúllépés**
S0	S1	SErr	!!!!!
S1	S1	S2	!!!!!
S2	S1	S2	SErr
SErr	S1	SErr	!!!!!

A !!!!! esetek lesznek az invalid átmenetek. Ezek gyakorlatilag azok az esetek, ahol az időtúllépést nem értelmezzük.

Használati eset tesztek

Amikor egy szoftvert egy külső szereplő, például egy felhasználó vagy egy másik program szemszögéből vizsgálunk, akkor úgynevezett használati eseteket tudunk megfogalmazni. Ezekben azt írjuk le, hogy a program hogyan reagál a külső szereplő behatásaira. Egy-egy használati eset jellemzően nem egy akció-reakció párosból áll, hanem egy egész folyamat leírásából, hogy honnan indulunk, milyen tevékenységet hajtunk végre, és hová érkezünk. Mivel itt főként a felhasználó oldaláról közelítjük meg a programot, ezért általában jó alapot képeznek ezek a tesztek az ügyfél elégedettségének eléréséhez. A használati esetek leírásának nincsen kötött formája, ezért az cégenként, sőt ügyfelenként változó lehet. Általában valahogy szabad szöveges formában, jól tagoltan, pontokba szedve írjuk le, hogy a felhasználó milyen műveleteket végez a programmal, és hogy erre milyen reakciót várunk. Mindenképpen adjuk meg, hogy milyen feltételek

teljesülése esetén tudjuk elkezdeni az egyes eseteket, például kikapcsolt navigációval nem fogunk tudni navigálni. Arra is érdemes figyelni egy-egy használati eset leírásánál, hogy ne legyen túl hosszú a feladat, azaz 15-20 lépésnél többet nem érdemes egyetlen használati esetbe zsúfolni. Ilyenkor inkább bontsuk több, rövidebb esetre! Például ha el akarok navigálni az aktuális pozíciómból valahova, akkor ne egyetlen használati esetbe zsúfoljuk bele az üzembe helyezés lépéseit, az útvonal tervezés lépéseit, és magát a navigációt. Ezek mind külön használati esetek legyenek, ahol az egyik utófeltételei megegyeznek a következő előfeltételeivel.

Nézzük példának az útvonal megtervezését. Ehhez adottnak kell lenni egy bekapcsolt, üzemkész navigációnak, ahol a térképen jelölve van az aktuális pozíciónk. Ez lesz az előfeltétele a tesztesetnek. És a következő lépéseket hajtsuk végre:

1. Válasszuk ki a célpont megadását a menüből. Ekkor jelenjen meg egy felület, ahova be tudjuk írni a kívánt címet.
2. Adjuk meg az elérni kívánt hely címét, város, utca, házszám formában. Ekkor a térképen jelenjen meg az adott helyszín.
3. Válasszuk ki az útvonalopciók közül, hogy a leggyorsabb útvonalat szeretnénk.
4. Indítsuk el az útvonal tervezést.

Ezek után a programtól a következőket várjuk el:

- Jelenjen meg a térképen egy kijelölt útvonal az aktuális pozíciónk, és a megadott cél között.
- Írja ki, hogy milyen hosszú az út, mennyi ideig tart, figyelembe véve az egyes sebesség korlátozásokat.
- Illetve a menetidő és az aktuális időpont figyelembe vételével adja meg, hogy mi a tervezett érkezés időpontja.

Ha vannak bizonyos feltételek, amiket szeretnénk, hogy teljesítsen a program, akkor azokat is meg kell itt adnunk, például, hogy az útvonal tervezésnek egész Európában működni kell. Mivel egy navigációnak az útvonal megtervezése elég lényeges része, ezért azt is jelölnünk kell, hogy ez a felhasználói eset gyakran fog előfordulni a szoftver használata során, valamint meglehetősen fontos, hogy jól működjön.

Az ilyen felhasználói eseteket összegyűjtjük, kitöltjük konkrét adatokkal, fontosságtól és feltételektől függően esetenként több tesztet is létre hozunk különböző adatokkal. Például

Magyarországon belüli és kívüli célponttal is, valamint kipróbáljuk, Európán kívüli célponttal is, ez itt egy invalid teszteset lesz.

