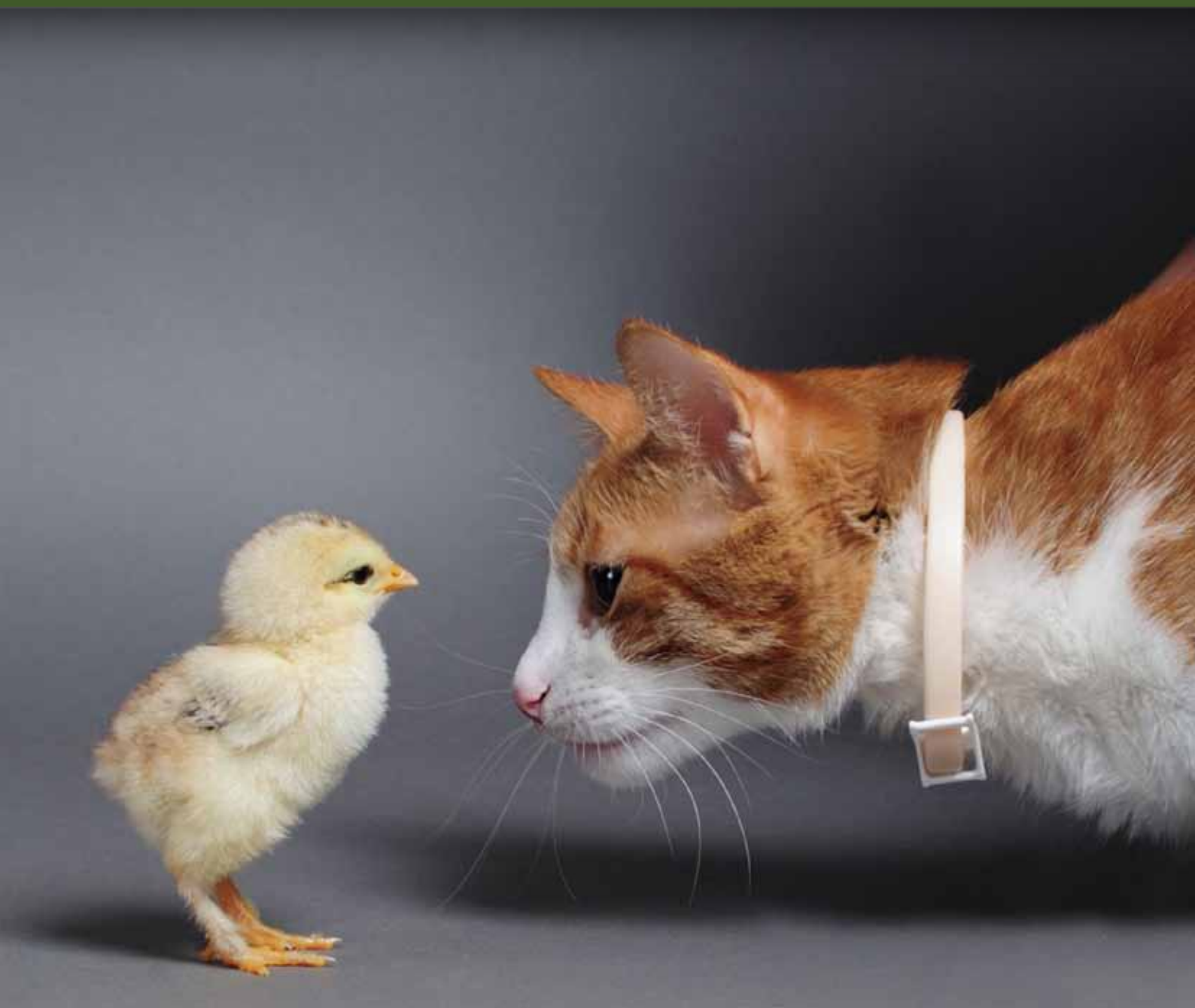
2011 / JÚNIUS

# TESZTELÉS A GYAKORLATBAN

A SZAKÉRTŐ TESZTELŐK LAPJA



## ALKALMAZÁSOK BIZTONSÁGA EGY PEN-TESTER SZEMÉVEL

## *Kedves Olvasó*

Mindig rejtélyesnek, misztikusnak tűnik, amikor biztonsági tesztekkel foglalkozó emberek mesélnek a munkájukról. Habár valószínűleg számomra is megtanulható lenne, mégsem vettem eddig a fáradtságot, hogy elmélyedjek a biztonsági terület szakmai fogásaiban. Pedig módfelett érdekfeszítő (és egyben hátborzongató) történetekkel tudnak szórakoztatni a szakemberek.

A vicc az, hogy a két oldal ugyanolyan furcsán néz egymás feladataira. Egyik sem tudja, hogyan végzi a másik a munkáját. Milyen eszközöket használ, milyen módszer alapján dolgozik.

Amikor az újságot kezdtük szerkeszteni több helyen is próbáltuk elérni, hogy biztonsági szakemberek is írjanak cikkeket. Nagyon jó lenne, ha a két tábor kicsit közelebb tudna kerülni egymáshoz. Ugyanis a biztonsági rés, amelyet egy ethical hacker megtalál, az alkalmazás tervezése és fejlesztése alatt kerül bele a rendszerbe. Talán ha a tesztelés során jobban fókuszálnánk ezekre a tesztekre kevesebb biztonsági hiba kerülne ki az éles környezetre.

Egymás szervezeteiről, tanfolyamairól, konferenciáiról is alig tudunk valamit. Említhetnénk a nagy sikerű ITBN-t, vagy az Ethical Hacking konferenciát, esetleg a Hactivity-t. De ugyanez fordítva is igaz, a biztonsági teszteléssel foglalkozó embereknek sem mond semmit az, hogy HTB, ISTQB, vagy az IIR és IDG tesztelési konferencia.

Úgy tűnik, mintha két egymástól teljesen különálló világ létezne. Pedig szerintem nagyon is közeliek. Remélem egy pici szerepet mi is játszhatunk abban, hogy a két tábor közelebb kerüljön egymáshoz.

Üdvözlettel,

Pongrácz János

# Tartalomjegyzék









**Kedves Olvasónk,**

Sok olyan tesztelő, tesztvezető dolgozik az országban, aki a saját területén egyedi, maradandó, kiemelkedő értéket alkot. Őket kívánjuk felkutatni, összefogni, hogy egy szakmailag profi lapot tudjunk folyamatosan szerkeszteni.

**Szeretnél publikálni, csak eddig nem találtál megfelelő platformot?**

A Tesztelés a Gyakorlatban magazinban a mindennapi szoftvertesztelési munkákat, tapasztalatokat mutatjuk meg a nagyközönségnek oly módon, hogy azt közvetlenül a szakemberektől, vagyis Tőled hallják.

## Gyere és írj cikket Te is!

Célunk, hogy gyakorlati tudást adjunk át a szoftvertesztelési projektekhez, munkákhoz.

Amennyiben kedvet érzel publikálni kérünk, vedd fel velünk a kapcsolatot a publikacio@tesztelesagyakorlatban.hu, címen és küldd el anyagodat, hogy minél átfogóbb, a trendet követő friss irányzatokat, tapasztalatokat bemutató magazint tudjunk szerkeszteni a Te segítségeddel.

Üdvözlettel,
**Tesztelés a gyakorlatban** – A szakértő tesztelők lapja
magazin szerkesztői


# *Tesztelés Java környezetben*

**A szoftverfejlesztés egyik szükséges – és ma már bizonyára közhellyé lényegülten sokszor elmondott – következménye az elkészült program tesztelése. Mindazonáltal még mindig számos olyan fejlesztési feladat, projekt végeredménye hagyja el a „szoftvergyárat”, ahol a fejlesztett termék nincs megfelelően letesztelve. A kisebb-nagyobb, részben „házon belüli”, jellemzően egyedi szoftverfejlesztési projektek során nehezebb egy koncepció mentén kialakítható tesztelési folyamat végigvezetése.**

A programozók saját fejlesztéseik, részfeladataik végzése közben a programkód növekedésével – természetszerűleg – egyre nagyobb kihívással szembesülnek, ami az áttekinthetőséget illeti. A több ezer soros programokban ennek megfelelően hibák keletkezhetnek, de ez nem is lehet kérdés.

A kérdés az, hogyan lehet megtalálni a keletkezett hibákat. Jellemzően azoknál az egyedi szoftverfejlesztéseknél, ahol a fejlesztői teszten (amikor a fejlesztők saját maguk munkáját, a programrész működését ellenőrzik) átesett végtermék – további szervezett tesztelések nélkül – a megrendelőhöz kerül, ott derülnek ki a hibás működéssel kapcsolatos problémák. Ez igen könnyen visszaüthet a szállítóra, ha a megrendelő ügyfél számára ez jelentős kényelmetlenséget, plusz költséget okoz.

A tesztelés, mint folyamat beépítése a szoftverfejlesztési projektekbe azért is fontos, mert a fejlesztők a saját maguk által írt programkódjukat egyszerűen képtelenek más szemszögből nézni. Még a jól felépített koncepcióba is csúszhatnak hibák, ezért kritikus fontosságú, hogy az elkészült terméket egy különálló, de természetesen a funkcionális specifikációt ismerő tesztelő csoport tesztelje.

A tesztelésekkel kapcsolatban általánosan ismert követelmények a józan ész és a tapasztalatok alapján kialakult alapelvek köré csoportosulnak. Például:

- Mindig tervezettnek és megismételhetőnek kell lennie.
- Más (is) teszteljen, mint aki fejlesztett.
- Az egyes komponensek tesztelésétől kell haladni a teljes rendszer tesztelése felé. Érdemes a komponenseket külön letesztelni összeállítás előtt.
- A komponensek még értelmezhető kisebb egységeinek tesztelését érdemes meghagyni fejlesztői tesztként, mert ilyen mélységben csak a fejlesztők ismerik a kódot.
- A funkcionális működés ellenőrzésén túl szükség van a fejlesztett termék integrációs tesztjére a működési környezetbe helyezéskor.
- A funkcionalitás megfelelőségének ellenőrzésén felül a terhelhetőség (teljesítmény) és a stabil működés (fenntarthatóság) ellenőrzése is fontos.

A Java-alapú szoftverfejlesztési feladatok során a tesztelési fázisban természetesen ugyanezek az alapszabályok érvényesek.

A következőkben három tesztelési lépést, azok egy-egy lehetséges módszerét/eszközét vesszük sorra, amelyek Java környezetben írt programok tesztelését segíthetik:

### a) Komponens (modul) tesztelés

A komponensek belső működésének ellenőrzése a strukturális tesztek fogalomkörébe tartozik. Mivel a fejlesztett alkalmazás belső struktúráját vizsgáljuk, ezért white-box teszteknek is nevezik őket (ellentétben a funkcionális tesztekkel, ahol fekete dobozként tekintünk az alkalmazásra).

Célszerűen a különálló modulok egy egységbe építését megelőzően használható módszer, így kiküszöbölendő sok, a nagyobb egységgé gyúrt rendszerben fellépő hibák keresésével járó vesződség.

A komponens-szintű tesztelés lényege, hogy a szoftver metódusait, azok működését tesztesetekben meghatározott tesztértékekkel ellenőrizzük. Az egyes modultesztek sikerességét a lefedettséggel szokás jellemezni.

A lefedettség a programsorok relatív számával (teszt által elért programkód sorainak az összes sorhoz viszonyított száma), vagy a meghívott függvények relatív számával (végrehajtott függvények számának és az összes függvény számának hányadosa), jellemzően százalékos formában adható meg. Ide tartozik még pl. az elágazások, vagy az összes lehetséges út lefutásának ellenőrzése is. A lefedettség mérése mind a memória, mind a processzor számára jelentős terhelést jelenthet, ezt a tesztek tervezésénél időzítés, terhelés szempontjából figyelembe kell venni. Másik vonatkozása a lefedettség mérésnek, hogy az adott projekt költségét jelentősen növelheti, ez pedig általában erős befolyásoló tényező.

Az ismétlődő végrehajtás, a tesztesetek számossága, a teljes vagy részleges végrehajtás szükségessége és az eredmények megjelenítése, rendszerezése, továbbá az automatizált végrehajtási igények kezelése és mindezek együttes bonyolultsága nélkülözhetetlenné teszi az ilyen feladatokra készített keretrendszerek használatát.

A JUnit (http://www.junit.org/) az egyik legelterjedtebb keretrendszer, amit az egyes komponensek tesztelésére használnak. Bár lefedettséget nem tud mérni (erre a célra kiegészítő eszközöket lehet alkalmazni, ilyenek: pl. Cobertura, Emma), ezzel együtt ez az egyik legnépszerűbb komponens-tesztelő eszköz, amelyet a leggyakrabban használt fejlesztői környezetek (pl. Eclipse, NetBeans) is támogatnak. Egyszerű használhatósága mellett lehetőséget nyújt automatikus tesztfuttatásra a kód egészére, vagy egy részére vonatkozóan, továbbá a teszteredmények megjelenítése mellett strukturált riportok készítése is lehetséges.

A JUnit-ban hierarchikusan rendezett tesztesetekből állíthatunk össze tesztkészleteket (test suite), amelyekben a teszteseteket együtt futtathatjuk. A tesztek ellenőrzésére szolgálnak az ún. Assert metódusok, amelyekkel a teszt eredményét hasonlítjuk össze az elvárt eredménnyel.

A metódusok tesztelésekor is figyelemmel kell lenni arra, hogy a metódusok ne önmagukban kerüljenek letesztelésre, hanem az üzleti funkciójukat lássák el (rendszerben gondolkodás).

A tesztek tervezésében többnyire az emberi képzelőerőre (vagy kiegészítő eszközre) van szükségünk, a JUnit önmagában nem támogat tervezést. A tesztesetek tervezése eléggé hosszadalmas feladat, ráadásul az a kihívás is előttünk áll, hogy megtaláljuk az arany középutat a tesztesetek szükséges és elégséges száma (lefedettség), valamint az aránytalanul nagy energia ráfordítás között.

Tesztesetek tervezésekor figyelembe kell vennünk az egyes változók szélsőértékeit, a beállítandó intervallumokat, érvénytelen karaktereket stb.

Az egyes modulok működésének tesztelését követően az integrációs tesztekkel célszerű ellenőrizni a modulok „valós” környezetbe illesztésekor a tranzakciós folyamatokat és a modulok egymásra hatását.

### b)Funkcionális (rendszer) tesztelés

A funkcionális tesztekre akkor kerülhet sor, ha az összeállított alkalmazás egyes komponensei unit és modulszinten (modul alatt itt pl. egymásra ható unit-okat értünk) megfelelően működnek.

A funkcionális tesztelés már jobban elkülöníthető a fejlesztőktől. Általános esetben egy különálló tesztcsapat feladata, hogy a tesztelésre kapott rendszer üzleti funkcionalitását figyelembe véve, de magát a megvalósítási módot fekete dobozként kezelve megvizsgálja az egyes funkciók működését. Ismerve a lehetséges bemeneteket, le kell ellenőrizni a lehetséges kimeneteket. A funkcionális tesztek egy-egy tesztesetre építve egy-egy felhasználói funkció, üzleti folyamat lépéseit ellenőrzik.

