SZÉCHENYI 2020

SZOFTVERTESZTELÉS A GYAKORLATBAN

Boda Béla (4., 5., 6. fejezet)

Bodrogközi László (3. fejezet)

Gál Zoltán (7., 8. fejezet)

Illés Péter (9., 10. fejezet)

Készült: 2014

Terjedelem: 8.4 ív

Kézirat lezárva: 2014. augusztus 31.

A tananyag elkészítését a Munkaerő-piaci igényeknek megfelelő, gyakorlatorientált képzések, szolgáltatások a Debreceni Egyetemen Élelmiszeripar, Gépészet, Informatika, Turisztika és Vendéglátás területen (Munkaalapú tudás a Debreceni Egyetem oktatásában) **TÁMOP-4.1.1.F-13/1-2013-0004** *számú projekt támogatta. A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg.*

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

SZÉCHENYI 2020

"Things should be made as simple as possible, but no simpler"

Albert Einstein

"Nature wasn’t designed but debugged into perfection"

A Recognita fejlesztőinek jelmondata

SZÉCHENYI 2020

TARTALOMJEGYZÉK

Előszó **8**

Bevezetés **9**

Tesztelési és tesztelői alapfogalmak **10**

A TESZTELŐ *10*

A szoftver kapcsolatai a fejlesztési folyamat során *10*

Szoftverek a világban *11*
- Üzleti szoftverek 11
- Vezérlő és rendszer szoftverek 12
- Játékszoftverek 13

A meghibásodás okai *14*
- Hardver hiba 14
- Hálózati hiba 15
- Konfigurációs hiba 16
- Kezelői hiba 17
- Nem megfelelő környezet 17
- Adatsérülés 18
- A szoftver készítése során indukált hibák 18
  - Emberi tévedés, elírás 18
  - Ismerethiány 19
  - Segédprogramok hibái 19
  - Rohammunka és szervezési hibák 20

A szoftverfejlesztés/tesztelés körülményei *20*

Tesztelés szerepe *20*

Minőségirányítás, Minőségbiztosítás, Minőség *21*
- Minőségirányítás 21
- Minőségbiztosítás 21
- Minőség 21

Tesztelés mennyisége és a kockázat *24*
- Tesztelés mennyisége 24

Mi a tesztelés? *25*

Általános tesztelési alapelvek *28*

A tesztelés alapvető folyamata *30*
- Teszttervezés 30
- Tesztelemzés és műszaki teszttervezés 30
- Tesztirányítás 32
  - Teszt megvalósítása (test implementation) 32
  - Teszt végrehajtás 33
- A kilépési feltételek értékelése és jelentés 34
- Tesztlezárás 34

A tesztelés pszichológiája *36*
- Ki teszteljen? 36
- Függetlenség szintjei 37

SZÉCHENYI 2020

Milyen a jó tesztelő? ... 38
Kommunikációs problémák elkerülése ... 39

Etikai kódex ... 40

Szoftverfejlesztési modellek ... 41

V-Modell ... 41

Iteratív-inkrementális fejlesztési modellek ... 43

Tesztelés egy életciklus-modellen belül ... 44

Tesztszintek ... 49

Egységtesztek ... 49
Az egységtesztek igénye ... 49
Egységtesztek karakterisztikái ... 50

Integrációs teszt ... 51
Rendszerintegrációs tesztek ... 52

Függőségek emulálása ... 53
Stubbing ... 54
Mocking ... 54

Egység- vagy integrációs teszt? ... 55

Rendszerteszt ... 55
Automatizálás ... 56
Speciális funkcionális rendszertesztek ... 56
Smoke-teszt ... 56
Sanity-teszt ... 57

Átvételi teszt ... 57

Egy példa alkalmazás ... 58
Igény ... 58
Architektúra ... 60

xUnit ... 60
jUnit ... 63
jUnit Test Class & Test Method ... 63
Több tesztmetódus ... 64
jUnit Assert ... 66
jUnit életciklus ... 67
jUnit Timeout ... 73
jUnit Expected ... 74
jUnit Suite ... 75
jUnit Category ... 76
jUnit Parametrized ... 78
Szolgáltatás egységteszt ... 81
Stubbing példa ... 82
Mocking példa ... 86

Teszttípusok ... 96

Tesztszintek és teszttípusok viszonya ... 96

Funkcionális tesztek ... 97

SZÉCHENYI 2020

Alfa-teszt ... 97
Béta-teszt ... 98

Nem funkcionális tesztek ... 98
Teljesítmény teszt (Performance Testing) ... 98
Terheléses teszt (Load Testing) ... 99
Stresszteszt (Stress Testing) ... 100
Kitartási teszt (Soak / Endurance Testing) ... 100
Csúcsteszt (Spike Testing) ... 101
Mennyiségi teszt (Volume Testing) ... 101
Használhatósági teszt (Usability Testing) ... 101
Nemzetköziesség teszt (Internationalization - i18n & Localization - l10n Testing) ... 104
Biztonsági teszt (Security Testing) ... 105
Skálázhatósági teszt (Scalability Testing) ... 108
Visszaállíthatósági teszt (Recovery Testing) ... 108
Hordozhatósági teszt (Portability Testing) ... 108
Kompatibilitási teszt (Compatibility Testing) ... 110
Hatékonysági teszt (Efficiency Testing) ... 110
Megfelelési teszt (Contract Acceptance Testing) ... 110
Szabályossági teszt (Compliance Testing / Regulation Acceptance Testing) ... 110
Linearitás vizsgálat ... 111
Megbízhatósági teszt (Reliability Testing) ... 111
Regressziós teszt (Regression Testing) ... 113
Strukturális teszt (Structural Testing) ... 113
Karbantartási teszt (Maintenance Testing) ... 114
Hatáselemzés (Impact Analisys) ... 114

Tesztelési technikák ... 115

Statikus technikák ... 115
Csoportos értékelések (structured group evaluations) ... 115
Átvizsgálás (review) ... 116
A folyamat fázisai ... 117
Szerepkörök (roles and responsibilities) ... 118
Főbb típusok ... 119
Tesztelők, mint bírálók ... 120
Tanácsok a sikeres átvizsgálásokhoz ... 120
Gyakorlás ... 120

Statikus elemzés (static analysis) ... 127
Adatfolyam jelenségek ... 128
Vezérlési folyam jelenségek ... 128
Ciklomatikus komplexitás (cyclomatic complexity) ... 129
Gyakorlás ... 129

Dinamikus technikák (dynamic analysis) ... 132
Fekete doboz technikák ... 134
Ekvivalencia particionálás (equivalence partitioning) ... 134
Határérték-analízis ... 136
Állapotátmenet tesztelés (state transition testing) ... 138
Logika alapú technikák (logic based techniques) ... 140
Döntési tábla tesztelés (decision table testing) ... 140
Használati eset alapú tesztek (use-case-based testing) ... 142
Összefoglalás ... 143
Gyakorlás ... 143

SZÉCHENYI 2020

Fehér doboz technikák ... 151
Utasítás szintű tesztelés (statement testing) ... 151
Döntés alapú tesztelés (decision/branch testing) ... 152
Feltételek tesztelése (test of conditions) ... 153
Többszörös feltételek tesztelése (multiple condition testing) ... 153
Feltétel meghatározásos tesztelés (condition determination testing/minimal multiple condition testing) . 153
Bejárási út alapú tesztelés (traversal based testing) ... 153
Összefoglalás ... 154
Gyakorlás ... 160
Intuitív és tapasztalat alapú módszerek ... 166
Összefoglalás ... 167

Tesztmenedzsment ... 169

Bevezetés ... 169

Teszt menedzsment elemei ... 170
Teszt stratégia ... 170
Teszttervezés és becslés ... 174
WAG és SWAG ... 177
Súlyozott átlag (Weighted Average) ... 177
Delphi ... 178
Összehasonlító becslés (Comparative vagy Historical Estimating) ... 178
Teszt monitorozás ... 179
Konfiguráció menedzsment ... 181
Kockázatok ... 183
Incidens menedzsment ... 185

Tesztelő csapat ... 188
Tesztelő (Teszt végrehajtó) ... 190
Teszttervező ... 190
Teszt Menedzser (Teszt Vezető) ... 190

Tesztautomatizálás ... 192

Automatizált teszt ... 192

Érvek az automatizált szoftvertesztelés mellett ... 193

Miért nem egyszerű az automatizált szoftvertesztelés? ... 193

Mikor nem javasolt a tesztautomatizálás használata? ... 194

Automatizált szoftvertesztelés – Generációk ... 194

Lineáris automatizálás ... 194

Adatvezérelt automatizálás ... 196

Kulcsszó vezérelt automatizálás ... 199
Összegzés ... 201

Eszközválasztás ... 202

Eszköz tulajdonságok ... 204

Tesztautomatizálás bevezetése ... 206

A megfelelő folyamat kiépítése ... 213

Befektetés / megtérülés nyomon követése ... 214

SZÉCHENYI 2020

Összefoglalás ... *219*

Függelék ... *220*

Piacvezető tesztautomatizálási eszközök ... 220

SZÉCHENYI 2020

Előszó

A szoftvertesztelés az utóbbi időben kezd átalakulni az informatikán belül a "szükséges rossz", a "néhány hallgató majd megcsinálja", a "fejlesztőnek nem jó, no majd csinálunk belőle tesztelőt" szemléletből a minőségi szoftvert csak jó tesztelőkkel tudunk előállítani gyakorlat felé. Egyre többen vallják azt a felfogást (köztük jelen sorok írója is), hogy a tesztelés egy önálló szakma, ami más kompetenciákat követel, mint a fejlesztői kompetenciák.

A magyar felsőoktatásban ma a tesztelés egy-egy tantárgyban, esetleg egy szakirány-jellegű tantárgyi blokkban jelenik meg és korántsem általánosan.

Jelen tananyag egy tananyagsorozat első elemeként jelenik meg, egy konkrét tantárgyhoz kapcsolódóan, a nemzetközi ipari szabványokhoz igazodva.

Azonban itt és most nem egy többedik magyar nyelvű teszteléssel foglalkozó jegyzet jött létre az eddig már elérhetők mellett, hanem szemléletében egy kimondottan gyakorlatorientált, jó gyakorlatokat, esettanulmányokat, személyes tapasztalatokat felvonultató, néhol szubjektív elemeket is tartalmazó mű született meg.

A tananyagot a piacon már bizonyított, a tesztelés területen hosszú évek óta dolgozó, sok projektet végigélt (netán irányított) szakemberek állították össze és írták meg.

A szerkesztőnek végig gondot jelentett, hogy az egyedi stílusokat hogyan lehet egységessé gyúrni, vagy hogy egyáltalán kell-e ilyen irányban sokat dolgozni. Aztán maradt a középút: van ugyan egyfajta egységesítés, de minden fejezet magán viseli írójának személyiségjegyeit.

A szerzők munkájának elismerése mellett szeretnék köszönetet mondani három olyan embernek, akik végig segítettek a megfogalmazásban, anyaggyűjtésben, formálásban, ők: Kundra Zoltán, Mucsina Norbert és Balla Tibor.

Juhász István

Bevezetés

Jelen tananyag a Debreceni Egyetem Informatikai Karán a programtervező informatikus, mérnökinformatikus, gazdaságinformatikus alapszakok mindegyikén szabadon választható *Kompetens szoftvertesztelés a gyakorlatban* című tantárgyhoz készült. Elsősorban a tárgy előadásainak és gyakorlatának követésére való, de alkalmas önálló feldolgozásra is.

A tananyag alapjául az International Software Testing Qualifications Board (ISTQB), illetve annak magyar tagozata, a Hungarian Testing Board által kidolgozott nemzetközi tanúsítvány rendszer alap szintjének koncepciója és sillabusza, valamint fogalomtára szolgált. Az elmélet nagyban támaszkodik (gyakran hagyatkozik) az alapfokú tanúsítványhoz szükséges ismereteket tartalmazó könyvre, amit magyar nyelven az Alvicom jelentetett meg . Azonban a tananyag kimondottan gyakorlatorientált módon közelíti a témát.

A tananyag igyekszik mindenütt magyar terminológiát használni. Egy-egy fogalom, eszköz, módszer első felbukkanásánál gyakran a magyar elnevezés mellett megjelenik az angol elnevezés is.

A felépítés spirális, sokszor előfordul, hogy egyes fogalmak az első megjelenésük után a tananyag későbbi részében kerülnek kifejtésre, részletezésre, netán alkalmazásuk finomabb elemzésére is így bukkanhatunk rá.

A kurzus és így a tananyagbeli ismeretek elsajátításához előfeltétel (az általános digitális írástudáson túl) egy OO nyelvben szerzett programozási jártasság, a programozási, alkalmazásfejlesztési alapfogalmak és alapeszközök ismerete. A tananyag maga programozási nyelvektől független (pontosabban bármely nyelvi környezetben használható és értelmezhető) ismereteket ad.

A tananyag tíz fejezetből áll, ezek a fejezetek lényegében egy egyetemi szemeszter struktúrájához igazodnak. A felépítés logikája miatt önálló feldolgozás esetén javallott a fejezetek sorrendjében való tanulás.

SZÉCHENYI 2020

Tesztelési és tesztelői alapfogalmak

A TESZTELŐ

Amikor a hétköznapokban végezzük a napi tevékenységeinket nem is jut eszünkbe, hogy ma már szinte nincs olyan feladat, amelyben valamilyen informatikai eszköz nem vesz részt. Teljesen természetes számunkra, hogy bárhol bármikor tudjunk telefonálni, de ettől lényegesen egyszerűbb és természet közelibb feladatok, mint a kirándulás sem képzelhető el a nélkül, hogy valamilyen informatikai eszköz ne működött volna közre ebben. Hiszen ha más nem a ruha, amely rajtunk van biztosan olyan eszközökkel készült, amelyben ezek az eszközök részt vettek.

Ezekről a hétköznapi emberek nem is igazán vesznek tudomást, pedig a munkahelyükön Ők is biztosan közelében vannak ilyen eszközöknek.

Ahogy egyre több eszköz válik "okossá", úgy válnak ezek az eszközök a hétköznapi életünk aktív részévé, ahol a felhasználó nem képzett az adott eszközzel kapcsolatban egyszerűen csak szeretné használni. "Nem akar tudomást venni a szoftverről."

Ha azonban valamely eszköz nem működik, akkor bizony bosszús lesz, hiszen ez megnehezíti vagy megakadályozza Őt valamilyen már eltervezett tevékenységében. Ha az Olvasó is találkozott már ezzel az érzéssel, akkor tudja, hogy miről van szó.

Ki a tesztelő? Egy olyan szakma képviselője, aki azért dolgozik, hogy az ügyfél megfelelően működő szoftvert használjon, és azzal elégedett legyen.

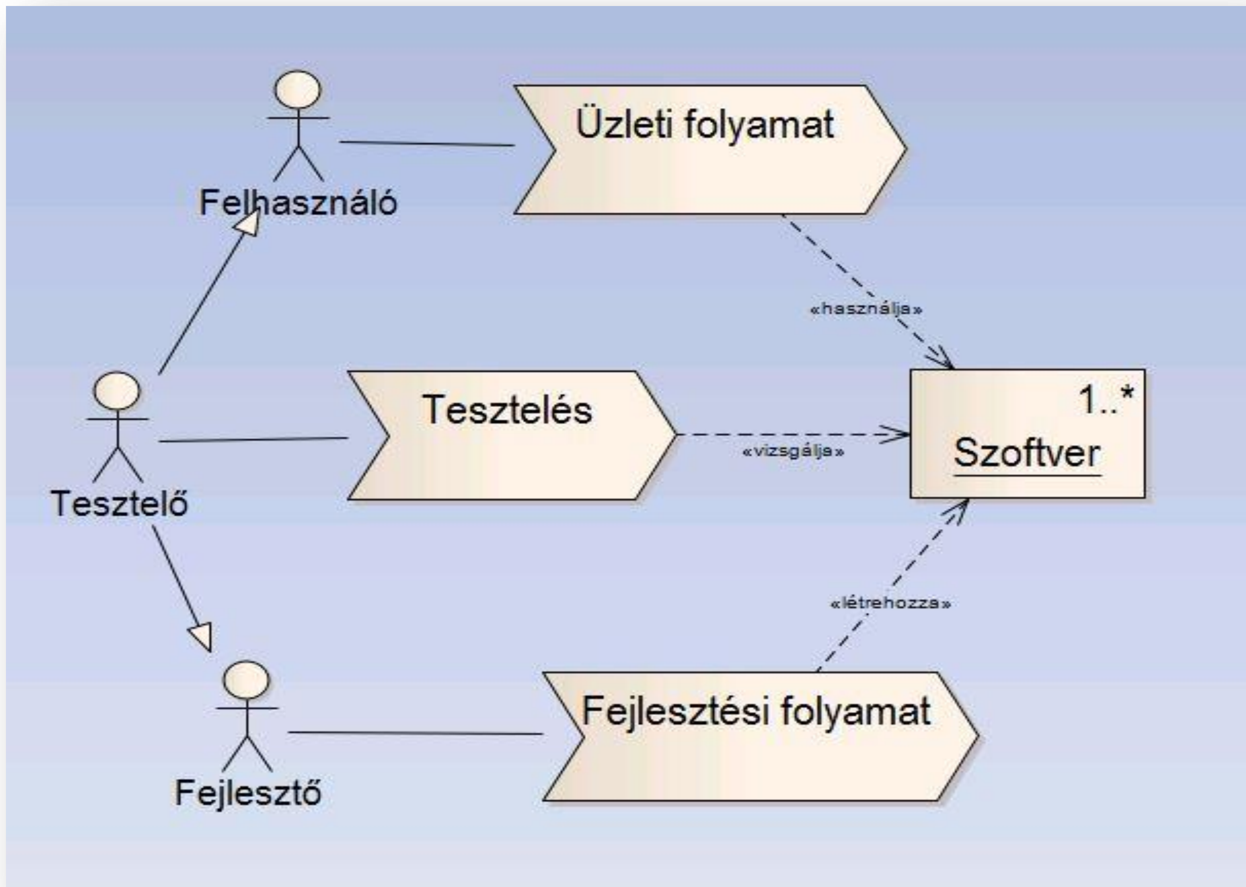
Mi a tesztelő célja? Az hogy a fent részletezett csalódottság érzés lehetőleg soha ne forduljon elő és az ügyfél továbbra is azt érzékelje, hogy az az eszköz, amit használ, segíti őt a tevékenységében.

A szoftver kapcsolatai a fejlesztési folyamat során

A szoftverrel a különböző szereplők másként vannak kapcsolatban a szoftver életciklusa során:

- A **fejlesztő** a fejlesztési folyamaton keresztül létrehozza a szoftvert.
- A **tesztelő** a tesztelés segítségével vizsgálja a szoftver megfelelőségét.
- A **felhasználó** különböző tevékenységeken keresztül pedig használja a rendszert.

Amikor a **szoftverről** és az egyes szereplőkről beszélünk fontos, hogy megértsük ezeket a lényeges különbségeket.



Szoftverek a világban

Amint látjuk, ma már a szoftverek minden részét behálózzák életünknek, azonban fontos, hogy a különböző típusú szoftvereket a tesztelés szempontjából is elkülönítsük egymástól, hiszen céljuk, felhasználási módjuk és a használat körülményei is jelentősen eltérnek egymástól. Ezek azért fontosak, mert a tesztelés szempontjából akár teljesen más módon kell ezeket megközelíteni.

Üzleti szoftverek

Az üzleti szoftverek azok, amelyek egy szervezet (vállalat, közigazgatás) működésnek folyamatait közvetlenül támogatják a szoftver használatán keresztül (pl. ügyviteli folyamatok, értékesítés, logisztika). Sok esetben már elképzelhetetlen, hogy manuálisan kezeljék le azt az időszakot, amikor a szoftver nem működik megfelelően.

Az üzleti szoftverek általában az adott szervezet működésének részterületét kizárólagosan támogatják, így a velük kapcsolatos minőségi problémák az adott terület működését befolyásolják jelentősen.

Amennyiben vállalatról beszélünk, akkor az értékesítési szoftver hibája csökkentheti vagy megakadályozhatja a cég bevétel termelési képességét, így akár a korábbi hasonló időszakkal való összehasonlítással pontosan kimutatható, hogy milyen üzleti kár érte a céget. Fontos, hogy az ilyen anomáliákból adódó veszteség minden esetben legyen pontosan kimutatva, elsősorban azért, hogy a menedzsment objektív módon lássa, hogy a megfelelő minőségű tesztelés milyen károk bekövetkezésétől mentesíti a vállalatot.

Az üzleti szoftverekkel kapcsolatban általában el lehet mondani, hogy a működési problémáik üzleti veszteséget jelentenek, de közvetlenül nem veszélyeztetnek emberi életeket.

Vezérlő és rendszer szoftverek

Ezek olyan szoftverek, amelyek valamilyen hardver eszközt közvetlenül működtetnek. A szoftverek a tervezésük során is már a vezérelt hardver eszközökhöz készülnek és azokon kívül mással nem is tudnak működni.

Jellemzően a hardver eszköz specifikus működését irányítja és/vagy a hardver eszköz által generált jeleket dolgozza fel.

A vezérlő szoftverek működését a rendszer szoftverek hangolják össze, így ezek biztosítják a hardver eszköz megfelelő működését.

A vezérlő és rendszer szoftverek működésének problémái a hardver eszköz hibás működéséhez vezethetnek így az akár fizikai károsodást is szenvedhet. Ez a fizikai károsodás az anyagi káron túl akár közvetlenül életet is veszélyeztethetnek.

Ezért ezeknek a tesztelése során egészen más megközelítést kell alkalmazni. Ezeknél a hardver-szoftver komponenseknél gyakori a többszörözés (kritikus eszközök esetén páratlan számút alkalmaznak), ahol az egyik elem egyértelmű hibája esetén a többiek átveszik a meghibásodott elem funkcióját.

Természetesen ezen esetekben a tervezésnél és a tesztelésnél is figyelembe kell venni azokat a hibakezelő folyamatokat, amelyek ilyen esetekben elindulnak.

A vezérlő szoftverek és hardverek együttes hibája bizonyítottan és valószínűsíthetően is számos halálos áldozatokkal járó és nagy anyagi és erkölcsi károkat okozó katasztrófát okoztak.

SZÉCHENYI 2020

A repülés – bár jelenleg a legbiztonságosabb utazási forma – az egyik leginkább veszélyeztetett terület, mert az ott előforduló hibák – a többszörözött rendszerek ellenére is – végzetesek lehetnek az utasokra.

Példa: Az Airbus A320 repülőgép az egyik első kereskedelmi repülőgép, amely teljesen számítógépes vezérlő rendszerrel működik. Ez azt jelenti, hogy a repülőgép kormányai nincsenek közvetlen kapcsolatban a kormányfelületekkel, egy joystick és a pedálok segítségével adnak jeleket a vezérlő szoftvernek, amely működtetni a különböző hardver eszközöket. A rendszer szoftver folyamatosan figyeli a kormányokat és amennyiben úgy dönt, hogy a kormánykitérés "nem megfelelő", akkor felülbírálja és a csak a megfelelőnek ítélt módon téríti ki a kormánylapokat.

Hogy mi a "megfelelő" természetesen az előzetesen hatalmas tapasztalaton alapuló tervezés határozta meg. Azonban így is csúszhatnak be hibák, ez vezetett ahhoz, hogy az első prototípus leszállás közben katasztrófát szenvedett.

A baleset során a vezérlő szoftver felülbírálta a pilótákat, így a leszállás helyett a repülőtér melletti erőbe zuhant a repülőgép. A személyzet hét tagja meghalt. (http://www.youtube.com/watch?v=YAg-WauGrLU)

Az Airbus mérnökei megtalálták a hibát a szoftverben, kijavították és a típusból több mint 4500 darab repül és havi több mint 40 darabot adnak át a légitársaságoknak.

Játékszoftverek

A 1980-90-es években született generáció számára a játékszoftverek már a hétköznapi élet részei. Ezek elsődleges szerepe a szórakoztatás. Mára a játékszoftver ipar nagyobb üzlet, mint a filmipar, beleértve a Hollywoodi filmeket is.

Egy játék fejlesztési költségvetése meghaladhatja a legmonumentálisabb speciális effektekkel ellátott filmét is. A játékszoftverekben használt technológiák ma már megjelennek a szórakoztatás mellett a mérnöki területeken is. Gondoljunk bele, hogy a Formula 1 versenyzői még a tervező szoftverben lévő autót egy szimulátorban kipróbálva javaslatokat tehetnek annak megváltoztatására. A repülőgép pilóták ma a vészhelyzeteket azokban a szimulációs szoftverekben próbálják ki, amelyek játékokból fejlődtek valódi kiképzést segítő eszközökig.

A játékszoftverek általában mégis a szórakoztatás eszközei és elvárjuk, hogy szoftver hiba nélkül biztosítsák a maximális élményt. Ha a szoftver nem megfelelően működik, akkor az első játékosok vagy

újságírók által írt tesztek negatívan tüntethetik fel azt és ez akár el is döntheti egy játék kereskedelmi sikerét.

A meghibásodás okai

A szoftver meghibásodások oka származhat a szoftver működési környezetéből és magából a szoftverből.

A környezetből származó hibák a szoftver működésére ugyanúgy hatással vannak és a legtöbb esetben a felhasználó nem érzékeli, hogy mi a valódi kiváltó ok.

Hardver hiba

A szoftverek minden esetben valamilyen hardver környezetben működnek, így azok problémái közvetlen hatással vannak annak működésére. Az esetek egy részében a hardver hiba az egész eszköz működésképtelenségéhez vezet, ilyen esetben a kiváltó ok könnyen felismerhető. Ha a hardver hibája olyan típusú, hogy az eszköz látszólag működőképes, akkor ez nagyon megnehezítheti a hiba felismerését és kiküszöbölését.

Hardver hiba lehet például egy olyan érzékelő meghibásodása is, amely adatokkal lát el összetettebb szoftvereket. Például, ha egy hőfok vagy nyomás érzékelő hamis adatokat ad, akkor a rendszer szoftver a hamis adatok alapján vezérli az eszközt. Ez például egy autó motorban, akár a motor részleges vagy teljes meghibásodását is okozhatja, de egy repülőgépen akár súlyosabb baleset kiváltója lehet. Természetesen ez ellen lehet védekezni az egyes rendszerek többszörözésével és az egymástól eltérő értéket adó hardver eszközök közötti prioritás meghatározásával. Ez a tervezés feladata, a tesztelés során a kritikus rendszereket úgy kell vizsgálni, hogy az ilyen esetekben való működés is egyértelműen igazolható legyen.

A hardver hiba következhet nem csak a korábban működő eszköz meghibásodásából, hanem abból is, ha az adott hardver eszköz valamilyen gyártási hiba miatt nem hozza a megfelelő teljesítmény paramétereket.

Példa: Megtörtént eset, hogy egy szoftver a terheléses tesztelés során nem hozta az elvárt paramétereket. Az egyes komponensek vizsgálata alapján nem volt oka a lassulásnak.

Kisebb terhelésnél tökéletesen működött a rendszer, egy bizonyos terhelés felett azonban exponenciálisan megnőttek a válaszidők. Csak hálózati kapacitás problémára lehetett gondolni. Külön megvizsgálva a hálózatot a szükséges kapacitás többszörösét is gond nélkül biztosította, így ezt ki lehetett zárni. Maradtak a szerverek vagy a kliensek. A kliensek régen használatban lévő eszközök voltak, minden más rendszerrel működtek megfelelően. A szerverek viszonylag újak voltak, így a figyelem ezekre

irányult. A szervereket szétszedve kiderült, hogy a gyártó az összerakás során a 1 GB kapacitású hálózati kártya helyett 100 MB kapacitásút szerelt az eszközbe – rosszul felcímkézve – így ennek alacsony kapacitása okozta a problémát. (A gyártó 4 órán belül cserélte.) A vizsgálattal elveszett idő 100 munkaóra felett volt.

Hálózati hiba

Az egyes számítógépek együttműködését a különböző hálózatok biztosítják. Ezek lehetnek a jól ismert IP alapú hálózatok (LAN, Internet), de gondolni kell olyan hálózatokra is, amelyek valamilyen speciális számítógépeket kapcsolnak össze. A mai korszerű autókban 30-nál több különálló számítógép van. Az ezek közötti együttműködést is hálózatok, úgynevezett adatbusz rendszerek biztosítják.

Az adatok áramlása elsődleges fontosságú az egyes számítógépek között, így ha ebben hiba van, az lelassítja vagy megakadályozhatja a működést.

Amikor a hálózati hibákat kell vizsgálni, nincs könnyű helyzetben a tesztelő, hiszen a hálózatok önmagukban is sok komponensből álló rendszerek. Ráadásul itt a különböző rendszerek adatai "egy csőben" áramlanak, így egymásra is jelentős hatással vannak. A tesztelés tervezésénél és a rendszerek vizsgálatánál ezért kell külön szerepet szánni a hálózat megfelelő vizsgálatának.

A hálózat viselkedése jelentősen változik a terhelés függvényében. A növekvő terhelés a hálózat bedugulásához is vezethet, amely akár a szoftver rendszerünk leállását is okozhatja, hiszen az egyes szoftverkomponensek nem kapnak időben választ a többi komponenstől.