Már a véges automatáknál is vizsgáltunk hosszabb folyamatokat, a különbség a két technika között a nézőpont. Míg a véges automatánál a program nézőpontjából vizsgáltuk annak állapotváltozásait, a használati eseteknél a fókusz a felhasználón van. Itt azt figyeljük, hogy ha a felhasználó szeretne valamit elérni, akkor azt sikerül-e neki, míg az állapot átmenetek vizsgálatánál azt figyeltük, hogy különböző lehetséges események, és paraméterek kombinációjára hogyan reagál a rendszer. Az így leírt teszteseteket akár az ügyfél is megadhatja, ha pontosan lépésről lépésre tudja, hogy hogyan szeretné, ha működne a program. Nekünk csak konkrét adatokkal kell feltölteni a tesztet.

Funkcionális technikák összefoglalása

Az eddig részletezett technikák páronként csoportosíthatók és jellemezhetők. Ekvivalencia osztályozást és határérték-elemzést akkor használunk, amikor a program bemenő adatai számok, vagy olyan adatok, amiket valamely tulajdonságuk alapján jól meghatározható, egymástól elkülönülő csoportokra tudunk bontani. Ha két ilyen csoport határa összeér, és nem teljesen egyértelmű, hogy a határhoz közeli elemek melyik csoportba fognak tartozni, jellemzően intervallumok, ott érdemes határérték-elemzést használni. Amikor a csoportoknak nincs értelmezhető határuk, azaz nincsenek olyan elemeik, amik nagyon közel lennének egymáshoz, akkor az ekvivalencia osztályozás a célszerű.

A döntési táblák, és ok-okozati gráfok a bemenő adatokra felírt logikai állítások kombinációját vizsgálják. Tehát akkor érdemes őket használni, amikor a program eredménye több körülménytől is függ. Amikor a bemeneti adatokra felírt állítások mindegyike, vagy legalább is nagy százaléka szerepet játszik abban, hogy mi lesz a kimenet, akkor az ok-okozati gráfunkban sok él lesz, és ettől elég átláthatatlanná válik, viszont a döntési táblából könnyen ki lehet olvasni az eredményt. Ha azonban néhány bizonyos változóból egyértelműen lehet a végeredményre következtetni, akkor az ok-okozati gráf szemléletesebben mutatja az eredményt, a döntési táblának viszont feleslegesen sok oszlopa lenne.

Véges automaták és használati eset tesztek már nem csak egy-egy változót és az arra adott eredményt vizsgálják, hanem hosszabb tevékenység sorozat folytán felmerülő esetleges hibákat tárják fel. Itt a fő különbséget az jelenti, hogy kinek a szemszögéből vizsgáljuk a

szoftver működését. Ha a program viselkedését nézzük, akkor véges automatát használunk. Szerencsés esetben a szoftver leírásával együtt megkapjuk az állapot átmeneti ábráját is. Ekkor mindenképpen érdemes ezt felhasználni. Hogyha pedig azt szeretnénk kideríteni, hogy egy programban a felhasználó el tudja-e érni, amit szeretne, akkor használunk a tesztek megtervezéséhez használati eset leírásokat.

Ezeken kívül vannak más technikák az egyes említett csoportokban. Például a határérték –elemzés bővítéséből adódóan jött létre a domain analizis, azaz tartomány elemzés. Ezt akkor használjuk, amikor a paraméterek több dimenziósak, azaz egyszerre több bemenő adattól függ a kijövő, és minden bemenő adat valamilyen intervallumba esik, amiknek a határait tudjuk vizsgálni.

Gyakran előfordul, hogy a technikákat érdemes kombinálni, például felírjuk a program használati eseteit, majd a szükséges bemenő adatokat ekvivalencia osztályozással szabjuk meg.

Teszt tervezés automatizációs lehetőségei

Ahogy a teszt tervezésnek, úgy a tervezés automatizációjának is számos különböző módja van. Ahhoz, hogy ki tudjuk választani a megfelelő módszert és eszközt, először is ismernünk kell a különböző teszt tervezési technikákat, és ezen felül a különböző automatizációs lehetőségeket is. Előfordulhat, hogy egy programhoz egyszerűbb megírnunk a teszteket, mint rábízni egy automatára, hiszen ahhoz, hogy automatizálni tudjunk, kellenek a megfelelő formátumú dokumentumok vagy bizonyos konfigurációk a programba, amiket szintén meg kell írni. Tehát bármennyire is kecsegtető, hogy egy automata majd megtervezi a tesztjeinket, ha nem vagyunk tisztában azzal, hogy milyen automatizációs eszköznek milyen használati és konfigurációs feltételei vannak, akkor a teszt tervezés automatizálása csak jobban elbonyolítja a folyamatot, ahelyett, hogy leegyszerűsítené. Ezért mindig mielőtt elkezdünk automatizálni, mérlegelnünk kell, hogy vajon megéri-e.