A funkcionális tesztek során alapvetően az alábbi típusú tesztek kerülnek végrehajtásra (természetesen egyes módszerek alkalmazhatók pl. a modulok tesztelésénél is):

Ekvivalencia osztályok alapján történő tesztelés: A program bemeneti paramétereit a specifikáció szerint felállított ún. ekvivalencia-osztályokra (azonos viselkedési jelleggel bíró értékek halmazaira; pl. ilyenek az intervallumok) osztjuk, majd minden osztályból tesztértékeket véve teszteseteket készítünk. A jól megválogatott tesztesetek (pl. érvényes-érvénytelen, határérték-középérték) így a lehetséges inputokat nagymértékben lefedik, így egy olyan halmazt állítunk elő a teszteléshez, amely a lehető legtöbb hibát felderítheti.

Határérték tesztelés: Tulajdonképpen az ekvivalencia osztályok alkalmazásának speciális esete, ugyanis az osztályok határértékeire fókuszálva, a hibák rendszerint itt sűrűsödnek. Határérték-tesztelésnél a kimeneti értékek vizsgálata is szükséges lehet, azaz érdemes a tesztekhez olyan bemeneti értékeket is választani, ami a kimeneti határértékeket fogja vizsgálni.

Ok-okozati tesztelés: A funkcionális tesztelés talán legfontosabb célja, hogy kiderüljön, milyen bemeneti paraméterek hatására mi történik a kimeneten. Ennek vizsgálatára egy ok-okozati gráfot, vagy táblázatot célszerű készíteni, amit a tesztesetek kidolgozásakor használhatunk. Természetesen a lehetséges bemeneti-kimeneti kombinációk száma óriási lehet, ezért sokszor szinte képtelenség minden egyes esetet megvizsgálni. Ezért fontos azoknak a teszteset kombinációknak az összeállítása, amelyekkel a leghatékonyabban tudjuk feltérképezni a hiányosságokat, illetve a specifikációnak ellentmondó hibákat.

Az így összeállított teszteseteket egy ún. teszt-lefedési mátrixon megjelenítve feltérképezhetjük, hogy az adott teszt(eset) mely funkció(k) vizsgálatára alkalmas, valamint hogy maradt-e ellenőrzés nélküli funkció.

Véletlenszerű tesztelés: A véletlenszerűen előállított tesztadatokkal elvégzett vizsgálatok célja azoknak a hibáknak a feltárása, amelyek a determinisztikus tesztek tervezésénél esetleg elkerülték a figyelmet. A véletlenszerű tesztelés hibafeltáró képessége rendszerint alacsony, de általában megéri a ráfordítást, mert a tesztadatok (véletlen számok) viszonylag gyors előállítása újabb területen fedheti fel az eddig elrejtett hibákat.

A funkcionális tesztelés, a tesztesetek végrehajtása részben automatizálható, azonban néhány esetben szükségszerű lehet a manuális tesztelés is, mint kiegészítő módszer.

Az automatizálás történhet pl. script-ek segítségével parancssoros alkalmazásoknál, de akár egy vagy több különálló grafikus megjelenítésű tesztszközzel is, pl. webes alkalmazásoknál.

Miért hasznos a tesztelés automatizálása? Függetlenül a tesztelés módszerétől, javítások után gyakori a regressziós tesztelés szükségessége, a tesztesetek akárhányszor futtathatók, valamint lerövidül a visszajelzések ideje a fejlesztők számára, így hatékonyabbá válik a fejlesztési folyamat (ez az előny felhasználható pl. az agilis szoftverfejlesztésnél vagy az extrém programozási technikánál).

A webes alkalmazások automatikus tesztelésére használható eszközök egyike a Selenium (http://seleniumhq.org/), amellyel funkcionális teszteket is végre tudunk hajtani.

A Selenium segítségével a leggyakrabban használt webes böngészőkben futtathatunk egy-egy tesztesetet, vagy egész tesztforgatókönyvet is úgy, mintha egy felhasználó futtatná őket. A Selenium többféle operációs rendszert is támogat, így felhasználhatósága széleskörűnek mondható a webes felületek tesztelésében.

A Selenium több komponensből áll, amelyek az automatizált tesztelés másfajta





megközelítését szolgálják. A fő komponensek az alábbiak:
Selenium IDE: Fejlesztői környezet, amellyel rögzíthetők a böngésző felületén elvégzett feladatok (kattintások), melyek aztán lefuttathatók, szerkeszthetők és hibakeresésre használhatók. Hátránya, hogy csak Firefox böngészőben futtatható. A teszteket futtató keretrendszer neve Selenium Core, ami az IDE és az RC motorja is egyben.
Selenium RC (Remote Controller): Szerver és kliens könyvtárak összetevőiből álló, többféle programozási nyelvet támogató API, melynek segítségével automatikusan – akár távoli gépen lévő böngészőben is – futtathatók az elkészített tesztesetek. (A Selenium WebDriver a legújabb verziójú Selenium egyik komponense. Az RC modulhoz hasonló funkcionalitással bíró API, amellyel az RC komponenst fogja kiváltani a jövőben.)
A Selenium egy saját, egyszerű „nyelvvel" rendelkezik, amelyből könnyen összeállíthatók a tesztesetek és ezek más programozási nyelvre is elmenthetők a fejlesztői környezetből. A Selenium alap nyelvi összetevői az alkalmazások állapotának manipulálására szolgáló parancsok (command), az állapotok vizsgálatára és tárolására szolgáló ún. Accessor elemek és az alkalmazás állapotát az elvárt eredménnyel összevető Assertions elemek.
A rögzített vagy átszerkesztett tesztesetek exportálhatók többféle formátumba (pl. Java, Perl, PHP) és más környezetben is futtathatók az API segítségével.
Selenium Grid: Ez a komponens párhuzamosan több szerveren futtatható tesztek elvégzését teszi lehetővé, így jelentős idő takarítható meg a webes tesztelések során, valamint egyszerre tesztelhető egy teljesen heterogén környezet.
Természetesen nem mindig éri meg a felülettesztek automatizálása: ha a határidő nagyon rövid, vagy a felhasználói felület gyakran változik, akkor a tesztesetek újraírása hosszabb időt vehet igénybe, mint a manuális tesztelés.
A funkcionális tesztekhez sorolható a rendszer integrációs tesztelés, ami az interfészeken kapcsolódó más rendszerek hatását is ellenőrző tesztfázis.
A funkcionális teszteken túl szükséges azokat a nem funkcionális teszteket is elvégezni, amelyek a működő rendszer használata közbeni viselkedést vizsgálják.

### c) Terheléses tesztek

Amikor az elkészült alkalmazások funkcionálisan már működőképesek, végső működési környezetükbe kerülés előtt fontos ellenőrizni a rendszer teljesítőképességét. Ezt általában már a tesztelési folyamatot lezáró végső felhasználói teszt keretében végzik el. A tervezés során specifikált hardverigény, a tervezett párhuzamos felhasználók száma, a rendszer adott időszakban történő elérhetősége és válaszideje mind olyan követelmények, amelyeket az elkészült alkalmazásnak teljesítenie kell. Ezeknek a feltételeknek a vizsgálatára szolgálnak a terheléses tesztek.
A terheléses tesztekben többféle módszerrel vizsgálhatjuk az elkészült rendszert. Vizsgálhatjuk a kívánt paraméterek szerinti teljesítményt vagy akár a rendszer túlterheltségének hatását, a huzamosabb ideig tartó csúcsterhelés hatását, alkalmazhatjuk a fokozatos terhelés módszerét stb.
A terheléses teszteknél különösen célszerű ügyelni arra, hogy a tesztrendszer paraméterei összemérhetők legyenek az éles rendszer paramétereivel. Ezzel biztosíthatjuk, hogy a tesztrendszerben mért teljesítmény-értékek az éles rendszerben is mérvadóak legyenek.
A webes alkalmazások (de akár fájlszerver kapcsolatok, adatbázis kapcsolatok, Java objektumok, web szolgáltatások stb.) teljesítmény tesztelésére használható egyik eszköz a több operációs rendszeren is működő JMeter (http://jakarta.apache.org/jmeter/). A JMeter-ben egy grafikus felület segítségével lehet a teszteseteket összeállítani, a tesztek eredményeit táblázatokban, grafikonokon tudjuk megtekinteni. Egy beépített proxy szerver segítségével a küldött üzenetek rögzíthetők és újra lejátszhatók.
A JMeter több szálon futó tesztelést is képes végrehajtani az ún. Thread Groupok segítségével, így szimulálva az egyes felhasználókat, akik különböző oldalakat próbálnak elérni. Az időzítők segítségével pedig a terhelés elosztását is szabályozni lehet.
A JMeter-rel elvégzendő tesztek tervezésekor használatos elemek a tesztet vezérlő elemek: az ún. Sampler (ami a szerver felé történő kommunikációt indítja és a választ várja) és a logikai vezérlő (logika annak eldöntésére, hogy a kérés milyen feltétellel, mikor induljon).
A szervertől visszaérkező válaszok elvárt eredménnyel való összehasonlítására használt elem az ún. Assertion elem.
A Listener-ek feladata pedig az adatok összegyűjtése és az igények szerinti riportokhoz szükséges adatok előállítása.
Ezekből a JMeter-ben használatos alapelemekből építhető fel a különféle szerverek teljesítménytesztjéhez szükséges teszterv. A részletesebb beállítások természetesen már a teszt céljától és tárgyától függően mások lesznek, ám a JMeter működésének megértéséhez ezek az elemek nélkülözhetetlenek.

Végezetül, megemlítendő még a tesztelés, mint a szoftverfejlesztési életciklus egyik fontos szakasza, ami a projekt fejlesztési költségeinek jelentős részét teheti ki. A tesztek egy része automatizálható, ahogy a fenti példák is mutatják, persze a tesztesetek felépítése itt sem fogja csökkenteni a ráfordítást. Ám a jól megírt tesztek hatékonyabbá tehetik a tesztelést magát, ráadásul az ismétlődő tesztek már egyszerűbben elvégezhetők a bejáratott úton, ami összességében csökkentheti a projekt ráfordítását.


***Csáki István***

IT Architect, Atoll Technologies Kft.

Villamosmérnöki és közgazdasági végzettséggel rendelkezik. A távközlési iparágban kezdte pályafutását, majd főleg pénzintézeti és államigazgatási szektorban töltött 10 évet szoftverfejlesztési projekteken. Az IT területén szakértőként számos nemzetközi projektben vett részt. Munkája során többnyire szervezési és tesztmenedzsment feladatokat lát el.

# A tesztmenedzsment eszközök akadályozzák az agilis munkát

**Ha agilis csapatban dolgozol és specializált tesztmenedzsment eszközt használsz, akkor van egy fontos üzenetem számodra:**

**Állj meg. Komolyan. Hagyd abba, csak hátráltat téged.**

Ha hozzá vagy szokva a tesztmenedzsment eszközökhöz, lehet nem is látod, hogy az inkább csak teher, mint segítség az agilis életben. Bizton állíthatom, hogy az agilis csapatoknál kivétel nélkül mindig hátráltató tényező egy ilyen eszköz.

Nem könnyen teszek ilyen kijelentéseket és nem is várom el, hogy csak úgy elhidd nekem. Szóval enged meg, hogy elmagyarázzam.

## Agilis alternatíva a tesztmenedzsmenthez

Ezek azok a dolgok, amikre szükség van egy agilis környzetben, hogy kezelni lehessen a tesztelési erőforrásokat:

- feladatlista - backlog
- forráskód kezelő rendszer - SCM
- folyamatos integrációt nyújtó rendszer - CI
- automatikus regressziós tesztek.

Ennyi. Nincs szükség más eszközre vagy követő mechanizmusra.
Minden más teszt-specifikus eszköz használata növelni fogja az információ duplikációt és még szükségtelen erőforrás költség is hozzáadódik, hogy szinkronban tartsuk a sok másolatot. A plusz eszközök használata valószínűsíti, hogy létre kell hozni és karban kell tartani egy csomó meta adatot, mint például a követhetőségi mátrixok (traceability matrices), hogy összekapcsoljuk a különböző eszközökben tárolt adatokat. Ezek mind magas fenntartási költséggel járnak és nem adnak több értéket a teszteléshez, mint amit az SCM, a CI és a feladatlista már eleve biztosít.

## De, de, de …

Sok ellenvéleményt kaptam azzal kapcsolatban, hogy az agilis csapatoknak nem kell tesztmenedzsment eszköz. Ezekre most jöjjön néhány válasz:

## De akkor hol tároljuk a teszteket?

A teszttel kapcsolatos dolgok két helyre kerülhetnek:

- a magas szintű elfogadási kritériumok, teszt tervek, felderítő tesztelés (exploratory testing) a feladatlistához tartozik a hozzá kapcsolódó tennivalóval együtt;
- a technikai dolgok, beleértve a teszt automatizációt és a manuális regressziós teszt scripteket (ha van ilyen) a Source Control System-hez tartozik a hozzá kapcsolódó kódokkal.

## És hogyan rögzítjük a tesztelési becsléseket?

Az agilis fejlesztésben a lényeg az elvégzett feladat. Ami kódolva van de még nincs tesztelve, az nincs kész. Így a tesztelési erőforrást úgy becsüljük, mint az egész feladat (Story) végrehajtásának egy részét, ha muszáj valami távoli hozzávetőleges becslést készíteni. Azaz mi nem elkülönítve becsüljük a tesztelési erőfeszítést, ez azt jelenti, hogy nem kell külön hely a tesztelési becsléseknek.

## Hogyan priorizálom a tesztjeimet?

Az agilis csapatok egy priorizált feladatlistából (Backlog) dolgoznak. A tesztek helyett a végrehajtandó komplex feladatokat (Stories) teszik fontossági sorrendbe. A feladatok (Stories) vagy el vannak végezve vagy nem. Ebben a környezetben nincs semmi értelme a tesztek elkülönített fontossági sorrendjének.

## Hello, én a való világban élek. Soha nincs elég idő a tesztelésre. Hogyan állítok fel fontossági sorrendet a szűkre szabott idő alatt?

Ha a feladat (Story) elég fontos, hogy programozzák, akkor elég fontos hogy teszteljék is. Pont. Ha agilis környezetben dolgozol akkor rendkívül fontos, hogy ezt a csapatban mindenki megértse.

## De a tesztelés soha nincs befejezve. Hogyan döntöm el mit teszteljek?

Ez igazából nem tesztmenedzsment probléma. Ez egy követelménybeli, minőségi és tesztelési probléma amire a tesztmenedzsment eszközök ajánlanak látszólagos megoldásokat.

Tehát ne pocsékoljunk időt arra, hogy kitaláljuk hogyan irányítsuk a tesztelési folyamatot egy tesztmenedzsment eszközzel vagy, hogyan priorizáljuk a megírt teszteseteinket. Minden perc amit a tesztmenedzsment eszközre fordítunk, egy perc, amit nem arra fordítottunk, hogy megértsük a fejlesztés alatt álló rendszerünk valódi állapotát.

Fektessünk inkább abba az energiánkat, hogy ténylegesen közvetlenül előrefelé mozdítsuk a projektet. Értsük meg a megrendelő elvárásait, automatizált elfogadási teszteken keresztül ellenőrizzük ezeket a követelményeket és felderítő teszteléssel (exploratory testing) fedjük fel a kockázatokat és sebezhetőségeket.

## Mi a helyzet a riportokkal?

A hagyományos tesztmenedzsment eszközök sokféle riporttal szolgálnak. Pl: becsült és valós végrehajtási idő, tervezett és végrehajtott tesztesetek, helyes és hibás tesztfuttatások aránya, stb. A legtöbb ilyen fajta információ haszontalan az agilis környezetekben.