Általános tapasztalat, hogy a szoftverrendszerek tervezése és tesztelése során mondjuk egy vállalati hálózatot "végtelen" kapacitásúnak tekintik. A hálózat eszközök fejlődése miatt, ma már sok lokális hálózat ténylegesen jelentős kapacitásfelesleggel működik. Amikor új rendszereket telepítünk, egyre több adatot áramoltatunk át a hálózaton, ha erre a szoftverek tervezői és tesztelői nem figyelnek oda, akár az új rendszer hozhatja el azt a telítettségi szintet, amely már problémát jelent. Természetesen ilyenkor egy hálózati eszköz hibája a terheltség miatt végzetes hibát okozhat a rendszerek működésében.

Nehezíti ennek tesztelését, hogy a tesztrendszereket általában elkülönítetten kezelik az éles rendszerektől ezért nehéz valós körülményeket előállítani a tesztelés során.

Gondoljuk bele: melyik vállalat szeretné, hogy a napi működése közben az új rendszer terheléses vizsgálatát ugyanazon a hálózaton végezzük vizsgálva, hogy esetleg nem okoz-e problémát.

Konfigurációs hiba

A rendszerek tervezésénél fontos szempont az újrafelhasználhatóság. Ha a rendszereket csak a szoftverkód módosításával lehetne különböző környezetekbe telepíteni, az igencsak költségessé és lassúvá tenné azok elterjedését. Ezért már nagyon korán megjelent a lehetőség a rendszerek konfigurálhatóságára. A konfiguráció segítségével – az előre meghatározott lehetőségek alapján – jelentősen befolyásolhatjuk a rendszerek működését. Ez jelentősen kitágítja a rendszer felhasználási lehetőségeit, azonban ez azt is jelenti, hogy az egyes konfigurációs beállításokhoz meg kell határozni a megfelelő működést.

Sok rendszer akár több száz paraméter segítségével konfigurálható és azok értéktartománya is több tíz elemből állhat.

Belátható, hogy ez a tesztelést nagyon komoly kihívások elé állítja. Nagyon felértékelődik az adott rendszer szakértőinek tudása. Természetesen ilyen esetekben is előfordulhat olyan konfigurációs beállítás, ami nem érvényes és a rendszer nem megfelelő működéséhez vezet.

A rendszer tervezésekor ez már látszik, így célszerű olyan önellenőrző mechanizmusokat bevezetni, amelyek vizsgálják, hogy az aktuális konfiguráció érvényes-e.

Ez viszont nem ad választ olyan esetekre, amikor az egyik rendszer érvényes konfigurációja egy másik rendszerrel való együttműködésben már nem megfelelő működést okoz.

Példa: Tipikus probléma a változó terhelés (ez számos esetben megjelenik). Egy nagyvállalati környezetben működő rendszer adatbázisa általában több rendszert szolgál ki. Ennek nyilvánvalóan erőforrás optimalizálási okai vannak. Amikor az adatbázisokat telepítik, meghatározzák azokat a paramétereket, amelyekkel optimálisan tud működni. Ilyen paraméter az engedélyezett kapcsolatok száma. Ha ennek a paraméternek a beállítása nem követi egy, a funkciók bővülésével egyre növekvő terhelésű rendszer igényeit, az előbb-utóbb azt eredményezi, hogy "elfogynak" a rendelkezésre álló adatbázis kapcsolatok és az alkalmazás nem tud kapcsolatot felépíteni. Ez egy üzleti szoftver esetében általában egyenlő azzal, hogy működésképtelenné válik. Ebben az esetben is fontos felhívni a figyelmet arra, hogy a tesztelés során törekszünk arra, hogy az "éles" környezethez a leginkább hasonló működési módon teszteljünk ez azonban legtöbbször a különböző – egyébként egymástól független – rendszerek egymásra hatásai miatt nehezen kivitelezhető.

SZÉCHENYI 2020

Kezelői hiba

Azoknál a rendszereknél ahol a működésnek szerves része az emberi beavatkozás, ott természetesen az emberi hibát is számításba kell venni.

Ezekre a tervezésnél is érdemes felkészülni és a felhasználó tevékenységét "vezetni" kell. Ha egy meghatározott vállalati folyamatról beszélünk, akkor természetesen általában jól vezethető a felhasználó. Be kell építeni megfelelő érvényesítési eljárásokat az egyes beviteli mezőkhöz és számos olyan eszköz van, amellyel csökkenthető az ilyen hibalehetőség. Kizárni azonban ezeket sajnos nem lehet.

Példa: Egy vállalati rendszerben az ügyfél megkeresése a személyes adatai alapján (név és születési idő) történik. A felhasználónak azonban nagyon bonyolult a neve és a telefonos beszélgetés alapján a kezelő rosszul rögzíti azt a rendszerben. Ráadásul a születési idejét is elgépeli. Mind a kettő érvényes adat a rendszer szempontjából, azonban az ügyfelet nem lehet a keresési eljárás segítségével megtalálni a rendszerben. Ilyenkor érkezik egy hibajelzés, hogy eltűnt az ügyfél. Természetesen a vizsgálat megállapítja, hogy a rendszer megfelelően működik, de sajnos a felhasználó számára ez komoly problémát jelent. A tervezés során olyan megoldásokat kell választani, amelyik csökkentik a kezelői hibák valószínűségét és hatását.

Nem megfelelő környezet

Ahogy a konfigurációs fejezetben is említettük, a rendszereknek számos különböző környezetben kell működnie.

A tervezés során meghatározásra kerül az a számítógépes környezet, ahol a rendszernek működnie kell. Amennyiben a környezet nem olyan, mint amire a rendszert tervezték, nem garantálható a rendszer megfelelő működése. A környezet alatt értjük a szoftverkomponenseket (operációs rendszer, web böngésző, adatbázis-kezelő stb.), a szükséges hardver elemeket (szerverek, kliens), a hálózatot is.

Manapság gyakori példa az Internet világában, hogy egy elkészített szoftver meghatározott böngésző típus(okkal) és azok verzióival kompatibilis, azonban az új verziókkal vagy más típusú böngészőkkel nem, így amikor ezekkel használjuk, az alkalmazás nem működik megfelelően.

Nagyon fontos, hogy amikor egy rendszer elkészül, akkor meghatározásra kerüljön az a környezet, amelyben a megfelelő működés biztosított.

Ezért amikor egy hibajelentés készül minden esetben nagyon fontos, hogy az tartalmazza, hogy milyen környezetben állt elő a hiba, ezzel rengeteg többletmunka elkerülhető.

SZÉCHENYI 2020

Környezet alatt érthetjük viszont azt a fizikai környezetet is, amelyben ennek a rendszernek működnie kell.

Gondoljunk bele, hogy egy repülőgép, amelyik felszáll egy sivatagi országban plusz 45 fokban, néhány perc alatt felemelkedik 12 000 méter magasra és ott akár mínusz 60 fok is lehet, de ezzel együtt a légnyomás is lecsökken így a nyomásértékek alapján működő szoftvereknek is figyelembe kell venniük ezt és a más módon kell működniük. Ez olyan extrém terhelésnek teszi ki a hardver eszközöket is, amelyet azoknak el kell viselniük.

Adatsérülés

A szoftver tervezésekor meghatározásra kerül, hogy milyen input és output adatokkal működik megfelelően.

A rendszer működése során eltárol bizonyos adatokat a memóriában vagy az adatbázisban. Ez a rendszer szempontjából zárt, tehát a rendszer nem "feltételezi", hogy ezek megváltozhatnak. Azonban ha valamilyen ok miatt az adott struktúrában letárolt adatok megsérülnek (HDD vagy memória hiba) és a rendszer erre nincs felkészítve, akkor ez a rendszer nem megfelelő működéséhez vezet.

Ha az adatok struktúrája sérül meg, akkor az felismerhető, és erre megfelelő hibakezelési eljárás építhető.

A hibák azonban lehetnek olyanok, amely nem érintik az adatok struktúráját, tehát a rendszer fel tudja azokat dolgozni (pl. egy numerikus érték esetén, ha a hibás érték még belefér a lehetséges értéktartományba). Ekkor a rendszer nem is érzékeli a nem megfelelő működést, de az output eredmények már hibásak lesznek. Ha az adatsérülés okozta nem megfelelő működés nagy kockázattal jár, akkor célszerű adatellenőrző algoritmusok használata, amely az adatok alapján különböző ellenőrző számokat generálnak, így ha az adatfeldolgozás során az ellenőrző szám nem egyezik a korábban eltárolt értékkel, akkor feltételezhető az adatsérülés és ennek kezelésére megfelelő eljárás felépíthető.

A szoftver készítése során indukált hibák

A szoftverek meghibásodása származhat a külső körülményeken túl olyan okokból, amely az elkészítéskor elkövetett hibákra vezethetők vissza.

Emberi tévedés, elírás

Az emberi tévedés az egyik legkomolyabb hiba ok. Ez jöhet az igény megfogalmazásakor elkövetett hibákból. Például a tervezés során kimarad valami vagy rosszul definiálunk adatokat, algoritmusokat, amelyek később az üzleti folyamat végrehajtása során okoznak problémát.

Ismerethiány

Egy szoftver tervezésénél nagyon fontos az adott területen szerzett tapasztalat, azonban a folyamatos fejlődés miatt még ilyen esetben is előfordulhat ismerethiányból következő olyan megoldás, amely nem megfelelő működést eredményez.

Az ismerethiány lehet technológia, fejlesztéstechnikai, módszertani vagy üzleti ismerethiány.

A legtöbb esetben a technológia ismertekkel rendelkező fejlesztő csapat nem rendelkezik olyan üzleti tudással, amely elegendő a szoftver részletes és az igényeket kielégítő megtervezéséhez. Ezért a különböző területeken tudással rendelkező csapattagoknak együttműködve kell létrehozniuk a megfelelő megoldást.

Ez az egyik leggyakoribb probléma. A korszerű szoftverfejlesztési módszertanok egyik legfontosabb előnye, hogy az üzleti tudással rendelkező szakembert épít be a fejlesztő csapatba és a szoftvert is prototípusok készítésén keresztül a lehető leghamarabb megismerteti a később végfelhasználók képviselőivel.

Segédprogramok hibái

A szoftverek számos komponensből épülnek fel, ezek megfelelő együttműködése valósítja meg szoftver célját.

A mai szoftverek esetében számos olyan komponens is a rendszer része, amelyek valamilyen külső forrásból érkeznek, ezek lehetnek számos szoftverbe beépülő segédprogramok. Ezek hibája a megvalósított szoftver működését is befolyásolja. A tesztelés során külön hangsúlyt kell fektetni arra, hogy jól elkülöníthetők legyenek ezek a modulok és az esetleges hibák esetén ezek könnyen felismerhetők legyenek. Az ezekben bekövetkező hibák azonban komoly gondot okozhatnak a szoftver készítőinek, hiszen ezek "házon kívül" vannak, így a hibajavítás időben nagyon elhúzódhat, illetve bizonyos esetekben a harmadik fél ezt nem is hajlandó elvégezni.

Ezekben az esetekben megtörténhet a probléma körbefejlesztése, ami általában azt jelenti, hogy a hibás működést okozó eseteket speciális kódrészletekkel lekezelik. Ez sem fejlesztési sem tesztelési szempontból nem túl ideális, mert külön figyelmet kell rá szentelni, és ha a segédprogram fejlesztője kijavítja a hibát, akkor ez a szoftverben újabb problémákat okozhat az által, hogy a korábbi hibás működést kezelő kód a most már jól működő segédprogrammal nem a megfelelő működést produkálja.

SZÉCHENYI 2020

Rohammunka és szervezési hibák

A szoftverfejlesztés általában költséges tevékenység. A projektek esetében ráadásul a piac és a technológia fejlődésével is meg kell küzdeni. 4-5 év alatt a szoftverfejlesztési technológiák gyakorlatilag kicserélődnek vagy olyan fejlődésen mennek keresztül, amely egy korábban készült rendszert teljesen elavulttá tehet.

Az üzleti szoftverek területén még az üzlet folyamatos változásával is meg kell küzdenie a szoftverfejlesztő és tesztelő csapatnak.

Ez sok esetben eredményezi azt, hogy egy projekt végrehajtására a piaci hatások miatt kevés idő áll rendelkezésre.

Ha ezt nem ismeri fel a megrendelő és a projektet végrehajtó csapat ebből komoly időnyomás alakul ki.

Ezeket természetesen nagyon nehéz kiküszöbölni ezért inkább olyan módszereket kellett kifejleszteni, ami kezeli ezeket a helyzeteket.

A projektek során az ismeretek hiánya, az időnyomás és a nem megfelelő szervezés generálja a legtöbb hibát, ezáltal a legtöbb időveszteséget.

A szoftverfejlesztés/tesztelés körülményei

A szoftverfejlesztés és a tesztelés igen komoly szellemi tevékenység, ami jó szellemi és fizikai állapotot kíván meg a projekt résztvevőitől.

A projekt tervezés során oda kell figyelni, hogy megfelelő körülmények biztosítottak legyenek a csapat számára.

E mellett fontos, hogy a csapat számára egyértelműek legyenek a célok és a feladatot olyan részekre bontsuk, amelyek önmagukban egyszerűek, könnyen értelmezhetőek és megvalósíthatók.

Tesztelés szerepe

A tesztelés feladata az, hogy a szoftver használata során fellépő hibák előfordulását csökkentse, a szoftver megbízhatóságát növelje és a szabványoknak, előírásoknak való megfelelőségét biztosítsa. **Tehát az ügyfél elégedettségét növelje!**

SZÉCHENYI 2020

Minőségirányítás, Minőségbiztosítás, Minőség

Minőségirányítás

A **minőségirányítás** egy általános fogalom: Az ISO 8402 (1986) megfogalmazása szerint a minőségirányítás "a teljes körű irányítás azon komponense, amely meghatározza, illetve megvalósítja a minőséggel kapcsolatos alapelveket ... magában foglalja a stratégiai tervezést, az erőforrásokkal való gazdálkodást, valamint más rendszeres tevékenységeket, mint amilyen a minőséggel kapcsolatos tervezés, működtetés és értékelés" (ISO, 2014)

Minőségbiztosítás

A minőségirányításnak az a része, amely a bizalomkeltés megteremtésére összpontosít arra vonatkozóan, hogy a minőségi követelmények teljesülni fognak. (MSZ EN ISO 9000:2005, 3.2.11. pont).

A **minőségbiztosítás** önmagában is egy rendszer, amely számos komponenst fog össze:

- fejlesztési szabványok, szabályzatok kialakítása
- képzés
- tesztelés
- hibaelemzés

A jelen tananyagban csak a szoftverteszteléssel foglalkozunk.

Minőség

A **minőség** definíciója: Az a szint, amikor egy komponens, rendszer vagy folyamat megfelel a meghatározott követelményeknek és/vagy a felhasználó/ügyfél elvárásainak. (IEEE, 2014)

(http://www.scribd.com/doc/101838061/IEEE-610-Standard-Computer-Dictionary, https://cours.etsmtl.ca/mgl800/private/Normes/ieee/intro%20standards%20IEEE.pdf)

A minőség önmagában nehezen definiálható, hiszen függ a körülményektől, a felhasználótól, a lehetőségektől, sőt a kortól is, amiben vizsgáljuk.

Nehéz pontos leírást adni egy jó minőségű termékre, mégis viszonylag könnyen felismerjük a számunkra minőségi termékeket, főleg ha ezt össze tudjuk hasonlítani egy olyannal, ami nem felel meg a mi minőségi elvárásainknak. Tehát a minőség a felhasználó szempontjából szubjektív. Ez a gyártó szempontjából azonban semmiképpen nem lehet az.

Ezért ha minőségi szoftver előállítása a célunk, akkor már a tervezés szakaszában definiálnunk kell azokat a feltételeket, amelyeket teljesítve a szoftver megfelelő minőségi szintet eléri.

A minőségi szint egy olyan mértékrendszer, amely felett a rendszer elfogadható (jó) minőségű, és amely alatt nem fogadható el (nem jó) minőségű.

Mivel a határ felhasználónként és felhasználási módonként változhat, azt mindenképpen figyelembe kell venni, hogy ha a szoftverünkről az a kép alakul ki, hogy nem jó minőségű, akkor ezt a véleményt nagyon nehéz és hosszú időt igénybe vevő módon tudjuk csak megfordítani. Azonban ha tartósan megfelelő minőségű termékeket állítunk elő, úgy ha hibák merülnek fel, akkor is sokkal toleránsabb lesz a felhasználói kör.

Határozzunk meg olyan szempontokat, amelyek befolyásolják a minőséget:

- funkcionális hibák
- megbízhatóság
- használhatóság
- hatékonyság
- karbantarthatóság
- hordozhatóság

Funkcionális hibák esetén meghatározzuk azt a mennyiséget, típust és gyakoriságot, amely mellett még elfogadható minőségről beszélünk.

A három szempontot külön-külön és együttesen is vizsgálni kell.

A típus esetében, ha a hibák olyan területet érintenek, amelyek a használhatóságot nem befolyásolják, akkor még a szoftver lehet jó minőségű. Tehát ebben az esetben azt kell meghatározni, hogy milyen típusú hibák nem fordulhatnak elő a rendszerben.

Ha a hibák mennyisége – típustól függetlenül – meghalad egy bizonyos mértéket, akkor a rendszer nem megfelelő. Ez elsősorban azért van, mert eléggé általános, hogy ahol hiba van a rendszerben, ott akár olyan egyéb hibák is vannak, amelyek a tesztelés során nem derültek ki (hibafürtök). Így fel kell tételezni, hogy ott további hibák is vannak, és ha ezek száma egy meghatározott mennyiséget meghalad, akkor

nem beszélhetünk megfelelő minőségről (még akkor sem, ha a hibák típusai olyanok, amelyek a használatot nem befolyásolják).

A hibák gyakorisága pedig attól függ, hogy a hibás funkció milyen gyakran van használatban. Ha a működés során az adott funkciót percenként használnánk, akkor ott nem fogadható el hiba, ha viszont egy olyan funkcióról beszélünk, amely évente egyszer használt és a következő használat is csak 10 hónap múlva lesz, addig pedig van idő azt kijavítani, akkor nyilvánvalóan ez a jelenlegi használatot és a minőséget nem befolyásoló hiba.

Megbízhatóság esetén a rendszer funkcióinak folyamatos működéséről beszélhetünk. Tehát, ha egy rendszer bár funkcionálisan nem tartalmaz hibát, de az adott környezetben folyamatosan leáll, belassul és így a funkciók nem használhatók, akkor a rendszer megbízhatatlan.

A **használhatóság (usability)** egy külön területté nőtte ki magát a szoftvertervezés területén. A usability célja, hogy a **felhasználói élmény** (**User eXperience**) minél jobb legyen. A használhatóságnál mindig azt kell vizsgálni, hogy az adott körülmények között, ahol a szoftvert használják, hogyan érhető el a maximális élmény.

Egyszerű példa, hogy ha készítünk egy weboldalt, amely számítógépen megjeleníthető és ennek elkészítjük a mobil készülékeken futó párját, teljesen mások a használhatóság szempontjai, így másképp kell működnie is annak.

Minél jobban ismerjük és vesszük figyelembe a tervezés során azokat a körülményeket, ahol a rendszert használni fogják, annál jobb lesz a rendszer használhatósága.

E mellett fontos szempont, hogy a felhasználó hogyan "éli meg" a szoftver használatát. Ha például két képernyő között – a rendszer működéséből adódóan – hosszú várakozási idő kell, akkor adhatunk a felhasználónak olyan információkat, amelyet áttekintve a képernyő váltás ideje nem tűnik olyan hosszúnak.

A **hatékonyság** esetén azt vizsgáljuk, hogy az adott szoftver hogyan gazdálkodik az erőforrásokkal. Egy adott funkcionalitást, minden tervező és programozó másként készít el. Az adott szakember előképzettsége és tapasztalata jelentősen befolyásolja az elkészített szoftver hatékonyságát.

Itt vizsgálhatjuk az adott funkciók, modulok futási sebességét, a lekérdezések sebességét és ezekre meghatározhatunk olyan mérőszámokat, amelyeket nem léphet túl az adott funkció. Például, ha van egy

SZÉCHENYI 2020

lekérdezés, amelyet egy perc alatt, csúcsidőben 100 alkalommal kérdezünk le, akkor meghatározhatjuk azt, hogy a lekérdezés válaszideje ne legyen hosszabb, mint 2 tized másodperc. Ehhez olyan hatékony megoldást kell választani, amely mellett biztosítható ez a lekérdezési sebesség.

Karbantarthatóság szempontjából, olyan megoldásokat kell választani, ami azt biztosítja, hogy a rendszer üzemeltetése során, könnyen áttekinthető legyen. Egyszerűen és biztonságosan elérhetők legyenek a rendszer naplói (a logok). A gyakran használt adminisztratív funkciók könnyen elérhetők legyenek.

Az üzemeltető csapat folyamatosan kapjon információt a rendszer működéséről, az esetleges hibák, amelyeket az üzemeltető nem lát, akár automatikusan bekerüljenek egy **issue tracking** (**ticketing**) rendszerbe, ahol azok a megfelelő hibakezelési folyamatot indítsák el.

Hordozhatóság szempontjából fontos, hogy a rendszer úgy épüljön fel, hogy az a környezet (hardver, szoftver) változásaira az előzetesen tervezettnek megfelelően tudjon reagálni.

A hordozhatóság szintje mindig egy fontos tervezési szempont és jelentős mértékben meghatározza a fejlesztés módját és idejét.

Ha példaként veszünk egy mobil alapú szoftvert, akkor láthatjuk, hogy jelenleg több száz különböző típusú eszköz használja az Android operációs rendszer különböző verzióit. Belátható, hogy ha csak a legelterjedtebb eszközök felbontását akarjuk kezelni a legelterjedtebb operációs rendszer verziókon nagyon komoly tervezési és tesztelési feladatok elé nézünk.

Tesztelés mennyisége és a kockázat

Tesztelés mennyisége

A tesztelés mennyiségének megállapítása során figyelembe kell venni azt, hogy a rendszer milyen célból készült és a szoftver egyes területeire milyen hatása lehet egy hiba bekövetkezésének. Ha ezt végiggondoljuk, akkor láthatjuk, hogy egy olyan funkciót, amelyet kevesen és nagyon ritkán használnak, valamint az ott bekövetkező hiba hatása is nagyon kicsi, nem érdemes olyan részletes tesztelésnek alávetni, mint egy gyakran használt és a folyamat szempontjából kritikus szoftverelemet.

Ezt végiggondolva láthatjuk, hogy a teszttervezés során milyen szempontokat kell figyelembe venni annak érdekében, hogy az optimális teszt mennyiségét meg tudjuk határozni.

SZÉCHENYI 2020

A tervezés során meghatározunk egy olyan optimum sávot, amely költségben még elfogadható és biztosítja a megfelelő minőségi szintet.

Ez így azonban még mindig nagyon általános ezért érdemes bevezetni a kockázat fogalmát. A **kockázat** (**risk**) az a tényező, amely a jövőben negatív következményeket okozhat. Általában, mint hatás és valószínűség jelenik meg.

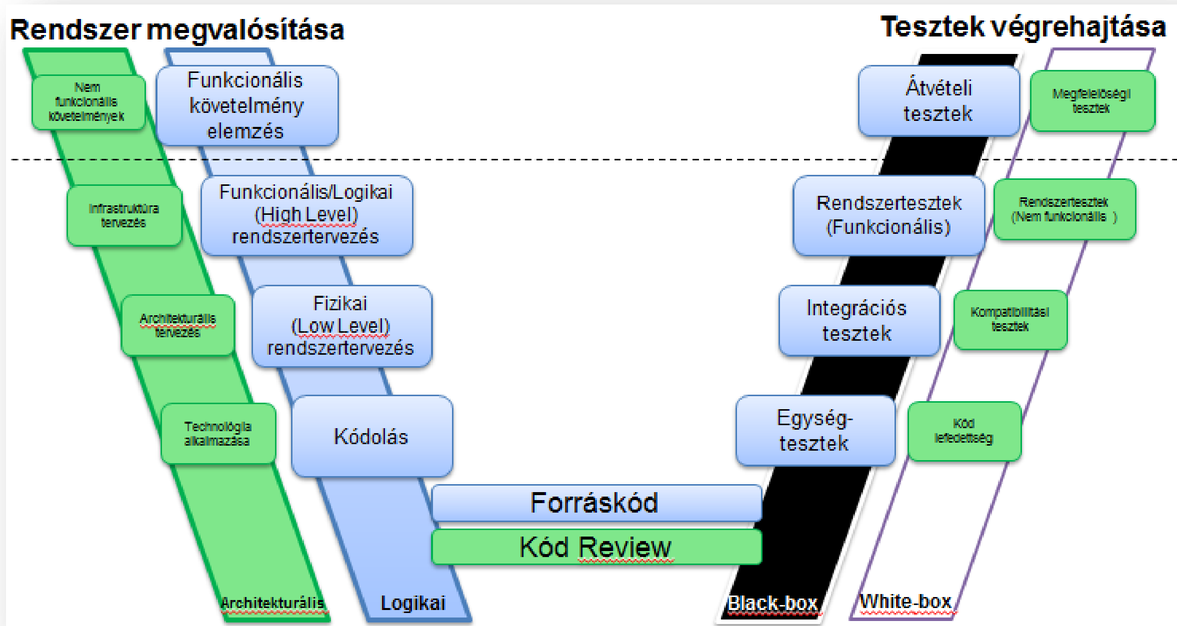
A **hatás** (**impact**) számítása folyamatonként elvégezhető vagy tapasztalati úton becsülhető.

Példa. Egy vállalati szoftver esetében, ha az adott folyamat nem használható 3 óráig, akkor 200 felhasználó, aki ezen a rendszeren keresztül dolgozik, nem tudja végezni a munkáját. Így a lemaradás 600 munkaóra lesz az elvégzett feladatokban. Ha ezt túlórával kell pótolni, akkor annak költsége 4 500 Ft/órával számolva 2 700 000 Ft bérköltséget jelent (az egyéb veszteségekkel még nem is számolva). Ha annak a valószínűsége, hogy ez a bevezetés után az első hónapban 4 alkalommal megtörténik 50%, akkor ez 5 400 000 Ft biztos veszteséget jelent.

Ha ennek csak a felét költjük tesztelésre, akkor már belátható, hogy megéri.

Mi a tesztelés?

A tesztelés a szoftverfejlesztési életciklus minden részéhez kapcsolódó akár statikus, akár dinamikus folyamat, amely kapcsolatban áll a szoftvertermékek tervezésével, elkészítésével és kiértékelésével, hogy megállapítsa, hogy a szoftvertermék teljesíti-e a meghatározott követelményeket, megfelel-e a célnak. A tesztelés felelős a szoftvertermékkel kapcsolatos hibák megtalálásáért. (HSTQB, 2014)



Az oktatási anyag és a CoSTiP kurzus ezt a definíciót követve visz végig a szoftvertesztelés módszerein és eszközrendszerén.

Fontos, hogy tisztában legyünk azzal, hogy a tesztelés a szoftver életciklusának a legelején a követelmények megfogalmazásakor is fontos szerepet tölt be. Ekkor a megfelelő képzettségű és tapasztalatú tesztelési szakemberek elemzik a dokumentumokat és meghatározzák azokat a fejlesztés és tesztelés szempontjából fontos lépéseket, amelyek a megfelelő minőségű szoftver létrehozásához szükségesek.

A tesztelési szakemberek részt vesznek a tervezési fázisban és érvényesítik azokat a szempontokat, amely biztosítja azt, hogy a szoftver megfelelően tesztelhető legyen.

A kódolás során az egység tesztek fontos elemei az egyes komponensek megfelelő működésének meghatározásában (!) és ellenőrzésében.

Az ezt követő integrációs tesztek biztosítják az egyes modulok megfelelő együttműködését.

A rendszertesztek során győződik meg a csapat, hogy a megvalósított szoftver megfelel a funkcionális és nem funkcionális megfelelőségi kritériumoknak.

SZÉCHENYI 2020

Az átvételi tesztek pedig az ügyfél bizalmát kell, hogy megerősítsék abban, hogy az elkészült megoldás megfelel az általa megfogalmazott elvárásoknak.

Látható tehát, hogy a tesztelés, mint tevékenység és mint szakma nagyon összetett és számos területet ölel fel.

A tesztelés felelős a szoftvertermékkel kapcsolatos hibák megtalálásáért.

Ez azt jelenti, hogy a tesztelés célja, az hogy azonosítsuk azokat a pontokat ahol a rendszer nem az elvárt módon működik, de a hiba okának megkeresése nem tesztelő feladata.

A hibakeresés (debugging) a szoftver meghibásodás okainak megtalálási, analizálási és eltávolítási folyamata.

A hibakeresés és a tesztelés két különböző folyamat, a **hibakeresés a szoftverfejlesztő feladata**. Természetesen a tesztelőnek fontos szerepe lehet abban, hogy a megfelelő hibajelentésekkel megkönnyítse a fejlesztő feladatát.