A teszt tervezés automatizációja során tulajdonképpen a dokumentumok elemzése az, amit rá bízunk az automatára. Attól függően tudunk megfelelő automatizációs módszert választani, hogy milyen dokumentumok, illetve specifikációk állnak rendelkezésünkre, vagy mi az, amit könnyebben meg tudunk írni, a meglévő információkból.

Modell-alapú tesztelés

Ha a programunk működéséről (funkcionalitásáról) készült valamilyen modell, akár felhasználói eset leírás, akár állapot átmeneti gráf, akkor ezt a modellt tudjuk elemezni. A modellek lehetnek grafikus vagy szöveges formában (különböző automatizáló szoftverek különböző képpen várják), de általában egy vagy több gráffal fel lehet őket rajzolni. A modell-alapú tesztelés ezeknek a különböző gráfoknak a bejárásaival foglalkozik. Mint ahogy azt a véges automatáknál leírtam, a tesztesetek tulajdonképpen úgy állnak elő a különböző gráfokból, hogy valamelyik csúcsból elindulunk, és az élek mentén haladva bejárjuk a gráfot, amíg valamilyen feltétel nem teljesül. Ennek a bejárásnak a végrehajtására készültek programok, amik megfelelő konfigurálás után megadják nekünk a kívánt gráf bejárást, azaz a teszteseteinket. Én főként a GraphWalker alkalmazást vizsgáltam. Ez széles körű lehetőséget biztosít tesztek generálására. A weboldalukon (http://graphwalker.github.io/) sok hasznos leírást és videót lehet találni a modell-alapú tesztelésről, valamint a GraphWalker beüzemeléséről és használatáról. Ezen kívül is azonban számos eszköz található az interneten. Vannak köztük nyílt

forráskódúak, és fizetősek, valamint válogathatunk köztük az alapján, hogy milyen típusú gráfot képesek feldolgozni. Az alábbi oldalon jónéhány ilyen eszköz össze van gyűjtve: http://mit.bme.hu/~micskeiz/pages/mbt.html

Tehát ha modell-alapú teszteléssel foglalkozunk, akkor nem a tesztesetek megalkotása van a középpontban (azt az automata majd megcsinálja), hanem a megfelelő modell fejlesztése. Ahhoz, hogy hatékony legyen ez a módszer, vagyis az automata gyorsan generáljon használható teszteseteket, érdemes odafigyelnünk néhány dologra.

A modellek általában a „valóság leegyszerűsített változatai", tehát fontos, hogy ne legyenek túl bonyolultak. Ha egy modell túl összetett – túl sok csúcsot és élet tartalmaz – akkor nem tudjuk átlátni, pedig pont az lenne a célja, hogy a rendszer vagy egy funkció működéséről egy jól áttekinthető képet adjon. Egy program modelljét megalkotni éppen ezért nem egyszerű feladat. Tudnunk kell, hogy mely részeket hagyhatjuk el belőle és mely részek szükségesek. Ha egy komplex rendszer modelljét akarjuk leírni, akkor először bontsuk kisebb részekre – akár a különböző funkciókra vagy felhasználói esetekre – és alkossuk meg az egyes funkciók modelljeit külön-külön. Ezek után fel tudjuk rajzolni azt a gráfot, ami ezek között a funkciók között megteremti a kapcsolatot.

Egy gráfot többféleképpen is be tudunk járni. Különböző gráfbejárások esetén különböző típusú teszteket kaphatunk. A teszt tervező eszközök nagy része ennek megfelelően paraméterezhető. Tehát ugyanahhoz a modellhez és programkódhoz néhány paraméter megadásával teljesen különböző célokra használható teszteket tudunk generálni. Ez egy hatalmas előnye ennek a technikának, hiszen ha ezeket a teszteket magunk akarnánk megtervezni, az lényegesen több időt venne igénybe. Nézzünk meg néhány ilyen – teszt tervezés szempontjából – fontos gráfbejárást.