A CI rendszer biztosítja azokat az információkat amelyek értékesek számunkra: az automatizált tesztfuttatási eredményket. És a legtöbbször ezeknek az adatoknak 100%-ig zöldnek (megfeleltnek) kell lenniük.

## Mi a helyzet a korábbi teszteredményekkel?

A legtöbb agilis csapat úgy találja, hogy a pillanatnyi CI jelentések fontosabbak, mint a korábbi eredmények. Ha a CI build pirosra vált bármilyen okból, akkor a csapat megáll és kijavítja azt. Így az agilis csapatokra nem vonatkozik a helyes/hibás tesztesetek arányának javulása úgy, mint a hagyományos csapatokra a fejlesztési fázisban. Ez azt jelenti, hogy a korábbi eredmények, trendek általában egyáltalán nem olyan érdekesek az agilis csapatok számára.

Azonban ha a csapat tényleg nyomon akarja követni a korábbi tesztfuttatások eredményeit (vagy rá vannak kényszerítve a belső szabályok miatt), akkor a teszteredmények elraktározhatóak az SCM-ben.

## A szabályozási kényszerről jut eszembe, hogyan juthatunk egyetértésre tesztmenedzsment rendszer nélkül?

Ha a környezetedben FDA, SOX, ISO vagy csak egy belső ellenőrzési szabályzat van, akkor valószínűleg egy olyan világban élsz ahol:

- ha nincs valami dokumentálva, akkor az meg sem történt
- megmondjuk mit csinálunk és azt csináljuk amit mondunk
- a tesztek megismételhetősége nagyon fontos

Ebben a környezetben a specializált tesztmenedzsment eszközök megoldásai ténylegesen mértékadóak, de nem a legjobb megoldások. Ha egy olyan rendszeren dolgozunk amiben a követelmények, tesztesetek és futási eredmények jól körülhatároltak, konkrétak, akkor válasszuk inkább az átvételi teszt által irányított fejlesztést (Acceptance Test Driven Development). Az ATTD módszer értékes adaléka, hogy végrehajtható követelményeket biztosít. Azzal szemben, hogy a tesztesetek és a követelmények csak megmondják hogy az alkalmazásnak hogyan kellene működnie, a végrehajtható követelményeket le is lehet futtatni, hogy megmutassuk tényleg azt csinálják.

Természetesen az ATTD sok erőfeszítést igényel, de egy különálló tesztmenedzsment eszköz adminisztrálása, az összes követhetőségi mátrix és a szükséges plusz dokumentációk fenntartása is.

## A vezetés előírta a tesztmenedzsment eszköz használatát. Most mi legyen?

Ajánld nekik ezt a cikket és kérd meg őket, olvassák el. Aztán kérdezd meg őket, hogy milyen plusz előnyőket kapnak a tesztmenedzsment eszköztől amit ne kapnának meg ha kihasználnák az SCM-et, a CI-t, a feladatlistát és az automatizált regressziós tesztet.

## Szóval meggyőztelek?

### *Elisabeth Hendrickson*

Elisabeth Hendrickson alapítója és elnöke annak a Quality Tree Software vállalatnak, amely tanácsadással és oktatással foglalkozik. A cég hatékony és tömör megoldási javaslatokkal próbálja segíti a fejlesztői csapatok munkáját. Ugyancsak ő alapította az Agilistry Studio-t, amely központi területévé vált az agilis szoftverfejlesztésnek a kaliforniai Pleasanton-ban. Több mint 20 éves szakmai tapasztalattal a háta mögött Elisabeth 2003 óta meghatározó tagja az agilis közösségeknek. 2006-2007 között segítette az Agile Alliance igazgatóság munkáját, valamint ő az egyike fő szervezője az Agile Alliance Functional Testing Tools programnak. Elisabeth megosztja idejét a tanítás, az előadás, az írás és az agilis csapatban való munkája között. Csapatában teszteléssel „fertőzött" programozók dolgoznak, akik becsülik az ő tesztelési megszállottságát.
Megtalálhatod a Twitteren @testobsessed, vagy olvashatod blogját http://testobsessed.com


# *A PDCA elv alkalmazása a tesztelésben*

**Dr. W. Edwards Deming munkásságából merítvén, kísérletet teszek arra, hogy az általa híressé tett PDCA elvet a programozók és tesztelők közötti munka ütemezésére, egymás megértésére alkalmazzuk.**

## A klasszikus út

Örök probléma, hogy mikor, milyen tartalommal, milyen céllal érkezik a tesztelésre a fejlesztés alatt álló alkalmazás új buildje; folyamatos a késés, állandóan előfordulnak jelentős, utolsó pillanatban a buildbe tett módosítások.

A teszt csapat meg ilyenkor áll, néz, hogy mi történt. Felkészülni, teszteseteket frissíteni, teszt futtatásokat ütemezni jellemzően már nincs idő. A helyzetet nem egyszer nehezebbé teszi ha az adott build “release candidate” állapotban érkezik, tehát még a teszteléshez alapvetően nem értő termékgazda / projektmenedzser is a tesztcsapat sarkában lohol, követelvén a mielőbbi kibocsátást.

Egy-két ehhez hasonló szituáció - és átvirrasztott éjszaka - nyomán kezdtem el keresni valami egyszerű, kézzelfogható, megvédhető megközelítést a fenti helyzet megismétlődésének elkerülése érdekében.

Deming PDCA elve egyszerűen átlátható, minden oldal által megérthető megoldási javaslatot nyújtott.

Alábbiakban nézzük át hogyan is lehetne a PDCA elvet alkalmazni főként a programozás és a tesztelés kapcsolatában:

## PLAN

A programozók feladata az új build tartalmának meghatározása, beleértve a hiba javítási, új fejlesztési feladatokat is. Az ütemezési kérdések tisztázása a projektvezető aktív részvételével. Egyeztetés a dokumentáció írásért felelőssel.
A build tartalmának publikálása egy a projekt összes részvevője által elérhető helyen (erre leginkább a hibajegy-, feladatkövető rendszerek alkalmasak, ahol fel kell állítani egy közös, globális szűrő feltételt, amelynek használatával mindenki „ugyanazon az oldalon” van)
A tesztelők feladata a közös területen elérhető tervezett build tartalom alapján, bárhol és bármilyen struktúrában is található, a teszteset “vagyon” frissítendő részének azonosítása; egyeztetés a programozói oldallal, a tervezett funkciók minél szélesebb körű megértése. Nagyon fontos, hogy a tapasztaltaktól eltérően, mindenképp kell, hogy legyen a tesztcsapatban erre szabad kolléga, aki nem az előző build elhúzódó tesztelésén dolgozik. Teszteset frissítés alatt jobb esetben az automatizált tesztek frissítését is érteni kell.

Projektvezetési kérdés, hogy a PLAN fázisnak mikor van vége, illetve van-e egyáltalán definiált vége. Tesztelői szempontból erőteljesen javasolt, hogy legyen.

## DO

A programozók feladata a build tervben meghatározott feladatok kivitelezése, hibajavítás, új funkciók fejlesztése. Kisebb mértékben, egyeztetve a teszteléssel, projekt vezetéssel a build tartalom változása is elképzelhető.

A tesztelőknek fel kell készülniük az új build tesztelésére: a meghatározott teszteset frissítési feladatok kivitelezésére, a tesztkörnyezet felállítására és a teszt futtatás ütemezésére.

## CHECK

A programozók az elkészült fejlesztési munkákat ellenőrzik, a unit teszt, code review stb. után buildbe fordítják, majd átadják a tesztelésnek.
A tesztelők az előre elkészített teszt környezeten, frissített tesztseteket futtatnak az alkalmazás új buildjén. Hibajegyeket rögzítenek, kommunikálnak a programozókkal a legjobb közös megértés érdekében.

## ACT

A programozók a projektvezetéssel közösen priorizálják a talált hibajegyeket, a fontosnak, javítandónak talált jegyeket javítják.

A projektvezető a projekt helyzete és a hibajegyek milyensége alapján dönt a javító build gyors összeállításáról, vagy a következő standard build ütemezésének megkezdéséről.

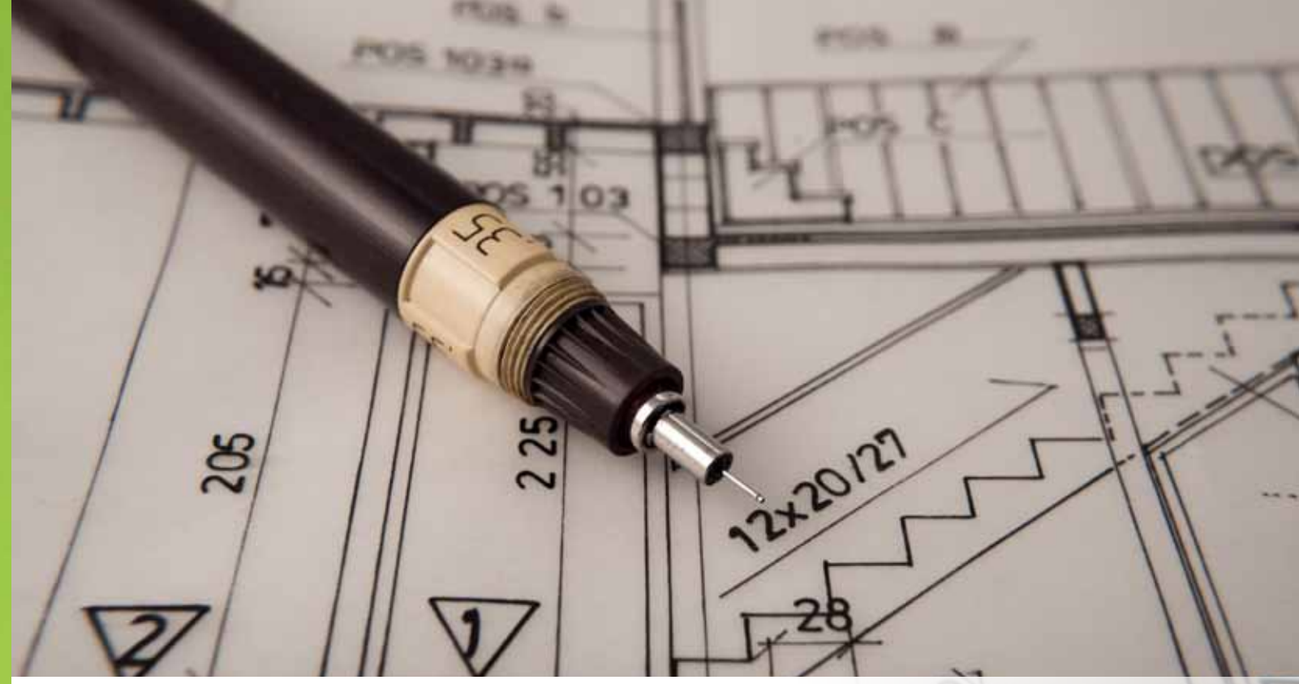
A tesztelők az esetleges javító build tesztelését végzik. A teszt futtatás során talált teszteset hibákat javítják, belső folyamatokat vizsgálnak, javítanak. Szükség szerint együttműködnek a programozókkal a hibajegyek megfelelő reprodukálása, javítása érdekében.

## Az agilis világ

A fentiek talán kicsit követhetőbbé, élhetőbbé teszik a tesztelők mindennapi munkáját egy klasszikus felfogású szervezetben. A 10 éve zászlót bontott agilis szoftverfejlesztési megközelítés a fenti elveket gyúrja össze. A programozók, tesztelők együtt fejlesztik a terméket, nincs lényegi időbeni különbség a teszt tervezés, program tervezés, a kódolás és a teszt eset írás között; az iteráció alatt a scrum master feladata a csoport megvédése a jelentős változtatásoktól. Dinamikus, határok, átadás átvételi pontok (és akár rögzített hibajegyek) nélküli, minden nap érvényesülő PDCA szerinti munkavégzés jellemzi az agilis projektet.

***Schaffhauser Balázs***

2001-től tesztvezetőként, az Egroupban, az Avon Cosmeticsben, a Lufthansa Systemsben és a 3DHistechben volt lehetőségem tanulni, megismerni számos iparágat a tesztelés szemszögéből. Tesztcsapatok felépítése, támogatása, eszközök kiválasztása, bevezetése tartozott a feladataim köré. Jelenleg a Scrum módszertant tanulom, Scrum masterként tevékenykedek.




# *Automatizált tesztelés II.*

**Az előző részben az automatizált tesztelés előnyeiről és hátrányairól, valamint az autotesztkörnyezet felépítésének mikéntjéről ejtettem szót. Kitértem a fejlesztés során felmerülő problémákra, egyszerűsítési lehetőségekre, alternatívákra. E második részben a tesztkörnyezethez szükséges keretrendszer tervezéséről, a tesztek megírásáról szeretnék minél több hasznos információt megosztani az olvasóval.**

## A keretrendszer

Ha úgy döntöttünk, hogy bevezetjük a tesztek automatizálását, és a tesztelendő rendszerünk elosztott rendszert alkot, akkor érdemes elgondolkozni egy keretrendszer fejlesztésén, amely lehetőséget biztosít az ilyen környezetben történő párhuzamos és problémamentes automatizált tesztfuttatásra. Magas költségekkel kell számolnunk, mivel egy ilyen rendszer megtervezése és megvalósítása akár hónapokat is igénybe vehet.

Korábban említettem, hogy a tesztkörnyezetet egy teszt konfiguráció manager (test CM) és egyéb teszteket végrehajtó komponensek alkotják. Mivel ezek erősen elkülönülnek egymástól, megállapíthatjuk, hogy két program elegendő a keretrendszer kialakításához, melyek megírásánál érdemes arra törekedni, hogy minél kevesebb külső programot használjon, de inkább egyet sem. Ez utóbbi azért is fontos, mert ezeket a keretrendszer felkonfigurálásánál újra és újra fel kellene telepíteni a gépekre, amely nagyszámú teszteket végrehajtó komponens esetén rengeteg időt venne igénybe.

A test CM-en lévő program lesz felelős a tesztek végrehajtásának vezérléséért, és kliens szerepet tölt be. A könnyű kezelhetőség végett minden releváns adatot ajánlott egy külön fájlban tárolni és nem paraméterként átadni. Érdemes a napjainkban kialakult trendek közül választani, mint például az XML, amellyel rendkívül egyszerű dolgozni. (A parseoláshoz számos előre megírt ingyenes XML library közül válogathatunk, mely igencsak megkönnyíti a dolgunkat.) Így ez a konfigurációs XML fájl lesz, ami tartalmazza majd a test set-eket, scenariokat és azokon belül is a test case-eket, így biztosítva, hogy egy konkrét „tesztcsokrot” tudjunk lefuttatni mindenféle felhasználói közbeavatkozás nélkül. Minden tesztesetnek kötelezően adjunk nevet, hogyha véletlenül a függőségek vizsgálatát is belevennénk a fejlesztésbe, tudjunk hivatkozni rájuk. Alapvető adatok még a teszteket végrehajtó komponensek elérhetőségei, mint ip és port, valamint a lefuttatandó teszt neve, ami a többi futtatáshoz szükséges fájllal együtt a test CM-en található meg.