SZÉCHENYI 2020

Általános tesztelési alapelvek

A következő tesztelési alapelvek minden, a szoftverfejlesztéssel és teszteléssel foglalkozó szakember számára alapvető iránymutatást nyújtanak.

Az alábbi pontok az ISTQB CTFL Syllabusból származnak. (HSTQB, 2014)

1. alapelv – Hibák látszólagos hiánya

Bár a tesztelés kimutathatja a hibák jelenlétét, de azt nem képes igazolni, hogy nincsenek hibák. A teszteléssel csökken annak az esélye, hogy a szoftverben felfedezetlen hibák maradnak, de ha nem találnak hibát, az nem annak a bizonyítéka, hogy a rendszer tökéletes (értsd: valóban nincs benne hiba).

2. alapelv – Nem lehetséges kimerítő teszt

Kimerítő teszt – azaz mindenre (a bemenetek és előfeltételek minden kombinációjára) kiterjedő tesztelés – a triviális eseteket leszámítva – nem lehetséges. A kimerítő teszt helyett kockázatelemzést és prioritásokat kell alkalmazni, ezáltal növelve a teszttevékenységek összpontosításának hatékonyságát.

3. alapelv – Korai tesztelés

A tesztelést a szoftver vagy rendszerfejlesztési életciklusban a lehető legkorábban el kell kezdeni, és előre meghatározott célokra kell összpontosítani.

4. alapelv – Hibafürtök megjelenése

A tesztelést a feltételezett, illetve a megtalált hibák eloszlásának megfelelően kell koncentrálni. A kiadást megelőző tesztelés során a megtalált hibák többsége néhány modulban van, vagy legalábbis ezen modulok felelősek a működési hibák többségéért.

5. alapelv – A féregirtó paradoxon

Ha mindig ugyanazokat a teszteket hajtjuk végre, akkor az azonos a tesztkészlet egy idő után nem fog új hibákat találni. A "féregirtó paradoxon" megjelenése ellen a teszteseteket rendszeresen felül kell vizsgálni, és új, eltérő teszteseteket kell írni annak érdekében, hogy a szoftver vagy a rendszer különböző részei kerüljenek futtatásra, és ezáltal további programhibákat találhassanak.

6. alapelv – A tesztelés függ a körülményektől

SZÉCHENYI 2020

A tesztelést különböző körülmények esetén különbözőképpen hajtják végre. Például egy olyan rendszert, ahol a biztonság kritikus szempont, másképp kell tesztelni, mint egy e-kereskedelmi oldalt.

7. alapelv – A hibamentes rendszer téveszméje

A hibák megtalálása és javítása hasztalan, ha a kifejlesztett rendszer használhatatlan, és nem felel meg a felhasználók igényeinek, elvárásainak.

A tesztelés alapvető folyamata

Teszttervezés

A **tesztelés céljának meghatározása**, amely a megvalósítandó szoftvertermék típusának, céljának, kockázati szintjének figyelembevételével történik.

A tesztelés céljából következően el kell végezni a tesztelési tevékenységek meghatározását. Ennek során a célnak megfelelő tesztelési módszerek, tevékenységek, erőforrások, eszközök kerülnek meghatározásra. Ez alapján készíthető el a projekt tesztelési feladataihoz kapcsolódó pénzügyi terv is.

A fentiek alapján a teszttervezést végzők együttműködve a projekt többi tagjával meghatározzák a határidőket. A határidők meghatározása során fel kell készülni az esetleges nem várható eseményekre is. Ezt erőforrás vagy idő tartalék biztosításával lehet megtenni.

Tesztelemzés és műszaki teszttervezés

A teszttervek elkészítése után a tesztelő csapat elkezdi a tesztbázis átvizsgálását.

A **tesztbázis** (**test basis**) az összes olyan dokumentum, amelyből a komponensekre vagy rendszerekre vonatkozó követelmények származnak. Ezek azok a dokumentumok, amelyeken a tesztesetek alapulnak. Ha egy ilyen dokumentumot csak formális változáskezelési folyamat során módosíthatnak, a tesztbázist ún. fagyasztott tesztbázisnak nevezik. (HSTQB, 2014)

Ezt követően a teszttervezéssel foglalkozó szakemberek **értékelik a megvalósítandó szoftvert tesztelhetőség szempontjából**.

Ez alapján meghatározásra kerülnek a tesztfeltételek és azok prioritása.

Ezt követően a tesztesetek megtervezése és priorizálása történik. Ennek során olyan tesztesetek kialakítása a cél, amely a megfelelő szinten lefedik a rendszer funkcionális és nem funkcionális követelményeit. A tesztfedettség célszámának meghatározása is a tervezési folyamat része.

A tesztesetek alapján meghatározhatók a tesztadatok típusai és az adatok köre.

A tervezés fontos eleme a tesztkörnyezet megtervezése. A tesztkörnyezet tervezés része az is, hogy az ehhez szükséges beruházási terveket is el kell készíteni és a projekt pénzügyi tervében meg kell jeleníteni.

A tesztelés fontos eleme, a tesztelés dokumentálása és a nyomon követhetőség kialakítása, a teszttervezés során az ehhez szükséges eszközöket és feladatokat is meg kell határozni.

SZÉCHENYI 2020

A követhetőségnek fontos eleme, hogy a hibák kategorizálására megfelelő rendszert vezessünk be, amely jelzi a hiba **súlyosság**át (**severity**).

Ez lehet 1-5 fokozatú, de elterjedtebb az 1-8 fokozatú.

Az alábbi egy példa a súlyosság szintekre:

1 – Wish A hiba nincs hatással az üzleti folyamatokra, vagy az általa okozott működésbeli változás nem releváns

3 – Minor A hiba javításra szorul, de vagy nem érinti az üzleti folyamatokat, vagy egy nagyon kis ügyfélkört érint (a napi adatfeldolgozás kevesebb mint 1%-át érinti)

5 – Normal Üzleti funkcionalitást érintő folyamatok hibája, nagyobb ügyfélkört érint (a napi adatfeldolgozás kevesebb mint 30%-át, de több mint 1%-át érinti)

7 – Important Működést hátráltató, részfunkciókat ellehetetlenítő, bizonyos adatkörökre teljes funkciókat érintő, fennmaradása súlyos üzleti veszteséget okozhat, nagyobb ügyfélkört érint (a napi adatfeldolgozás több mint 30%-át érinti)

8 – Blocker A rendszer leállt, használhatatlan. Üzletkritikus, működést nagymértékben befolyásoló, minimális funkcionalitást is ellehetetlenítő hiba, esemény

9 – Security Minden olyan hiba, ami az ügyféladatok szempontjából biztonsági kockázatot jelent

A teszttervezés fontos része a teszt kilépési feltételeinek meghatározása. Ezek természetesen függnek a szoftver típusától.

Ilyenek lehetnek például:

- Meghatározott százaléka a tesztеseteknek megfelelően lefutott
- Hány százalékban kell a forráskódot tesztesetekkel lefedni
- A meghatározott adatkörre a tesztek megfelelően lefutottak
- Nincs 3-as súlyosságúnál nagyobb hiba

Tesztirányítás

A tesztirányítás a teszt megvalósítást és a teszt végrehajtást foglalja magában.

Teszt megvalósítása (test implementation)

A teszt megvalósítás során a teszttervben szereplő a teszteléshez szükséges eszközöket állítják elő.

Létrehozzák a **teszt eljárásokat**, amelyek leírják a tesztelés folyamatát, az egyes lépések egymásra épülését és a prioritásokat.

Létrehozzák a **teszteseteket**. Ezeket a tesztfejlesztéssel foglalkozó szakemberek fejlesztik, ezek a teszttípusok és az alkalmazott módszerek miatt eltérően valósulnak meg.

Példa. A unit tesztek esetén a tesztesetek fejlesztése maga is szoftverfejlesztési (kódolási) feladat. A funkcionális tesztek esetén fejleszthetünk a manuális tesztekhez alkalmazott dokumentumokat (leírásokat), alkalmazhatunk automatikus teszteszközöket (pl. Selenium) stb.

A nem funkcionális tesztek esetén készíthetünk megfelelő szkripteket pl. a terheléses tesztek elvégzéséhez.

Amint látható a teszt megvalósítás a tesztelés nagyon komoly szakértelmet, tapasztalatot igénylő része. A teszteseteket úgy kell létrehozni, hogy azok többször megismételhetők legyenek, valamint teszteset futtatásának eredménye egyértelműen kiértékelhető legyen.

A megvalósított teszteseteket természetesen szintén ellenőrizni kell, hogy megfelelnek a teszttervekben foglalt követelményeknek.

A tesztesetek létrehozása utána megfelelő tesztadatokat is elő kell állítani.

A **tesztadatok** előállítása akár nagyon időigényes feladat is lehet, hiszen a különböző bemenetekre és kimenetekre is elő kell állítani ezeket.

E mellett komoly feladat, hogy a tesztadatok akár egy tesztelés során is elavulhatnak, így ha nagyszámú vagy ismétlődő teszteket kell végrehajtani, akkor azonos adatkörből is a várható teszteknek megfelelő számút kell előállítani.

Belátható, hogy a terheléses tesztek esetén hatalmas egymástól eltérő adatokra is szükség lehet. Ilyenkor a tesztadat generálásra is külön szoftvert hoznak létre.

SZÉCHENYI 2020

A teszt megvalósítás része a **környezetek felépítése**, kialakítása. Egy nagyobb fejlesztés során a tesztkörnyezet felépítése akár összetettebb feladat is lehet, mint a későbbi éles környezeté, hiszen a normál működés mellett ki kell elégítenie a teszteléssel kapcsolatos elvárásokat is.

A teszt megvalósítás során a teszt követéséhez szükséges rendszer létrehozása és beállítása is fontos feladat.

Teszt végrehajtás

A tesztirányítás során a tesztterv és a teszt eljárások alapján folyamatosan hajtjuk végre az abban meghatározott feladatokat, futtatjuk a teszteseteket.

A végrehajtás során folyamatosan dokumentáljuk a teszt végrehajtást. A korszerű teszteszközök ehhez már nagy segítséget adnak, a tesztadatok és a teszteredmény rögzítésével.

A tesztek végrehajtása során folyamatosan figyeljük az elvárt és a kapott adatok közötti eltéréseket.

Ha eltérés mutatkozik, akkor azt **incidens**ként (**issue**) rögzítjük az *issue tracking* rendszerben.

A bejegyzésnek egyértelműen hivatkozni kell a futtatott tesztesetre és teszt futtatás körülményeire.

Folyamatosan össze kell hasonlítani az eredeti teszt végrehajtási tervet és az aktuális előrehaladást

A tesztirányítás során rendszeres időközönként a teszt- és a fejlesztő csapattal közös megbeszéléseket kell tartani, ahol meg kell osztani szóban is a tapasztalatokat és a problémákra közösen kell megoldást találni.

A megbeszéléseken fontos, hogy ne csak a negatív dolgokról, hanem a sikerekről is beszéljünk ezzel növelve a csapat motivációját.

Amennyiben eltérés mutatkozik, értékelni kell, hogy mi az oka az eltérésnek és lépéseket kell tenni az eltérések kezelésére.

Az eltérések kezelésére alkalmazhatók a következők:

- tesztesetek módosítása
- tesztadatok körének módosítása
- szükséges erőforrások változtatása

- szükséges környezetek módosítása
- a szoftver módosítása
- a projekt terv módosítása

A tapasztalatokat és a szükséges lépéseket meg kell vitatni a projektmenedzsmenttel.

A kilépési feltételek értékelése és jelentés

A tesztelés során az előzetes terveknek megfelelően elérjük azt a pontot, ahol a teszteket zárni kell. Ekkor megvizsgáljuk, hogy a tesztelés teljesíti-e a kilépési feltételeket.

Megvizsgáljuk, hogy a tesztnaplók alapján a tesztek megfelelő mennyiségben és minőségben végrehajtásra kerültek-e.

Döntést hozunk, hogy szükséges-e további teszt.

Ezt követően **Tesztösszefoglaló jelentést** készítünk a "Szoftverteszt Dokumentáció Szabvány" szerint (IEEE 829, 2008)

Tesztlezárás

A **tesztlezárás** a szoftvertesztelési folyamat fontos része, ezzel foglaljuk össze az elvégzett munkát. A szoftvertesztelés szakszerű lezárása azért is fontos, mert szinte biztosan nem zárult le a tesztelt szoftver fejlesztése, ezért amikor az újabb szakaszába lép a tesztelést folytatni kell. Tehát az előző projektben felhalmozott tudás és eszközök hasznosulni fognak a jövőben is.

Milyen feladatokból áll:

A tervezett átadandók ellenőrzése során átvizsgáljuk, hogy a tesztelés során elkészített dokumentumok megfelelnek minden formai és tartalmi követelménynek. Az anyagok hasznosíthatók a jövőben. Ilyenkor érdemes még a csapatban benne lévő tudással kiegészíteni azokat, hiszen az idő elmúlásával ezek feledésbe merülnek.

Az **incidens jelentések lezárása** során az incidensek állapotát rögzíteni kell. A lezárult ügyeket a megfelelő kiegészítésekkel el kell látni annak érdekében, hogy az átvevő csapat tisztában legyen az adott incidenssel kapcsolatos feladatokról.

A **rendszer átvételéről szóló dokumentáció elkészítése** a dokumentációnak tartalmaznia kell, hogy milyen állapotban került átadásra, milyen ismert hibák vannak és az üzemeltetés során mire kell felkészülnie az üzemeltetés során. Ez a dokumentum általában komoly jog relevanciával is bír.

A **tesztver, és a teszt infrastruktúra véglegesítése és archiválása későbbi felhasználásra**. A **tesztver** (**testware**) a tesztfolyamat során keletkezett különböző termékek, például dokumentáció, programkód, inputok, várt eredmények, eljárások, fájlok, adatbázisok, környezetek illetve bármilyen egyéb szoftver. (HTB Glossary, 2013)

A tesztver átvizsgálása és archiválása során elkészítjük azokat az utasításokat és javaslatokat, amely a tesztver újbóli felhasználása során a tesztelő csapatok számára megfelelő információ lesz a tesztek elindításához.

A **tesztver átadása a karbantartást végző szervezet részére**. A tesztver archiválása után azt át kell adni az üzemeltető szervezet számára, amely ezt megfelelően dokumentáltan eltárolja és/vagy elkészíti belőle a rendszer éles környezetét támogató un. **support** környezetet. Ez a környezet az éles környezethez hasonló és a célja az, hogy a működés közben talált rendellenességeket ezen vizsgálja ki az üzemeltető csapat.

A **tapasztalatok feldolgozása a jövőbeni termékekhez vagy projektekhez**. A tapasztalatok során mindenképpen végezni kell írásos és szóbeli elemzést is. Az ebből készült összefoglalókat át kell adni a fejlesztő, a projektmenedzsment csapatnak és az üzleti területnek is. Ezekben célszerű ajánlásokat is megfogalmazni az egyes területek számára

Az **információk hasznosítása a tesztfolyamat érettségének fejlesztéséhez**. A tapasztalatok alapján meg kell tenni az ajánlásokat a teszt csapat számára is és ezek alapján fejleszteni kell a teszt folyamatokat.

SZÉCHENYI 2020

A tesztelés pszichológiája

Azt már látjuk, hogy a szoftvertesztelés egy nagyon összetett feladat. Önmagában is önálló szakmaként érdemes értelmezni.

Mint a legtöbb szakma ez is megfelelő szakmai felkészültségű és megfelelő beállítottságú embereket kíván.

A szoftvertesztelés, mint összetett folyamat megkívánja, hogy megvizsgáljuk a tesztelést végzőket személyiségük alapján is.

Ki teszteljen?

Vizsgáljuk meg, hogy a szoftverfejlesztés folyamatában részt vevő különböző szakemberek részvétele a tesztelés folyamatában milyen előnyökkel és hátrányokkal jár.

Független tesztelő szakember alkalmazása a tesztelés során:

- Más szemlélet, máshol vannak a vakfoltok. Nyilván ő nem elsősorban a szoftver kódolása szempontjából vizsgálja a megoldást. A tesztelő elsősorban az igényelt megoldás szempontjából és nem az elkészült szoftver szempontjából vizsgálja azt, így akár az is kiderülhet, hogy valami elkerülte a szoftverfejlesztő figyelmét.

- Nem ellenérdekelt, hiszen neki az a célja, hogy jó minőségű szoftver készüljön. Viszont nem érdekelt abban, hogy a hibák el legyenek fedve.

- Tesztelési technikák ismerete miatt olyan eszközöket és módszereket alkalmazhat, amit egy szoftverfejlesztő nem.

Amikor a szoftver fejlesztője végzi a tesztelést, akkor előny az, hogy

- Ismeri a kódot, így ha talál valami hibát, akkor akár azonnal meg tudja oldani és tudja folytatni a tesztelést. Ez meggyorsítja a tesztelési folyamatot, azonban veszélyes is lehet, ha a fejlesztő nem elég körültekintő a módosítás hatásait tekintve.
- Ismeri a technológiát, így számára a technológiából adódó eszközök eleve adottak.
- Ismeri a követelményeket (elvileg). Hiszen ő volt az, aki azt szoftverkódban implementálta. Azonban a szoftverfejlesztő a követelmények egy adott értelmezésére készíti el az adott szoftvert, így ez akár komoly veszélyt is jelent. Minden esetben célszerű "külső szemmel" is megvizsgálni az elkészített megoldást

SZÉCHENYI 2020

- A jó fejlesztő teszteli a munkáját, így számára egyértelmű, hogy az elkészült kódot vizsgálja át és csak úgy adja tovább.

Függetlenség szintjei

A tesztelés függetlenségének szintjei meghatározzák, hogy a tesztelés során kik és hogyan végzik el a tesztelést. Az adott szoftverfejlesztési projekt és annak kockázati szintje határozza meg, hogy milyen szintű és összetettségű a csapat, amely a tesztelést végzi.

A fejlesztők tesztelik a saját kódjukat. Ekkor lehetőleg tesztelési ismeretekkel rendelkező fejlesztők végzik a saját szoftver tesztelését.

A fejlesztő csapaton belül van tesztelő szakember, aki a tesztelési folyamat tervezésért és végrehajtásáért felel. Ez előnyös akkor, ha a szoftver mérete nem igényli önálló csapat kialakítását. Előnye még az, hogy a csapat tagjaként együttműködve a fejlesztőkkel a céljuk a legjobb minőségű szoftver létrehozása.

Független tesztelő csapat közös vezetéssel, ekkor már a szoftvertesztelő csapat önállóan végzi a szoftvertesztelési tevékenységet, de miután a vezetés közös így gyorsan és hatékonyan megtörténhet a problémák átbeszélése. Ha szükséges, akkor a beavatkozás is egyszerű a közös vezetés által.

Független tesztelők alkalmazása a felhasználótól bizonyos tesztfolyamatok esetében nagyon hasznos, hiszen ők tisztában vannak a való életbeli folyamatokkal és a felhasználás tényleges módjával, így azonnali visszajelzést tudnak küldeni a fejlesztő csapat számára. Elsősorban funkcionális tesztek végrehajtására érdemes az ilyen csapatokat használni.

Szakosodott tesztelői szakértő alkalmazása elsősorban speciális tesztelési feladatoknál célszerű, ilyen például a terheléses teszt, hálózati tesztek, speciális integrációs tesztek. Ezek a szakemberek általában nem üzleti területre, hanem teszt metodikára specializálódnak és ebben tudják a legnagyobb segítséget nyújtani a projekt számára.

Külső tesztelő cég alkalmazása általában a legdrágább megoldás, de az ezekben együttlévő specifikus tudás minden területen képes segíteni a projektet. Ebben az esetben azonban már a projektmenedzsmentnek kell összehangolni a különböző szállító cégek munkáját, ami mindenképpen többleterőforrás igény.

Milyen a jó tesztelő?

A jó tesztelő nem csak szakmailag, hanem emberileg is alkalmas erre a feladatra. Tekintsük át az ehhez szükséges személyiségjegyeket:

- Kíváncsiság, hogy az új feladatok érdekeljék. Olvasson utána az adott területnek, képes legyen jó kérdéseket feltenni és azokat megfelelően szortírozni.
- Szakszerű pesszimizmus, hogy ne "bízzon meg" abban, hogy ha valaki állít valamit, hanem objektív módon alá is tudják támasztani azt.
- Kritikus szemlélet, az üzleti igénnyel, a csapattal és a megoldással szemben. Így tud olyan kérdéseket feltenni, ami elgondolkodtatásra ösztönzi a szereplőket, a legjobb megoldás megtalálása érdekében.
- Részletekre való odafigyelés, hiszen nem lehet elsiklani azok felett a megfelelő minőség érdekében.
- Tapasztalat a kommunikációhoz, hiszen a tesztelés során a cél az, hogy a hibákat megtaláljuk, és ha valakinek a munkájában hibát találunk azt általában nehéz kommunikálni, hiszen az védekezésre készteti az adott embert. Ha azonban világos a közös cél és valamiért és nem valakik ellen dolgozik a csapat, akkor az ilyen szituációkat is könnyebb kezelni.
- Tapasztalat a hibasejtéshez. Rendelkezik azzal az elemző képességgel, amely tanulva a korábbi tapasztalatokból előre képes megmondani, hogy mik egy rendszernek a kritikus pontjai és oda koncentrálni a megfelelő erőforrásokat a hibák csökkentésének érdekében.

Követelmény egy tesztelő számra:

- Olvasási készség, amellyel képes értelmezni a dokumentumokat, az üzleti igényt, a szükséges eszközöket. Ehhez megfelelő tapasztalat kell.
- Széles látókör és érdeklődés a világ iránt, hiszen így a máshol megismert ismerteket is be tudja építeni a napi munkájába.
- Jelentések elkészítésének készsége, amellyel következetes és lényegre törő, a fontos dolgokat kiemelő jelentéseket képes készíteni.
- Megfelelő szintű jártasság
    - Az alkalmazási és üzleti területen, hogy az adott területen tudjon a "felhasználó fejével" gondolkodni. Értse mi az feladat, amire a szoftver készítik.
    - A technológiában, annak érdekében, hogy tudjon kommunikálni a fejlesztővel, javaslataival, ötleteivel segítse a hibakeresési folyamatot.

- A tesztelésben, hogy szakmailag megfelelő módon közelítve meg a feladatot, a lehető leghatékonyabban képes legyen a munkáját elvégezni.

Kommunikációs problémák elkerülése

A tesztelés során a felmerülő hibák és problémák kommunikációja fontos feladata a tesztelő csapatnak.

A kommunikáció módja elsődleges annak érdekében, hogy a fejlesztő csapat azt érezze, hogy a hibák megtalálása a közös cél – a jó minőségű szoftver elkészítése – érdekében történik és nem azért, hogy bárki munkáját kritizálják.

Ezért az alábbi kommunikációs eszközöket célszerű alkalmazni:

- Együttműködő szándék jelzése, a közös cél elérése érdekében.
- A probléma tárgyilagos előadása. Objektíven. Nem a felelős keresése, hanem a cél elérése érdekében.
- Meg kell próbálni megérteni a másik felet, hogy megértsük a probléma gyökerét és megtaláljuk azt a módot, hogy ezt a későbbiekben el lehessen kerülni.
- Ellenőrizzük, hogy értjük-e egymást. Minden esetben kérdezzünk vissza és több oldalról is vizsgáljuk meg a kérdést, hogy biztosan ugyanazt értjük egy adott probléma alatt.
- A jót is jelezzük vissza, ezzel a csapat kohézióját erősítsük és adjunk pozitív visszacsatolást a csapat tagjai számára.

Etikai kódex

A szoftvertesztelőnek, mint egy fontos szakma képviselőjének rendelkeznie kell azzal az etikai megközelítéssel, amivel elismertté válhat az informatikai szakmai és a piac többi szereplője számára.

A szoftverek tesztje során a résztvevők bizalmas és kiváltságos információkhoz juthatnak hozzá. Az etikai kódexre többek között annak biztosítására van szükség, hogy az információkat nem használják fel illetéktelenül.

Elismerve az ACM és az IEEE mérnökökre vonatkozó etikai kódexet, az ISTQB® a következő etikai kódexet határozza meg (HSTQB, 2014):

KÖZÉRDEK – A képesített tesztelőknek következetesen a közérdeknek megfelelően kell tevékenykedniük.

MEGRENDELŐ ÉS MUNKAADÓ – A képesített szoftvertesztelőknek úgy kell tevékenykedniük, hogy a megrendelőik, illetve munkaadóik igényeit legjobban kiszolgálják, ugyanakkor a közérdekkel ne kerüljenek szembe.

TERMÉK – A képesített szoftvertesztelőknek biztosítaniuk kell, hogy az általuk tesztelt, átadásra kerülő termék, vagy rendszer megfelel a legmagasabb szakmai szabványoknak.

VÉLEMÉNY – A képesített szoftvertesztelőknek feddhetetlennek és függetlennek kell maradniuk a szakmai véleményalkotáskor.

MENEDZSMENT – A képesített szoftver tesztmenedzsereknek és vezetőknek azonosulniuk és támogatniuk kell az etikai kódexet a szoftvertesztelés menedzsmentje felé.

SZAKMA – A képesített szoftvertesztelőknek elő kell segíteniük a szakma hírnevét és feddhetetlenségét a közérdeknek megfelelően.

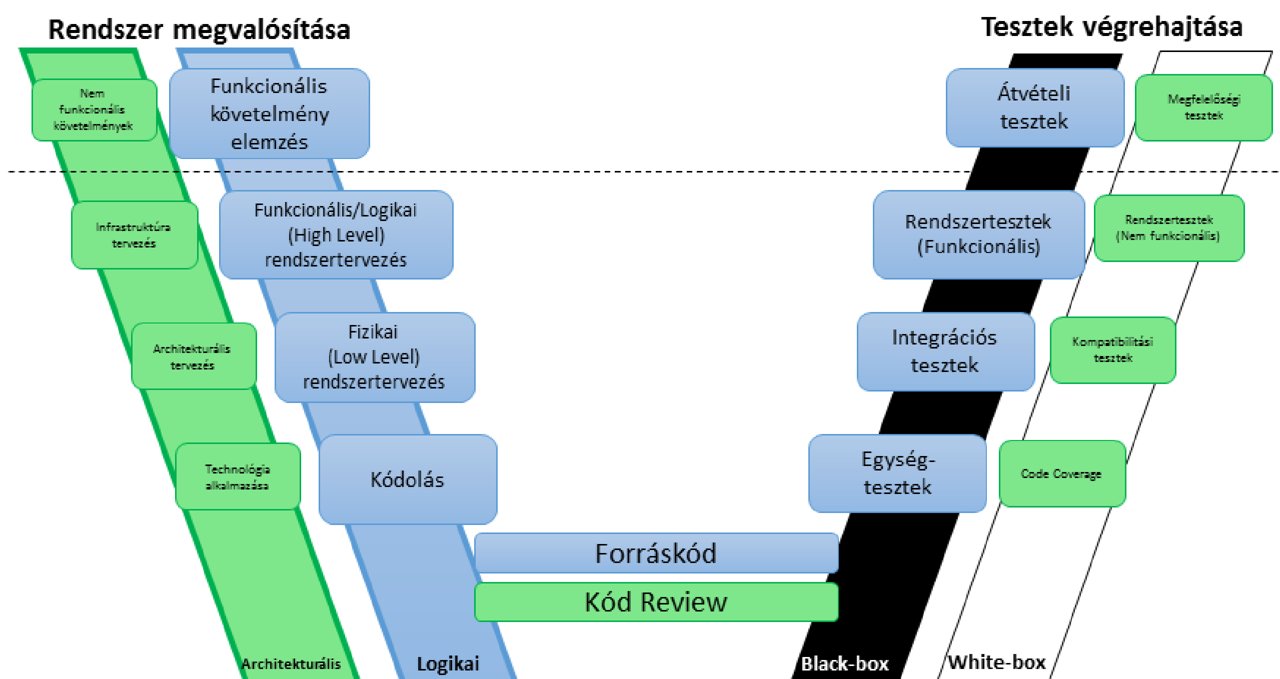
MUNKATÁRSAK – A képesített szoftvertesztelőknek korrektül és támogatóan kell fellépni a munkatársaikkal szemben és segíteniük kell a szoftverfejlesztőkkel való együttműködést.

SZEMÉLYES – A képesített szoftvertesztelőknek életük végéig tanulniuk kell a szakmájukban és munkájuk végzése során figyelniük kell arra, hogy az etikai kódex a munkájuk részévé váljon.

Szoftverfejlesztési modellek

V-Modell

A V-Modell egy korai szoftver fejlesztési modell, mely a vízesés modellt egészíti ki a verifikációs és validációs fázisokkal. Számos változata létezik, melyek közös tulajdonsága, hogy az egyes megvalósítási fázisokat megfelelteti az ellenőrzési és tesztelési fázisoknak. Ma már inkább egy hasznos elméleti modellként tartjuk számon, mely szisztematikusan felsorolja az egyes szinteken elvégzendő és ellenőrizendő feladatokat, bár vannak, akik továbbra is a V-Modell rendíthetetlen hívei.