Ha kijelöljük, hogy honnan induljunk el, hova szeretnénk megérkezni, és a lehető legrövidebb úton jussunk el oda, akkor nem járjuk be a gráf összes csúcsát, viszont néhány ilyen esettel megadhatjuk, hogy mik azok az állapotok, amiket mindenképpen érinteni szeretnénk. Ezzel a bejárással egy úgynevezett smoke tesztet kapunk, aminek lényege, hogy a szoftver fontosabb funkcióit leteszteljük. Ez főként olyankor hasznos, amikor valamilyen változtatás történt, és azt akarjuk ellenőrizni, hogy ez a változtatás nem rontotta-e el a program egyéb részeit.

Amikor a hangsúlyt arra fejtetjük, hogy a gráf minden élét (és ezáltal minden csúcsát) bejárjuk, akkor biztos, hogy ennek a tesztnek a során minden egyes funkcióját kipróbáljuk

a programunknak. Ez funkcionális tesztet fog eredményezni. Amikor először teszteljük az elkészült programot és meg akarunk győződni arról, hogy minden része úgy működik, ahogy azt elvárjuk tőle, olyankor mindenképpen érdemes funkcionális tesztet futtatni.

Ezen kívül paraméterezhetjük úgy is a gráf bejáró algoritmust, hogy mindig egy tetszőleges élet válasszon (mindegy, hogy jártunk-e már arra vagy sem), és egy lépésszámot vagy időkorlátot adunk meg arra, hogy meddig folytassa a gráf bejárását. Ezzel tulajdonképpen egy átlagos felhasználást tudunk szimulálni.

Általában azok a programok, amik elemzik a modelleket, és legenerálják a teszteseteket, magukat a teszteket nem hajtják végre, viszont képesek együttműködni tesztautomatizáló eszközökkel, azaz olyan outputot generálnak, amit egy automata rendszer végre tud hajtani. Tehát ha összekötjük a két rendszert (azaz az egyik által generált eredményeket átadjuk a másiknak), akkor az egész tesztelést – a tesztesetek létrehozásától kezdve a tesztek végrehajtásáig – automatizálni tudjuk. Ez két féle képpen történhet.

Amikor offline tesztgenerálásról beszélünk, akkor a tesztek generálása (azaz a modellünk bejárása) teljesen külön történik a tesztesetek végrehajtásától. Ekkor a program – ami bejárja a szoftverünk modelljét – egy file-ba elmenti a bejárás eredményét, azaz, hogy mely csúcsokból milyen élek mentén haladunk, vagy más szóval mely állapotokban milyen akciót hajtsunk végre, és ezáltal milyen állapotba kerülünk. Ezt a file-t utána lépésenként végrehajthatjuk vagy átadhatjuk egy tesztfuttató rendszernek, ami automatikusan végrehajtja. Az offline generálás előnye, hogy a teszteket, amiket egyszer legeneráltunk és elmentettünk, azokat többször is le tudjuk futtatni. Így vizsgálhatjuk a különböző futtatások közötti különbségeket: mik javultak, vagy romlottak el, illetve performanciát – vagyis a szoftver teljesítményét (sebesség, memória használat, stb…) – is mérhetünk.

Online tesztgenerálás esetén a tesztek generálását és végrehajtását párhuzamosan végezzük. Ekkor a modell bejárásának minden lépésében végre is hajtjuk a tesztet. Ilyenkor nem kell foglalkoznunk azzal, hogy a tesztek lépéseit eltároljuk, egyből a tesztelés eredményét kapjuk meg. Online generálást akkor célszerű használni, ha a rendszerünk nemdeterminisztikus, azaz nem biztos, hogy egy akció végrehajtásával mindig ugyan oda jutunk. Ilyenkor nem tudjuk előre megmondani, hogy milyen lépéseket hajtsunk végre egymás után, hiszen előfordulhat, hogy olyan állapotba kerülünk ahol az előre meghatározott következő lépés nem értelmezhető. Tehát minden egyes akció után

újra kell vizsgálni, hogy mik a lehetséges további lépések. Másik hasznos felhasználása az online tesztgenerálásnak, amikor hosszabb ideig (például egész éjjel) szeretnénk teszteket futtatni. Ilyenkor ahelyett, hogy előre legenerálnánk egy hatalmas lépésszámú tesztet, és azt futtatnánk, célszerűbb megadni, hogy meddig szeretnénk, hogy fusson a tesztünk, és a rendszer addig generálja és futtatja a tesztet, amíg el nem éri a megfelelő időtartamot.