**Példa konfigurációs fájlra:**

```
<run name=”almafa”>
        <scenario name=”test”>
        <testcase name=”B” exec=”run/
script.sh” url=”socket://127.0.0.1:1111”>
        <depends name=”A”/>
        <resource path=”run”/>

        <stdout to=”out.txt”/>
        <stderr to=”err.log”/>

        <result from=”R1”/>
        <result from=”R2”/>
        </testcase>
        <testcase name=”A” exec=”ls
-l” url=”socket://127.0.0.1:1112”/> …
        </scenario>…
</run>
```

Mint a példán látható, egyéb megadható opciók lehetnek még a futtatáshoz szükséges fájlok helye (resource), a logok helye és neve (stdout, stderr), melyek a test CM-en jönnek létre a futtatás során. Beállíthatóak az eredmények, azaz a tesztek által generált fájlok illetve mappák neve is (result). Egy vagy több másik teszteset nevének megadásával beállíthatjuk a függőségeket (depends), így ezek sikeres lefutásától lesz függő a beállított teszt végrehajtásának elindulása.

Ha a keretrendszert nem csak egy-egy teszt lefuttatására szeretnénk használni, hanem több tesztesettel, és mondjuk

ugyanazon teszteket végrehajtó komponensen, akkor mindenképpen ajánlott a függőségek vizsgálatának fejlesztése. Egyéb esetben minden teszteset egyszerre fut le az adott gépen, és ez különböző problémákhoz vezethet. Ez az egyszerű pár soros metódus a konfigurációs fájl feldolgozása után ellenőrzi, hogy az egyébként irányított gráfot alkotó tesztesetek tartalmaznak-e kört. Ha igen, még a tesztek lefuttatása előtt megszakítja a program működését, és figyelmezteti a felhasználót, hogy mely tesztesetek érintettek az ügyben. Ez azért fontos, mert az érintett tesztek kör esetén soha nem futnának le. Ezzel a funkcióval rengeteg hibát kerülhetünk el.

Mindeközben a teszteket végrehajtó komponenseken szerverként figyel a tesztek futtatását végző program, és várja, hogy a test CM-től megkapja a végrehajtáshoz szükséges információkat, ami akkor történik meg, ha függőségi vizsgálat esetén egy irányított körmentes gráfot kaptunk. Ezután a komponensek lefuttatják a teszteket, és a stdout-ot és stderr-t folyamatosan küldik át a test CM-nek, így hiba esetén mindig tudjuk majd hol bukott el a teszt.

### Tesztek

A keretrendszer működése független a tesztek megírásától, minden futtatható állományt vagy parancsot elfogad tesztesetnek. Innentől egyértelmű, hogy a teszteket bármilyen programozási nyelven írhatjuk, erre semmilyen megkötés nincsen. Sőt, a programozást abszolút elkerülhetjük, ha olyan rendszert tesztelünk, melynek tesztelését le tudjuk vezényelni különböző teszteszközökkel, amennyiben azok rendelkeznek felvétel és visszajátszás funkcióval. Ekkor a tesztet lefuttatjuk manuálisan, miközben minden lépést felveszünk a speciális teszteszközzel, majd az elmentett tesztet adjuk meg a konfigurációs fájlban, így az a futtatás során minden lépést újra végrehajt majd a tesztelést végrehajtó komponenseken.

Cikkem előző részében említettem, hogy ha nem használunk felvétel funkcióval rendelkező teszteszközöket, akkor írjuk meg úgy a teszteket, hogy minél kisebb komponensekre bontsuk fel azokat. Így nem csak a karbantartás jár alacsonyabb költségekkel, hanem sokkal tesztelőbarátabbá tehetjük az egész rendszert.

### Tesztelő-barát, de hogyan?

Még ha a legkisebb komponenseket a fejlesztők írják is, a teszteket mindenképpen a tesztelőknek kell előállítaniuk a hatékonyság érdekében (értem ezalatt, hogy a fejlesztői fázisban ne fordulhasson elő, hogy a fejlesztőt tesztelők tartják fel tesztek megírásával kapcsolatban). Hogyan történhet meg mindez, ha tesztelőink nem rendelkeznek elegendő tapasztalattal és tudással a programozás terén? Az előző cikkemben taglalt GUI fejlesztése lehetővé teszi, hogy tesztelőink könnyedén előállíthassák a követelményeket lefedő teszteket, anélkül, hogy programozniuk kellene. Ehhez mindössze arra van szükség, hogy a GUI lehetőséget biztosítson a tesztek összeállítására, méghozzá úgy, hogy a fejlesztők által már korábban megírt tesztlépéseket elég legyen csak összefűzni egyetlen tesztesetté a grafikus felületen. A GUI-n korábban kialakított tágabb jogkörrel rendelkező manager felhasználó pedig most válik igazán fontossá, mivel ő lesz az, aki jóváhagyja majd az összeállított teszteket, elkerülve ezzel a káoszt, ha a tesztelő netán nem megfelelően végezné a munkáját és rosszul összeállított tesztek futtatásába kezdene.

A magazin előző és mostani számában megjelenő cikkeim többnyire azoknak a vállalatoknak nyújtanak hasznosabb információkat, ahol még csak mostanában jött szóba a tesztjeik automatizálása és már a tervezést fontolgatják. Remélem, hogy e rövid cikksorozattal elértem célomat, és az érdeklődő olvasókat sikerült útbaigazítanom jövőbeli keretrendszerük felépítésével és működésével, valamint tesztjeik hatékony automatizálásával kapcsolatban.

***Horváth Nóra***

V&V gyakornok

Végzős hallgató az ELTE Informatika Karán, Programtervező Informatikus szakon. 2009 májusa óta gyakornok a GE HealthCare-nél. Automatikus és hagyományos tesztelés területeken tevékenykedik. Ezen felül a GE által kiírt Öveges Ösztöndíj Program keretében a szakdolgozatát írja.





# *Interjú - Simon György Ferenccel*

### Mikor és hogyan kerültél kapcsolatba a szoftverteszteléssel?

Felsőoktatási éveim utolsó szakaszában projektmenedzseri gyakornokként dolgoztam a T-Online Magyarország Zrt.-nél. Munkám során kimutattam, hogy a strukturált tesztelés, a tesztelési folyamat menedzselése terén még van mit fejleszteni a cég szervezetén belül. Megbíztak ezzel a feladattal, ami a tesztelési módszertan kidolgozásából és a tesztelési folyamat támogatásához szükséges teszteszközök kiválasztásából és bevezetéséből állt. Ekkor (2006-ban) kezdődött a tesztelői pályafutásom.

### Volt olyan pillanat a karriered során, amikor elhatároztad, hogy többé nem akarsz tesztelő lenni?

Nem volt ilyen pillanat, ezt a szakmát véleményem szerint a kreativitás, kihívás jellemzi. Sosincs egyforma tesztelési munka. Persze, nem mondanék igazat, ha azt mondanám, hogy nem voltak és nem lesznek nehéz pillanatok. Ahogy látom a problémák többsége a nem megfelelő minőségű, nem jól végiggondolt követelmények menedzseléséből fakad. Sok esetben a tesztelés közben alakulnak ki a végleges megoldások, legyen szó üzleti igényről, vagy akár a fejlesztésről. Erre még sok iterációval is nehezen lehet felkészülni, teszteseteket tervezni, erőforrást és tesztelési időt becsülni.

### Tudnál mesélni arról, hogy jelenleg milyen módszerrel folyik a tesztelés a Telekom-ban?

A Magyar Telekomon belül az egységes fejlesztési-üzemeltetési, ezen belül az egységes tesztelési módszer kialakításában az elmúlt időszakban nagy előrelépés történt: néhány évvel ezelőtt még a T-Mobile, T-Com, T-Online, T-Kábel saját IT portfólióval, fejlesztési technológiával, és saját módszertannal is rendelkezett. 2008-ban a portfólió és a szolgáltatások körének növelése, a költséghatékonyság és az integrált működés előnyei kihasználásának érdekében ezeket a cégeket összeolvasztották. Ekkor született meg a T-Home márka is. A szervezeti integrációt első lépésben az IT rendszerek integrációja, és az egységesítés felé tett lépések követték, majd ezután következett a munkafolyamatok egységesítése, a szabályozások integrációja, és az egységes tesztelési folyamatok kiterjesztése a cég egészére, amely még nem zárult le.

A Magyar Telekomban a fejlesztést elsősorban külső szállítók, a tesztelést pedig a Magyar Telekom saját szervezetei végzik.

Az egységes tesztelési folyamatok, módszerek a következő elveken alapulnak:

- Követelmény-alapú tesztelés: a tesztelést a priorizált üzleti követelmények vezéreljék.
- Dokumentált tesztelés: ad-hoc tesztelés helyett megtervezett, jóváhagyatott tesztforgatókönyvek alapján történjen a tesztelés, és a tesztek eredményei is dokumentálásra kerüljenek.
- Dedikált tesztkörnyezetek ahh, hogy a tesztelés az éles üzemi környezethez és adattartalomhoz minél jobban hasonlító környezetben történhessen.
- Tesztelési eszközök használata a tesztelés egyes fázisainak támogatására (tesztmenedzser eszköz a tesztek megtervezéséhez és a teszteredmények nyomon követéséhez, hibakövető rendszer, stb.).
- Mérések, minőségi mutatók kerültek definiálásra a tesztelési munka kontrolálásához.
- Automatizált tesztek készülnek a regressziós tesztelés támogatására.

**Mekkora csapattal és milyen eszközökkel teszteltek?**

A Magyar Telekomban az egységes tesztelési folyamat bevezetéséhez szervezeti átalakulás is tartozott: a teszteléshez a Magyar Telekom létrehozott egy központi tesztelési szervezeti egységet, melyet Test Factory-nak hívunk.

A Test Factory-ban jelenleg több tucat (külső és belső) munkatárs dolgozik, de az átszervezés még nem zárult le.

A teszteléshez a következő támogató eszközöket használjuk:

- Követelménykezelésre és tesztmenedzsment eszközként: HP Quality Center
- Hibakezelésre és kiadás menedzsmentre: IBM Rational ClearQuest
- Modell alapú teszt automatizálásra: IBM RSA -> Quick Test Pro (illetve, a régebbi teszteknél: WinRunner)

**Milyen ismérvei vannak egy jó tesztelőnek? Kik azok akik sikeresen pályázhatnak egy Telekom-os tesztelői pozícióra?**

Alapvetően szétbontanám teszttervező és tesztvégrehajtó szerepkörre. Egy jó teszttervezőnél nagyon fontos a kommunikáció képesség, az üzleti terület megértésére, üzleti folyamatok megismerésére való törekvés. Képes legyen megérteni az üzleti igényeket, és az ehhez tartozó IT megvalósításokat. Ezenkívül nagy előny a strukturált gondolkozás, ez főleg a dokumentációk elemzésénél, tesztterv készítésénél fontos.

A tesztmérnök inkább a fejlesztőkkel fog kommunikálni a munkája során. Legyen alapos és még egy kis pesszimizmus sem hátrány. Az a jó tesztmérnök, aki abban leli a kihívást, hogy magát a rendszert, a mögöttes folyamatait, valamint a rendszerek közti kommunikációt minél jobban megértse, elemezni tudja.

Ami mind a kettőre igaz, hogy képesek legyenek folyamatokban, rendszerekben gondolkozni. Nagyon fontos a határidők pontos betartása, a megfelelő adminisztráció, agilitás, proaktivitás, információra való éhezés.

**Mennyire vagy elégedett a mostani tesztelők képzettségével?**

Azt mondanám, hogy az 5 évvel ezelőtti állapothoz képest sokat javult a helyzet. Most már sokkal nagyobb a lehetőség a megfelelő munkaerő képzésére. Különböző tesztelési konferenciákon elhangzott, hogy már a főiskolák és egyetemek is komolyabban foglalkoznak a teszteléssel, külön szakirányok, kutatási területek is vannak. Külön kiemelném az ISTQB képzést, melyet minden teszteléssel foglalkozó személynek javasolnék.

**Milyen tanáccsal tudnád azokat a tesztelőket segíteni, akik most kezdenek ismerkedni a szoftvertesztelés-sel?**

A lelkesedés, az információ szomj, a hozzáállás sok esetben többet ér, főleg az elején, mint a technikai tudás. Ugyanakkor a szakmai képzés nagyon fontos, hogy a munkánk minőségében idővel előre tudjunk lépni.

**Eddig a múltról és jelenről volt szó, most érdekelne, hogyan látod a jövőt? Szerinted milyen irányban fog fejlődni a tesztelés?**

Szerintem a jövőben alapvetően két tesztelési típus fog kialakulni. Az egyik inkább technikai, fejlesztőkkel együtt dolgozó, azok munkáját segítő tesztelő. Főleg belső fejlesztést alkalmazó és azon belül a Scrum típusú módszertant követő cégeknél figyelhető ez meg. A másik tesztelői típus felfejlődik a business analyst réteg mellé és hasonló szerepkört fog betölteni, vagyis átmeneti réteget képez az üzleti és az IT terület között. Ez a típus inkább a tesztelési szolgáltatást nyújtó, illetve külső fejlesztéssel dolgozó multicégekre jellemző.

Úgy látom, hogy a manuális tesztelés támogatására is születtek új megoldások. Ezt az is bizonyítja, hogy kezdenek piacra törni a „fél automatizált“ eszközök, melyek nem váltják ki a manuális tesztelést, viszont azokat sokkal gyorsabbá, hatékonyabbá tudják tenni, ezzel is sok időt spórolva a tesztelési csapatnak.

Nagy előrelépés várható a Cloud tesztelés és a virtuális tesztkörnyezetek terén.

**Van ötleted, hogyan lehetne a szakmát jobban elfogadtatni? Hogyan kaphat nagyobb figyelmet a tesztelés?**

A jelenlegi irány szerintem biztató, a tesztelés egyre nagyobb tekintélyt kap. Nagyon gyorsítani a folyamatokon véleményem szerint nem lehet. A legfőbb javaslatom, hogy képezzük magunkat, menjünk el szakmai fórumokra, képzésekre, olvassunk blogokat. Cégen belül, meg ügyeljünk arra, hogy ne ad-hoc módon, hanem standardizált folyamatokon, módszertanokon keresztül teszteljünk. Ilyen esetben sokkal jobban tudjuk mérni, felmutatni az eredményeinket is. A tesztelés, már egy elismert és megbecsült szakma, dolgozzunk e szerint.

### *Simon György Ferenc*

Tesztelési munkairányító

2006-ban szerezte diplomáját a Budapesti Műszaki Főiskolán. Már a tanulmányai mellett is gyakornokként tesztelési és projektmenedzseri feladatokat látott el a T-Online Magyarország Zrt.-nél. A diploma megszerzése után munkáját ugyanennél a cégnél folytatva tesztelési módszertant alakított ki és vezetett be, valamint részt vett projektmenedzsment és fejlesztési módszertanok kidolgozásában. Azóta számos stratégiai projektben (2Play/3Play, T-Home, Kapcsolatprogram) tesztmenedzseri feladatokat látott el. Jelenleg tesztelési munkairányítóként dolgozik a Magyar Telekom Nyrt.-nél.