Az elterjedtebb agilis módszerek újra definiálták a szoftverfejlesztés folyamatát, azok azonban sokkal inkább a munkaszervezésre koncentrálnak, mint a jelentkező feladatokra. Így a V-Modellt használhatjuk arra, hogy rögzítsük, hogy egy-egy szoftverrész kidolgozásakor milyen lépéseket kell végrehajtanunk, mely feladatok hogyan épülnek egymásra. Például, találkozhatunk olyan problémával, mely megkívánja a részletes és teljes egységteszt lefedettséget. Ekkor úgy határozzuk meg az egységtesztek fókuszát, hogy az ilyen osztályok, metódusok azonnal a lehető legnagyobb teszt lefedettséggel rendelkezzenek. Azokat a részeket pedig, melyeknél inkább az együttes működés érdekes, ellenőrizzük integrációs szinten. A V-Modell segítségével mindig tudni fogjuk, hogy milyen szinten dolgozunk, és az adott projekt esetén milyen elvárásokat fogalmaztunk meg az adott szinten.

Azonban az új irányzatok előretörésével sok kritika fogalmazódik meg a V-Modell-lel kapcsolatban:

- A vízesés modellből ered.
- Rugalmatlan és képtelen jól alkalmazkodni a változásokhoz.
- Helytelenül, lineáris képet fest a szoftver fejlesztés folyamatáról.
- A tesztelést is rugalmatlan folyamatként kezeli, ahelyett, hogy a tesztelőkre bízná a legjobb módszer felkutatását az adott probléma vizsgálatához.
- Sok változatának köszönhetően sok a bizonytalanság is a V-Modell-lel kapcsolatban, mely vitákra adhat okot akár egy-egy projekten belül is.

Tekintsünk akár csak egy kisebb céget, mely egy új üzleti modellt dolgozott ki, és szeretné az új üzletmenetet szoftverrel támogatni. Ha a fenti V-Modell szerint dolgozunk, természetesen a követelmények elemzésével kezdünk. Már itt, az elején problémába ütközünk. A feladat ugyanis az, hogy a rendszerrel kapcsolatos minden igényt rögzítsünk és dolgozzunk ki. Esetünkben az új üzleti modellről kellene minden szoftver elvárást a fejlesztés megkezdése előtt feketén-fehéren megfogalmazni. Az esély a feladat sikertelenségére arányos a feladat méretével. Nézzük meg, miért:

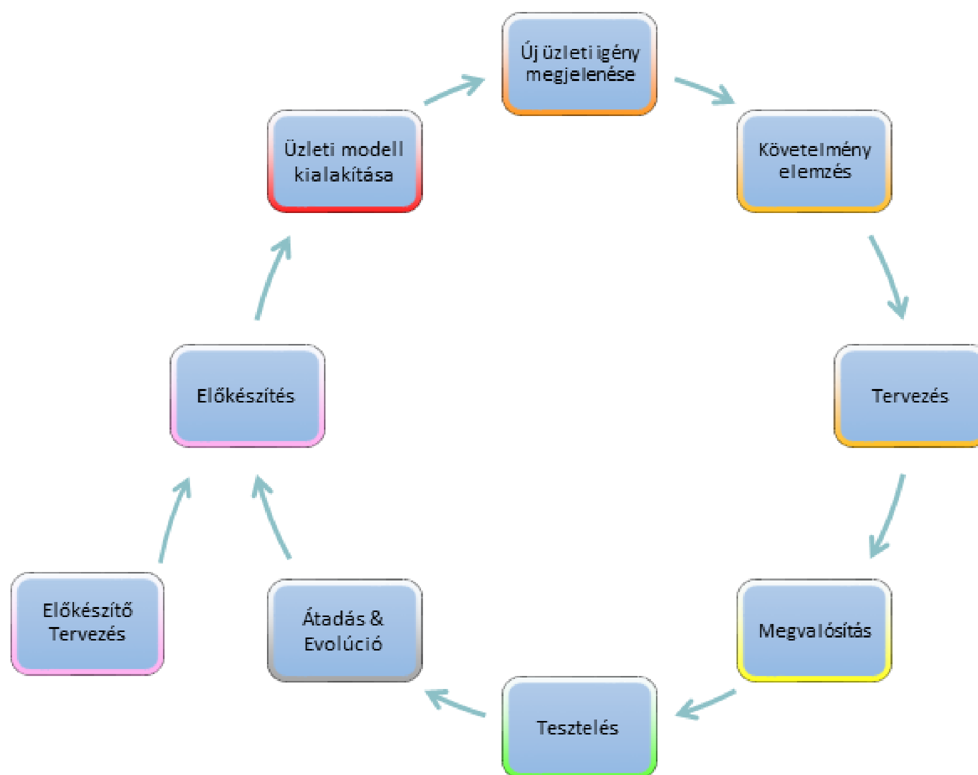
- Az üzleti modell folyamatosan változik.
- Ezáltal vagy a megvalósítás tolódik folyamatosan,
- vagy a rendszer, mely elkészül, nem fog megfelelni a valós (megváltozott) elvárásoknak.
- Ha meg akarunk felelni az elvárásoknak, folyamatosan karban kell tartani az igényeket, mely a megvalósítási fázisok előtt is folyamatos munkát igényel, látható eredmény nélkül.
- A változó igények kezeléséről nem nyilatkozik a modell, így az alkalmivá (ad hoc) és szervezetlenné válhat.

Tételezzük fel, hogy minden nehézség ellenére sikerült rögzíteni a rendszerrel kapcsolatos elvárásokat. A magas és alacsony szintű tervezési fázisokban, valamint a kódoláskor is hasonló problémákkal találkozhatunk, ha a projekt mérete nagy. Tételezzük fel, hogy a projektünk több felhasználói folyamatot, több modul segítségével valósít meg, továbbá a fejlesztés méretei és a határidők megkívánják, hogy több csapat dolgozzon a megvalósításon. A V-Modell arról sem rendelkezik, hogy az egyes csapatok által előállított tervek, termékek vagy résztermékek hogyan kerülnek egyeztetésre a fejlesztés során. Pedig egy hosszú fejlesztési feladat esetén elengedhetetlen a folyamatos konzultáció, az interfészek megfeleltetése.

A főbb problémákat, tehát, a változó igények és a feladat mérete jelentik a V-Modell esetén. E problémákra adnak választ a későbbi iteratív-inkrementális modellek és azok hibrid változatai.

Iteratív-inkrementális fejlesztési modellek

Az első, amit talán tisztázni kell az elnevezés. Mit is jelent a két szó: iteratív és inkrementális? Az **iteratív** jelző azt jelenti, hogy a fejlesztési feladatot önállóan értékelhető és megvalósítható feladatokra bontjuk, melyeket aztán külön-külön azonos folyamat szerint valósítunk meg. E folyamat lépéseit nevezzük az iteráció lépéseinek. Ezt az iterációs folyamatot hajtjuk végre újra és újra előállítva a rendszer egyes **inkrementum**ait. Ezzel el is jutottunk az inkrementális fogalomig, mely azt jelenti, hogy ezeket az inkrementumokat időről időre átadjuk, integráljuk az éles rendszerbe, így növelve a rendszer funkcionalitását.



Hogyan vagyunk képesek ezzel a modellel hatékonyan reagálni a példánkban szereplő új üzleti modell változásaira? Egyrészt a részfeladatokra bontás egyszerűsíti a teljes feladatot és a jelentkező változás megmutatja, hogy melyek azok a részek, amiket érint, és melyek azok, amiket nem. A részfeladatok priorizálhatók, így lehetőségünk van arra, hogy azt a fejlesztést vegyük előre, amiről a legprecízebb információk állnak rendelkezésünkre. Így a legkidolgozottabb rész fog megvalósulni először. Így azzal a feladattal foglalkozunk, mely esetén feltételezhetően kevesebb változás lesz. Továbbá az üzlet is motivált, hogy a számára legfontosabb részeket dolgozza ki a legpontosabban és a leghamarabb, mert azok fognak legelőször megvalósulni. Másrészt pedig tudjuk azt, hogy a folyamat támogatja a változást,

SZÉCHENYI 2020

sőt természetesnek veszi, és azt is megmondja, hogy hogyan kezeljük. Alapvetően a rendszert a megfogalmazott változások (igények) alapján fejlesztjük, és elfogadjuk azt, hogy a rendszerrel szemben támasztott igények annak élete során folyamatosan változnak, és ehhez alkalmazkodni kell. Az iteratív-inkrementális modellek jellemzője, hogy bevezeti a prioritás fogalmát, és megköveteli az igények megvalósítási sorrendjének meghatározását azok prioritása szerint. A prioritást akkor alkalmazzuk helyesen, ha egy listaként tekintünk rá, melyben az elemek szigorúan egymást követik, azaz nincs két azonos prioritású elem.

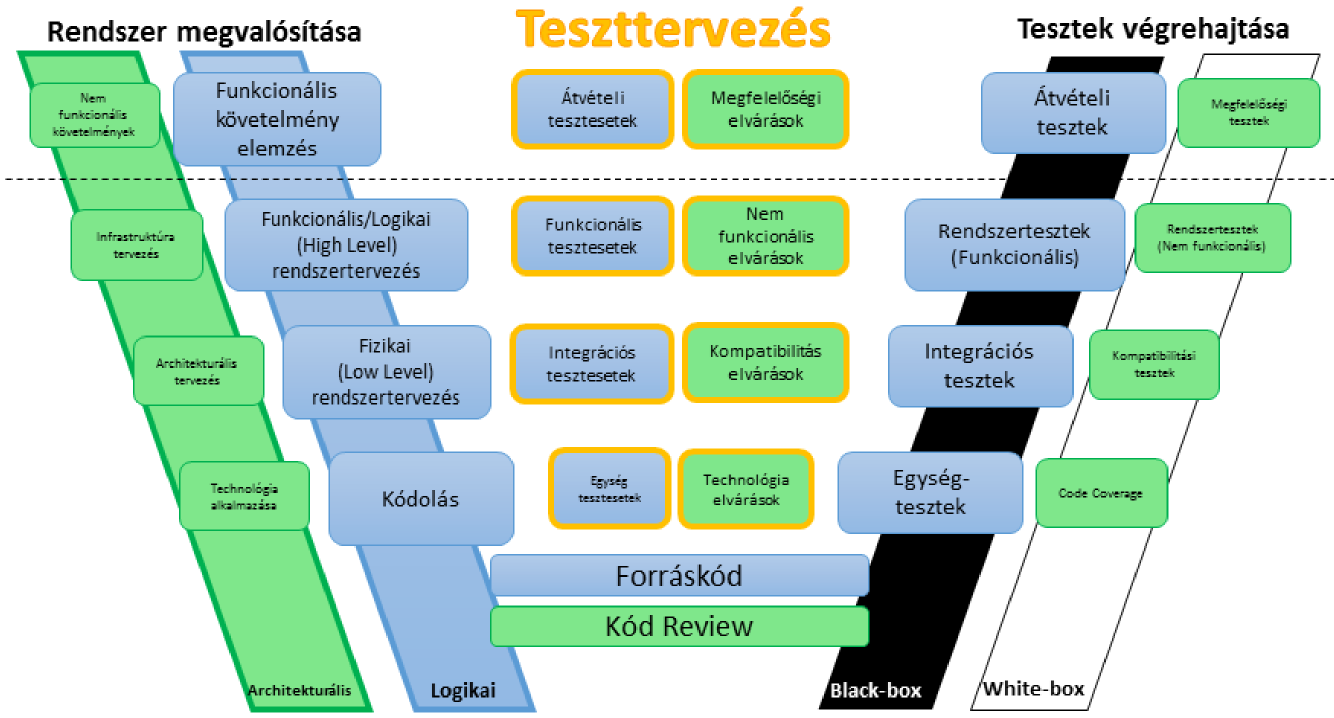
Tesztelés egy életciklus-modellen belül

A tesztelőtől vagy a tesztelői feladatoktól életciklus modellenként vagy fejlesztési módszertanonként más-más elvárásokkal találkozhatunk. A vízesés és V-Modell általában a minőségbiztosító szerepét rója a tesztelőre. Ezt aztán úgy értelmezzük, hogy tulajdonképpen a tesztelő tehet arról, ha hiba van a rendszerben. Azt az irreális elvárást támasztja a tesztelővel szemben, hogy hibátlan rendszert adjon ki a kezei közül. Az agilis módszerek e téren is szemléletbeli változást hoznak, és racionálisan közelítik meg a tesztelő szerepét és annak feladatkörét is. Az agilis tesztelés szerint a minőséget a csapat, mint egység, biztosítja. Az tesztelőnek pedig az a feladata, hogy az adott üzleti feladattól és lehetőségeitől függően a lehető legtöbb olyan hibát vagy problémát megtalálja és jelezze, melyek az ügyfél számára nehézséget jelenthetnek vagy ellehetetlenítik a rendszer használatát. Az agilis módszerek ezért is fektetnek nagyobb hangsúly a tesztelési technikák ismeretére. A tesztelőre bízzák a technikák alkalmazását a fejlesztésnek abban a fázisában, amikor azt a kód vagy a funkcionalitás készültsége lehetővé teszi: Ha megvan a forrás, az információ, ami alapján tesztelni lehet, akkor készüljenek és fussanak a tesztek a lehető leghamarabb.

A korábbi fejlesztési modellek a hibák keresésére korlátozták a tesztelők feladatkörét. Azon belül is sokszor csak a logikai, működésbeli hibákra koncentráltak, mert esetleg csak arra volt idő. Ráadásul sokszor a gyakorlatban mindez csak magas szintű, GUI tesztekkel történt. A modern módszerek kiterjesztik a tesztelők feladatkörét, hatáskörét vagy sokkal inkább kiterjesztik a tesztelési feladatokat. A funkcionális hibakeresésen túl, nem funkcionális (pl. használhatósági) teszteket is végrehajtunk, ellenőrzési és visszajelzési feladatokat végzünk, illetve a magas szintű felületi, funkcionális tesztek esetén javaslattételi feladatokat is elláthatunk, mint tesztelők. Ugyanis egy kényelmetlen, rossz struktúrával felépített képernyő legalább akkora probléma lehet a felhasználó számára, mint egy logikai hiba. Az alacsony szintű (kód közeli) tesztek esetén pedig a tesztelők sokszor maguk a fejlesztők, azaz nincsenek éles határok fejlesztők és tesztelők között. Természetesen a négy szem elvet érvényesíteni kell, azaz lehetőség szerint a kódot fejlesztő ne legyen ugyanaz a személy, aki teszteli is. Ha ez előbbi nem lehetséges, akkor valaki más ellenőrizze a kódot, és a tesztet annak fejlesztőjétől függetlenül.

Az agilis módszerek magukkal hozták a priorizálás fontosságát és jelentőségét, és ez a teszttervezésre is kifejtette hatását. A tesztelőknek meg kell határozniuk a tesztelés fókuszát egy-egy iterációban, és a kritikus részekre fektetni a hangsúlyt. Törekedni kell a minél nagyobb vagy teljes kódlefedettségre, azonban erre sokszor nincs lehetőség vagy nem is szükséges ahhoz, hogy az átadandó rész helyes működését elfogadjuk. Sokkal fontosabb az, hogy mind a fejlesztők, mind a tesztelők tisztában legyenek azokkal a részekkel, melyek hibás működése esetleg ellehetetleníti a használatot és veszélyezteti az adott inkrementum sikerességét. Fontos gyakorlat az agilis módszerek esetén a regresszió. Azaz a tesztek időről időre (akár fordításonként) történő - lehetőleg automatikus - újra futtatása. Ezáltal az egyes változások kontrolláltak lesznek, és mindig beláthatjuk a rendszer egészéről, hogy megfelel a teszteseteknek. Ennek megvalósulásához járul hozzá a CI (Continuous Integration), mely folyamatosan integrált és automatikusan tesztelt állapotban tartja a kódot.

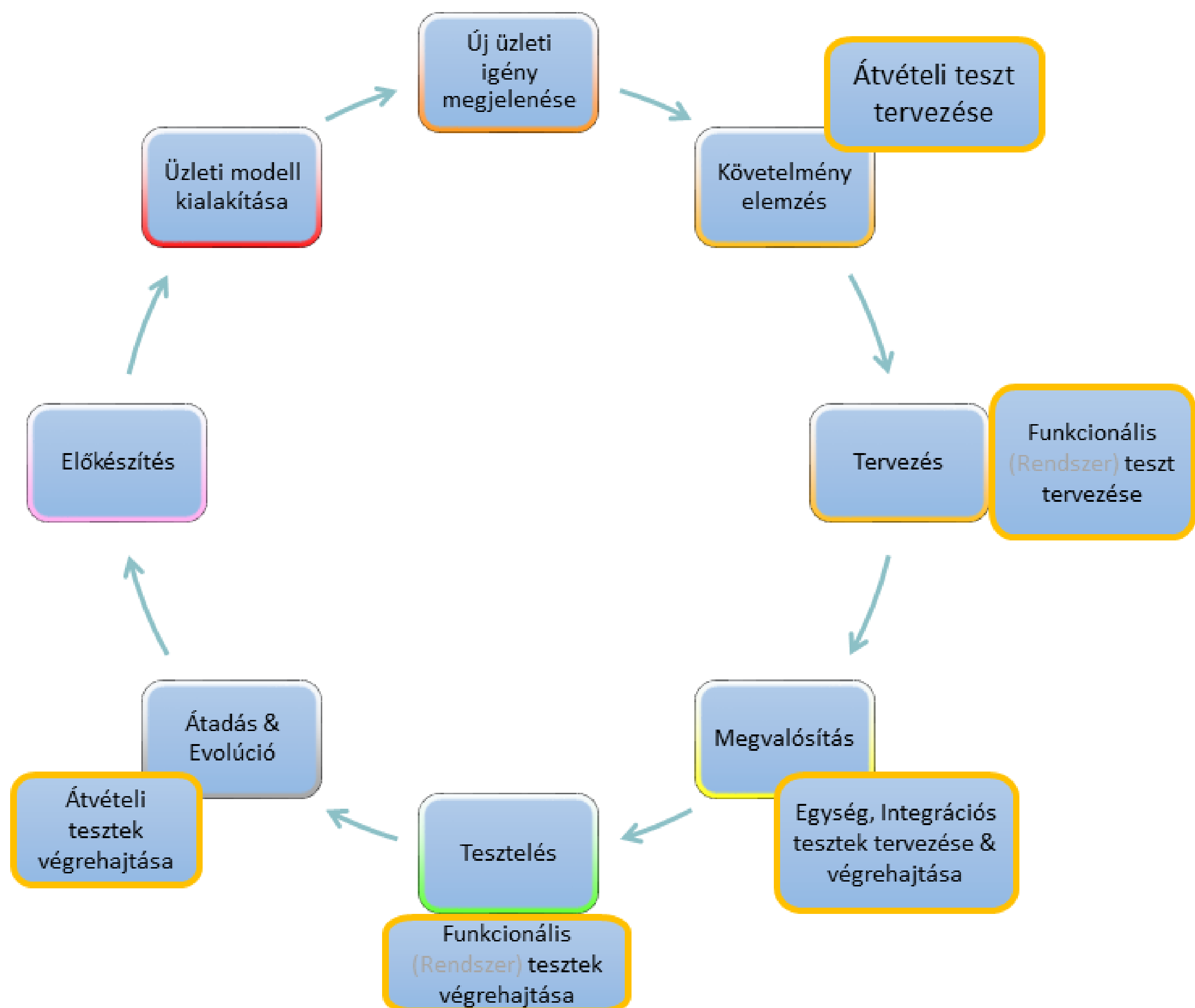
Tekintsünk vissza a V-Modellre, mint sorvezetőre, és vizsgáljuk meg, hogy az egyes szinteken hogyan feleltetjük meg egymásnak a megvalósítás, a teszttervezés és a tesztelés elemeit.



Bár a V-Modell a tesztek végrehajtását időben az adott szint megvalósítása után hajtja csak végre, ez a V-Modell szerint sem jelenti azt, hogy a tesztelők munkája a fejlesztések lezárultával kezdődik. Amint előállnak a funkcionális és nem-funkcionális követelmények, kezdhetjük az átvételi tesztesetek és a megfelelőségi elvárások tervezését. Ugyan ez igaz a funkciók terveire, a felhasználói folyamatokra és

azok magas szintű funkcionális tesztjeire. Még alacsonyabb szinten a moduláris tervezések után integrációs és kompatibilitási elvárásokat fogalmazhatunk meg. A kódolandó programegységek esetében pedig szükség van az interfészeik specifikációja alapján elkészíthető egységtesztek eseteire.

Nézzük meg, hogyan vonja be a tesztelőt a kezdetektől a munkába az iteratív-inkrementális modell illetve hogyan vet fel tesztelési feladatokat a fejlesztési életciklus során.



A V-Modellhez hasonlóan - láthatjuk, hogy a feladatok és azok rendszerezése nem mutatnak nagy eltérést - az idő elteltével egyre mélyebb szinten végezzük a teszttervezési és konkrét tesztelési feladatokat, ahogy azt az elkészülő termékek engedik, majd a validációs fázisokban egyre feljebb haladunk, és eljutunk az átvételi tesztig. Ezért is mondhatjuk, hogy azok a technikák, melyek a V-Modell alapján kapták az elnevezésüket, a többi modellben is megállják a helyüket, csak a munkaszervezésben és az eszközök alkalmazásában tapasztalunk eltérést.

SZÉCHENYI 2020

Helyezzük el a már említett ellenőrzést, visszajelzést és javaslattételt, mint tesztelői feladatokat az előző modellünkben. A cél megint az, hogy a tesztelőket vonjuk be a fejlesztési folyamatba amint lehet, és építsünk az ellenőrző munkájukra a fejlesztés minden szintjén. Ez különösen fontos a példánkban említett új üzlet támogatásakor. A tesztelők esetleges más projektekből hozott szakterületi ismerete illetve az ellenőrző, kritikus szemlélet az új üzleti modell építését túllendítheti a kezdeti fázisok nehézségein, és akár kész, más területeken már bizonyított megoldásokat hozhat a projekt korai fázisaiban. Továbbá az esetleges hiányosságok, felesleges redundanciák és ellentmondások is kiszűrhetők, ha a tesztelő folyamatosan jelen van, és képviseli a megfeleltetést már az igények megjelenésekor. Érvényesülhet a korai fázisokban is javaslattétel akár nem funkcionális megoldások esetén is. Például korábbi tapasztalatokból származó GUI (UX) megoldás vagy felhasználói folyamat.

Új üzleti igény megjelenése
Követelmény elemzés
Követelmény Review
Tervezés
Funkcionális & Fizikai terv Review
Megvalósítás
Kód Review
Tesztelés
Felhasználói dokumentáció Review
Átadás & Evolúció
Validáció
Előkészítés
Üzleti modell kialakítása

Kicsit kivételt képez megint csak a forráskóddal kapcsolatos ellenőrzés, mert azt -, ahogy az egységteszteket is - általában a fejlesztők végzik. Azonban a feladat jellegét tekintve szorosan kapcsolódik a minőségbiztosításhoz és ahhoz, hogy a csapat belássa: jól dolgozott, és az átadandó kód minősége megfelelő.

SZÉCHENYI 2020

Tesztszintek

Ebben a fejezetben funkcionálisan vizsgáljuk az egyes tesztszinteket, melyek a következők:

- Egység vagy komponensteszt
- Integrációs teszt
- Rendszerteszt
- Átvételi teszt

Általában a funkcionális tesztek jelentik a leghangsúlyosabb feladatot egy-egy rendszer tesztelésekor. Minden más aspektust, teszttípust a Teszttípusok fejezetben tárgyalunk, és vizsgáljuk azok gyakorlati jelentőségét az egyes szinteken.

Egységtesztek

Az egységtesztek igénye

Az egységteszt a rendszer legkisebb önállóan működő egységeit teszteli. Ezek az egységek OO környezetben az osztályok és azok metódusai, eljárásorientált környezetben az eljárások és a függvények. A tesztelés korábban a programozás utáni feladat volt, mely csakis az elkészült funkciók felhasználói szintű vizsgálatát jelentette. Azaz a rendszer működése közben tettek kísérletet a hibák felkutatására. Az, hogy ez volt az egyetlen tesztelési technika, a gyakorlatban komoly többletmunkát jelentett. Nézzünk meg néhány okot, melyek életre hívták az alacsonyabb szintű tesztelési technikákat:

- A rendszer legmagasabb szintű felhasználói felülete és az ott működő folyamatok összetett működést produkálnak, melynek tesztelése szintén összetett feladat és karbantartásuk is sok ráfordítást igényel.
- Sokszor egy-egy újrafelhasznált egység több funkcióban és részfunkcióban okozhat helytelen működést. Ennek azonosítása felhasználói szinten nehézkes. Azzal a jelenséggel találkozhatunk, hogy időről időre bizonyos egység módosítások után a magas szintű teszteseteink közül jó néhány elbukik első ránézésre érthetetlen okokból. Vagy hosszas elemzés, vagy a tesztesetek karbantartása szükséges ahhoz, hogy a problémát orvosoljuk.
- A fejlesztési folyamat késői fázisában derül fény a hibás működésre, mert minden egység vagy komponens fejlesztését meg kell várni ahhoz, hogy a felületen tesztelni tudjunk.
- A felhasználói felület tesztjei gyakran csak futás közben jelzik, hogy megváltozott az adott funkcionalitás interfésze, struktúrája. Ezzel szemben alacsonyabb szinten fordítási időben

derülhet fény az adott egység interfészének változására. Egy fejlesztői IDE azonnal jelzi is ezt a fajta változást.

Természetesen a felhasználói folyamatok szükséges tesztjei nem maradhatnak el egy rendszer fejlesztése során, azonban nagyban csökkenthető azok elbukása, sikertelensége és a hibák analizálására fordított munka, ha alacsonyabb szintű teszttechnikákat is alkalmazunk.

Egységtesztek karakterisztikái

Tekintsük át azokat a főbb karakterisztikákat, melyekkel az egységteszteknek (unit teszteknek) rendelkezniük kell. Ezek a jellemzők azért is fontosak, mert az eszközök, melyekkel előállítjuk a rendszert (Maven, Ant), illetve automatizálva, regressziósan végrehajtjuk a teszteket, feltételezik azt, hogy a programozó tisztában van ezekkel a mintákkal, javaslatokkal, és helyesen alkalmazza azokat. A következő jellemzőket szabályokként is tekinthetjük, melyeket be kell tartani ahhoz, hogy egy konkrét projektben értékes egységteszteket készíthessünk:

- **Egy egységteszt pontosan egy programegységet tesztel önállóan**

  Első hallásra triviálisnak tűnhet a megszorítás, azonban ha jobban belegondolunk: mikor működik, például, egy osztály önmagában? A válasz: szinte sosem... Egy rétegelt architektúrában pedig különösen nehéz elképzelni olyan logikát, ami nem épít a perzisztencia rétegre vagy más egyéb használt eszközre, szolgáltatásra. A függőségek viselkedését, azonban, kontrollálnunk kell a tesztesetnek megfelelően.

- **Az egységtesztek nem lépik át saját moduljaik határát**

  Azaz, ha olyan programegységgel találkozunk, mely egy másik modul szolgáltatását használja, biztosak lehetünk benne, hogy valamilyen módon emulálnunk kell majd a másik modulban lévő szolgáltatás viselkedését.

- **Az egységtesztek egymástól függetlenül működnek**

  Minden tesztnek meg van a maga előfeltétele, melyet a futás előtt a teszt számára elérhetővé kell tennünk, és van végfeltétele, melynek a futás után meg kell felelnie. Egyik teszteset sem számíthat arra, hogy egy másik teszteset előállítja számára a kívánt előfeltételeket. Az előfeltételeket mindig a programozó álltja elő az adott teszteset részeként, mint ahogy a végfeltételeket is a teszteset részeként kezeli.

- **Az egységtesztek a futtató környezetüktől függetlenül működnek**

SZÉCHENYI 2020

Erre a megszorításra azért van szükség, mert az egységtesztek általában fordításkor, regressziósan futnak bárhol, ahol a forrás elérhető és fordítható. Ha az egységtesztekkel valamilyen külső adatbázishoz vagy külső rendszerhez szeretnénk kapcsolódni, akkor állandó konfigurációra és/vagy emulációra lenne szükség a külső kapcsolatokat illetően.

- **Az egységteszteknek nincs mellékhatásuk**

  Azaz futásuk után nem látunk változást a futtató környezetben. Elsősorban a globális változókra kell figyelnünk ez esetben. E megszorítást betartva könnyebb lesz betartanunk az előző két pontot is.

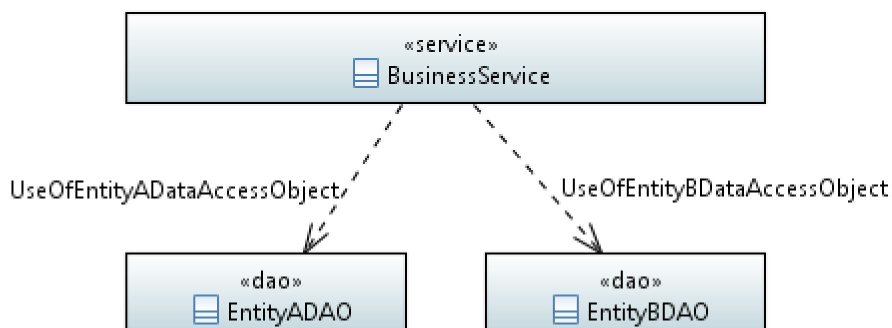
- **Az egységteszteket úgy valósítjuk meg, hogy fordításkor futtathatóak legyenek**

  Azok a tesztek, melyek környezetfüggetlenek, és hatékonyan képesek lefutni, a fordító- és a teszt-keretrendszer képes futtatni bárhol, megállítva a fordítást, ha valamelyik teszt elbukik.