Gherkin-alapú tesztelés

A Gherkin egy olyan nyelv, amivel egy szoftver funkcióinak működését tudjuk leírni. A nagyszerűségét az adja, hogy tulajdonképpen elég néhány apró szabályt és kulcsszót megtanulni ahhoz, hogy használni tudjuk. Ráadásul rengeteg nyelvre lefordították, így tényleg bárki által könnyen értelmezhető formában írhatjuk le vele a programunk egyes részeit. Ez azért is jó, mert ezt a leírást ugyanúgy érti a tesztelő, a fejlesztő, és az ügyfél is, így könnyebb elkerülni a félreértéseket, és kielégíteni az ügyfél kívánságait. Ebből a leírásból aztán gyorsan és egyszerűen tudunk a programhoz dokumentációt és teszteket generálni, ráadásul a tesztek során készült riportokba (azaz a teszt eredményéről készült leírásokba) is kerülnek be innen információk. Amikor egy szoftver fejlesztését (és tesztelését) ebből az irányból közelítjük meg, azt viselkedés-alapú fejlesztésnek (Behaviour Driven Development – BDD) hívjuk. Ez a módszer ugyan megkönnyíti a teszt tervezést és végrehajtást, de a fő célja nem csak ez, hanem hogy a szoftver fejlesztéséhez mutasson egy irányt.

Mint a modell-alapú tesztelés esetében, itt is fontos szempont, hogy legyen egy olyan leírásunk, aminek segítségével könnyen át tudjuk látni a programunkat. Itt arra fektetjük a hangsúlyt, hogy a működés különböző „jeleneteit” határozzuk meg. Gyakorlatilag megírjuk a programunk forgatókönyvét. A Gherkin nyelv kulcsszavai és szabályai ehhez adnak egy eszközt a kezünkbe. Itt a szabályok nem korlátoznak, inkább egy keretet adnak ahhoz, hogy leírjuk, hogyan szeretnénk, hogy működjön a programunk. Különböző BDD eszközök használhatnak más leíró nyelveket, amik egyes kulcsszavakban és szabályokban eltérnek a Gherkintől, de a lényege mindegyiknek ugyanaz: valamilyen egységes formában leírni a program elvárt viselkedését.

Ehhez először felbontjuk a programot a különböző célú működési egységeire (Feature). Ajánlott minden egyes ilyen önálló funkcióról külön dokumentumot készíteni, ezzel is alátámasztva azt, hogy ezek az egységek egymástól függetlenül is képesek működni. A

Gherkin .feature kiterjesztésű file-jaiban nem is szerepelhet a Feature kulcsszó egynél többször. Minden ilyen dokumentum elején adunk egy hosszabb-rövidebb leírást a funkcióról. Ebben a részben érdemes feltüntetni az adott programrésszel kapcsolatos üzleti vagy felhasználói elvárásokat is. Az átláthatóság kedvéért célszerű szóközökkel és üres sorokkal tagolni a file-t. A dokumentum nem üres sorai általában valamilyen kulcsszóval kezdődnek. Ezalól kivételt képeznek a hosszabb funkció és jelenet leírások, amik több soron keresztül is tarthatnak.

Miután a főbb funkciókat meghatároztuk, ezeket az egységeket tovább bontjuk az egyes jeleneteire (Scenario, Scenario Outline). Minden jelenetnek egy önmagában végrehajtható egységnek kell lenni. A jelenet egyes lépései között nem várhatunk külső információra vagy esemény megtörténésére. Minden jelenetben leírjuk, hogy honnan indulunk, azaz milyen kezdeti feltételek, körülmények vannak (Given), milyen esemény történik (When), és az esemény hatására hová jutunk, vagyis milyen eredményt várunk el (Then). A jeleneteket nem tanácsos túl hosszúra írni, azaz egy jelenet általában 3-5 Given, When vagy Then lépésből kell, hogy álljon. Ezekből When lehetőleg csak egy legyen, ugyanis az egyes jelenetekben az egyes felhasználói interakciókat szeretnénk leírni, azok kezdeti feltételeivel, és eredményeivel. Ha egymás után több eseménynek kell történni, akkor úgy tudjuk őket külön jelenetekbe leírni, hogy az első eredményei (azaz Then lépései) a következő kezdeti feltételeivel (azaz Given lépései) egyezik meg. Több azonos típusú lépés között szerepelhet az And vagy a But kulcsszó is, ezt az alábbi példa szemlélteti:

```
Scenario: Route planning
     Given Start koordinates
     Given End koordinates
     When User asks for route planning
     Then Planned route should shown
     Then map viewpoint shouldn’t change
```

helyett:

```
Scenario: Route planning
     Given Start koordinates
     And End koordinates
     When User asks for route planning
     Then Planned route should shown
     But map viewpoint shouldn’t change
```