# *Webes tesztelés modell alapon*

**A webes applikációk tesztelésével több probléma van. Ezeken a felületeken a felhasználó eseményeket hoz létre. Pl. gombokra kattint, szövegmezőket tölt ki a billentyűzet segítségével, szöveget jelöl ki, vágólapra másol, majd beilleszt, sőt gondoljunk a Gmail-re, drag-and-drop funkciót is használhat. Ezekkel az eseményekkel egy baj van, túl sok van belőlük. Minél több eseményünk van, annál több sorrendiség létezik, így a lehetséges tesztesetek száma exponenciálisan növekszik.**

Az előző számban „A modell alapú tesztelésről” című cikkben már láttuk, mire jó ez a tesztelési megközelítés. Folytassuk most az ottani gondolatmenetünket és nézzük meg, hogyan tudjuk ezt weboldalak funkcionális tesztelésére használni!

## Egy kis ismétlés

Dióhéjban a modell alapú tesztelésről annyit, hogy a tesztelendő rendszerünkről (legyen az jelenesetben egy webes applikáció) készítünk egy modellt, ami a program viselkedését ábrázolja. Ezt a modellt felhasználjuk arra, hogy automatikusan generáljunk futtatható teszteseteket. A modell lehet egy gráf (véges állapotautomata), döntési táblázat, de tulajdonképpen bármi, ami szemléltetni tudja a rendszer működését és lehet belőle teszteseteket generálni (legutóbb említettünk még nyelvtanokat és a Markov-láncokat is). Ettől jobban nem szeretném most részletezni a megközelítést, aki többet szeretne olvasni róla, ne legyen rest, kapja elő az előző szám cikkét és vessen egy pillantást az első három bekezdésre. A továbbiakban látni fogjuk, hogyan segíthet nekünk az MBT egy weboldal tesztelésében.

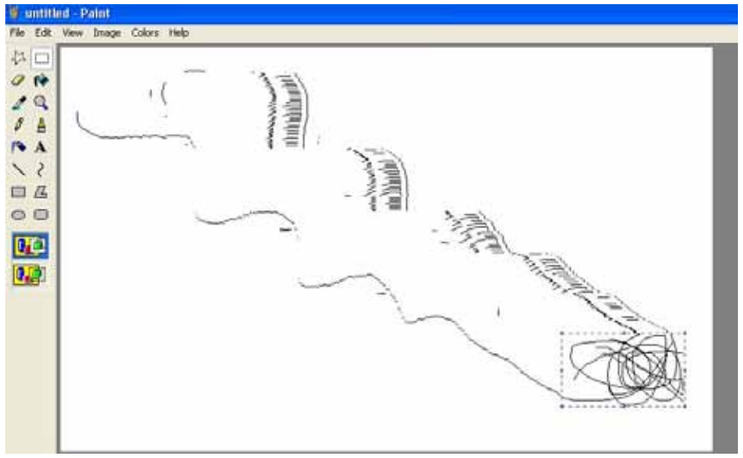
## Miért éppen web?

A webes applikációk tesztelésével több probléma van. Ugyanazokkal a problémákkal szembesülünk, mint egy GUI program tesztelésénél. Ezeken a felületeken a felhasználó eseményeket hoz létre. Pl. gombokra kattint, szövegmezőket tölt ki a billentyűzet segítségével, szöveget jelöl ki, vágólapra másol, majd beilleszt, sőt gondoljunk a Gmail-re, drag-and-drop funkciót is használhat. Ezekkel az eseményekkel egy baj van, túl sok van belőlük. Már egy kisebb programnál, vagy weblapnál is rengeteget tudunk megkülönböztetni. Az eventek sorrendje meghatározhat egy tesztesetet, ami elvezet minket egy elvárt vagy egy kevésbé elvárt eredményhez. Minél több eseményünk van, annál több sorrendiség létezik, így a lehetséges tesztesetek száma exponenciálisan növekszik. De érdemes-e ezeket a kombinációkat tesztelni?

## Motiváció

Ha a kedves olvasó Windows XP alatt dolgozik, próbálja ki a következőt. Nyissa meg a Paint-et, majd rajzoljon bele valamit. Megteszi egy pár fekete vonal a fehér háttéren. Ezek után hozzunk létre egy kijelölést a Select gombbal és mozdítsuk el a kijelölt területet. Lehetőleg a fekete vonalainkból jelöljünk ki valamit, majd látni fogjuk, hogy nyomva tartott egérgombbal át tudjuk helyezni a kijelölt területet. Fűszerezzük meg ezt a feature-t egy másik eseménnyel, tartsuk nyomva az áthelyezés alatt a SHIFT gombot. Máris 3D-ben rajzolunk! Ekkor ugyanis a Paint nem frissíti a rajzterületet. Egy jól ismert hibát csalogattunk most elő, nagyon egyszerű események kombinációjával. (1. ábra)

Ha már előttünk az XP lássunk egy másik hibát. Nyissuk meg a Computer Management programot, megtehetjük pl. úgy, hogy jobb egérgombbal kattintunk a My Computer-re, majd Manage. Itt is kattintsunk jobb egérgombbal a Computer Managment-re (local) a baloldalon, a fastruktúra legtetején, majd válasszuk ki a „Connect to another computer…” opciót. A felbukkanó ablakban válasszuk az „Another computer”-t, majd írjunk be olyan hosszú karaktersorozatot, amennyit csak enged a szövegmező! Kattintsunk az OK gombra. Természetesen egy nem létező számítógéphez akartunk kapcsolódni, ezért hibaüzenet fogad minket helyesen. Nyomjunk meg az OK




1. ábra – Hiba a Paint-ben

gombot a hibaüzenetnél, majd vegyük észre, hogy a baloldalon a fastruktúrában egyetlen ikon van, az iménti nem létező gép nevével. Kattintsunk jobb egérgombbal és nézzük meg a tulajdonságait (Properties). Ekkor a program összeomlik. (2. ábra)

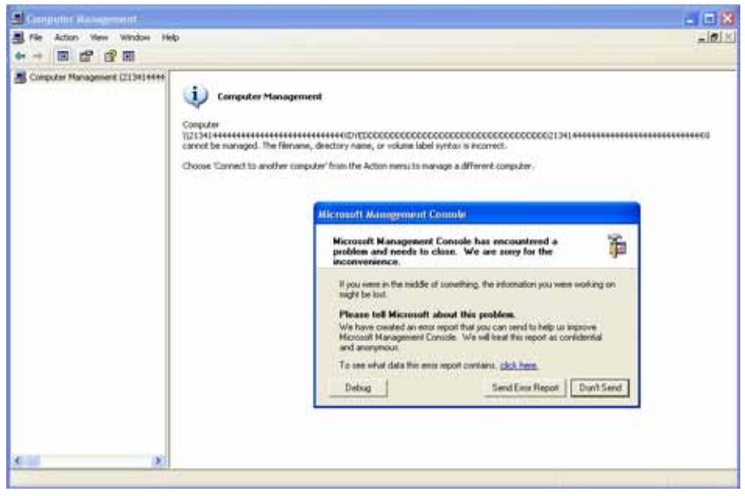
Az utóbbi példa Atif M. Memontól a Marylandi egyetem professzorától származik, aki egyik előadásában (http://www.youtube.com/watch?v=6LdsIVvxISU) említ egy webes hibát is. Egy amerikai légitársaság honlapján, ha a felhasználó bejelentkezett, majd közvetlen kijelentkezni próbált, azonnal egy „HTTP Error 500 Internal server error" hibával találta szembe magát, ha bármilyen más műveletet hajtott végre a kijelentkezés előtt és utána próbált meg kijelentkezni, ezzel a hibával nem találkozott.

## Az MBT közbeszól

A fenti hibák mindegyike egy eseménysor hatására jött létre, amit a tesztelők figyelmen kívül hagytak. Nem is csoda, hiszen már egyszerű alkalmazásnál is szinte lehetetlen mindent kipróbálni, annyi esemény van benne. De akkor mégis tesztelőként, hogyan tudunk ilyen típusú hibákat találni?

Az egyik lehetséges mód, a manuális tesztelés. Nem ritkán diákokat vagy szakembereket ültetünk a weboldal elé, majd kézzel végig kattintják azt, jobb esetben valamilyen követelmény-leírás alapján. Ezt a módszert nem kell tovább taglalnom. Rengeteg esetben elkerülhetetlen, sőt vannak olyan esetek, amikor csak és kizárólag manuálisan lehet tesztelni, gondoljunk pl. egy weboldalon lévő captchára (http://hu.wikipedia.org/wiki/Captcha). A kézzel való teszteléssel több baj van, ezek általános hátrányok, nem csak a webes tesztelésre igazak. Az egyik legnagyobb baj, hogy a tesztesetek nem használhatók fel újra. Ha ismét tesztelni akarunk, újra és újra végig kell kattintanunk az esetet. Vegyük még figyelembe, hogy a verifikálás is „csak" szemmel megy, azaz elképzelhető, hogy a tesztelő egyszerűen nem veszi észre a hibát. Ráadásul szinte biztosan csak a legáltalánosabb esetekre születnek tesztek, sok esetben kizárólag a sikeres felhasználói tesztesetek készülnek el. Tipikusan kevés számú eset áll rendelkezésre, így minimalizálódik annak az esélye, hogy a fentiekhez hasonló hibákat fedezünk fel manuálisan.

De akkor hogyan? Próbáljunk a fenti hátrányokra megoldást találni. Az egyik út a Capture/Replay eszközök használata. Ezek az eszközök képesek a felhasználói interakcióinkat elkapni, majd eltárolni, később pedig újrajátszani. Ezzel máris megoldódott az újrafelhasználási problémánk, sőt verifikálni is képesek, azaz még egy kérdést kipipálhatunk. Ilyen eszköz pl. a Selenium, amelyről most is szó lesz. A Selenium egy ingyenes nyílt forráskódú szoftver, amely akár közvetlenül a Firefox-ba beépülve tudja elkapni a felhasználói interakciókat. Ezek után nem csak visszajátszani tudja a tesztlépéseket, hanem exportálni is többféle programnyelven! Ez azért fontos, mivel innentől kezdve nagyobb szabadság van a kezünkben, mert különböző programnyelvek eszközeivel írhatunk teszteseteket. A futtatás a következőképpen alakul. Létezik egy Selenium Remote Control névre hallgató szerver folyamat, ami azért felelős, hogy fogadja a tesztesetünktől érkező kéréseket. Ilyen kérések pl. egy gombra való kattintás, szövegbevitel, várakozás egy bizonyos ideig, egyszóval a tesztlépéseink. Majd ezeket a lépéseket továbbítja a megadott böngészőnek, ami sorba végrehajtja azokat. Így tudjuk automatikusan, a már felvett tesztjeinket lefuttatni.




2. ábra – A Computer Management összeomlik

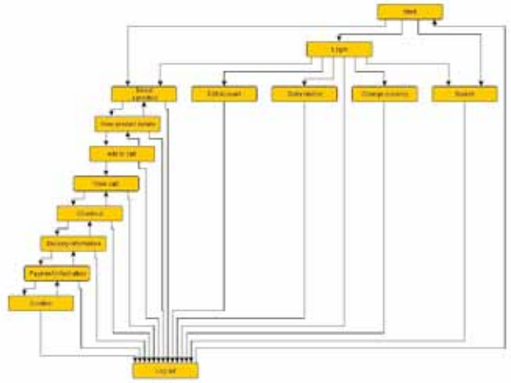
## A Selenium és a Graphwalker

Előző cikkemben az MBT eszköz a Graphwalker (http://graphwalker.org/) volt. A Graphwalker képes arra, hogy egy inputként megadott modellt (graphml formátumú véges gráfot) bejárjon és a megadott csomópontokban (állapotokban) egy bizonyos műveletet végezzen. Azaz ha építünk egy véges állapot automatát, jelen esetben egy gráfot, a programunk viselkedéséről, akkor a Graphwalker egy-egy tesztesetet jár végig a gráfban. Nincs más dolgunk, mint összekötni a gráf pontjait, éleit, egy-egy Java-ban implementált függvénnyel, ami a tesztlépéseket reprezentálja. Webes tesztelésnél ezt úgy hasznosíthatjuk, hogy az egyes függvényekbe a Selenium által generált lépéseket építjük be. Pl. az első állapot egy webshopot ábrázoló gráfban lehet a bejelentkező képernyő, az állapotátmenet lehet a „Login" gombbal való bejelentkezés, a következő állapot pedig az üdvözlő képernyő.

## Nézzük élesben

Legutóbb egy valós példát is néztünk, tegyük ezt most is! A célrendszerünk legyen az OpenCart, egy ingyenes webshop demo oldala (http://demo.opencart.com/). Rakjunk össze egy modellt (pl. az előző számban emlegetett yEd-del), ami mindössze annyit reprezentál, hogy be tudunk jelentkezni, ki tudunk valamit választani, kosárba rakni, megvenni, majd ki is tudunk jelentkezni. Helyezzük a hangsúlyt a kijelentkezés tesztelésére. A modell a mellékelt ábrán látható. (3. ábra)

A weblap modelljén jól látható, hogy minden állapotból (kivéve a Startból) ki tudunk jelentkezni (a „Logout" gomb mindig látható az oldalon és rá tudunk kattintani, ha már bejelentkeztünk). A modell természetesen nem teljes, a kijelentkezéssel kapcsolatos átmenetekre koncentrálunk. Járjuk be a gráfot a Graphwalker-rel! Ha azt állítjuk be, hogy minden élt, vagy pontot érintsen, akkor garantáltan minden utat le tudunk tesztelni a gráfban. Vagyis olyan események sorrendjét is, amit más tesztelési módszerekkel nehezen. Figyeljük meg, hogy pl. a bejelentkezés utáni közvetlen kijelentkezés is tesztelve lesz. Egyébként azt is választhatjuk, hogy véletlenszerűen járjuk be a modellt, össze-vissza bolyongva egy megadott ideig. Ezeknek a paramétereknek a variálásával növelhetjük a lefedettséget. Ha egy ilyen modellt összeállítunk, nincs más dolgunk, mint rögzíteni a tesztlépéseket a Selenium IDE-vel Firefox-ba. Így tudhatjuk meg, hogy hogyan mondjuk meg a Selenium RC-nek, hogy pl. hogyan „kattintson" rá a „Login" gombra. Ez egy olyan beépülő modul, amellyel rögzíteni tudunk közvetlen a Firefox-ból és ki tudjuk exportálni Javába a tesztlépéseket. A megfelelő utasításokat bemásoljuk a Graphwalker által végrehajtott Java forrásba. Ezt a forrást az előző cikkben leírtak alapján lefordítjuk, majd a Graphwalker-t elindítjuk parancssorból. Így kapunk teljesen automatizált teszteseteket. Az eseménysorrendek előállításáért a Graphwalker felel, a webes alkalmazás irányítását a Selenium végzi. Ha egy elég komplex gráfot építünk (még átlátható eseményhalmaz, de sok átmenet), kellően „sok" állapottal, akkor a véletlenszerű bejárás során olyan hibákat találhatunk amilyeneket a cikk elején említettünk. A véletlen bejárásnak azért van jelentősége, mert ha azt választjuk, hogy minden élt vagy pontot érintsünk, akkor egy előre meghatározott algoritmus alapján mindig ugyanazokat az eseménysorokat fogjuk visszakapni (jóllehet ezzel is foghatunk hibákat), míg a véletlen bejárás mindig újabb és újabb utakat, eseménysorrendeket generál.