Láthatjuk, hogy az egységteszteknek szigorú szabályoknak kell megfelelniük. E szabályok betartásában segítenek a stubbing és a mocking technikák. A további, magasabb szintek néhány szabály esetében engedékenyebbek lesznek, és a gyakorlatban sokszor inkább az definiálja majd a teszt fajtáját vagy szintjét, hogy e szabályok közül mit, milyen mértékben tartunk be.

Integrációs teszt

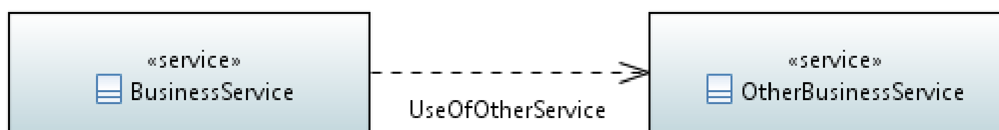
Az integrációs teszt (vagy modulintegrációs teszt) programegységeket, modulokat, azok egymással és a környezettel történő együttműködését teszteli. Osztályok, szolgáltatások és függőségeik valós működését teszteljük, ideértve az esetleges adatelérést is.



Integrációs tesztek esetén a logika és az adatelérés együttes tesztelése az egyik hangsúlyos feladatunk. Miután az egységtesztekkel beláttuk, hogy a logika jól működik, az integrációs tesztekkel kipróbálhatjuk azok perzisztens működését is. Az adatbázis ez esetben nem feltétlenül egy élő, különálló adatbázis. Sok

esetben elégséges egy memória adatbázis is, mely a teszt (-gyűjtemény) indulásakor feláll a szükséges tesztadatokkal (**fixture**), majd a végrehajtás után megszűnik.

Másik fontos feladatunk az integrációs tesztekkel az, hogy a szolgáltatások, logikák, modulok együttes működését vizsgáljuk. Lehetőségünk van olyan tesztek írására, melyben egy-egy programegység a valós működést hajtja végre, másokat pedig valamilyen módon emulálunk.



Rendszerintegrációs tesztek

Vállalati környezet esetén gyakran előfordul, hogy több egymástól fizikailag is különálló alrendszert kell összekapcsolni. E rendszerek tesztjei során találkozunk az integrációs tesztek egy speciális fajtájával, az un. rendszerintegrációs teszttel. Nem keverendő össze a következőkben tárgyalt rendszerteszt szinttel! A rendszerintegrációs tesztek esetében is használhatjuk az integrációs tesztek technikáit: ekkor a stubbing és a mocking a külső rendszerek emulálásában segítenek. Ha betartjuk azt a szabályt, hogy csak saját interfészt emulálunk, akkor a tesztek megvalósítása pontosan ugyanúgy történik, mint modulintegrációs esetben. Tulajdonképpen a külső rendszerek interfészeit 3rd party modul függőségként kezeljük, és elrejtjük egy saját interfész mögé.

Amennyiben a rendszerintegrációs teszt célja az, hogy magát a kapcsolódást és a kommunikációt tesztelje - akár csak egy tényleges adatelérés esetén -, újabb eszközöket kell alkalmaznunk. Tegyük fel, hogy a teszt során a külső SOAP interfésszel rendelkező rendszert meg szeretnénk szólítani, és ellenőrizni akarjuk azt, hogy a távoli kommunikációhoz használt library-k megfelelőek, vagy az interfészek kompatibilisek egymással. Ezt úgy tudjuk megtenni, ha egy un. távoli mockservice-ként emuláljuk a külső rendszert, és valós távoli, hálózati kapcsolatot építünk fel teszt során. A mockservice felállítása történhet például a tesztek futása előtt, és kezelhető, mint előfeltétel (fixture), vagy egy adott környezetben létrehozzuk ezt a bizonyos mock szolgáltatást, és ahhoz kapcsolódunk a tesztünkkel. Ez utóbbi esetben vigyáznunk kell, mert - akárcsak egy külső adatbázis kapcsolat esetén -, környezetfüggővé válik a tesztünk, és vagy paraméterezni kell a tesztben használt távoli elérést, vagy a teszt futását kell az adott környezetre korlátozni pl. teszt gyűjtemények segítségével.

SZÉCHENYI 2020

Az integrációs tesztek futás ideje sokkal nagyobb lehet, mint az egységteszteké. Ennek oka, hogy, amint láttuk modulintegrációs tesztek esetén a memória adatbázis, rendszerintegrációs tesztek esetén az esetleges külső kapcsolatok emulálása erőforrás igényes feladat lehet. Emiatt a megnövekedett futási idő elsősorban az előfeltételek felállításakor jelentkezik, nem a tényleges tesztesetek működésekor.

Néhány technológiai elem, melyeket jó, ha ismerünk ahhoz, hogy alacsony szintű, távoli rendszerintegrációs teszteket készítsünk:

- WSDL, SOAP
- Message Queue
- Remote EJB
- RESTful
- JSON
- Szinkron és aszinkron architektúrák

Függőségek emulálása

Láthatjuk, hogy a jó egységtesztek és bizonyos integrációs tesztek kulcsa az, hogy a tesztelendő programegység függőségeit a tesztesetünknek megfelelően legyünk képesek működtetni, kontrollálni. Ezt úgy tudjuk megtenni, hogy a függőségek valódi példányát lecseréljük egy ugyanolyan interfésszel rendelkező, általunk megvalósított vagy konfigurált másik példányra. Ezáltal nem csak a létező függőségünk esetleges hibáit küszöböltük ki, hanem a hiányos vagy nem létező megvalósítások sem jelentenek problémát.

Egységtesztek esetében minden függőséget kötelezően emulálunk. Integrációs tesztek esetén a teszteset és a függőségek készültsége határozza meg, hogy alkalmazunk-e valamilyen emulációs technikát vagy sem.

Csak saját típust emulálhatunk! Minden 3rd party eszközt, API-t saját interfész mögé kell rejteni, majd függőségként annak egy megvalósítását használni a saját logikánk implementációjakor. Gondoljunk csak a már említett adatelérésre: az ajánlás szerint nem emulálhatjuk az EntityManager-t vagy a JDBC API-t. Ezért vezetjük be a DAO réteget, mellyel elfedjük az adatbázis műveleteket. A saját DAO interfészeinket már tetszőlegesen emulálhatjuk. Láthatjuk, hogy csak azt az alkalmazást tudjuk helyesen tesztelni alacsony szinten, melynek az architekturális felépítése támogatja és segíti a tesztelést.

A függőségek emulálására két technika vált elterjedtté: a **stubbing** és a **mocking**. A mocking az őt megelőző stubbing technikából fejlődött, és adott további lehetőségeket a tesztelők kezébe. Sajnos sokszor találkozhatunk azzal, hogy a két technikát keverik vagy helytelenül használják a két kifejezést.

Stubbing

Az egységtesztek megjelenésével azonnal igényként jelentkezik a függőségek viselkedésének kontrollálhatósága. Az első válasz a stubbing technika, mely úgy működik, hogy az adott függőség interfészét megvalósító saját típust készítünk, és az eredeti megvalósítást lecseréljük az általunk készített stub példányra. Ezáltal a tesztelt programegység számára transzparens módon jelenik meg a tesztelő által megvalósított stub viselkedés. Egy stub implementációban tetszőleges vizsgálatot, eredményt, viselkedést programozhatunk, ám gyakran tesztesetenként más és más működést kell produkálnunk, ami tesztek karbantarthatóságát nehezíti.

Mocking

A mocking technika - vagy sokkal inkább egy-egy mocking keretrendszer - a stubbing technikával újra és újra megvalósított tesztelési részfeladatokat fogja össze és ad egy egységes keretet a tesztesetek megvalósításához. A mocking technika kisebb hangsúlyt fektet a mock objektum tényleges kódolására, inkább speciális un. elvárásokat rögzít adott metódushívásokra, és definiálja, hogy bizonyos paraméterek esetén milyen eredményt adjuk vissza az adott metódus. Ezzel a megközelítéssel inkább a paraméter adatokra és a visszatérés eredményére helyeződik a hangsúly. A mocking terminológiában találkozhatunk azzal, hogy az ilyen típusú elvárásokat stubbingnak nevezik. Ebből is láthatjuk, hogy a mocking keretrendszerek támogatást adnak a gyakori feladatok elvégzéséhez, és sokkal inkább kiterjesztik, standardizálni próbálják a stub/mock objektumok alkalmazását. Szűkebb értelemben véve azt a műveletet, amikor egy mock keretrendszerrel a visszatérés eredményét definiáljuk, nevezhetjük stubbingnak.

Azon túl, hogy a stubbing technikát leegyszerűsítik számunkra, a mocking keretrendszerek további eszközöket adnak a kezünkbe: viselkedéssel (behavioral) kapcsolatos elvárásokat is rögzíthetünk a keretrendszer eszközrendszerével. Például, hogy meghívódott-e a metódus bizonyos paraméterekkel, vagy hányszor hívódott meg az adott metódus. A mocking keretrendszereknek ezt a lehetőségét **ellenőrzés**nek (**verification**) nevezzük, mely új megközelítést, az un. gray-box technikák, kialakulását és elterjedését teszi lehetővé. A **gray-box** technika magába foglalja a black-box és white-box technikák elemeit: specifikáció alapján állítjuk elő a tesztesetet, és vizsgáljuk az eredményeket, továbbá ellenőrizzük is a program belső működését vizsgálva azt, hogy az egyes utasítások hogyan, milyen körülmények között hajtódtak végre. (A stubbing technikával is van lehetőségünk arra, hogy

viselkedéssel kapcsolatos jelenségeket vizsgáljunk, azonban sokkal nehézkesebb, és több programozást igényel.)

Egység- vagy integrációs teszt?

A gyakorlatban találkozhatunk azzal a szituációval, amikor egy egységteszt elkezd "átalakulni" integrációs tesztté. Ennek oka lehet az üzleti logika változása, architekturális változás vagy a teszteset tovább fejlesztéséből adódó változások. Azért fordulhat ez az átalakulás elő, mert ugyanazokkal az eszközökkel és technikákkal dolgozunk mindkét esetben. Hiba az, ha ilyen esetben engedékenyek vagyunk, és hagyjuk, hogy az egységtesztünk "kicsit" integrációs teszt is legyen.

Akkor lesznek helyesek a tesztjeink, ha élesen megkülönböztetjük az egységteszteket és az integrációs teszteket. Ne engedjük, hogy hibrid tesztek kezdjenek elterjedni a rendszerben. Pontosan tudnunk kell minden tesztről, hogy hogyan működik, és melyik tesztelési szinthez tartozik. Ehhez alkalmazhatjuk azt, hogy az integrációs tesztek egy másik névtérbe (package) vagy másik fordítási egységbe (project, artifact) kapnak helyet.

Rendszerteszt

Lépjünk egy szinttel feljebb, és teszteljük a teljes integrált rendszert. Lehetőség szerint nem alkalmazunk emuláló technikákat, a rendszert olyan vagy ahhoz hasonló környezetben teszteljük, ahogy majd éles környezetben is működni fog. A funkcionális rendszerteszt legfőbb jellemzője, hogy oly módon teszteljük a rendszert, mint ahogy azt a felhasználó használni fogja, azaz:

- Felhasználói felületen keresztül (GUI)
  - Kontrollok
  - Képernyő állapotok
  - Üzenetek, hibajelzés
  - Konzisztencia
  - Folyamatok és azok állapotai
- Input és output állományok
  - CSV állományok
  - XML
  - PDF
  - Jelentések (reports)
- Az esetleges alrendszerek állapotait ellenőrizve
- Feljogosítás (authorizáció) a User Story-kban, követelményekben megjelenő szerepek alapján

Rendszerteszt szinten a tesztjelentések alapján be kell látnunk azt, hogy a rendszer olyan állapotban van-e, hogy átadhatjuk az ügyfélnek vagy sem. Azaz minden funkcionális igényt teszteltünk, verifikáltunk. A tesztjelentések minden funkcionális igényt tárgyalnak, és tartalmazzák az ismert hibákat. Ez a dokumentum ad alapot a döntés meghozatalához.

Automatizálás

Rendszerteszt szinten a tesztautomatizálás, azaz a tesztek automatikus futtatása jelenti az egyik legnagyobb kihívást és feladatot. Sokszor a rendszer jellegéből adódóan elégséges, ha maradunk a manuális módszernél, és amint egy-egy újabb funkció elkészül, teszteljük azt és az esetlegesen érintett egyéb funkciókat. Azonban érdemes meggondolni, hogy a rendszer bizonyos részein, funkcióin nem lenne-e praktikus, ha bevezetnénk az automatikus futtatást.

Rendszertesztek automatikus futtatásának előnyei:

- A teszt futtatása gyors és hatékony
- Hosszútávon költség hatékony
- Érdekesebb, mint manuálisan kitölteni újra és újra a formokat
- Az eredmények azonnaliak és könnyen eljuttathatók az érdeklődőknek

Rendszertesztek automatikus futtatásának hátrányai:

- Drága eszközök
- A tesztszkriptek fejlesztése miatt a tesztelés kezdetben kevésbé hatékony
- A tesztek karbantartása sok ráfordítással járhat
- A teszteszközök korlátokkal rendelkeznek

Speciális funkcionális rendszertesztek

Az élet rendszerteszt szinten is hozott néhány praktikus megoldást. Mivel a rendszertesztek futtatása sok ráfordítást igényel - akár a manuális módon, akár automatizálva működünk -, a tesztgyűjtemény futása előtt lefuttatunk néhány speciális tesztet, melyek képet adnak arról, hogy maga a rendszer tesztelhető állapotban van-e.

Smoke-teszt

A "smoke" elnevezés a hardver világból jött, mely arra utal, hogy bekapcsolás után az adott eszköz lángra lobban-e vagy sem. A szoftvertesztelésben egy széles és sekély teszt gyűjteményt jelent, melynek célja, hogy meggyőződjünk arról, hogy a rendszer minden része működőképes és készen áll arra, hogy feladatokat hajtsunk végre vele. A széles jelző azt jelenti, hogy lehetőleg a rendszer minden

SZÉCHENYI 2020

funkcionalitását érintsük. A sekély jelző pedig arra utal, hogy az egyes funkcionalitásokat csak minimális ráfordítással vizsgáljuk. A smoke-tesztet úgy kell tervezni, hogy lehetőleg megmutassa az adatbázis séma helytelenségét, az integrációs kapcsolatok működésképtelenségét és a működésre képtelen komponenseket, szolgáltatásokat. Ha a smoke-teszt elbukik, annak azt kell jelentenie, hogy az alkalmazás architekturálisan alkalmatlan a működésre. Általában automatizálva futtatjuk, mert a rendszer minden funkcionalitását érintenünk kell, és gyors eredményre van szükség.

Sanity-teszt

A sanity-teszt néhány irodalomban - helytelenül - a smoke-teszt szinonimájaként szerepel. Van azonban néhány fontos különbség a két teszttípus céljait és végrehajtását illetően, ami miatt meg kell őket különböztetnünk. A sanity-teszt azt szeretné megmutatni, hogy bizonyos módosítások után van-e egyáltalán értelme tovább tesztelni a rendszert. Ez esetben keskeny és mély tesztről van szó. Ami azt jelenti, hogy a módosítások által érintett funkcionalitást vizsgáljuk részletekbe menően. A sanity-tesztet először általában manuálisan hajtjuk végre, mert egy új vagy módosult funkcionalitást ellenőrzünk annyira, hogy beláthassuk: a további tesztelésnek igenis van értelme.

Átvételi teszt

Az utolsó tesztelési szint, mely elválasztja a rendszerünket a való életben történő működéstől az átvételi teszt szint. Az átvételi teszteket az ügyfél vagy annak megbízottja végzi, tehát a fejlesztői csapattól független tesztelő. Az átvételi tesztek célja nem az, hogy az ügyféllel hibát kerestessünk, hanem az, hogy kiépítsük a bizalmat a rendszerrel és annak a való életben történő alkalmazásával kapcsolatban.

Tágabb értelemben minden olyan ellenőrzést és feladatot ide értünk, ami támogatja az ügyfelet abban, hogy meghozhassa a döntés a rendszer élesítéséről. Általában az átvételi tesztek részét képezik a megfelelési és szabályossági tesztek, melyeket részletesen a Teszttípusok fejezetben tárgyalunk.

Az átvételi teszteket általában más környezetben hajtjuk végre, mint a rendszer szintű teszteket. Ezért érdemes az átvételi teszteket is smoke- illetve, ha módosításról is szó van, akkor sanity-tesztekkel kezdeni, és belátni, hogy van értelme a rendszer további tesztelésének.

SZÉCHENYI 2020

Egy példa alkalmazás

Igény

A fejezetben található konkrét példák a következő igények szerint megvalósítandó rendszert tesztelik.

- Mint számlázó felhasználó, szeretnék a felületen megkeresett és kiválasztott vevő ügyfél számára új számlát kiállítani, mert a napi feladataink döntő többségét ez a művelet képezi.
- Szükség van arra, hogy a rendszer ajánljon fel fizetési határidő dátumot (8 üzleti nappal későbbre), mert ez a legáltalánosabb.
- Ha a felületen nem adjuk meg (kitöröljük) a fizetési határidő dátumot, akkor automatikusan állítsa be a számlán erre a dátumra, az átállítás után lehet, hogy mégis meggondolja magát a felhasználó, és egyszerűbb, ha törléssel jelzi a szándékát, továbbá, ha a rendszer nem kap határidő dátumot bármilyen más interfészről sem, akkor is legyen ez az alapértelmezett érték.
- Legyen lehetőségünk a számla kiállítása közben új ügyfél létrehozására, melyet azonnal megadhatunk a számla vevőjeként, mert gyakran előfordul, hogy új vevő számára állítunk ki számlát.
- A számla tétel származtatott mezői: nettó összes, bruttó összes, áfa összes automatikusan számolódjanak, mert ez a rendszer alapfunkcionalitását képezi.
- A számla származtatott mezői: nettó összes, bruttó össze, áfa összes automatikusan számolódjanak úgy, hogy a számlatételeket összesítik, mert ez a rendszer alapfunkcionalitását képezi.

Számtalan igényt megfogalmazhatnánk még egy számlát kiállító funkcionalitással kapcsolatban. Például az áfa összesítő, amit tekintsünk egy későbbi iterációban megvalósítandó feladatnak. A szemléltetéshez elég lesz ez a néhány egyszerű igény is. A fenti igénylistához a következő képernyő terv és facade interfész készült. A facade interfész metódusaiban a képernyő műveleteihez, paramétereiben és struktúrájában pedig a képernyőn bekért és megjelenő adatokhoz igazodik.

SZÉCHENYI 2020

Kibocsátó:

lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation

Vevő: Ügyfél keresés

Ügyfél Fo	Cím Foo	Adószám
Ügyfél2 F	Cím2 F	Adószám

Ügyfél Fo

Cím Foo

Adószám: Adószám Foo

Létrehoz

Alaphelyzet

Az "Alaphelyzet" gomb törli a formadatokat és a kiválasztást.

A keresés mezőbe beírva a név kezdetét, megjelennek a találatok egy táblázatban, melynek elemei választhatók.

Fizetési mód	Teljesítés	Kelt	Fizetési határidő
Átutalás	2014.05.27.	2014.05.27.	2014.06.08.

A "Fizetési határidő" kezdeti értékét a szolgáltatás rétegtől kapjuk.

A "Létrehoz" gombra kattintva létrehozzuk a begépelt adatok alapján az ügyfelet. Ha az ügyfél adatok a keresőből kerültek be, akkor a gomb disabled állapotú.

Megnevezés	Mennyiség	Egységár	Nettó össz.	ÁFA %	ÁFA össz.	Bruttó össz. (Ft)	
Termék Foo1	1 db	2000 Ft	2000 Ft	27%	540 Ft	2540 Ft	-
Termék Foo2	2 db	1500 Ft	3000 Ft	27%	810 Ft	2810 Ft	-

\+

'Termék Foo2' tétel törlésének visszavonása

Nettó összesen: 5000 Ft

ÁFA összesen: 1350 Ft

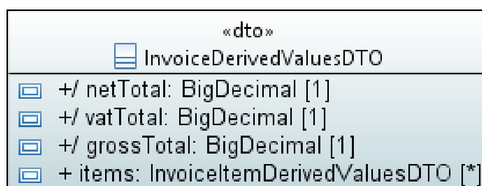
Bruttó végösszeg: 6350 Ft

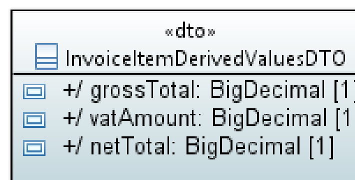
Mentés

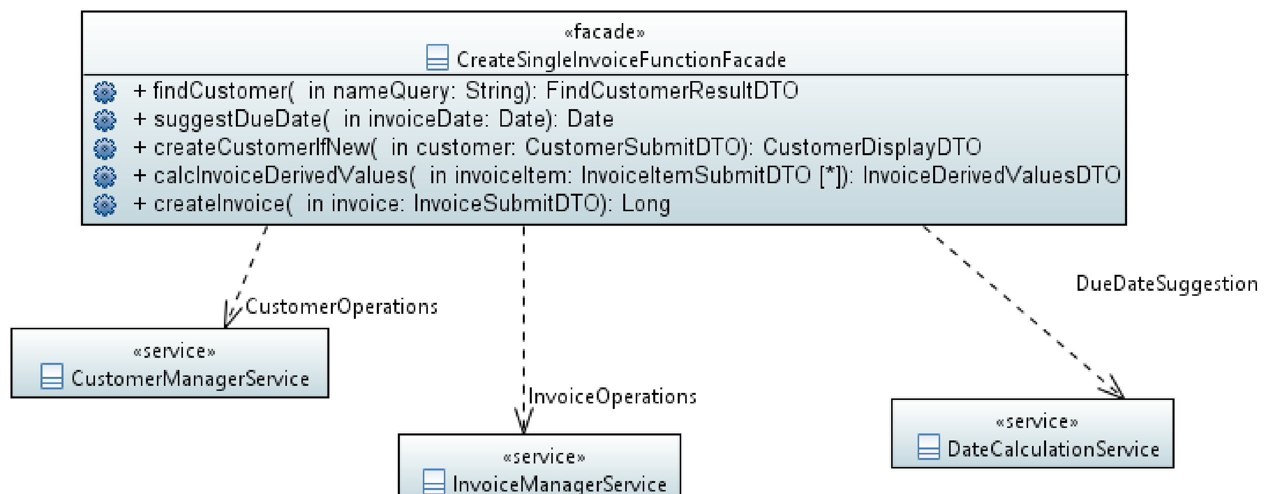
A "+" gombra kattintva újabb tétel jelenik meg a táblázatban, melynek adatai a cellákban kitölthetők. A származtatott értékek szolgáltatás segítségével számolódnak.

A "Mentés" gomb létrehozza a számlát a szolgáltatás segítségével.

Ha megnyomtuk a "-" gombot, a tétel törlődik a felületen, és megjelenik a "Tétel törlésének visszavonása" felirat, melyre kattintva a tétel újra megjelenik. A felirat 3 másodperc múlva elhalványul, és a tételt ezután ténylegesen töröljük a számla objektumból.



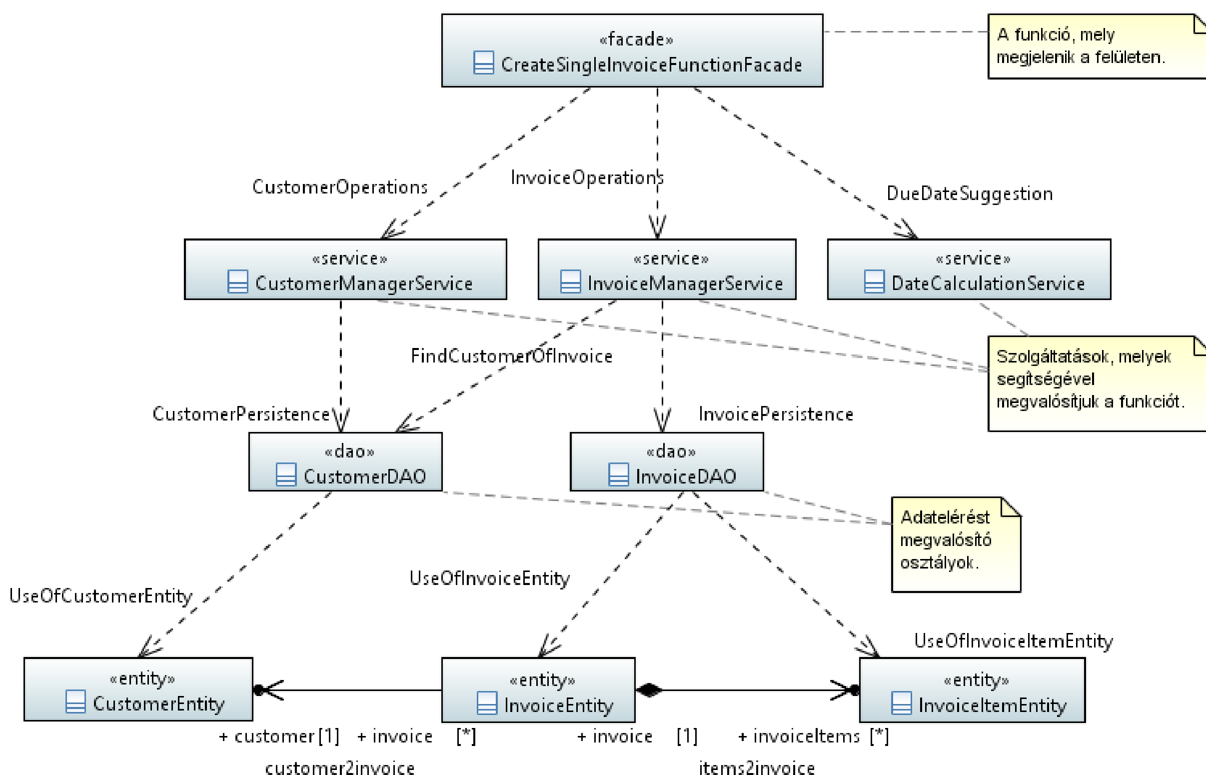




SZÉCHENYI 2020

Architektúra

A teljes, rétegelt architektúra az alábbiak szerint néz ki:



Az egyes rétegek tesztjeit az alábbi szinteken végezhetjük:

Réteg	Egységteszt	Integrációs teszt	Rendszerteszt	Átvételi teszt
UI	N	N	I	I
<<facade>>	I	I	I	N
<<service>>	I	I	N	N
<<dao>>	N	I	N	N

xUnit

Az programozói szintű teszttechnikák eszközeit és az azokhoz kapcsolódó filozófiai, módszertani elgondolásokat az "xUnit" elnevezés fogja össze. A fentebbi okok miatt szinte minden nyelven, platformon megvalósították azokat a konkrét eszközöket, melyekkel hatékonyan tudunk alacsony szintű teszteket írni. Minden xUnit eszközök őse a Smalltalk világában jelent meg Kent Beck tollából SUnit

névvel. Hamarosan követte a jUnit, mely már egy jóval elterjedtebb platform - a Java - világába hozta el az egységteszteket.

Az xUnit és az egységtesztek célja, hogy a programozó számára keretet adjon ahhoz, hogy hatékony, regressziós teszteket írjon fókuszálva egy adott egységre, annak specifikációjára és megfelelő működésére. Az xUnit eszközök a Driver szerepét töltik be a Driver & Stub technikák esetében.

Az xUnit eszközökre vonatkozó ajánlás az alábbi eszköz architektúrát javasolja az egyes megvalósítások számára:

- **Tesztfuttató keretrendszer (Test Runner)**
  Feladata, a tesztek futtatása egy adott xUnit keretrendszerben és a teszteredmények rögzítése. A tesztfuttató gondoskodik arról, hogy egy teszt példány futtatásra alkalmas állapotba kerüljön, meghatározhatja a futtatandó tesztek körét, illetve a teszteredményeket az általa meghatározott módon gyűjti össze és jelenítteti meg az xUnit keretrendszerrel.

- **A teszt megvalósítását támogató eszköz (Test / Test Case)**
  A teszteset implementációjakor szükségünk van arra a eszközrendszerre, mellyel az xUnit keretrendszer támogatja a tesztek fejlesztését. Ez jelenthet egy ős osztályt, annotációkat vagy egyszerű függvénygyűjteményt. Ezek alkalmazására egy dedikált programegységben van lehetőségünk, amit tesztként jelölünk meg az xUnit keretrendszer és a futtató számára.