Ahhoz, hogy a Gherkin file-okból teszteket tudjunk generálni és futtatni, a leírt lépéseket hozzá kell kötni a programunk egyes részeihez. Azaz meg kell mondanunk, hogy egyes Given, When és Then lépések a programunk mely részét hajtsák végre, milyen változóknak adjanak értéket és az eredménynek mivel kell egyenlőnek lenni. Ha ezzel készen vagyunk, akkor a kiválasztott BDD eszközzel tesztelhetjük is a programunkat, aminek a végén megkapjuk, hogy mely lépések hajtódtak végre helyesen, és hol tapasztalt a rendszer valamilyen rendellenes működést. A tesztek lefutásáról riportot is generáltathatunk, amelybe a funkciók és jelenetek leírásai is bele kerülnek, illetve, hogy hol történt a hiba a program futása közben. Hibátlan tesztlefutás esetén a riportok arra is jók lehetnek, hogy az ügyfél elfogadja a programunkat, hiszen ha jól írtuk meg a feature file-jainkat, akkor bele kerül a riportba, hogy az ügyfél mely elvárásainak teljesülését melyik teszt lefutások bizonyítják.

Összegzés

Láthattuk, hogy a tesztelés ugyanolyan szerves része a szoftverfejlesztésnek, mint a kódolás. A tesztek értő megtervezésével rengeteg kódolási idő megspórolható, ehhez azonban az kell, hogy a teszt tervezés már a szoftver tervezési fázisában elkezdődjön. Sőt, célszerű, hogyha a szoftver tervezési és a teszt tervezési csapat szorosan együttműködve hozza létre a program fejlesztési és tesztelési tervét. Ehhez azonban szükséges ismerni a különböző technikákat és eszközöket. Ha tudjuk, hogy milyen esetekben melyiket célszerű használni, akkor ezek kreatív kombinálásával, nagyobb valószínűséggel kerülhetjük el a hibákat.

Az említett példákból az is látható, hogy egyetlen szoftver termék teszteléséhez gyakorlatilag az összes tervezési technikát fel lehet használni. Fontos tehát, hogy a tesztek tervezésénél ne ragaszkodjunk egyetlen technikához, hanem tudjuk kreatívan kombinálni azokat. Gyakran fordul elő az is, hogy a tesztelők tapasztalatára kell hagyatkozni, mert a különböző technikákkal nem tudunk minden hibát felfedni.

Mindemellett fontosnak tartom megjegyezni, hogy a tesztelés és teszt tervezés sokszor nagyon intuitív és kreatív feladat, amihez ugyan vannak hasznos technikák, amiket megtanulva az ember jobb szakemberré válhat, de a technikák elsajátítása mellett a gyakorlat is nagyon fontos. Minél több hibalehetőséggel találkozunk a tesztelés folyamán, annál jobban fel tudjuk mérni egy új szoftvernél, hogy mi mindenre érdemes jobban odafigyelni.

Irodalomjegyzék

[01] Beizer, Boris: Software Testing Techniques, Dreamtech, 2003, [550], ISBN: 8177222600 9788177222609.

[02] Anette Mette Johanssen Hass: Guide To Advanced Software Testing, Artech House, 2008, [460], ISBN-13: 978-1-59693-285-2.

[03] Forgács István, Kovács Attila: Practical Test Design, 2018, preprint.

[04] Dorothy Graham, Erik Van Veenendaal, Rex Black: Foundations of Software Testing, Cengage Learning EMEA, 2008, [258], ISBN-13: 978-1844809899.

[05] Horváth László: Szoftvertesztelés a gyakorlatban (diplomamunka, ELTE IK), 2007 (http://people.inf.elte.hu/holaci/Diploma/Szoftvertesztel%20a%20gyakorlatban.pdf)

[06] http://graphwalker.github.io/MBT_How_to/, 2018

[07] http://mit.bme.hu/~micskeiz/pages/mbt.html, 2018

[08] http://mit.bme.hu/~micskeiz/papers/szoftverteszteles-2016/micskei-modell-alapu-teszteles.pdf, 2018

[09] https://docs.cucumber.io/guides/, 2018