3. ábra – OpenCart modellje a yEd-ben

## Vannak még kihívások

A fenti technika jól használható, ha nem túl bonyolult a weblapunk (relatív kevés esemény), ill. ha megfelelően implementált tesztlépéseink vannak. Már az OpenCart-nál is rengeteg funkciónk/eseményünk létezik. Ha mindent bele akarunk építeni egy modellbe, akkor bonyolult és átláthatatlanná válik. Ilyenkor érdemes funkciók szerint több modellt építeni. Szót kell ejteni a Selenium kevésbé előnyös képességeiről is. Aki ismeri az tudja, hogy ha rögzítünk vele valamit, sokszor visszajátszani sem tudjuk a Firefox-ból. Ha kívülről Selenium RC-n keresztül próbálkozunk, akkor további nehézségekbe ütközünk. Egyszóval a kiexportált kódok után nagyon sokat kell még „kézzel" igazítani. Itt meg kell jegyeznem, hogy sok kereskedelmi szoftver lényegesen jobb ebben. Szintén a Selenium számlájára írhatjuk a hordozhatóság problémáját, egyáltalán nem biztos, hogy ami Firefox-on futtatható teszt az Chrome alatt is az. További problémák még a váratlan események, pl. előfordulhat, hogy le kell kezelnünk a hirtelen felugró pop-up ablakokat stb. Aztán felmerül még több elméleti kérdés is. Mekkora modellt kell építenünk? Bele kell vennünk az összes létező eseményt? Ha igen, az összes sorrendet le kell tesztelnünk? Minden esetben tudunk verifikálni? Egyszóval sok kérdés kering még a GUI és a webes felületű tesztelések körül egyaránt. Valamint fejlődniük kell még a tesztelési módszertanoknak és az eszközöknek is.

## Végezetül

Azt bátran állíthatom, hogy weboldalak manuális tesztelésénél eredményesebb a fenti technika. A kezdeti nehézségek után (sok konfigurálás, kódjavítás) egy megbízható modellel sokat lehet nyerni. Mindenkinek ajánlom, hogy próbálja meg alkalmazni a modell alapú tesztelést weboldalak tesztelésére is, sok esetben megéri.

**Tóth Árpád**

Szoftvertesztelő mérnök

Tóth Árpád okleveles programozó matematikus. 3 éve foglalkozik szoftverteszteléssel. Kezdetben a Nokia Siemens Networks, jelenleg pedig az IT-Services tesztmérnöke. Érdeklődési területe a modell alapú tesztelés, valamint a terheléses tesztelési technikák. Korábban kifejezetten funkcionális, jelenleg pedig performance tesztekkel foglalkozik. A szakmán kívül, többszörös magyar bajnok formációs latin táncos. Showtánc Európa- és világbajnok.
Arpad.Toth@t-systems.com

TESZTELÉS A GYAKORLATBAN
A SZAKÉRTŐ TESZTELŐK LAPJA

## Tapasztalattal rendelkezel a tesztelés területén?

*Oszd meg másokkal!*

*Tesztelés a Gyakorlatban*
*A szakértő tesztelők lapja*

# A mai alkalmazások biztonsága egy pen-tester szemével

**Mi is az a betörési tesz és miért fizet valaki ilyesmiért? A biztonsági betörési teszt célja egy rendszer sebezhetőségeinek felmérése gyakorlatias megközelítést alkalmazva, a valódi támadók eszköztárának felhasználásával. A betörési tesztet végző szakemberek alapvetően ugyanúgy fő elfoglaltságként a sebezhetőségek felderítésével és kihasználásuk mikéntjével foglalkoznak mint a számítógépes bűnözők.**

Mindkét térfélen rengeteg időt és energiát fektetnek a támadási technikák kifejlesztésére és demonstrálására vagy éppen tényleges végrehajtására. Az alapvető különbség a célokban rejlik, míg az egyik esetben a rendszer hibáinak kijavítására és a biztonság megerősítésére, a másikban jogosulatlan haszonszerzésre, lopásra és károkozásra kell gondolni. Egy jó biztonsági szakértőnek mindenekelőtt érteni kell az elterjedt programozási nyelvek, alkalmazás platformok (.NET, Java), adatbázis-kezelő rendszerek és operációs rendszerek biztonsági vonatkozásaihoz továbbá az ismert sebezhetőségek kihasználásának módjaihoz és hacker eszközök használatához. Szükséges továbbá egy kis csavar ami megkülönbözteti az elvileg hasonlóan jól képzett rendszergazdáktól és fejlesztőktől, ez pedig a szemléletmódban rejlik. Tudnia kell ugyanis a bűnözők fejével gondolkodni. A tapasztalatok szerint erre a főállású fejlesztők és rendszergazdák csak nagyon kis hányada képes. A gyakorlatban a „készen” átadott rendszerek szinte kivétel nélkül tartalmaznak súlyos biztonsági hibákat, amiket egy későbbi biztonsági betörési tesz a felszínre hoz. A betörési tesztek célja az ilyen hiányosságok felderítése és a biztonsági szakértők javaslatai alapján a kijavítása. Mi történik ha ezeket nem javítják ki? Lehet hogy hosszú ideig semmi. Aki olvas híreket mostanában, annak látnia kell viszont: a tendenciák a játszma eldurvulására utalnak. A „rosszfiúk” valóban léteznek és a minket is körülvevő világ az ő játszóterük is. Ha valaki az utóbbi időkben történt komoly biztonsági incidensekről szeretne tájékozódni, a következő javasolt keresőszavak alapján szemezgethet: Sony, EMC/RSA Inc., Lockheed-Martin, L3 Communications, Anonymous vs HBGary, Stuxnet, GhostNet, Operation Aurora. Hazai példaként említhetjük az Ozeki nevű magyar fejlesztőcég informatikai „kirámolását”. A felsorolt példák részletezésétől eltekintenék, mert kiváló és kevésbé kiváló cikkek tucatjai születtek már róluk a külföldi és a hazai sajtóban egyaránt. A statisztikákat kedvelők számára érdekes olvasmány lehet a témában a Verizon által évente publikált „Data Breach Investigations Report”.

Ennyi bemelegítés után térjünk a lényegre, vagyis mit kell tudni az alkalmazások biztonságáról. Ebben a cikkben az alkalmazás szintű betörési tesztekkel kapcsolatos saját tapasztalatok alapján szeretnék pár gondolatot megosztani. Az információs infrastruktúra (hálózati eszközök, szerver operációs rendszerek, alkalmazás szerver platformok, adatbázis kezelők) biztonsági tesztelését ez a cikk nem tárgyalja. Hogy éppen black-box vagy white-box megközelítést alkalmazunk azt természetesen előre egyeztetjük az ügyféllel, aszerint milyen alkalmazást vizsgálunk. Az interneten elérhető publikus, vagy zárt hálózatban elérhető belső alkalmazások esetében más-más fenyegetettségekre és támadókra (angolul threat agent) kell felkészülni.

A több felhasználós alkalmazások eléréséhez manapság klienseket szokás használni, ami lehet vékony kliens (tipikusan web böngésző) vagy vastag kliens (Java, .NET, vagy esetleg natív kódú futtatható program). A szerver oldalon állhat közvetlenül egy adatbázis-kezelő (ekkor beszélünk kétrétegű alkalmazás architektúráról) vagy egy alkalmazás-szerver és mögötte az adatbázis-kezelő (ezt nevezik 3 vagy többrétegű architektúrának). A mobiltelefonon futó (nem a mobilos web browserekre optimalizált, hanem a klienset is magában foglaló) alkalmazások logikailag a kétrétegű architektúra alá sorolhatóak. Biztonsági szempontból hatalmas különbség van a két megközelítés között. A kétrétegű alkalmazások a gyakorlatban biztonsági szempontból eleve problémás helyzetből indulnak. Az alkalmazásokat a felhasználók általában különböző jogosultságokkal érik el. Csak a példa kedvéért léteznek rögzítő felhasználók, akik tranzakciókat rögzíthetnek (pl. pénzátutalás, részvény vétel/eladás) és jóváhagyó felhasználók, akik

az alájuk rendelt felhasználók tranzakcióit jóváhagyják (a pénz ténylegesen elutalódik a forrás számláról a célszámlára). Léteznie kell még legalább egy felhasználónak, aki a többi felhasználót létrehozza/törli és a jogosultságaikat állítgatja. A helyzet tovább bonyolódik: vegyük hozzá az előbbiekhez, hogy egyes felhasználók csak egyes ügyfelek/vállalatok adataihoz férhetnek és kizárólag a hozzájuk tartozó számlák/értékpapírok felett rendelkezhetnek. Ebből már látható hogy a jogosultságok kezelésére (autorizáció) funkció szinten (melyik felhasználó mely műveleteket hajthatja végre) és adat szinten (mely felhasználó mely ügyfelek adatait olvashatja/módosíthatja) egyaránt oda kell figyelni.

Az alkalmazások biztonsági tesztelésénél alkalmazható a statikus megközelítése, itt alapvetően az alkalmazás forráskódjából kiindulva próbáljuk a hibákat megtalálni. Ez történhet automatizált eszközökkel, melyek a tipikus programozási hibákat tudják beazonosítani. Így kiszűrhető például az SQL utasítások összerakása felhasználói bemenetről származó sztringeket használó kódrészlet, ami SQL injekció sebezhetőségekhez vezethet. Szintén kiszűrhetőek így a nem biztonságos függvény/metódushívások a C vagy C++ programok forráskódjából, melyek integer/buffer overflow jellegű sebezhetőségeket eredményeznek. Ilyen automatizált eszköz nagyon sok létezik akár ingyenesen, akár megvásárolható termékként. Az egyik gyakorlati probléma a használatukkal kapcsolatban, hogy rengeteg figyelmeztetést generálhatnak. Ezeken egy hozzáértő szakembernek egyesével végig kell menni és a nagy számú hamis riasztásból kiszűrni a lényegeseket. Léteznek olyan programozási hibák, amiket a jelenlegi statikus kódellenőrző eszközök nem képesek kiszűrni (logikai hibák, információszivárgás, race-condition, back-door, magic number jellegű konstansok használata, privilégiumszintek nem megfelelő használata, kriptográfia nem megfelelő alkalmazása). A forráskódot emberi erővel átnézni biztonsági hibákat keresve drága, időigényes és ritkán alkalmazott módszer (jellemzően több hetes speciális szakértelmet igénylő munka az alkalmazás összetettségétől is függően).

A dinamikus megközelítés lényege, hogy az alkalmazást működés közben vizsgáljuk biztonsági hibákat keresve. Szintén léteznek automatizált eszközök (angolul: application security/vulnerability scanner) webes alkalmazások biztonsági tesztelésére, hatékonyságukról és összehasonlításukról az interneten részletes elemzéseket találhatunk. A gyakorlati tapasztalatok alapján elmondható róluk: a leggyakoribb biztonsági hibák (Cross-Site Scripting, SQL Injection, Directory Traversal, Forceful Browsing) jelentős részét különböző hatékonysággal képesek megtalálni, jellemzően számos hamis riasztást generálva. Megjegyezzük, hogy az összes „alap” biztonsági hibához léteznek olyan körmönfontabb esetek, mikor a humán intelligenciával ellentétben egy automata nem képes a problémát detektálni. A fenti hibák részletes leírása meghaladja jelen cikk kereteit, az érdeklődők hasznos információt találhatnak az owasp.org oldalon, az OWASP (Open Web Application Security Project) honlapján. Érdekes olvasmányok lehetnek még leggyakoribb biztonsági hibákat taglaló elemzések, mint például az évente publikált „OWASP Top 10” eredményhirdetése. Végül még mindig maradnak olyan biztonsági hibák, melyeket a legfejlettebb automatikus sérülékenység ellenőrző eszközök sem tudnak megtalálni: jogosultságkezelési rendszer (authorization system) kikerülése, session kezelés hibái, több lépésből álló folyamatokban nem azonnal jelentkező hibák. Ez utóbbit pár példával talán érthetőbbé tudjuk tenni. Például egy pénzügyi tranzakciókat is lehetővé tevő rendszerben egy pénz/értékpapír ügylet feladása rendszerint többlépcsős, általában külön jóváhagyási lépést is tartalmazó folyamat. Ha például a folyamat sokadik lépésénél (értsd: sokadik kitöltendő HTML form a webes alkalmazásban) egy Man in The Middle Proxy (pl. Paros) alkalmazásával a HTTP kérést elfogva (a böngésző és a szerver közé ékelődve) kicseréljük a cél és a forrás számlaszámokat, vajon az alkalmazás szerver része észreveszi-e a turpisságot? Ha nem, jogosulatlanul tudunk utalni más számlájáról a sajátunkra, vagy más nevében tudunk értékpapírokat venni/eladni. Az elküldött tranzakciók részletezésénél, a lekérdező (hovatovább a jóváhagyó) funkciónak küldött HTTP GET/POST kérésben a sajátunktól eltérő tranzakció azonosítót elküldve az alkalmazás megmutatja/jóváhagyja-e a máshoz tartozó tranzakciót? Ha nincsenek ilyen jogosultság ellenőrzések az alkalmazásba építve, szabadon turkálhatunk mások pénzügyeiben. Gyakorlati tapasztalataimra hivatkozva kijelenthetem, az évek során eddig tesztelt rendszerek majd mindegyikében sikerült hasonló hibákat azonosítani. Jellemző tévképzet a fejlesztők fejében, hogy ha egy menüpontot nem rajzolunk ki a felhasználó elé rakott GUI-ra, vagy egy adatot nem listázunk ki, mert ahhoz a felhasználónak egyébként nem lenne jogosultsága akkor az adott dolgot biztonságilag jól megvédtük. Ez sajnos nem igaz: az elrejtett funkció URL-jére a megfelelő kérést ettől függetlenül elküldhetjük a szerver felé, ha az nem ellenőriz jogosultságot és az „elrejtett” adatot az alkalmazás memóriaterületéről közvetlenül kiolvashatjuk, vagy ott tetszés szerint módosíthatjuk. Az alkalmazandó ökölszabályok: minden alkalmazás logikához tartozó döntést a szerver oldalon kell meghozni; az alkalmazás működése során a klienstől származó minden adata eredete megbízhatatlannak tekintendő.

A problémák egyébként már a bejelentkezésnél elkezdődnek. Kétrétegű alkalmazásoknál tipikus, hogy a kliens egy darab adatbázis szintű felhasználó és jelszó birtokában azonosítja magát az adatbázis-kezelő felé. Az alkalmazás felhasználó ezt a jelszót normál esetben nem látja, ő az adatbázisban tárolt (jó esetben titkosított) alkalmazás jelszavát adja meg bejelentkezéskor. Az alkalmazás felhasználó az alkalmazáshoz tartozó összes funkciót és adatot elérheti, attól függetlenül ki ül a klienst futtató számítógép előtt. Az alkalmazás szintű azonosítás (autentikáció) egy kliensoldalon lejátszódó ellenőrzés és döntés eredménye. Minden kliens oldalon történő döntés és onnét származó adat szigorúan megbízhatatlannak tekinthető. A kliensből kinyerhető az adatbázis felhasználó jelszava, az azonosítást megvalósító alacsony szintű utasítások debuggerrel megtalálhatóak és működésük módosítható. Ha az alkalmazás rossz kezekbe kerül, a támadó bármit megtehet, amire az alkalmazás felhasználónak van jogosultsága. A gyakorlati tapasztalatok szerint ezt pár napos munka után a gyakorlatban is demonstrálható az egyébként először jellemzően kétkedő ügyfelek számára.