- **Teszt előfeltételeket támogató eszköz (Test Fixtures)**
  Feladata, hogy a teszt futása előtt a teszteset számára megfelelő, előírt állapotba hozza az alkalmazást, majd a teszt futása után eltüntessen minden, a teszt előfeltételeivel és futásával kapcsolatos változtatást. A mesterséges (laboratóriumi) környezet felépítése és lebontását szeretnénk ezzel az eszközrendszerrel megvalósítani.

- **Tesztfutás és életciklus (Test Execution & Lifecycle**)
  A teszt előfeltételeit és a futás által hátrahagyott módosításokat vagy a lefoglalt erőforrásokat a teszt-keretrendszer által kezelt életciklus metódusokkal kezelhetjük. A tesztek futásának függetlenségét biztosíthatjuk ezzel az eszközzel.

- **Tesztgyűjtemények (Test Suites)**
  A tesztgyűjtemények segítségével lehetőségünk van csoportosítani a teszteket. A csoportosítás alapját képezhetik logikai, infrastrukturális vagy környezeti tényezők. Legtöbbször a tesztek

SZÉCHENYI 2020

előfeltételeinek egyezését tekintjük a csoportosítás alapjának, és minden egyéb szempontot kategorizálással vagy címkézéssel valósítunk meg.

- **Teszt elvárások (Assertions)**

  A teszt futásának utolsó fázisai között szerepel az, amikor a vizsgált programegység vagy folyamat eredményét megfeleltetjük egy elvárt állapotnak. Az elvárások általában valamilyen logikai feltételt megvalósító függvények, melyek vagy "sikeres" vagy "elbukott" eredményt adnak. Így jelezik a programhibákat és befolyásolják a teljes teszt futásának sikerességét. Egy teszt több elvárást is definiálhat. Azt, hogy egy elvárás nem teljesült és a teszt elbukott általában valamilyen kivétellel jelzik a tesztfuttatók a keretrendszer számára.

- **Tesztjelentés**

  A teszteredményeket valamilyen formában elérhetővé kell tenni a teszt-keretrendszert futtató környezet számára. Erre és a jelentés formátumának testre szabására ad támogatást maga az xUnit megvalósítás.

Tekintsük át a jUnit segítségével, hogy az xUnit architektúra hogyan jelenik meg Java platformon.

jUnit

"Never in the field of software development have so many owed so much to so few lines of code."

Martin Fowler a jUnitról

jUnit Test Class & Test Method

A jUnit alap eszközei a tesztosztály és a tesztmetódus. A tesztosztály egy POJO, melynek metódusait annotációkkal láthatjuk el. A jUnit futtató rendszere a `@Test` annotációval ellátott tesztmetódusokat fogja futtatni.

```
import org.junit.Test;

public class Test000Basic {

        @Test
        /**
         * Minden teszteset leírását az adott metódus
         * előtt megjegyzésben részletezzük
         */
        public void testMe() {
                System.out.println("Hello Test World");
        }
}
```

Az első példánkban a `DateCalculationService` szolgáltatást hívjuk jUnit tesztből. Ennek a szolgáltatásnak nincsenek függőségei, így bátran írhatunk egységteszteket anélkül, hogy bármiféle emulációs technikát kellene alkalmaznunk. A jUnit alap eszközeit e szolgáltatás segítségével tekintjük át.

SZÉCHENYI 2020

```
import org.junit.Test;
...
/**
 * A tesztosztály a banki napok kalkulációját ellenőrzi,
 * a DateCalculationService.calculateBusinessDay metódusának
 * működését teszteli.
 */
public class Test001DateCalculationService {

        private static DateCalculationService dateCalculationService =
                new DateCalculationServiceImpl();;

        @Test
        /**
         * Az adott nap (fromDate) és a napok számából (days)
         * eredő banki nap ugyanazon a héten van.
         */
        public void testSameWeek() {

          // Tesztadatok
          Date fromDate = new GregorianCalendar(
            2014, Calendar.MAY, 27).getTime();
          int days=2;

          // Tesztelendő programegység
          dateCalculationService.calculateBusinessDay(fromDate, days);
        }
}
```

Hogyan alkalmazzuk helyesen a tesztosztályokat és a tesztmetódusokat, hogy megfeleljünk a korábbi szabályoknak?

- Egy tesztosztállyal egy adott üzleti osztályt vagy szolgáltatást teszteljünk. Egy üzleti osztályhoz készíthetünk több tesztosztályt.
- Egy tesztmetódus egy teszteset.

Több tesztmetódus

Lehetőségünk van egy tesztosztályban több tesztmetódust definiálni. Ezek futása egymástól független. A példányszintű tagok minden tesztmetódus saját hatáskörének tekinthetők, mert minden tesztfutás új

SZÉCHENYI 2020

tesztosztály példányt hoz létre. A statikus tagok a szokásos globális hatáskörrel jelennek meg. Alkalmazásukkor meg kell győződnünk arról, hogy csak olvassuk a referenciát, illetve csak használjuk az állapotot vagy állapottal egyáltalán nem rendelkezik a statikus példány.

A lenti példában azt látjuk, hogy a szolgáltatás nem rendelkezik állapottal és értéket sem adunk a statikus változónak, így használhatjuk a tesztmetódusokban.

```
import org.junit.Test;
...
/**
 * A tesztosztály a banki napok kalkulációját ellenőrzi,
 * a DateCalculationService.calculateBusinessDay metódusának
 * működését teszteli.
 *
 */
public class Test002DateCalculationServiceMoreTestMethods {

        private static DateCalculationService
         dateCalculationService =
                new DateCalculationServiceImpl();;

        @Test
        /**
         * Az adott nap (fromDate) és a napok számából (days)
         * eredő banki nap ugyanazon a héten van.
         */
        public void testSameWeek() {

          Date fromDate = new GregorianCalendar(
            2014, Calendar.MAY, 27).getTime();
          int days=2;

          dateCalculationService.calculateBusinessDay(
         fromDate, days);
        }

        @Test
        /**
         * Az adott nap (fromDate) és a napok számából (days)
         * eredő banki nap között van egy hétvége.
         */
        public void testOneWeekendBetween() {
```

SZÉCHENYI 2020

```
          Date fromDate = new GregorianCalendar(
            2014, Calendar.MAY, 27).getTime();
          int days=6;

          dateCalculationService.calculateBusinessDay(
         fromDate, days);
        }
}
```

jUnit Assert

Egy teszteset végfeltételeit un. "assert"-ek segítségével definiáljuk. Egy tesztesethez tartozhat több assert is, melyek az adott eset működésének több aspektusát vizsgálják.

```
import org.junit.Test;
import static org.junit.Assert.*;
...
/**
 * A tesztosztály a banki napok kalkulációját ellenőrzi,
 * a DateCalculationService.calculateBusinessDay metódusának
 * működését teszteli.
 *
 */
public class Test003DateCalculationServiceAsserts {

        private static DateCalculationService
         dateCalculationService =
                new DateCalculationServiceImpl();;

        @Test
        /**
         * Az adott nap (fromDate) és a napok számából (days)
         * eredő banki nap ugyanazon a héten van.
         */
        public void testSameWeek() {

          // Tesztadatok
          Date fromDate = new GregorianCalendar(
            2014, Calendar.MAY, 27).getTime();
          int days=2;
```

SZÉCHENYI 2020

```
            // Tesztelendő programegység
            Date calculateBusinessDay = dateCalculationService.
              calculateBusinessDay(fromDate, days);

            // Elvárt eredmény és ellenőrzése
            Date expectedBusinessDay = new GregorianCalendar(
              2014, Calendar.MAY, 29).getTime();
            assertEquals(expectedBusinessDay, calculateBusinessDay);;
          }

          @Test
          /**
           * Az adott nap (fromDate) és a napok számából (days)
           * eredő banki nap között van egy hétvége.
           */
          public void testOneWeekendBetween() {

            // Tesztadatok
            Date fromDate = new GregorianCalendar(
              2014, Calendar.MAY, 27).getTime();
            int days=6;

            // Tesztelendő programegység
            Date calculateBusinessDay =
              dateCalculationService.calculateBusinessDay(
               fromDate, days);

            // Elvárt eredmény és ellenőrzése
            Date expectedBusinessDay = new GregorianCalendar(
              2014, Calendar.JUNE, 4).getTime();
            assertEquals("Business day does not match
             the expected one!",
              expectedBusinessDay, calculateBusinessDay);
          }

}
```

jUnit életciklus

A következő példában kipróbáljuk az osztályszintű (`@BeforeClass` és `@AfterClass`) valamint a példányszintű (`@Before` és `@After`) életciklus metódusokat. A megfelelő annotációval ellátva egy

SZÉCHENYI 2020

tetszőleges metódust, a keretrendszer az életciklus adott fázisában lefuttatja azt számunkra. Az életciklus metódusokkal van lehetőségünk arra, hogy a tesztosztályt illetve a tesztpéldányt a tesztmetódusok futtatására alkalmas állapotba hozzuk. Az osztályszintű életciklus metódusokkal olyan eszközöket hozhatunk létre, mely minden tesztmetódus számára elérhető. Vigyáznunk kell az osztályszintű tagokkal, mert azok állapota nem változhat a tesztek futása során! Ugyanis, ha változna az a tesztek függetlenségét sértené. Az osztályszintű tagok általában infrastrukturális és/vagy architekturális objektumokat hivatkoznak. A lenti példában ilyen eszköz lesz a tesztelt `DateCalculationService` szolgáltatás példány. A példányszintű életciklus metódusokkal az előfeltételeket építhetjük fel és bonthatjuk le, melyek minden egyes tesztmetódus futása előtt és után működésbe lépnek. Ezáltal teszik lehetővé számunkra, hogy ugyanabba a tesztosztályba szereplő tesztek ugyanazzal az előfeltétel rendszerrel fussanak. A különböző tesztmetódusokban szereplő eltérő tesztadatokkal tudjuk az egyes teszteseteket futtatni. Példánkban az egyik - állandó - tesztadat (`fromDate`), az elvárt eredmény (`expectedBusinessDay`) és a szolgáltatás visszatérési értéke (`calculateBusinessDay`) kap helyet példány szinten. A különböző tesztesetek csak a napok számában térnek el, ezért azok szerepelnek a tesztmetódusokban, mint tesztadatok.

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Assert;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
...
/**
 * A tesztosztály a banki napok kalkulációját ellenőrzi, a
 * DateCalculationService.calculateBusinessDay metódusának működését teszteli.
 *
 */
public class Test004DateCalculationServiceLifeCycle {

        private static final Logger log = Logger.getLogger(
         Test004DateCalculationServiceLifeCycle.class.getName());

        private static DateCalculationService
         dateCalculationService;
```

```
/**
 * Példány (tesztfutás) szintű adattag!
 * Az induló dátum minden teszteset esetén ugyan az,
 * így a @Before metódusban állítjuk be.
 */
private Date fromDate;

/**
 * Példány (tesztfutás) szintű adattag!
 * Az elvárt dátum tesztesetenként eltér,
 * így azt a @Test metódusokban állítjuk be.
 */
private Date expectedBusinessDay;

/**
 * Példány (tesztfutás) szintű adattag!
 * A @Tesztmetódusok futása során előállított érték,
 * melyet vizsgálni fogunk esetenként.
 */
private Date calculateBusinessDay;

@BeforeClass
public static void initTestClass() {
  dateCalculationService = new DateCalculationServiceImpl();
  log.info("Date calculation tests started...");
}

@Before
public void initTestRun() {
  fromDate = new GregorianCalendar(
    2014, Calendar.MAY, 27).getTime();
  log.info("fromDate day of week: " + sdf.format(fromDate));
}
```

SZÉCHENYI 2020

```
@Test
/**
 * Az adott nap (fromDate) és a napok számából (days)
 * eredő banki nap ugyanazon a héten van.
 */
public void testSameWeek() {

  int days = 2;
  log.info("Next bank day in " + days + " bank days");
  calculateBusinessDay = dateCalculationService.
    calculateBusinessDay(fromDate,days);
  expectedBusinessDay = new GregorianCalendar(
    2014, Calendar.MAY, 29).getTime();
}

@Test
/**
 * Az adott nap (fromDate) és a napok számából (days)
 * eredő banki nap között van egy hétvége.
 */
public void testOneWeekendBetween6() {

  int days = 6;
  log.info("Next bank day in " + days + " bank days");
  calculateBusinessDay = dateCalculationService.
    calculateBusinessDay(fromDate,days);
  expectedBusinessDay = new GregorianCalendar(
    2014, Calendar.JUNE, 4).getTime();
}
```

```
@Test
/**
 * Az adott nap (fromDate) és a napok számából (days)
 * eredő banki nap között van egy hétvége.
 */
public void testOneWeekendBetween8() {

  int days = 8;
  log.info("Next bank day in " + days + " bank days");
  calculateBusinessDay = dateCalculationService.
    calculateBusinessDay(fromDate,days);
  expectedBusinessDay = new GregorianCalendar(
    2014, Calendar.JUNE, 6).getTime();
}

@Test
/**
 * Az adott nap (fromDate) és a napok számából (days)
 * eredő banki nap között van két hétvége.
 */
public void testTwoWeekendBetween() {

  int days = 9;
  log.info("Next bank day in " + days + " bank days");
  calculateBusinessDay = dateCalculationService.
    calculateBusinessDay(fromDate,days);
  expectedBusinessDay = new GregorianCalendar(
    2014, Calendar.JUNE, 9).getTime();
}
```

SZÉCHENYI 2020

```
@Test
/**
 * Az adott nap (fromDate) és a napok számából (days)
 * eredő banki nap között több hétvége van.
 */
public void testManyWeekendBetween() {

  int days = 40;
  log.info("Next bank day in " + days + " bank days");
  calculateBusinessDay = dateCalculationService.
    calculateBusinessDay(fromDate,days);
  expectedBusinessDay = new GregorianCalendar(
    2014, Calendar.JULY, 22).getTime();
}

@Test
/**
 * 0 eltelt nap vizsgálata.
 */
public void testZeroDaysBetween() {

  int days = 0;
  log.info("Next bank day in " + days + " bank days");
  calculateBusinessDay = dateCalculationService.
    calculateBusinessDay(fromDate,days);
  expectedBusinessDay = new GregorianCalendar(
    2014, Calendar.MAY, 27).getTime();
}

@After
public void evaulateExecution() {
  log.info("expectedBusinessDay of week: "
               + sdf.format(expectedBusinessDay.getTime()));
  Assert.assertEquals(expectedBusinessDay, calculateBusinessDay);
}

@AfterClass
public static void finalizeTestClass() {
  log.info("Date calculation tests finished.");
}

private SimpleDateFormat sdf = new SimpleDateFormat("(E) yyyy.MM.dd");
```

SZÉCHENYI 2020

```
}
```

A példa érdekessége még, hogy az ellenőrzés kikerült a tesztmetódusokból és az @After életciklus metódusban találhatjuk meg. Erre azért van lehetőségünk, mert egyrészt minden adat, ami az ellenőrzéshez szükséges megtalálható példányszinten, másrészt az @After metódus minden tesztmetódus után (azaz minden tesztpéldány esetén) újra, és újra lefut.

jUnit Timeout

Az előző példánkat kiegészítjük egy újabb elvárással. A @Test annotáció timeout attribútuma lehetőséget ad arra, hogy egy felső időkorlátot szabjunk a teszt futásának. Az időkorlátot milliszekundumokban adjuk meg. Ha a teszt nem fut le az adott idő alatt, akkor elbukik.

Használhatjuk egységteszteknél, vizsgálva például azt, hogy az adatok mennyiségének lineáris növekedésével lineáris vagy exponenciális-e a futási idő, illetve rendszerintegrációs teszteknél, amikor a külső kapcsolódás idejét szeretnénk egy bizonyos időkorlát közé szorítani.

A specifikációban találhatunk arra utalást, hogy egy-egy képernyőváltásnak vagy műveletnek milyen válaszidővel kell rendelkeznie. Ezeket a válaszidőket általában egy-két szűk keresztmetszettel rendelkező művelet határozza meg, melyek tesztjeire érdemes ezt a timeout eszközt használni.

Természetesen a futásidőt erősen befolyásolja a hardver és a környezet, amin a teszt fut, ezért az időkorlátot érdemes körültekintően megválasztani.

```
...
        @Test(timeout=4)
        /**
         * Az adott nap (fromDate) és a napok számából (days)
         * eredő banki nap között van egy hétvége.
         */
        public void testOneWeekendBetween() {

          int days = 6;
          log.info("Next bank day in "+days+" bank days");
          calculateBankDay = dateCalculationService.
            calculateBusinessDay(fromDate, days);
          expectedBankDay = new GregorianCalendar(
            2014, Calendar.JUNE, 4).getTime();
```

```
        }
...
```

Általában nem szoktunk ilyen rövid időintervallumot megadni, mert a jUnit párhuzamos működése és a JVM működése is okozhatja a teszt bukását.

jUnit Expected

Az eddigiekben azt vizsgáltuk, hogy helyes paraméterek esetén, milyen eredményt ad a szolgáltatás. Most azt fogjuk vizsgálni, hogyan reagál helytelen paraméterekre. Azt várjuk el, hogy ha null dátum paramétert adunk át, akkor `NullPointerException`-t fogunk kapni. A metódusunkkal kapcsolatban ez szerepel a specifikációban, tehát szükség van olyan tesztesetre, mely ezt is teszteli.

Azonban tudjuk, ha kivételt érzékel a jUnit futtató, akkor automatikusan elbuktatja a tesztet. Ezt természetesen megoldhatjuk kivételkezelővel is, de ehhez kapunk egy sokkal praktikusabb eszközt a jUnittól: a `@Test` annotáció `expected` attribútumának megadunk egy kivételosztályt, mellyel azt mondjuk meg az adott tesztmetódusról, hogy abban az esetben legyen sikeres, ha a megadott kivételt dobja, egyébként bukjon el.

```
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
...
/**
 * A tesztosztály a banki napok kalkulációját ellenőrzi,
 * a DateCalculationService.calculateBankDay metódusának
 * működését teszteli null értékekre.
 */
public class Test006DateCalculationServiceExpectedException {

        private static final Logger log = Logger.getLogger(...);

        private static DateCalculationService dateCalculationService;

        @BeforeClass
        public static void initTestClass() {
          dateCalculationService = new DateCalculationServiceImpl();
          log.info("Date calculation null tests started...");
        }
```

SZÉCHENYI 2020

```
        @Test(expected=NullPointerException.class)
        /**
         * A fromDate null, a days nem null.
         */
        public void testFromDate() {
          dateCalculationService.calculateBusinessDay(null, 1);
        }

        @AfterClass
        public static void finalizeTestClass() {
          log.info("Date calculation null finished.");
        }
}
```

jUnit Suite

Szükségünk lehet arra, hogy bizonyos tesztosztályokat összefogjuk, és egyszerre futtassunk. Ez akkor merülhet fel például, ha ugyanahhoz a tesztelt programegységhez tartozó, de eltérő jellegű vagy eltérő előfeltételekkel rendelkező teszteseteit akarjuk egy tesztosztályba foglalni. A jUnit a Suite.class futtatót és a @SuitClasses annotációt adja ehhez. Az alapértelmezett futtatót a @RunWith annotációval tudjuk módosítani, ahogy az alábbi példában látható.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ Test004DateCalculationServiceLifeCycle.class,
        Test006DateCalculationServiceExpectedException.class })

public class Test007Suite {

}
```

SZÉCHENYI 2020

A jUnit Suite segítségével lehetőségünk van osztály szinten összefogni a teszteket, és azokat együtt futtatni. Ekkor minden olyan tesztmetódus futni fog, amely a `Suite` osztályaiban szerepel. A következő eszközzel tesztmetódus szinten van lehetőségünk a tesztek csoportosítására.

jUnit Category

Lehetőségünk van arra, hogy a tesztmetódusokat - tesztosztályuktól függetlenül - kategóriákba soroljuk. Egészítsük ki a `Test006DateCalculationServiceExpectedException` és a `Test004DateCalculationServiceLifeCycle` példáinkat úgy, hogy néhány tesztmetódushoz felveszünk egy `@Category` annotációt, melynek megadunk egy saját interfész osztályt, ami az adott kategóriát reprezentálja. A példánkban egy új `ExtremeValueTest` interfészt készítettünk, mellyel azokat a teszteket jelöljük meg, melyek szélsőséges értéket tesztelnek.

```
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.experimental.categories.Category;
...
/**
 * A tesztosztály a banki napok kalkulációját ellenőrzi,
 * a DateCalculationService.calculateBankDay metódusának
 * működését teszteli.
 */
public class Test008DateCalculationServiceExpectedExceptionCategory {
        ...

        @Test(expected=NullPointerException.class)
        @Category(ExtremeValueTest.class)
        /**
         * A fromDate null, a days nem null.
         */
        public void testFromDateNull() {

          dateCalculationService.calculateBusinessDay(null, 1);
        }
        ...
}
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Assert;
```

```
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.experimental.categories.Category;
...
public class Test008DateCalculationServiceLifeCycleCategory {
        ...
        @Test
        @Category(ExtremeValueTest.class)
        /**
         * 0 eltelt nap vizsgálata.
         */
        public void testZeroDaysBetween() {

          int days = 0;
          log.info("Next bank day in "+days+" bank days");
          calculateBankDay = dateCalculationService.
            calculateBusinessDay(fromDate, days);
          expectedBankDay = new GregorianCalendar(
            2014, Calendar.MAY, 27).getTime();
        }
        ...
}
```

Miután megjelöltük az egyes tesztmetódusokat, létrehozunk egy tesztosztályt a `Categories` futtatóosztállyal, melyben összefogjuk őket. A `@Categories.IncludeCategory` annotációval megmondjuk, hogy mely interfésszel megjelölt tesztmetódusokat akarjuk futtatni ebben a tesztosztályban a `@SuiteClasses` annotációval megadott tesztosztályokból.

```
import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import org.junit.runners.Suite.SuiteClasses;
...
@RunWith(Categories.class)
@Categories.IncludeCategory(ExtremeValueTest.class)
@SuiteClasses({ Test008DateCalculationServiceExpectedExceptionCategory.class,
                 Test008DateCalculationServiceLifeCycleCategory.class })
public class Test008GroupingExtremeValueTests {
```

SZÉCHENYI 2020

```
}
```

Arra is van lehetőségünk, hogy bizonyos kategóriával megjelölt tesztmetódusok ne fussanak az adott gyűjteményben. Ezt a @Categories.ExcludeCategory annotációval tudjuk definiálni.

```
import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import org.junit.runners.Suite.SuiteClasses;
...
@RunWith(Categories.class)
@Categories.ExcludeCategory(ExtremeValueTest.class)
@SuiteClasses({ Test008DateCalculationServiceExpectedExceptionCategory.class,
        Test008DateCalculationServiceLifeCycleCategory.class })
public class Test008GroupingNoExtremeValueTests {

}
```

A `@Categories.IncludeCategory` és a `@Categories.ExcludeCategory` annotációkban több típust is megadhatunk. Alapértelmezetten bármely felsorolt kategória típussal rendelkező tesztmetódus befoglalását jelenti.

jUnit Parametrized

Lehetőségünk van arra, hogy paraméterek vizsgálatára olyan teszteket készítsünk, melyeknek jellemzője, hogy minden logikai, infrastrukturális és strukturális elem megegyezik a tesztesetek esetén, egyedül tesztadataikban térnek el. A bemenő paraméterek és az elvárt eredményeket kollekcióban gyűjtjük össze, és a kollekció elemeivel futtatjuk a teszteket.

```
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
...
```

SZÉCHENYI 2020

```
/**
 * A tesztosztály a tetszőleges kezdő dátum és napok szám
 * alapján teszteli a dátumkalkulációs szolgáltatást
 */
@RunWith(Parameterized.class)
public class Test009DateCalculationServiceParameterized {

        private static final Logger log = Logger.getLogger(...);

        private static DateCalculationService dateCalculationService;

        private Date fromDate;

        private int days;

        private Date expectedBankDay;

        private Date calculateBankDay;

        private static final String FROM_DATE = "2014.05.27";

        @Parameterized.Parameters
        public static Collection data() {
                return Arrays.asList(new String[][] {
                        { FROM_DATE, "2", "2014.05.29" },
                        { FROM_DATE, "6", "2014.06.04" },
                        { FROM_DATE, "8", "2014.06.06" },
                        { FROM_DATE, "9", "2014.06.09" },
                        { FROM_DATE, "40", "2014.07.22" },
                        { FROM_DATE, "0", "2014.05.27" } });
        }

        private SimpleDateFormat paramSdf =
                new SimpleDateFormat("yyyy.MM.dd");

        public Test009DateCalculationServiceParameterized(
          String fromDate,
          String days,
          String expectedBankDay) throws ParseException {
                this.fromDate= paramSdf.parse(fromDate);
                this.days= Integer.valueOf(days);
                this.expectedBankDay= paramSdf.parse(expectedBankDay);
        }
```

```
    @BeforeClass
    public static void initTestClass() {
      dateCalculationService = new DateCalculationServiceImpl();
      log.info("Date calculation tests started...");
    }

    @Test
    public void testCalculationService() {

      log.info("Date Calculation Case: from "
             + sdf.format(fromDate) + " "
             + days + " days expected " + sdf.format(expectedBankDay));
      calculateBankDay = dateCalculationService.
        calculateBusinessDay(fromDate,days);
      Assert.assertEquals(expectedBankDay, calculateBankDay);
      log.info("calculateBankDay of week: "
        + sdf.format(calculateBankDay));
    }

    @AfterClass
    public static void finalizeTestClass() {
      log.info("Date calculation tests finished.");
    }

    private SimpleDateFormat sdf = new SimpleDateFormat(
      "(E) yyyy.MM.dd");
}
```

A Parametrized futtató esetén van néhány megszorítás, melyeknek meg kell felelnie a tesztosztályunknak:

- A `@Parameterized.Parameters` annotációnak egy statikus, `java.util.Collection` visszatérési értékű metódust kell jelölnie, mely a tesztadatokat tartalmazza. A kollekció tömböket tartalmaz, melyek mérete megegyezik, és elemeik száma a teszt paramétereinek számával egyezik. Ide értjük az esetleges elvárt értékeket is.
- A tesztosztálynak kell lennie egy olyan konstruktorának, mely pontosan annyi paramétert tartalmaz, ahány eleműek a kollekcióban visszaadott tömbök, és típusuk megegyezik a tömb elemeinek típusával.

SZÉCHENYI 2020

- A konstruktorban értékül kell adni a kapott paramétereket példányszintű tagoknak.
- A tesztmetódusokban ezeket a példányszintű tagokat kell használni paraméterként.

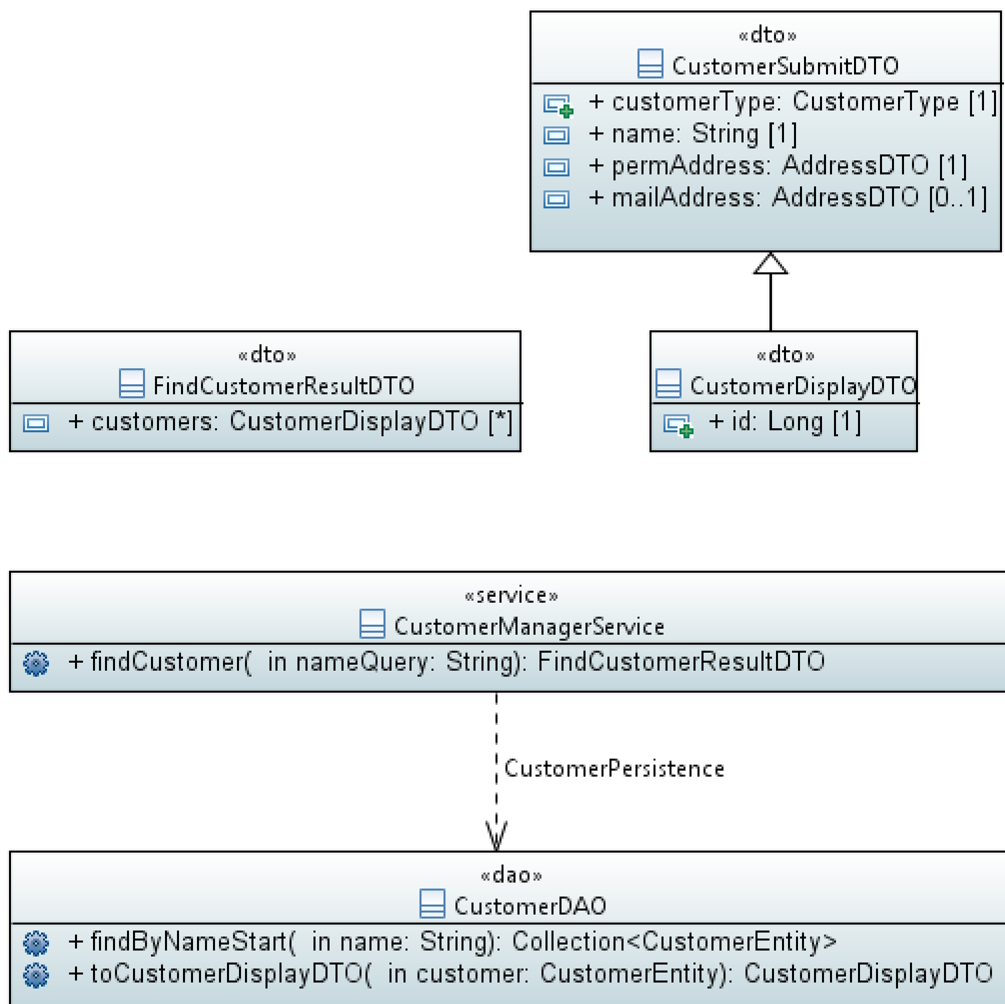
Most, hogy megismerkedtünk a jUnit alapeszközeivel nézzünk meg néhány összetettebb példát. A következőkben függőséggel rendelkező szolgáltatásokat tesztelünk, melyeknél alkalmazzuk a különböző emulációs technikákat.

Szolgáltatás egységteszt

Az életben sokkal gyakoribbak az olyan szolgáltatások, melyek más szolgáltatásokat vagy alsóbb rétegeket használnak, függenek tőlük. Ahhoz, hogy szabályos egységteszteket készítsünk, vagy helyettesítsünk még nem létező szolgáltatást integrációs tesztek esetén, szükségünk van arra, hogy ezeket a függőségeket helyettesítsük a tesztek futása során.

Az első példánk egy ügyfélkezelő szolgáltatás lesz - `CustomerManagementService` -, melynek a kereső függvényét – `findCustomer` – fogjuk tesztelni. Ez a metódus a `CustomerDAO` két metódusát a `findByNameStart`-ot és a `toCustomerDisplayDTO` metódusokat hívja. Segítségükkel keresi meg az adatbázisban az entitást, majd transzformája a képernyőn megjelenítendő struktúrába azt. Azt szeretnénk tesztelni – a tényleges lekérdezés futtatása nélkül –, hogy ez a bizonyos transzformáció és a visszatérési érték előállítása megfelelő-e.

SZÉCHENYI 2020



Stubbing példa

Névtelen típussal létrehozunk egy stub objektumot és a customerDAOStubObject változónak adjuk értékül. A `CustomerDAOImpl` mindegyik metódusát felüldefiniáljuk, ezáltal kontrolláljuk a működést. Majd ezt a stub objektumot állítja be az `init()` metódus `CustomerManagerServiceImpl` objektumnak, mint `CustomerDAO` példányt. A tesztmetódus ezután mit sem sejtve hívja a szolgáltatást, és vizsgálja a visszatérési értéket.

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.Before;
import org.junit.Test;
...
```

```
/**
 * Teszteli az ügyfél keresés eredményének entitás objektumokból
 * DTO objektumokká történő konverzióját. Két teszt entitás
 * visszatérését szimulálja.
 */
public class Test010CustomerManagerServiceFindCustomerByName {

        private CustomerManagerService customerManagerService;

        private ArrayList<CustomerEntity> testCustomers;

        @Before
        public void init() {

          CustomerManagerServiceImpl customerManagerServiceImpl =
                new CustomerManagerServiceImpl();

          // Stub objektum definíciója
          CustomerDAO customerDAOStubObject = new CustomerDAOImpl() {

            @Override
            public CustomerDisplayDTO toCustomerDisplayDTO(
                    CustomerEntity customer) {
              // Hívja, és ezáltal teszteli az eredeti metódust
              return super.toCustomerDisplayDTO(customer);
            }

            @Override
            public Collection<CustomerEntity> findByNameStart(
             String nameQuery) {

              testCustomers = new ArrayList<CustomerEntity>(2);

                  // Tesztadatok
                  CustomerEntity testCustomer = new CustomerEntity();
                  testCustomer.setId(1L);
                  testCustomer.setName("Mr. Dennis M. Ritchie");
                  testCustomer.setCustomerType(CustomerType.B2B);
                  testCustomer.setPermAddress("4032 Debrecen,
                  Egyetem tér 1.");
                  testCustomers.add(testCustomer);
```

SZÉCHENYI 2020

```
            testCustomer = new CustomerEntity();
            testCustomer.setId(2L);
            testCustomer.setName("Mr. James Gosling");
            testCustomer.setCustomerType(CustomerType.B2B);
            testCustomer.setPermAddress("4028 Debrecen,
            Kassai út 26.");
            testCustomer.setMailAddress("4028 Debrecen,
            Kassai út 28.");
            testCustomers.add(testCustomer);

        return testCustomers;
      }

      @Override
      public CustomerEntity findById(Long id) {
        throw new RuntimeException(
                    "This method should not be
              called in this test!");
      }

      @Override
      public Long create(CustomerEntity customer) {
        throw new RuntimeException(
                    "This method should not be
              called in this test!");
      }
    };

    // Stub objektum beállítása függőségként
    customerManagerServiceImpl.setCustomerDAO(
          customerDAOStubObject);

    customerManagerService = customerManagerServiceImpl;
  }

  @Test
  /**
   * Teszteli, hogy a kereső metódus helyesen végzi-e a
   * konverziót.
   */
  public void testFindCustomer() {
```

SZÉCHENYI 2020

```
        FindCustomerResultDTO findCustomerResult = customerManagerService
                        .findCustomer("Mr.");

        assertNotNull("findCustomerResult should not be null!",
                        findCustomerResult);

        CustomerDisplayDTO[] customers = findCustomerResult.getCustomers();

        assertEquals("", testCustomers.size(), customers.length);
        for (int i = 0; i < customers.length; i++) {
          assertNotNull(
            "Null should not be elements in the customer result!",
            customers[i]);

          // Ellenőrzi, hogy az eredményként visszakapott
        // customers[i] objektum megfelel-e a testadatokban
          // visszaadott testCustomers[i] objektumnak.
          assertEquals(customers[i].getId(),
             testCustomers.get(i).getId());
          assertEquals(customers[i].getName(),
                 testCustomers.get(i).getName());
          assertEquals(customers[i].getCustomerType(),
            testCustomers.get(i).getCustomerType());
          assertEquals(customers[i].getPermAddress(),
            testCustomers.get(i).getPermAddress());
          assertEquals(customers[i].getMailAddress(),
            testCustomers.get(i).getMailAddress());
        }
      }
}
```

A toCustomerDisplayDTO super hívása talán érdekesnek tűnhet, hiszen ezzel egy másik osztályt is tesztelünk, és így már nem csak azt az osztályt futtatjuk a tesztünkkel, amit tesztelni akarunk. Annak a szabálynak azonban eleget teszünk, hogy a toCustomerDisplayDTO működését kontrolláljuk a teszten belül. Jelen esetben pedig a kontrollálás annyit tesz, hogy tovább hívunk az eredeti metódusra, azért, mert a stub műveletek is pontosan ugyanazok lennének, mint az eredeti. Tehát önállóan, eredeti állapotában csak a szolgáltatás működik, minden mást a teszt kontrollál.

Mocking példa

Nézzük meg a fenti tesztet a Mockito keretrendszerrel.

```
import org.junit.Before;
import org.junit.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
...
/**
 * Teszteli az ügyfél keresés eredményének entitás objektumokból
 * DTO objektumokká történő konverzióját. Két teszt entitás
 * visszatérését szimulálja.
 */
public class Test011CustomerManagerServiceFindCustomerByName {

        private CustomerManagerService customerManagerService;

        // Mock objektum definíciója
        @Mock
        private CustomerDAOImpl customerDAOMockObject;

        @Before
        public void initTests() {

          //Init Mockito
          MockitoAnnotations.initMocks(this);

          CustomerManagerServiceImpl customerManagerServiceImpl =
            new CustomerManagerServiceImpl();

          // Mock objektum beállítása függőségként
          customerManagerServiceImpl.setCustomerDAO(customerDAOMockObject);

          customerManagerService = customerManagerServiceImpl;
        }

        @Test
        /**
         * Teszteli, hogy a kereső metódus helyesen végzi-e
         * a konverziót.
         */
        public void testFindCustomer() {
```

SZÉCHENYI 2020

```
String nameQuery = "Mr.";

// Mock objektum viselkedésének beállítása:
// Hívja, és ezáltal teszteli az eredeti metódust
when(customerDAOMockObject.toCustomerDisplayDTO(
  any(CustomerEntity.class))).thenCallRealMethod();

ArrayList<CustomerEntity> testCustomers =
  new ArrayList<CustomerEntity>(2);

testCustomers = new ArrayList<CustomerEntity>(2);

// Tesztadatok
CustomerEntity testCustomer = new CustomerEntity();
testCustomer.setId(1L);
testCustomer.setName("Mr. Dennis M. Ritchie");
testCustomer.setCustomerType(CustomerType.B2B);
testCustomer.setPermAddress("4032 Debrecen, Egyetem tér 1.");
testCustomers.add(testCustomer);

testCustomer = new CustomerEntity();
testCustomer.setId(2L);
testCustomer.setName("Mr. James Gosling");
testCustomer.setCustomerType(CustomerType.B2B);
testCustomer.setPermAddress("4028 Debrecen,
      Kassai út 26.");
testCustomer.setMailAddress("4028 Debrecen,
      Kassai út 28.");
testCustomers.add(testCustomer);

when(customerDAOMockObject.findByNameStart(
  nameQuery)).thenReturn(testCustomers);

// Ellenőrizzük, hogy a mock objektum eddig
// érintetlen maradt-e.
Mockito.verifyZeroInteractions(customerDAOMockObject);

// A tesztelendő metódus hívása
FindCustomerResultDTO findCustomerResult =
  customerManagerService.findCustomer("Mr.");

assertNotNull("findCustomerResult should not be null!",
```

SZÉCHENYI 2020

```
            findCustomerResult);

          CustomerDisplayDTO[] customers =
            findCustomerResult.getCustomers();

          assertEquals("", testCustomers.size(), customers.length);
          for (int i = 0; i < customers.length; i++) {
            assertNotNull(
              "Null should not be elements in the customer result!",
              customers[i]);

            // Ellenőrzi, hogy az eredményként
            // visszakapott customers[i]
            // objektum megfelel-e a tesztadatokban visszaadott
            //  testCustomers[i] objektumnak.
            assertEquals(customers[i].getId(), testCustomers.get(i).getId());
            assertEquals(customers[i].getName(),
                   testCustomers.get(i).getName());
            assertEquals(customers[i].getCustomerType(),
              testCustomers.get(i).getCustomerType());
            assertEquals(customers[i].getPermAddress(),
              testCustomers.get(i).getPermAddress());
            assertEquals(customers[i].getMailAddress(),
              testCustomers.get(i).getMailAddress());
          }

          // Ellenőrizzük, hogy megtörténtek-e az elvárt hívások
          // a mock objektumon. (gray-box eszközök)
          Mockito.verify(customerDAOMockObject).findByNameStart(nameQuery);
          Mockito.verify(customerDAOMockObject,
            Mockito.times(testCustomers.length)).
              toCustomerDisplayDTO(any(CustomerEntity.class));

          // Ellenőrizzük, nem maradt-e ellenőrizetlen hívás a mock
          // objektumon, azaz nem történt-e olyan hívás, melyre
          // nem számítottunk. (gray-box eszközök)
          Mockito.verifyNoMoreInteractions(customerDAOMockObject);
        }
}
```

A fenti példában láthatjuk a Mockito elvárásokkal kapcsolatos és verifikációs eszközeit, melyek a viselkedés módosításához és ellenőrzéséhez adnak eszközöket számunkra könnyen olvasható formában.

SZÉCHENYI 2020

Nézzük az alábbi utasítást:

```
...
when(customerDAOMockObject.toCustomerDisplayDTO(
        any(CustomerEntity.class))).thenCallRealMethod();
...
```

Ezzel az utasítással azt mondjuk meg a keretrendszernek, hogy amikor (when) a `customerDAOMockObject.toCustomerDisplayDTO()` metódusát bármilyen (any) `CustomerEntity.class` típusú objektummal meghívják, akkor hívjon tovább az eredeti metódusra (`thenCallRealMethod`).

Hasonlóan olvasható, és érthető az alábbi elvárás is:

```
String nameQuery = "Mr.";
...
ArrayList<CustomerEntity> testCustomers =
  new ArrayList<CustomerEntity>(2);
...
when(customerDAOMockObject
  .findByNameStart(nameQuery)).thenReturn(testCustomers);
```

Amikor a `nameQuery` paraméterrel meghívja valaki `customerDAOMockObject.findByNameStart()` metódusát, akkor adja vissza a `testCustomers` tesztadat gyűjteményt.

A tesztmetódusunk végén láthatjuk a **gray-box** eszközök megjelenését, melyekkel sokkal meggyőzőbb teszteket és teszteredményeket készíthetünk. E vizsgálatokkal belátjuk, hogy a teszt tisztában van a teszteset lefutásával és érti a működést.

SZÉCHENYI 2020

```
        ... WHEN elvárások

        //Ellenőrizzük, hogy a mock objektum eddig érintetlen maradt-e.
        Mockito.verifyZeroInteractions(customerDAOMockObject);

        ... TESZT FUTÁS
        ... ASSERT-ek

        // Ellenőrizzük, hogy megtörténtek-e az elvárt hívások
        // a mock objektumon. (gray-box eszközök)
        Mockito.verify(customerDAOMockObject).
          findByNameStart(nameQuery);
        Mockito.verify(customerDAOMockObject,
        Mockito.times(testCustomers.length)).
          toCustomerDisplayDTO(any(CustomerEntity.class));

        // Ellenőrizzük, nem maradt-e ellenőrizetlen hívás a mock
        // objektumon, azaz nem történt-e olyan hívás, melyre
        // nem számítottunk. (gray-box eszközök)
        Mockito.verifyNoMoreInteractions(customerDAOMockObject);
```

A következő példában a Mockito általános válaszára (Answer) és annak egyik lehetséges alkalmazási módjára látunk példát. A számlalétrehozó szolgáltatást teszteljük úgy, hogy mockoljuk az `InvoiceDAO` és a `CustomerDAO` objektumokat. A tesztünk célja egyrészt, hogy egyrészt a leképezés helyességét vizsgáljuk, másrészt a származtatott értékek számítását ellenőrizzük.

```
import static org.mockito.Matchers.*;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.invocation.InvocationOnMock;
import org.mockito.stubbing.Answer;
...
```

SZÉCHENYI 2020

```java
/**
 * Teszteli a számla létrehozása közben, a származtatott
 * értékek meghatározásakor végrehajtott számításokat.
 */
public class Test012InvoiceManagerServiceCreateInvoice {

        private InvoiceManagerService invoiceManagerService;

        // Mock objektum definíciója
        @Mock
        private CustomerDAOImpl customerDAOMockObject;

        @Mock
        private InvoiceDAOImpl invoiceDAOMockObject;

        @Before
        public void initTests() {

          // Init Mockito
          MockitoAnnotations.initMocks(this);

          InvoiceManagerServiceImpl invoiceManagerServiceImpl =
            new InvoiceManagerServiceImpl();

          // Mock objektum beállítása függőségként
          invoiceManagerServiceImpl.setCustomerDAO(customerDAOMockObject);
          invoiceManagerServiceImpl.setInvoiceDAO(invoiceDAOMockObject);

          invoiceManagerService = invoiceManagerServiceImpl;
        }

        private InvoiceEntity resultInvoiceEntityForAsserts;

        @Test
        /**
         * Teszteli, hogy a kereső metódus helyesen
         * végzi-e a konverziót és a számításokat.
         */
        public void testCreateInvoice() {

          // Tesztadatok
          BigDecimal vatPercent = new BigDecimal("27");
```

```
CustomerEntity invoiceCustomers = new CustomerEntity();
invoiceCustomers.setId(1L);
invoiceCustomers.setName("Mr. Dennis M. Ritchie");
invoiceCustomers.setCustomerType(CustomerType.B2B);
invoiceCustomers.setPermAddress("4032 Debrecen, Egyetem tér 1.");

InvoiceSubmitDTO testInvoice = new InvoiceSubmitDTO();

InvoiceItemSubmitDTO[] testInvoiceItems =
  new InvoiceItemSubmitDTO[2];

testInvoiceItems[0] = new InvoiceItemSubmitDTO();
testInvoiceItems[0].setProductCode("PC1");
testInvoiceItems[0].setProductName("Product1");
testInvoiceItems[0].setQuantity(2L);
testInvoiceItems[0].setUnit("db");
testInvoiceItems[0].setUnitPrice(new BigDecimal("500"));
testInvoiceItems[0].setVatPercent(vatPercent);

testInvoiceItems[1] = new InvoiceItemSubmitDTO();
testInvoiceItems[1].setProductCode("PC2");
testInvoiceItems[1].setProductName("Product2");
testInvoiceItems[1].setQuantity(4L);
testInvoiceItems[1].setUnit("db");
testInvoiceItems[1].setUnitPrice(new BigDecimal("2500"));
testInvoiceItems[1].setVatPercent(vatPercent);

testInvoice.setInvoiceItems(testInvoiceItems);
testInvoice.setCustomerId(1l);
testInvoice.setBuyerName("Mr. Dennis M. Ritchie");
testInvoice.setBuyerAddress("4032 Debrecen,
    Egyetem tér 1.");
testInvoice.setSellerName("XY");
testInvoice.setSellerAddress("The Address of XY");
testInvoice.setSerialNumber(null);
testInvoice
      .setInvoiceDate(new GregorianCalendar(2014,
               Calendar.MAY, 27).getTime());
testInvoice.setFulfillmentDate(new GregorianCalendar(2014,
               Calendar.MAY, 26).getTime());
testInvoice.setDueDate(null);
testInvoice.setPaymentDate(null);
```

SZÉCHENYI 2020

```
// Mock objektum viselkedésének beállítása:
// Teszt ügyfél visszaadása
when(customerDAOMockObject.findById(any(Long.class))).
       thenReturn(invoiceCustomers);

// Új számla sorszám előállítása
when(invoiceDAOMockObject.getNextInvoiceSerialValue())
                .thenReturn(1000L);

// Működés közben létrehozott InvoiceEntity
// objektum megszerzése ellenőrzés céljából
when(invoiceDAOMockObject.create(
       any(InvoiceEntity.class))).
       thenAnswer(new Answer() {
           public Long answer(InvocationOnMock invocation)
             throws Throwable {
           resultInvoiceEntityForAsserts =
                         (InvoiceEntity)invocation
                         .getArguments()[0];
           return 1L;
       }
    });

// Ellenőrizzük, hogy a mock objektum eddig
// érintetlen maradt-e.
verifyZeroInteractions(
       customerDAOMockObject,invoiceDAOMockObject);

// A tesztelendő metódus hívása
invoiceManagerService.createInvoice(testInvoice);

assertEquals(11000L,
       resultInvoiceEntityForAsserts
       .getNetTotal().longValue());
assertEquals(2970L,
       resultInvoiceEntityForAsserts
       .getVatTotal().longValue());
assertEquals(13970L,
       resultInvoiceEntityForAsserts
       .getGrossTotal().longValue());
assertEquals("1000",
       resultInvoiceEntityForAsserts
       .getSerialNumber());
```

SZÉCHENYI 2020

```
// Ellenőrizzük, hogy 8 nappal későbbi üzleti
// napra állította-e a fizetési határidőt a rendszer.
assertEquals(new GregorianCalendar(
  2014, Calendar.JUNE, 6).getTime(),
        resultInvoiceEntityForAsserts.getDueDate());

Set invoiceItems = resultInvoiceEntityForAsserts
               .getInvoiceItems();
for (InvoiceItemEntity invoiceItemEntity : invoiceItems) {
      switch (invoiceItemEntity.getProductCode()) {
      case "PC1":
        assertEquals(1000L, invoiceItemEntity
          .getNetTotal().longValue());
        assertEquals(270L, invoiceItemEntity
          .getVatAmount().longValue());
        assertEquals(1270L, invoiceItemEntity
          .getGrossTotal().longValue());
              break;
      case "PC2":
        assertEquals(10000L, invoiceItemEntity
          .getNetTotal().longValue());
        assertEquals(2700L, invoiceItemEntity
          .getVatAmount().longValue());
        assertEquals(12700L, invoiceItemEntity
          .getGrossTotal().longValue());
              break;
      default:
              break;
      }
}

// Ellenőrizzük, hogy megtörténtek-e az elvárt hívásokat
// a mock objektumon. (gray-box eszközök)
verify(customerDAOMockObject).findById(any(Long.class));
verify(invoiceDAOMockObject).getNextInvoiceSerialValue();
verify(invoiceDAOMockObject).create(any(InvoiceEntity.class));

// Ellenőrizzük, nem maradt-e ellenőrizetlen hívás a mock
// objektumon, azaz nem történt-e olyan hívás, melyre
// nem számítottunk. (gray-box eszközök)
verifyNoMoreInteractions(customerDAOMockObject,
  invoiceDAOMockObject);
```

```
        }
}
```

A fenti példában az alábbi konstrukció az, ami a kód belső működését figyeli, használja, és a paraméterként átadott adatot elérhetővé teszti a teszt számára. A példában a perzisztálás előtti entitás példányt tudjuk ily módon ellenőrizni.