Rövid ismertető következik a biztonsági teszteléshez leggyakrabban használt eszközökről:

**MITM (Man in The Middle) proxy:** A kliens és szerver közé beékelődésre szolgáló program, jellemzően lehetővé teszi az éppen elfogott kérés/válasz módosítását és naplózását továbbküldés előtt. A beékelődést általában a hálózati kommunikáció, vagy rendszerhívások eltérítésével érik el. A fejlettebbek alkalmasak az SSL titkosítás mellett kliens oldali tanúsítványok kezelésére is (ennek nagy jelentősége lehet, de itt hely szűke miatt ezt nem részletezem).

**Debugger:** A programok futás közbeni nyomkövetésére alkalmas eszközök. A program működése közben elemezhetjük és akár módosíthatjuk annak viselkedését a megfelelő memóriaterületeken található adatok vagy alacsony szintű utasítások módosításával. A natív kódú alkalmazásokhoz, Javás alkalmazásokhoz és a .NET platformra írt alkalmazásokhoz egyaránt találhatunk jól használható megoldásokat. A legfejlettebben képesek arra, hogy scriptek alkalmazásával automatikusan és futásidőben módosítják egy program működését: például egy töréspont elérésekor adott feltételek teljesülése esetén adatokat vagy utasításokat „ütnek át” közvetlenül a számítógép memóriájában. A lehetőségek szinte határtalanok, könyvespolcokat lehetne megtölteni elemzésükkel.

**Rendszermonitorozó eszközök:** Egy futó alkalmazás működését valós időben monitorozzák. Például rögzítik a kiválasztott operációs rendszer processz hálózati kommunikációra, fájlműveletekre, Windows registry írás/olvasás műveletekre vagy szálak indítására/leállítására irányuló aktivitását. Alkalmazásukkal hasznos információkat nyerhetünk egy ismeretlen forráskódú és komplex alkalmazásról annak működésével kapcsolatban.

**Fuzzer:** Egy alkalmazás adott funkcióinak (hálózati szolgáltatás, interfész metódushívások, adatfájl betöltés) hibáit nagy mennyiségű különböző véletlenszerű mintákat tartalmazó bemenet átadásával és az alkalmazás működésének figyelésével próbálja felderíteni. A használható eredményt érdekében érdemes a használt formátum (leírómezők, hosszmezők) részleteit ismerni és ennek megfelelően értelmes, vagy szándékosan elrontott bemenetekkel próbálkozni. Alkalmas adatbeolvasást és saját protokollra jellemző üzenetformátumok feldolgozását végző kódrészletek tesztelésére. A témának külön szakirodalma van, például natív kódú alkalmazásokban integer/buffer overflow típusú hibák felfedezésére szokták bevetni jól használható eredményeket produkálva. Az olyan komplex, de széles körben elterjedt alkalmazások, mint a Microsoft Outlook, irodai alkalmazások (pl. Word, Excel, Adobe Reader) és Web böngészők biztonsági hibáinak jelentős részét valószínűleg ezzel a módszerrel találják napjainkban.

Az alábbiakban következzen pár a trivialitásokon túlmutató hiba bemutatása, amiket tapasztalataim szerint az egyébként jól képzett, jó munkát végző, fejlett keretrendszereket alkalmazó és a biztonságra odafigyelő fejlesztők is rendszeresen elkövetnek.

**Megfelelő bemeneti szűrés hiánya:** Vegyük a következő helyzetet. Az adott alkalmazás más alkalmazásokkal is kommunikál a már régóta használt alkalmazáshoz később „hozzábarkácsolt” interfészeken keresztül. Az interfészeken utazó adatok formátuma nem teljesen azonos módon értelmezett a két oldalon. Az egyik alkalmazás adatot tölt át a másikból például egy felhasználótól származó bemenet által meghatározott szűrés/keresési feltétel alapján. Az egyik oldalon sincs megfelelő bemenetszűrés megvalósítva, attól félve nehogy a kommunikációba belenyúlva működési hiba keletkezhessen vagy azt gondolva: biztosan megoldották már ezt a másik oldalon. A félreértés eredményeként egyes mezőkben olyan speciális karaktereket tudunk bevinni, melyek végül közvetlenül megváltoztatják a háttérben lefutó SQL kifejezés működését és egy támadó saját SQL parancsot futtathat le a háttérben.

**Session kezelés hibái:** Az adott vastag klienst alkalmazó alkalmazás az éppen belépett felhasználókat egy egyedi és véletlenszerű érték (session cookie) segítségével azonosítja. A kliens a bejelentkezés során a szervertől kapott válaszban található értékek alapján jeleníti meg a felhasználó jogosultságai alapján a számára elérhető funkciók menüpontjait. Képzeljük el, hogy ezt a választ a bejelentkezéskor azelőtt módosítjuk, hogy a kliens megkapta volna, például a felhasználó szerepkörét magasabb jogosultságokat biztosító szerepkörre módosítjuk, vagy amúgy nem elérhető menüpontokat bekapcsolunk. Egy támadó így alacsony jogosultságú felhasználó birtokában magas jogosultságú funkciókat érhet el (pl. felhasználók adminisztrációja, bizalmas üzleti adatok elérése).

**Jogosultság ellenőrzési (autorizációs) hiba:** Képzeljünk el egy webes alkalmazást, ami ügyfelek (pl. hitelesek, biztosítottak) kezelésére szolgál. A működés során az ügymenethez hozzátartozik, hogy az adott igény elbírálásához igazolásokat kell tárolni (pl. beszkennelt hivatalos iratok) és az igény sikeres elbírálása esetén szerződés keletkezik (pl. hitelről, biztosításról). A tárolt dokumentumokat mondjuk állományként a fájlrendszerben vagy egy adatbázis táblában BLOB típusú értékként tárolja az alkalmazás. Az adott tárolt dokumentumot egy numerikus értékkel azonosítja az alkalmazás és mindenhol ez alapján hivatkozik rá. Jellemzően ez a numerikus érték új dokumentum keletkezésekor pontosan egyel nő az eddig utolsóként tárolt azonosítójához képest. A dokumentum letöltésekor/megnézésekor elküldött kérésben az azonosítót átírva például egyel kisebbre máris hozzáférhetünk a más felhasználóhoz tartozó dokumentumhoz, mert az alkalmazás nem ellenőrzi hogy a kérő felhasználónak tényleg lenne-e joga a kért dokumentum megtekintéséhez.

Konklúzióként elmondhatjuk, hogy az alkalmazások biztonságossá tételét nem bízhatjuk kizárólag a fejlesztőkre. A megfelelő biztonsági teszteléshez sajátos gondolkodásmód, speciális szaktudás és tapasztalat szükséges. A megfelelő képességek megszerzése és fenntartása teljes embert kívánó feladat, ezért a legtöbb cég házon belül nem képes megfelelő kompetencia létrehozására és megtartására. A megfelelő belső erőforrásokkal rendelkező legnagyobb fejlesztőcégek (pl. Google, Microsoft) is pénzbeli jutalommal honorálják a külsős biztonsági tesztelőket. Ha valaki egy módszertan jellegű áttekintést szeretne kapni a témáról, javasolt az Open Source Testing Methodology (OSSTM) és az OWASP Testing Guide tanulmányozása. A biztonságra törekvést és biztonsági tesztelést komolyabb fejlesztéseknél javasolt már a projekt korai fázisaiba beépíteni, erre specializálódott szakértőket igénybe véve. Ezzel elkerülhető az az effektus, amikor egy új alkalmazás verzió élesbe állítása a biztonsági hibák kijavítása miatt csúszik.

### *Major Marcell*

Marcell jelenleg a Deloitte IT-biztonsági csapatának tagja. Több mint 5 éves tapasztalattal rendelkezik betörési tesztek és egyéb informatikai biztonsági vizsgálatok terén. Tanulmányait a szoftverfejlesztésre és az informatikai biztonságra fókuszálva végezte. Tapasztalatokat szerzett az alkalmazások biztonsági tesztelése, a reverse engineering, a kriptográfiai algoritmusok és protokollok megvalósítása terén. Az előadók névsorában rendszeresen megtalálhatjuk hazai és külföldi hacker konferenciákon.


# TestLink

**Írásomban egy olyan tesztelést támogató eszköz (TestLink) használatának ismertetését fogom bemutatni - amellyel több éve dolgozok kisebb nagyobb projekteken. A folyamatos fejlesztés egyre szélesebb teret enged a TestLink eszköz használatának, ugyanakkor állandó szakmai kihívások elé állítja azokat, akik módszertani szemléletet visznek bele a tesztelési folyamatok kidolgozásába.**

Napjainkban egyre több tesztelést támogató eszköz jelenik meg az IT piacon, gyártóik különböző marketing fogásokkal ösztönzik az ügyfeleket, szakembereket az ingyenes kipróbálásra, miközben kivétel nélkül azt ígérik, hogy hatékonyabbá válik alkalmazásukkal a tesztelés. Jogos a kérdés; használjuk-e eszközöket a tesztelés támogatására, ha igen melyiket válasszuk? Hogyan ágyazzuk be fejlesztési folyamatainkba, mikorra fog ez a befektetés megtérülni?

A kilencvenes évek elején a Microsoftnak dolgoztuk itthon és külföldön egyaránt - a tesztelést már akkor eszköztámogatással és módszertan alapján kezdtem elsajátítani. Ez indított el a tesztelés – mint szakma - irányába, mert már akkor megtapasztaltam, hogy van hatékonysága az eszközök használatának, ha biztosítottak a megfelelő körülmények és tesztelési folyamatok.

## Rövid áttekintés

A TestLink egy web alapú nyílt forráskódú Teszt Menedzsment rendszer, amely Apache 1.3.x vagy 2.x web szerveren fut MySQL 4.1.x vagy 5.x adatbázissal Linux és Windows operációs rendszereken. Az eszköz használata Firefox 3+; IE7+; Opera 7+ böngészőkre optimalizált.
A TestLink 2003-ban indult útjának és folyamatosan fejlesztették, a jelenlegi 1.9.2 verzió számos hibajavítást és új funkciót tartalmaz, amely kényelmesebbé, és hatékonyabbá teszi a felhasználók munkáját.
A rendszer telepítése és konfigurálása, némi rendszergazdai tapasztalatot azért igényel, azonban a vállalkozó kedvűeknek a telepítés & konfigurálás útmutatója részletesen megtalálható a csomaghoz mellékelt telepítési és konfigurációs leírásban. Szintén itt útmutatást találunk a funkciók értékkészleteinek beállítására vonatkozóan és válaszokat olyan gyakran feltett kérdésre, mint például a verziófrissítés vagy ékezetes karakterek kezelése.
Bár több nyelvre lefordították magyarra lokalizált változata még nincs, azonban ez folyamatban van, ha elkészül nagyobb lehetőség nyílik arra, hogy a rendszert magyar nyelvű projekteken is használni lehessen.

Tapasztalataim alapján a rendszer megbízhatóan működik – megbirkózik a nagyszámú felhasználó kezeléssel is – így nagyobb projektekre vagy több projekt egyidejű kezelésére is bátran ajánlható.

## A TestLink felépítése

### A TestLink rendszernek 3 sarokköve van:

- Teszt Projekt
- Teszt Terv
- Felhasználó

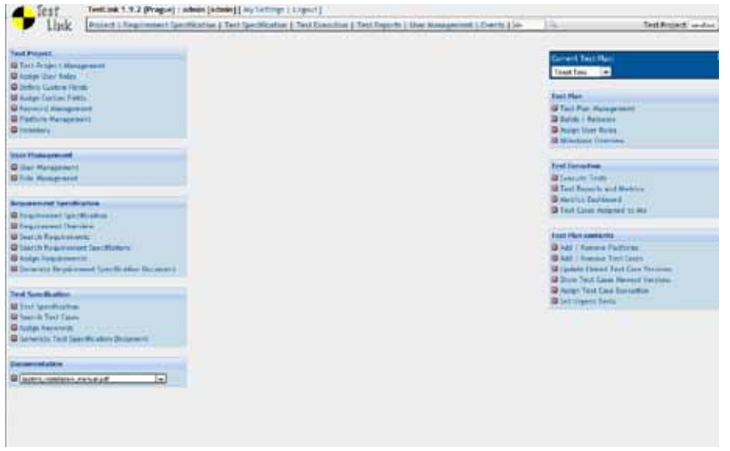
A többi funkció és adat relációja vagy tulajdonsága e három sarokkőnek. A működés könnyebb megértéséhez szükséges tisztában lenni az alapvető tesztelési terminológiával. A csomag tartalmaz egy rövid angol nyelvű terminológiát, ezt érdemes elolvasni azoknak, akik nem járatosak a tesztelésben, tapasztaltaknak nem sok újdonságot nyújt.

A belépés után a rendszer a Project oldalra navigál, ahol jogosultságtól függően megjelennek az elérhető funkciók. Admin jogosultsággal végezhető a tesztelési folyamatok kialakítása és testre szabása a projekt jellegétől függően.

A funkciók 7 nagy csoportba vannak szervezve ezek az alábbiak:







1. ábra

- Test Project – Tesztelési projektek létrehozása és kezelése
- User Management – jogosultságok, szerepkörök beállítása
- Requirement Specification – Követelmény kezelés
- Test Specification – Teszt tervezés
- Documents -
- Test Plan – Teszt forgatókönyvek létrehozása összerendelése
- Test Execution – Teszt futtatások kezelése
- Test Plan Contents – Tesztforgatókönyvek összeállítása és menedzselése
- Platform Management - platformokat hozhat létre a projekt jellegétől függően
- Inventory – Host eléréseket definiálhat
- Test Project Management funkció segítségével - létrehozza a tesztprojektet
- Létrehozza a Teszt Tervet a hozzátartozó build-el – Builds / Releases funkcióval
- Kialakítja a Követelmény Specifikáció struktúráját – Requirement Specification funkcióval
- Importálja a követelményeket a kialakított struktúrába

A vízszintes menüsorból egy ötletes megoldással a főbb funkciók gyorsabb elérése lehetséges. (1. ábra)

## Tesztelési folyamatok kialakítása

TestLink-ben a tesztelési folyamat kialakítása az alábbi főbb lépésekből áll, jelen esetben feltételezzük, hogy a tesztcsapat rendelkezik a felsorolt hat szerepkörrel. A TestLink-ben ezek: guest; tester; senior tester; test designer; leader; admin illetve még beállítható két speciális szerepkör is.

Az Adminisztrátor a User Management segítségével létrehozza a felhasználókat

Ezek után a Vezető tesztelő kialakítja az alábbi folyamatlépéseket:

- Assign User Roles funkcióval - beállítja a Teszt Projekt szerepköröket
- Define Custom Fields - egyedi mezőket definiálhat
- Assign Custom Fields - egyedi mezőket célzottan szignálhatja
- Keyword Management - Kulcsszavak definiálhat a könnyebb kereshetőség érdekében

Ritkább esetekben van a tesztcsapatban külön teszt tervező, a tesztprojektekre inkább az a jellemző, hogy egy személy több szerepkörben dolgozik, ami nem igazán szerencsés.

- A Teszt tervező a Test Specification funkció segítségével kialakítja a tesztesetek struktúráját és definiálja a teszteseteket
- A Vezető Tesztelő az Assign Requirements funkció segítségével összerendeli a követelményeket a tesztesetekkel, a Test Plan Content funkciók segítségével kialakítja tesztforgatókönyvet, kiosztja a teszteseteket.
- A tesztelő az Execute Tests funkcióval végrehajtja a kapott teszteseteket és dokumentálja az eredményeket.
- A Vezető tesztelő a TestReports and Metrics funkcióval összesíti a tesztfuttatások eredményeit, jegyzőkönyveket és riportokat készít.

A következő részben nem célom a funkciók további részletes bemutatása, inkább gyakorlati és szakmai szempontból közelítem meg a TestLink használatát.

## Teszt Projektek kezelése

Egy teszt projekt definiálását rövid idő alatt el lehet végezni, a rendszer lehetőséget biztosít egyszerre több projekt párhuzamos aktiválásához és használatához. A projektek közötti átjárhatóságot a Role Management / Assign Test Project Role biztosítja ahol beállíthatjuk a Tesztprojektben résztvevők szerepköreit. A beállított jogosultságnak megfelelően egy felhasználó csak a beállításoknak megfelelően láthat és kezelhet teszt projektet.

Egy külön teszt projekt kialakítása lehetővé teszi a végfelhasználók részére is – például átadás - átvételi tesztek idején - hogy a kiosztott teszteseteket a TestLink-ben futtassák. Szerepköröket és jogosultságot nem csak a projekthez hanem a teszttervhez is lehet definiálni. Így tesztprojekt és teszterv szinten is célirányosan konfigurálható mi az, amit egy felhasználó elérhet a rendszerben.

## Követelmény kezelés

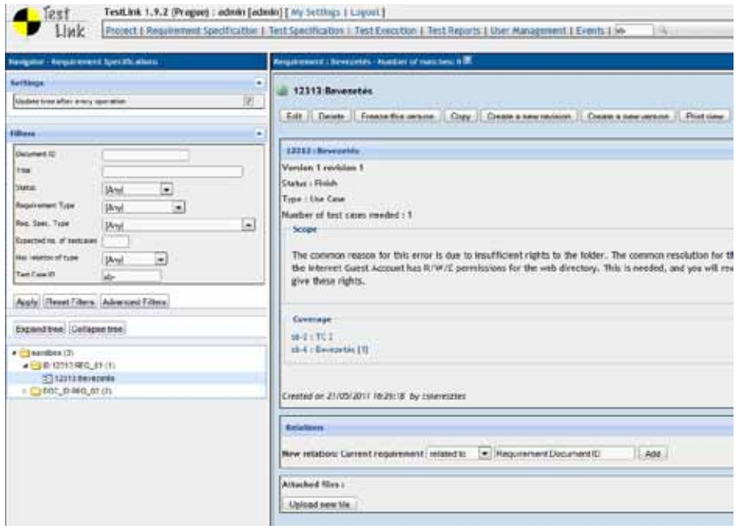
A Követelmény specifikálás részben sok mindent javítottak és újdonság, hogy van verziókezelés a követelmények szintjén is.
A tesztelés során mindig nagy kihívás, hogyan tudjuk a követelmény alapú teszttervezést megvalósítani? Lehetséges ez Excel és Word párosítással is azonban ennek a módszernek hatékonysága messze elmarad a tesztelést támogató eszközök hatékonyságától és lehetőségeitől.

További kérdés - ha szükséges a követelmény alapú tesztelés kialakítása - milyen stratégia szerint építjük fel a struktúrát és hogyan kerülnek be a követelmények a kidolgozott struktúrába? (2. ábra)

A TestLink-ben erre 3 módon van lehetőség:

1. A követelmények másolása – Ctrl+C & Ctrl+V funkciókkal – ez viszont nem túl hatékony módszer.
2. Követelmény címének és rövid leírásának létrehozása majd a követelmény leírásának csatolása (Word, Excel, XML) a kialakított struktúrába.
3. Követelmények importálása – a rendszer csak XML formátumot enged azonban TestLink telepítő csomagban létezik egy olyan előre definiált sablon - RequirementXLSTOXML.xls - amelybe strukturáltan feltölthető a teljes követelmény leírás.

Az import után összerendelhetőek a követelmények vonatkozásai szülő-gyerek kapcsolat, összefüggések. Ezután beállíthatjuk a követelmények státuszát – alapesetben ez a Normal és Not testable



2. ábra

- további státuszok - (Draft, Review, Rework, Finish stb.) - és típusát (Use Case, Feature, UI, Non functional stb.) valamint azt is, hogy egy adott követelményhez hány tesztesetet kell tervezni. Ez a szám adja majd a követelmények és a tesztesetek összerendelése után a lefedettség mutatót. A lefedettség mutatót befolyásolja a Not testable státuszú követelményeket a rendszer nem számolja bele a mutatószámokba. (2. ábra)

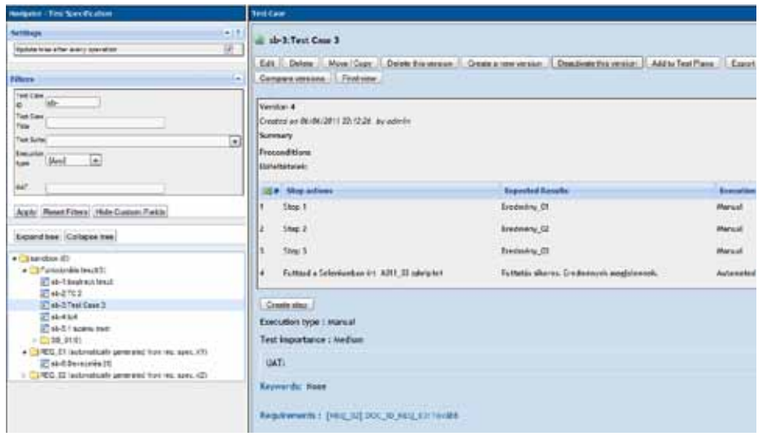
A Navigátor részen szűrési feltételek könnyítik a célzott információ gyors elérését.

Egy követelmény mappa kiválasztása után jobboldalon megjelennek a mappához tartozó követelmények ezekből automatikusan teszteseteket generálhatunk – ezek követelmény specifikációhoz hasonló struktúrában jönnek létre a követelmény neve (Title) és a Scope mező tartalmát másolja át a rendszer a tesztesetbe.

Az importált Követelmény specifikációból riportot is készíthetünk, előtte beállíthatjuk azokat az opciókat, amiket látni szeretnék a generált HTML, Word, és OpenOffice Writer formátumú riportokban. A Word dokumentum generálása Firefox-ban ajánlott mivel itt működik igazán.

## Teszt tervezés

A teszt tervezést segíti és egyszerűsíti az automatikus teszteset generálás, amennyiben jól felépített a követelmény kezelés úgy sok szükséges információ átkerül a tesztesetbe, könnyítve a tesztlépések kidolgozását. Az tesztlépések kidolgozását egy szerkesztő felületen keresztül végezhetjük. Ez egy FCKEditor-nak nevezett szerkesztő, amely a konfigurációs fájlokban leírt útmutatás segítségével egyedileg beállítható. A Szerkesztőt tovább fejlesztették így lehetőség van tesztlépések létrehozására és mentésére, a rendszer minden tesztlépéshez egyedi számot rendel. (3. ábra)



3. ábra

A régebbi verziók egy mezőben kezelték az összes lépést ezért szükséges volt a lépések és az elvárt eredmények számozása az egyértelműség miatt. További új lehetőség, hogy a teszteset egy vagy több teszttervhez is hozzárendelhető egyidejűleg, illetve a tesztesetnek új verziója is létrehozható.

A teszttervezést tovább segíti, hogy Excel és XML formátumban is importálhatjuk a teszteseteket. A tervezéshez itt is található egy sablon fájl - TestCaseXLSTOXML.xls – amelyben a struktúra is kialakítható. A teszteseteket a Copy és Move funkciókkal könnyen másolhatjuk mozgathatjuk. A tesztesetek halmazából a Generate Test Specification funkcióval teszt specifikációt generálhatunk opcionális beállításokkal attól függően, hogy milyen tartalmat akarunk látni a dokumentumban.

Olyan esetekben amikor a teszteseteket szeretnénk egy meglévő projektből – új projektbe másolni – az Export funkciót használhatjuk amely XML formátumban generálja a kijelölt teszteset halmazt. Az új projektbe az Import funkcióval tudjuk azonos struktúrában átmásolni a szükséges teszteseteket.

Itt is a Navigátor részen szűrési feltételek könnyítik a célzott információ gyors elérését.
(3. ábra)

## Teszt futtatás

A tesztfuttatás előfeltétele, a tesztesetek kiosztása továbbá a megfelelő verzió létrehozása. Execute Tests módban nincs lehetőség a tesztesetek módosítására. Alapértelmezettként 4 féle eredmény van konfigurálva, ezt azonban testre szabhatjuk vagy bővíthetjük a tesztelési folyamatainknak megfelelően. Alapesetben a tesztestesek státusza Not run állapotban van – státusz váltás esetén nincs mód a Not run állapot visszaállítására. A futtatás során talált észrevételeket beírhatjuk a Notes mezőbe, státusz váltás és mentés után azonban eltűnik a mezőből. A Notes leugró mezőben azonban megtekinthető.

**Megjegyzés:** Ha hozzá akarunk adni további információt az észrevételekhez, státusz váltás és mentés esetén felülírja az eredeti bejegyzésünket!
(4. ábra)





## Riportok készítése

A rendszer többféle riportot képes előállítani, előtte beállíthatjuk azokat az opciókat, amelyeket látni szeretnék a generált HTML, Word, és OpenOffice Writer formátumú riportokban. A Word dokumentum generálása itt is Firefox-ban ajánlott. A következő riportok széles skálája generálható ez egészen kellemes egy nyílt forráskódú ingyenesen elérhető rendszernél.

- Test Report – tartalmazza a tesztek futtatásának eredményét
- General Test Plan Metrics – táblázatos formában összesíti a tesztfuttatások eredményeit
- Results by Tester per Build – személyenként megjeleníthető a tesztfuttatások eredménye
- Test Case Assignment Overview – tesztesetek kiosztásának áttekintése
- Query Metrics – van lehetőség feltételek szerinti metrikák lekérdezésére
- Test result matrix – az utolsó tesztfuttatások eredményeit mutatja meg verziókra
- Failed Test Cases – hibára futott tesztesetek listája
- Blocked Test Cases – blokkolt tesztesetek listája
- Not run Test Cases – nem futtatott tesztesetek listája
- Test Cases without Tester Assignment – nem szignált tesztesetek megjelenítése
- Charts – Grafikonok generálása a teszteredmények alapján
- Requirements based Report – Követelmény alapú riport információt ad a teszteltségi állapotukról
- Bugs per Test Case – tesztesethez kapcsolt hiba lista megjelenítése – integrált hibakezelő eszköz esetén
- Test Cases not assigned to Any Test Plan – Teszttervhez nem rendelt tesztesetek listázása

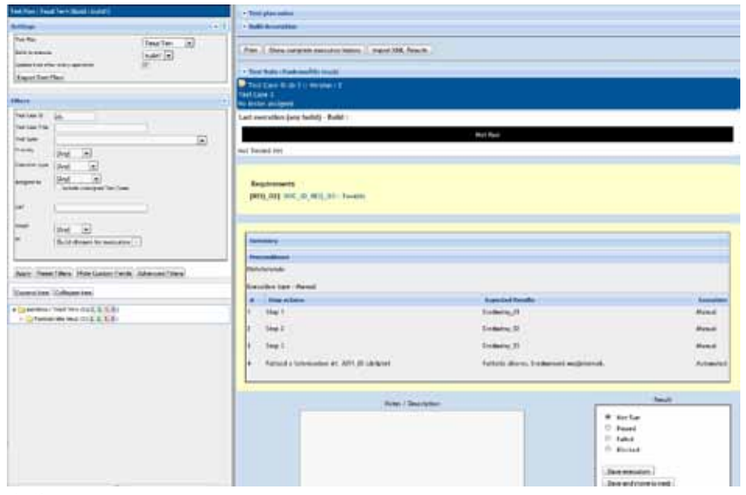
## A TestLlink és a hibakezelő eszközök kapcsolata

A rendszer további előnyére szolgál, hogy több ingyenes és fizetős hibakezelő rendszerrel integrálható. Ilyenek például: Bugzilla, Fogbugz, Jira, Mantis, Trac stb. további részletes információ a csomag - lib\bugtracking könyvtárában található.

A hibakezelés menete: a tesztelés során talált hibát rögzítjük az adott hibakezelő rendszerbe, ezután Execute Tests módban a Bug Management alatti ikonra kattintva megjelenik egy felugró ablak, ahol megadhatjuk a hiba egyedi azonosítóját. Mentés után a rendszer összekapcsolja a hibára futott tesztesetet a hibabejegyzéssel így a hibabejegyzés a TestLinkből is megnyitható a Bug Management alatti ikonra kattintással. Hibára futott tesztesetek listájának generálása esetén a hibákra utaló linkek a riportban is megjelennek és működnek.

## Összegzés

A TestLink egy olyan tesztelést támogató eszköz, amely megfelelő szakmai tudás mellett kiválóan alkalmazható nagyobb és akár komplexebb tesztelési projektek kivitelezéséhez. Írásomban nem volt cél a TestLink - fizetős vagy nem fizetős tesztelést támogató rendszerekkel való összehasonlítása ezért a használhatóság oldaláról igyekeztem a rendszert bemutatni. A hatékony eszközhasználathoz nem elégséges csupán azt megvizsgálni, hogy egy eszköz mennyire felel meg a célnak, hiszen a piacvezető fizetős eszközök esetében - a rosszul megtervezett folyamatok kezelése időigényes így csökkenti a hatékonyságot a minőség rovására. Az eszköz kiválasztás során egyéb tényezőket is szükséges figyelembe venni, például nagy szerepe van a szakértelemnek és tekintettel kell lenni a projekt tényezőkre is. Az ügyfél elégedettség eléréséhez azonban nem csak az eredményesség szükséges, hanem az eredményhez vezető folyamatok minőségének biztosítása is, amely összhangban van az ügyfelek azon célkitűzéseivel, amelyek érdekében az IT megoldásaikat fejlesztik.



4. ábra

**Keresztes Csaba**

Tesztelői pályafutását 1994-ben a Microsoftnál a Windows termékek lokalizációs projektjein kezdte.
Tesztelés vezetőként dolgozott a GE HealthCare, a MÁV Informatika, a Vodafone, a Lufthansa Systems vállalatoknál valamint számos állami-, banki- és telekommunikációs projektben. 2006-tól tanácsadóként és tesztmenedzserként tevékenykedik. Főbb feladatai a módszertani tervezés és bevezetések mellett tesztfolyamat menedzsment, tesztcsapat építése - szakmai és operatív koordinálása.

## Szerzői jogok



## Impresszum

Kiadja az Acrobyte Informatikai Kft.

**Szerkesztőség:**
Alapító főszerkesztő:
Pongrácz János
info@tesztelesagyakorlatban.hu

**Design, grafika:**
Vadon Zsolt
www.vadondesign.hu
info@vadondesign.hu

**Internet:**
www.tesztelesagyakorlatban.hu

**Kapcsolat:**
1149 Budapest, Mexikói út 11/a
Telefon/Fax: 1 / 210 – 8424
info@tesztelesagyakorlatban.hu

**Kérdések:**
info@tesztelesagyakorlatban.hu

**Publikáció:**
Pongrácz János
publikacio@tesztelesagyakorlatban.hu

**Hirdetésfelvétel:**
Oros Attila
hirdetes@tesztelesagyakorlatban.hu

Szakmai ismeretek

Gyakorlati tapasztalatok

Új trendek, módszerek

Kizárólag szoftvertesztelésről szóló cikkek

Negyedévente olvashatod