```
public class Test012InvoiceManagerServiceCreateInvoice {
        ...
        private InvoiceEntity resultInvoiceEntityForAsserts;
        ...
        @Test
        /**
         * Teszteli, hogy a kereső metódus helyesen
         * végzi-e a konverziót és a számításokat.
         */
        public void testCreateInvoice() {
        ...
          when(invoiceDAOMockObject.create(
                any(InvoiceEntity.class))).
                thenAnswer(
              new Answer() {
                     public Long answer(InvocationOnMock invocation)
                       throws Throwable {
                         resultInvoiceEntityForAsserts =
                           (InvoiceEntity)invocation.getArguments()[0];
                         return 1L;
                     }
              });
        ...
        ASSERT-ek az InvoiceEntity példányra...
        }
}
```

SZÉCHENYI 2020

Teszttípusok

A teszttípusokat azok célja definiálja és elnevezésük is erre utal. Például, amikor azt mondjuk, hogy használhatósági (usability) tesztet akarunk végrehajtani, akkor az a célunk, hogy a rendszer használhatóságát ellenőrizzük. A teszttípusokat két nagy csoportba sorolhatjuk:

- Funkcionális
- Nem funkcionális

Mindkét csoportnak vannak elemei az egyes teszt szinteken.

Sok esetben bizonyos típusú tesztekhez specifikus eszközöket találhatunk, melyek keretet adnak az adott típusú teszttervezéséhez, végrehajtásához és értékeléséhez.

Tesztszintek és teszttípusok viszonya

Az alábbi táblázat tartalmazza a szintek, típusok, eszközök és technikák viszonyát, illetve funkcionalitás szerinti besorolásukat.

SZÉCHENYI 2020

<table>
<tr><th></th><th>Egység/Komponens/ Modul</th><th>Integrációs</th><th>Rendszer</th><th>Átvételi</th></tr>
<tr><td rowspan="2">Funkcionális</td><td colspan="2" rowspan="2">xUnit, Stubbing, Mocking,
Exception Handling,
Black-box, Gray-box, TDD</td><td>GUI Testing,
Exception Handling,
Exploratory, i18n,
Black-box
Alfa Testing</td><td>UAT,
Black-box,
Beta Testing</td></tr>
<tr><td colspan="2">Smoke, Sanity</td></tr>
<tr><td>Nem funkcionális</td><td>Complexity,
White-box,
Code Coverage,
Backward Compatibility,
Control Flow,
Code Review</td><td>Interface Compatibility</td><td>Platform Compatibility,
Usability,
Performance,
Load, Volume, Stress,
Security, Scalability,
Installation,
Backup/Recovery,
Accessibility,
Portability, Efficiency,
Reliability, White-box</td><td>Operational,
Compliance,
Contract Acceptance,
White-box</td></tr>
</table>

Funkcionális tesztek

A korábbi fejezetekben az egyes szinteket, az alacsony szintű eszközöket, technikákat funkcionális szempontból vizsgáltuk. Példákkal szemléltettük, hogy hogyan, milyen esetben kell/lehet őket használni.

A funkcionális tesztek magasabb (rendszer és átvételi) szintű típusait a végrehajtásukat és közönségüket illetően további két kategóriába sorolhatjuk.

Alfa-teszt

Azokat a teszteket, melyeket egy szűk, esetlegesen belső felhasználói kör segítségével hajtunk végre, alfa-teszteknek hívunk. Leginkább az úgynevezett dobozos alkalmazások esetén elterjedt ez a fajta terminológia, de alkalmazható egyedi megrendelés alapján készült alkalmazások esetén is. Ezeket a

teszteket még azelőtt hajtjuk végre, hogy külső felhasználók számára elérhetővé tennék, azaz béta-tesztre engednék az alkalmazást. E teszttípust rendszerteszt szinten végezzük, házon belül.

Béta-teszt

A béta-tesztre engedett verziót már a szélesebb közönség is használhatja. E verziót a gyártó instabil verzióként jelöli meg, és mindig elérhetővé teszi a korábbi stabil verziót. A béta verziók célja, hogy a kísérletező kedvű, az új funkciókat megismerni kívánó felhasználók betekintést nyerjenek a változásokban. Az új béta verziók javításokat is hozhatnak, melyek szintén érdeklődésre tartanak számot a felhasználók részéről. A béta verzióval együtt illik egy un. changelogot is kiadni, mely tartalmazza a változásokat, azok leírását. A béta-teszteket az átvételi teszt szintbe soroljuk.

Nem funkcionális tesztek

A továbbiakban a nem funkcionális tesztekre nézünk néhány példát. Meghatározzuk a szinte(ke)t, ahol alkalmazhatjuk őket, illetve ahol lehet, a korábbi számla alkalmazásunkkal szemléltetjük az egyes teszttípusok célját és jelentőségét. Az irodalom sokféleképpen rendszerezi ezeket a teszteket, és átfedéseket tapasztalhatunk vagy szinonim elnevezésekként jelennek meg bizonyos teszttípusok. A következőkben egy egyszerű rendszerezésben és felsorolásban megfogalmazzuk a teszt célját, rámutatunk a gyakorlati alkalmazásra, és igyekszünk a példa rendszerünkön szemléltetni néhány kritikus részt. Rokonságot figyelhetünk meg néhány teszttípus működését és alkalmazását illetően. Ezek a teszttípusok közel állnak egymáshoz, és bár céljaik csak néhány aspektusban térnek el, fontos, hogy mégis eltérő értéket hoznak a rendszerrel kapcsolatos bizalom kiépítésébe. A jelentőségük is eltérő attól függően, hogy milyen jellegű a rendszer.

Teljesítmény teszt (Performance Testing)

A legelterjedtebb nem funkcionális tesztek a teljesítmény (performancia) tesztek. Általában magas szinten történnek és az integrált rendszer hajtjuk végre őket. Mivel sok vizsgálati szempont, cél és módszer létezik, a performancia teszteken belül több típust is megkülönböztetünk, azonban mindegyik célja a rendszer válaszidejének, sebességének, viselkedésének vizsgálata bizonyos típusú terhelés mellett. Performancia tesztek tervezésekor meghatározzuk azoknak a funkcióknak a körét, melyek felhasználói szempontból kritikusak, majd tesztterveket készítünk és megvizsgáljuk a kijelölt funkciók működését a különböző terhelések esetén.

Minden performancia tesztnek szerves részét képezi az, hogy folyamatosan, lehetőleg online monitorozzuk a rendszert, valamint diagnosztikai naplófájlokat készítünk. A monitorozással azonnal észleljük a problémákat és látjuk azok jellegét illetve előfordulását, és akár arra is lehetőségünk van,

hogy konkrétan azonosítsuk azt a komponenst vagy elemet, amely a lassú működést okozza. A monitorok hagyományosan grafikonokkal ábrázolják a hardver erőforrásokat: memória, CPU. E grafikonok árulják el nekünk, hogy az adott hardveren futó alkalmazásban, komponensben folyik a memória, vagy túlzott számítási igény jelentkezik. Néhány korszerű eszköz a kód instrumentálásával akár metódus szinten képes megmutatni a futásidőt. Ezek az eszközök általában arra is képesek, hogy az adott tranzakcióban megmutassák a résztvevő metódusokat, és százalékos eloszlásban jelenítsék meg azok futási idejét.

Példákban felhasználói szempontból kritikus lehet az ügyfél keresése vagy a számla tételek és a számla származtatott értékeinek megjelenítése. Az előbbiről tudjuk, hogy valamilyen adatbázis műveletről van szó, a másik egyszerű számolás paraméterek alapján. Így performancia szempontból az ügyfél keresése tűnik kritikusabbnak. Ráadásul név kezdetre kell keresni, ami helyes indexelés nélkül lassan is működhet. Ha az igények úgy változnak, hogy nem csak a név kezdetre, hanem tetszőleges szövegrészre kereshetünk az ügyfél bármely adatából (nem csak a nevéből), akkor még több figyelmet kell majd fordítanunk e funkcióra performancia szempontból.

A következő néhány teszttípus tekinthető speciális performancia tesztnek:

- Terheléses teszt
- Stresszteszt
- Kitartási teszt
- Csúcsteszt
- Mennyiségi teszt

Terheléses teszt (Load Testing)

A terheléses tesztek célja, hogy bizonyos terhelés - ez lehet konkurens felhasználószám, tranzakciószám vagy kérés szám - mellett keressük a rendszer szűk keresztmetszeteit. Ilyen szűk keresztmetszet lehet például az adatbázis, valamilyen külső rendszerrel történő kommunikáció vagy valamilyen belső programozási hibából vagy figyelmetlenségből eredő erőforrás (memória, CPU) igényes működés.

Terheléses tesztek állandó eleme a rendszerbe való be- és kijelentkezés ezeknél a műveleteknél, ugyanis sokszor inicializációs és felszabadító folyamatok is zajlanak. Ha oly mértékben lassíthatják a működést, hogy a felhasználót zavarják és rontják az élményt, akkor javításra szorulnak. Ilyen esetben hasznosak a "lazy initialization" mechanizmusok, melyek csak akkor hoznak működésbe bizonyos komponenseket,

SZÉCHENYI 2020

amikor azokat használni akarja a rendszer. Ilyen mechanizmust használhatnak alkalmazás- és munkamenetszintű szolgáltatások is.

A számla létrehozásakor szűk keresztmetszetet jelenthet például az, ha a számla tételeket fel kell adni előírásként egy főkönyvi rendszerbe. Ha szinkron módon történik a kommunikáció, akkor meg kell várnunk a főkönyvi rendszer válaszát a létrehozáskor, mely akár több másodpercet is jelenthet, és ez nagyban rontja a felhasználói élményt. Ilyen esetben megoldás, ha rendszert úgy tervezzük, hogy a háttérrendszerekkel történő kommunikáció aszinkron módon történjen. Ha még is szükség van a háttérrendszer válaszára, akkor pollozunk, és jelezzük a felhasználónak, hogy milyen állapotban van a rendszer.

Stresszteszt (Stress Testing)

A stresszteszt célja, hogy megmutassa: a rendszer stabilitása az elvárásoknak megfelel. Jóval a normál működés feletti terhelést adunk a rendszernek, és figyeljük az egyes elemeket. Keressük a leggyengébb láncszemet a rendszerben, azaz addig növeljük a terhelést, míg a rendszer összeomlik vagy használhatatlanná válik. Az is célunk lehet, hogy figyeljük a rendszer összeomlásának mikéntjét, így nyerve tapasztalatot egy esetleges későbbi összeomláshoz, mely nem laboratóriumi körülmények között történik.

Mivel a stressztesztek célja, hogy a teljes rendszert - elsősorban a hardvert - a határain működtessék, felmerül annak a problémája, hogy hogyan állítjuk elő azt a számítási kapacitást, ami képes meghajtani ilyen mértékben a rendszerünket. A rendszerhez intézett kérések előállítása is hardverigényes feladat lehet.

Kitartási teszt (Soak / Endurance Testing)

A kitartási teszt is egyfajta terheléses teszt, melynek célja, hogy egy jól meghatározott terhelés alatt tartsa a rendszert hosszú időn keresztül. Ezzel azt akarjuk belátni, hogy a rendszer megfelelően viselkedik akkor is, ha huzamosabb ideig folyamatosan használat alatt áll, azaz nincsenek üresjárat időszakok.

Egyrészt tesztelhetjük vele azt, hogy a rendszer ugyanúgy, gyorsabban vagy lassabban válaszol-e bizonyos idő eltelte után, azaz ellenőrizhetjük a gyorsítótárak működését. Másrészt figyelhetjük a memóriaszivárgást és az esetleges garbage collector működését.

Csúcsteszt (Spike Testing)

Ez esetben a rendszert normál terhelés után, hirtelen nagy terheléssel árasztjuk el. Vizsgáljuk, hogy a válaszidők hogyan nőnek. Majd a nagy terhelést megszüntetjük, és újra normál terhelés alá helyezzük a rendszert. Ekkor azt akarjuk belátni, hogy a válaszidők visszaálltak a korábbi normál állapot válaszidőire.

Mennyiségi teszt (Volume Testing)

A mennyiségi tesztekkel az a célunk, hogy a rendszer működését nagy adatbázison, nagy rekordszám mellett vizsgáljuk. Előfordulhat ugyanis, hogy hiányos vagy nem megfelelő indexelés esetén az alkalmazás akkor válik majd használhatatlanná, mikor annak adatbázisa elért egy bizonyos méretet.

A valóságban sokszor problémát jelent a nagy mennyiségű adat előállítása. Mivel az adatok üzleti szempontból érdektelenek így lehetőségünk van olyan adatgeneráló technikák alkalmazására, melyek véletlenszerűen, akár választva, de bizonyos szabályoknak megfelelve állítanak elő tesztadatokat. A tesztadat előállítás sok ráfordítást igényelhet.

Példánkban az ügyfél keresése lehet ilyen szempontból kritikus. Milyen válaszidőket produkál a rendszer, ha nagy számú ügyfél kerül az adatbázisba?

Használhatósági teszt (Usability Testing)

A használhatósági tesztelés az egyik legrohamosabban fejlődő teszttípus. Számos módszert fejlesztettek ki, számos eszköz támogatja őket és számos cég végzi szolgáltatásként ezt a tevékenységet. A használhatósági teszt célja, hogy megmutassa a rendszerről azt, hogy mennyire felhasználó barát, mennyire könnyen használható. A **black-box** technikák közé soroljuk. Alapvetően három kérdésre keressük a választ egy adott rendszer kapcsán:

- Mennyire egyszerű használni?
- Mennyire egyszerű tanulni?
- Mennyire kényelmes a felhasználó számára?

A használhatósági tesztek az alábbi szempontokat veszik figyelembe:

- Tanulhatóság

  Mennyire egyszerű egy kezdő felhasználó számára feladatokat elvégezni úgy, hogy először látja a rendszert?

- Hatékonyság

SZÉCHENYI 2020

Milyen gyorsan tud egy tapasztalt felhasználó feladatokat elvégezni?

- Emlékezhetőség

  Képes-e egy felhasználó azonnal folytatni a hatékony munkát, miután hosszabb idő elteltével újra találkozik a rendszerrel, vagy újra kell tanulnia a kezdetektől?

- Hibás használat

  Hány hibát ejt egy felhasználó, milyen súlyosságúak ezek a hibák, és milyen gyorsan képesek orvosolni a hibát?

- Elégedettség

  Mennyire szereti a felhasználó használni a rendszert?

A használhatósági tesztek elsősorban a felhasználói felületre és a gép-ember interakciókra helyezik a hangsúlyt. A tesztek eredményei pedig a felületi tervezést (design) minősítik. A teszteket végrehajtók általában nem informatikai szakemberek vagy a projektben résztvevők, hanem vagy végfelhasználók, vagy véletlenszerűen, az utcáról kiválasztott emberek. Ennek az az elsődleges oka, hogyha a rendszerben nehézkesen végrehajtható feladat van, akkor azt a projekthez közelálló személy valószínűleg könnyedén meg tudja oldani, mivel vannak belső információi, és nem kapunk információt arról, ha valami nem működik gördülékenyen vagy nem egyértelmű. Ezzel szemben, ha a projekttől távoli felhasználókat választunk, akkor jó esély van arra, hogy megkapjuk azokat a véleményeket, melyek szerint a rendszer használata nehézkes, vagy akár lehetetlen.

SZÉCHENYI 2020

Kibocsátó:

lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation

Vevő: Ügyfél keresés

Ügyfél Fo | Cím Foo | Adószám: Adószám Foo

Ügyfél2 | Cím2 F | Adószám

Létrehoz

Alaphelyzet

Fizetési mód: Átutalás | Teljesítés: 2014.05.27. | Kelt: 2014.05.27. | Fizetési határidő: 2014.06.08.

Megnevezés	Mennyiség	Egységár	Nettó össz.	ÁFA %	AFA össz.	Bruttó össz. (Ft)	
Termék Foo1	1 db	2000 Ft	2000 Ft	27%	540 Ft	2540 Ft	-
Termék Foo2	2 db	1500 Ft	3000 Ft	27%	810 Ft	2810 Ft	-

+

'Termék Foo2' tétel törlésének visszavonása

Nettó összesen: 5000 Ft

ÁFA összesen: 1350 Ft

Bruttó végösszeg: 6350 Ft

Mentés

Vegyük sorra az egyes szempontokat a példa képernyőnkön. Tanulhatóság szempontjából jól állunk, mivel a képernyő elrendezése igazodik egy számla nyomtatvány elrendezéséhez. Így a felhasználó könnyel el tud igazodni a fogalmak között, és valószínűleg kérdezés nélkül ki tud állítani egy számlát. Ez annak köszönhető, hogy felhasználó számára ismerős képet alakítottunk ki. Hasonló okok miatt a hatékony munkavégzés és az emlékezhetőség is megvalósulhat. A hatékonyság és a hibakezelés mellett szól még, hogy ha egy tételt hibásan rögzített, azt törölheti úgy, hogy nem kérünk megerősítő képernyőt. Inkább néhány másodpercig egy "Visszavonás" funkciót jelenítünk meg számára, és ha tévesen törölt volna, akkor lehetősége van azt semmissé tenni, viszont, ha a törlés nem véletlen, akkor nem kell egy újabb gombra kattintania a megerősítéshez. Az elégedettség és kényelem érdekében az egérmozgást igyekszünk minimalizálni a képernyőn. Bizonyos műveletek a képernyőnek adott területeire orientáltak. Például az ügyféllel kapcsolatos műveletek a képernyő jobb felső részében vannak; a dátumok egy sorban helyezkednek el, és a tabindex megfelelően be van állítva; a számla tételek hozzáadása és rendezése a baloldalon, azok törlése a jobboldalon helyezkednek el. Egyedüli kényelmetlenséget az

SZÉCHENYI 2020

okozhat, hogy a mentés gomb a jobb alsó sarokba került, távol a hozzáadás művelettől. Ezen lehetne javítani úgy, hogy átrakjuk a képernyő bal oldalára.

Tekintsünk át néhány módszert arra, hogyan lehet egy használhatósági tesztet elvégezni.

Az un. Hallway Testing, ahogy a neve is utal rá, egy olyan teszt módszer, mely szerint néhány járókelőt beszólítunk a folyosóról, és velük hajtatjuk végre a feladatokat helyben az adott alkalmazásban. Egy-egy merőben új megjelenés vagy termék bevezetése előtt praktikus végrehajtani ezt a tesztet, hogy a tervezők lássák, hol tehetik könnyebbé az alkalmazást.

Ha nem tudjuk a tesztet helyileg megoldani vagy esetleg több országból szeretnénk felhasználókat bevonni, akkor a Remote Usability Testing módszert is választhatjuk. Ekkor valamilyen webes platformon online végezzük a tesztet, a kiértékelést, az egyeztetéseket. A online teszt folyhat szinkron és aszinkron módon. Szinkron esetben valamilyen online képernyő megosztó segítségével figyeljük a felhasználó ténykedését. Aszinkron esetben valamilyen beépített mechanizmus segítségével kapunk információt arról, hogy a felhasználó pontosan mikor milyen interakciót végzett, és az milyen eredménnyel zárult.

Az Expert Review módszer szerint egy vagy több használhatósági tesztekben jártas külső szakértőt vonunk be, akik értékelik a rendszert a Jakob Nielsen féle használhatóság szempontjai, heurisztikái alapján. Az e fajta használhatósági teszthez is használhatunk eszközöket, melyek szabályok és heurisztikák segítségével automatizálják az Expert Review módszert.

Az AB Testing módszer szerint ugyanazon funkcionalitást két változattal támogatunk, és egyszer az egyiket, máskor a másikat adjuk oda a felhasználónak. Az ilyen fajta tesztek elsősorban marketing és sales vonalon jelentősek, amikor akár a színvilágban vagy a formákban való kis eltérés is befolyásolhatja a konverziós értékeket. Akkor praktikus az ilyen vizsgálat, amikor a különbségek nem mérhetők objektíven, ezért rábízzuk a közönségre, hogy mutassák meg nekünk a jobb megoldást.

Nemzetköziesség teszt (Internationalization - i18n & Localization - l10n Testing)

Az I18n Testing elsősorban azt vizsgálja, hogy a rendszer képes-e több nyelven kommunikálni a felhasználóval, azaz le van-e fordítva több nyelvre. Az L10n Testing pedig a rendszernek azt a képességét ellenőrzi, hogy különböző országok szokásainak, szabályainak megfelelően működik-e. A két fogalmat együtt olykor globalization névvel illetik. Nézzünk néhány példát az egyes elvárásokra.

SZÉCHENYI 2020

Internationalization:

- Képernyő feliratainak fordítása megtörtént.
- Képernyőn megjelenő karakterek kódolása megfelelő (Unicode).
- A rendszer output dokumentumai megfelelő nyelven jelennek meg.
- A képernyőn megjelenő feliratok különböző nyelven, különböző hosszúsággal nem rontják el a képernyő elrendezését.
- Szám, dátum formátumok az adott nyelvnek megfelelőek.
- Balról jobbra és jobbról balra történő írás támogatása.
- Kis-, nagybetűk megfelelő kezelése.
- Film és videó anyagok fordítása hangsáv vagy felirat segítségével.

Localization:

- Különböző pénznemek támogatása.
- Telefonszám formátumok támogatása.
- Cím formátumok támogatása.
- Mértékegységek támogatása.

Biztonsági teszt (Security Testing)

A biztonsági tesztek azt vizsgálják, hogy a rendszer mennyire sebezhető rosszindulatú támadás vagy használat esetén. E témakörbe tartozik adatok védelme és a hálózati környezet biztonságossága, illetve a rendszer adatbázisaiban tárolt érzékeny adatok titkosítása, mint például a jelszavak.

Az alábbi fogalmakat kell lefednünk biztonsági tesztekkel:

- Titoktartás (Confidentiality)

  A rendszerből kikerülő információk védelmében olyan megoldásokat kell alkalmazni, hogy a jogosultakon és/vagy címzetteken kívül senki más ne férjen hozzájuk, ne használhassa jogosulatlanul olyan célra, mely ártalmas lehet.

- Sértetlenség (Integrity)

  A fogadó fél számára lehetővé teszi, hogy ellenőrizhesse a megkapott információ helyességét. A fogadó fél lehet a felhasználó és a rendszer is. A módszerek titkosítás helyett algoritmusokat

használnak (checksum, hash, Luhn), mivel az információ már a bevitelkor is sérülhetett - pl.: hibás gépelés.

- Hitelesítés (Authentication)

  Egy személy vagy másik rendszer azonosságát, egy objektum eredetét, az elnevezés és a tartalom megfeleltetését kell biztosítania a rendszernek.

- Meghatalmazás (Authorization)

  A rendszernek képesnek kell lennie eldönteni azt, hogy a rendszert használónak engedélyezett-e az adott művelet, használhatja-e az adott szolgáltatást.

- Elérhetőség (Availability)

  A rendszerhez jogosan hozzáférők számára elérhető a kért információ, amikor arra szükségük van.

- Letagadhatatlanság (Non-repudiation)

Az üzenet váltás folyamatában a küldő fél nem tagadhatja le, hogy elküldte az üzenetet, a fogadó fél pedig nem tagadhatja le, hogy megkapta.

Példánkban az hitelesítésen és a meghatalmazáson túl érdekes lehet az ügyfélkeresés funkció. Ha nem vagyunk körültekintőek, és az eredmény listában sok adatot engedünk megjelenni a felületen, akkor a felhasználó képes a teljes ügyfél adatbázisunkat lekérdezni ezen az egy funkción keresztül. Az ügyfél adatbázis érzékeny adathalmaz, így azt védeni kell. Megoldás lehet az, hogy a kereső mezőbe csak minimum 3 karakter hosszúságtól kezdünk el keresni, és 10 elemnél többet nem jelenítünk meg a listában. Minkét megszorítást megvalósítjuk a szerver oldalon is. Természetesen a karakterszámra vonatkozó megszorítást a kliensoldalon is implementálhatjuk, hogy elkerüljük a felesleges hálózati kommunikációt.

Ha kiterjesztenénk a számlakiállítónk funkcionalitását és lehetővé tennénk azt, hogy a felhasználó a termékeket vonalkód alapján adja hozzá a számlához, mint számlatételt, akkor felmerülne a vonalkód integritásának, azaz sértetlenségének problémája. Ha felismerjük a vonalkód típusát, és abban vagy kötelezően vagy opcionálisan megtaláljuk az ellenőrző számot, akkor még azelőtt meggyőződhetünk arról, hogy a felhasználó jól adta azt meg, hogy az adatbázis felé kérést indítanánk. Ezzel is erőforrást takarítunk meg. Az analóg-digitális folyamatok esetén általában előkerül az ellenőrző kód problémája. A

felhasználót mindig analóg "műszernek" tekintjük. Tehát ha őt kérjük meg arra, hogy adjon meg valamit szövegesen a felületen, ami a rendszer működése szempontjából releváns, akkor valamilyen ellenőrzési mechanizmust kell végrehajtani az adat sértetlenségét illetően. Pl.: jelszó vagy email megerősítő bekérése.

Ahhoz, hogy egy rendszerről elmondhassuk azt, hogy biztonsági teszten esett át a következő, részben egymásra épülő lépéseket kell végrehajtani:

- Felderítés (Discovery)
  A rendszer egészét vizsgáljuk, és keresünk az egyes komponensek esetleges elévült verzióit, melyek sérülékenységet okozhatnak.

- Sebezhetőségi vizsgálat (Vulnerability Scan)
  Automatikus teszteszközökkel keresünk ismert, gyakori biztonsági hibákat.

- Sebezhetőségi értékelés (Vulnerability Assessment)
  Az előző két lépés értékelése történik az adott környezet és rendszer esetén. Keressük és eltávolítjuk az automatikus értékelés eredményeként kapott álpozitív részeket a jelentésekből, és csak a releváns sebezhetőségi tételek maradnak.

- Biztonsági értékelés (Security Assessment)
  Manuális ellenőrzéssel győződünk meg a sebezhetőségi értékelés tételeiről a rendszer teljes terjedelmén. Az adott biztonsági hiba előidézése után, nem folytatunk, azt kihasználva, további gyengeségek keresését, a rendszerbe való mélyebb behatolást.

- Behatolási teszt (Penetration Test)
  A behatolási teszttel egy rosszindulatú felhasználó ténykedését szimuláljuk. Lehet white-box és black-box megközelítéssel is tesztelni, attól függően, hogy rendelkezésre állnak-e a rendszer részletei, forráskódja, vagy nem. Az előző eredmények alapján a rendszer mélyebb részeihez próbálunk hozzáférni a talált sebezhető pontokon. Behatolási tesztek esetén sok múlik a tesztelő tudásán és képességein, korábbi tapasztalatain.

- Biztonsági Audit (Security Audit)
  Egy magas szintű, kockázatokat feltáró folyamat, mely a korábbi lépésekre épít, illetve ellenőrzi azok meglétét, folyamatait.

SZÉCHENYI 2020

- Biztonsági Felülvizsgálat (Security Review)

  Verifikációs folyamat, mely vizsgálja, hogy rendszerben alkalmazták-e az iparági és a cég belső biztonsági előírásait. Minden lehetséges eszközt felvonultat, mely a rendszer kódjának, dokumentációjának, diagramjainak, architektúrájának átvizsgálását lehetővé teszi.

Skálázhatósági teszt (Scalability Testing)

E teszttel azt mérjük, hogy a rendszer mennyire képes a növekvő igényeknek megfelelni. A növekedés igénye jelentkezhet terhelésben, tranzakció számban, konkurens felhasználók számában, átviteli sebesség mértékében vagy adatmennyiség növekedés formájában. Azt vizsgáljuk, hogy a rendszer beállításait tudjuk-e úgy módosítani, hogy az ilyen fajta változásoknak eleget tegyen.

Vegyük azt az egyszerű esetet, hogy a rendszerünknek több memóriára lenne szüksége a megnövekedett felhasználó szám miatt. Ekkor a hardveres memóriabővítés után a folyamat számára is elérhetővé kell tenni a rendelkezésre álló memóriát: például JVM esetén a memória értékeket magasabbra kell állítani. Ez azonban a JVM leállításával és újraindításával jár.

Előfordulhat olyan eset is, amikor bizonyos részfunkcionalitások iránt nő meg az igény, azaz a rendszer funkciói között valamiféle prioritási sorrendet kell felállítani annak érdekében, hogy a kritikus funkciók számára mindig legyen erőforrás. Ilyen jellegű beállításokat általában valamilyen folyamat- vagy szálkezelő keretrendszerrel valósítunk meg.

Visszaállíthatósági teszt (Recovery Testing)

A visszaállíthatósági teszt azt vizsgálja, hogy a rendszer képes-e helyreállni, miután valamilyen külső hibából eredően lehetetlenné vált a működése.

Például vizsgálhatjuk, hogy ha egy hálózat kapcsolat hirtelen megszűnik - akár adatátvitel közben -, mi történik, ha újra visszakapcsoljuk a hálózatot. Képes a rendszer újra kommunikálni a másik féllel? Vizsgálhatjuk azt is, hogy mi történik, ha áramkimaradás következik be, vagy szabálytalanul állítjuk le a szervert, majd újraindítjuk. De az is érdekes lehet, hogy a kliensek hogy reagálnak egy esetleges szabályos szerver leállításra és újraindításra.

Hordozhatósági teszt (Portability Testing)

A hordozhatósági teszteket akkor és olyan alkalmazások esetén végezzük, amikor elvárás az, hogy több különböző platformon működőképesek legyenek. Például:

- Működnie kell Linuxon és Windowson is.
- Működnie kell Androidon és iOS-en is.

SZÉCHENYI 2020

- Működnie kell OpenGL-lel és DirectX-szel is.
- Böngészők tekintetében elvárás: Google Chrome, Mozilla Firefox, IE9, IE10

Sokszor egy működő platform után szeretnénk egy másik platformon is megjelenni. Keressük a választ arra is, hogy melyik megoldás a jobb: adoptálni a rendszert az új platformra vagy teljesen újraírni. A tesztek azt vizsgálják, hogy a megadott platformon milyen egyszerűen és gyorsan lehet használatba venni a rendszert, és a funkcionalitás sérül-e egy új platformon.

- Validáljuk (részlegesen) a rendszert azokon a platformokon, melyeket a hordozhatósági követelmények megfogalmaznak a környezettel kapcsolatban:
    - RAM és merevlemez helyigény
    - Processzortípus és -sebesség
    - Képernyőfelbontás
    - Operációs rendszer
    - Böngésző és annak verziója
- Vizsgáljuk a rendszer funkcionalitását és kinézetét (főképp a különböző böngészőkben történő megjelenést).
- Olyan hibákat keresünk és jelentünk, melyek feltehetően az adott platform miatt állnak fenn és nem valószínű, hogy az egység és integrációs tesztek képesek kimutatni.
- Meghatározzuk a rendszer készültségi szintjét az adott platformon.

A hordozhatósági tesztek az alábbiakat foglalják magukba:

- Telepíthetőség

  Azt vizsgálja, hogy a rendszert hogyan lehet telepíteni egy célplatformra. Kitér a tárhely ellenőrzésre, előfeltételek teljesülésére, a telepítés különböző módozataira, testre szabhatóságára, újra telepítésre és eltávolításra.

- Kompatibilitás és együtt élés

  Egymástól független rendszerek esetén akarjuk belátni, hogy nem befolyásolják egymás működését. Különös figyelem fordítunk erre olyan esetekben, ha mindketten használják ugyanazt a komponenst.

- Adaptálhatóság

  Az mutatja meg, hogy a rendszer minden platformon ugyanolyan működést produkál-e, és nincs szükség a normális interakciókon kívül egyéb műveletre platformspecifikus módon.

SZÉCHENYI 2020

- Helyettesítés

  Vizsgáljuk, hogyha egy rendszeren belül kicserélünk egy komponenst egy vele kompatibilis specifikációval rendelkező komponensre, akkor a működés nem változik.

Kompatibilitási teszt (Compatibility Testing)

A kompatibilitási teszttel legtöbbször a hordozhatósági teszt szinonimájaként találkozhatunk. Sokkal inkább annak részét képezi, és olyan esetben beszélünk kompatibilitási tesztről, amikor egy konkrét platformon vizsgáljuk az alkalmazás működését. Az adott platformon certifikálhatjuk a rendszert, ha megfelelt a kompatibilitási teszteknek. Továbbá bizonyos komponensek esetén szükséges a visszafelé kompatibilitást vizsgálni, ha az elvárásként jelenik meg az adott komponenssel kapcsolatban.

Hatékonysági teszt (Efficiency Testing)

A hatékonysági tesztek célpontja egy-egy funkció. Azt vizsgálják, hogy milyen mennyiségű kód és teszt szükséges ahhoz, hogy az adott funkcionalitást biztosítani tudjuk. Néhány aspektus, amiket vizsgálhatunk:

- Hány igényt elégítünk ki a rendszerrel?
- Milyen mértékben sikerült teljesíteni a specifikációban megfogalmazottakat?
- Milyen ráfordítást jelent a rendszer fejlesztése?

A tesztelés hatékonyságát is mérhetjük:

- Tesztek hatékonysága: (megtalált hibák az egység+integrációs+rendszer tesztekben) / (megtalált hibák az egység+integrációs+rendszer+megfelelési tesztekben)
- Tesztelés hatékonysága: (megtalált hibák / lejelentett hibák)*100

Megfelelési teszt (Contract Acceptance Testing)

A rendszer kapcsán megkötött szerződésekben foglaltaknak való megfelelést vizsgáljuk. Javasolt rendszerteszt szinten is vizsgálni a szerződésnek való megfelelést, hogy elkerüljük az esetleges kései módosításokat vagy vitákat. Ritkán kerül a belső operatív tesztelési feladatok közzé, mert a szerződés részletei a menedzsment hatáskörébe tartoznak. Gyakran a menedzsment nem fordít figyelmet arra, hogy a tesztelők számára szükséges információkat átadja a szerződésből, így a vagy egyáltalán nem, vagy az átadás kései fázisában kerül értékelésre ilyen szempontból.

Szabályossági teszt (Compliance Testing / Regulation Acceptance Testing)

Iparági szabályoknak, törvényi előírásoknak való megfelelést ellenőrzünk.

A számla létrehozó példánkban meg kell felelnünk a mindenkori áfa törvénynek. Az ilyen jellegű megfogalmazások magukban hordozzák a folyamatos jogkövetés igényét, mely azon túl, hogy az alkalmazás rendszeres felülvizsgálatát és esetleges módosítását jelenti, időről időre szakértő bevonását is követelheti meg.

Linearitás vizsgálat

E teszt célja, hogy az implementált algoritmusokról belássa, hogy nem exponenciális időben futnak akkor, ha a paramétereik száma vagy azok értéke lineárisan növekszik és az elvárás az, hogy a futás idő is maximum lineárisan változzon. Nem hatékonyak az olyan megoldások, melyek futásideje exponenciálisan változik. Nagyon nehéz eldönteni, hogy mely metódusokat vagy algoritmusokat kell vizsgálni, ugyanis a megvalósítás a legegyszerűbb igények esetén is lehet nagyon rossz: például helytelen ciklus használattal vagy rossz rekurzióval. Linearitás vizsgálatot azoknál a metódusoknál érdemes elvégezni, melyek futási ideje kiugróan nagy más metódusokhoz képest. Ezt a fajta tesztet tekinthetjük alacsony szintű performancia tesztnek is, mely a helytelen kódolásból eredő hibákat keresi. Olyan esetekben is találkozhatunk ilyen problémával, amikor az adatbázis rekordjainak vagy egyéb adatforrás elemeinek számossága kezd megnőni az rendszer élete során lineárisan, és a futási idők és válaszidők irreálisan megnőnek. Ekkor meg kell keresni a szűk keresztmetszetet jelentő metódust, és alacsony (egység vagy integrációs) szinten mennyiségi teszteket érdemes végrehajtani rajta és optimalizálni a megvalósítást.

Példánkban vizsgáljuk is az üzleti napokat kalkuláló szolgáltatásunkat, hogy változik-e a futási idő, ha nő a dátumintervallum. Az elvárás az lenne, hogy ne nőjön, mivel megvalósítható olyan algoritmussal, mely egyszerű aritmetikai műveleteket végez ciklusok használata nélkül. Ezért felveszünk egy tesztesetet, mely irreálisan nagy intervallumot vizsgál, és beállítjuk ugyanazt azt elvárt futási eredményt, mint az egyszerűbb eseteknél.

Megbízhatósági teszt (Reliability Testing)

A megbízhatósági teszt célja, hogy a rendszerrel elvégezhető feladatokról - lehetőleg mindről - megmutassa, hogy hosszú időn át képesek determinisztikusan és konzisztensen működni adott környezetben. Más aspektusból – alacsonyabb tesztszinten – azt várjuk el egy-egy művelettől, hogy ugyanazt az eredményt kapjuk, ha újra és újra végrehajtjuk adott körülmények között.

A megbízhatósági teszttípus magába foglalja, használja az alábbi teszttípusokat:

- Funkcionális tesztek
- Terheléses tesztek

SZÉCHENYI 2020

- Regressziós tesztek

Magas szintű funkcionális tesztekkel látjuk be, hogy az adott funkció megbízhatóan, időről időre konzisztensen működik. A terheléses tesztek a terhelés növekedéséből fakadó bizalmatlanságot hivatottak csökkenteni hosszabb időtávon. Azaz a rendszer megnövekedett terheléssel is megbízhatóan működik. A regresszió eszköztárát pedig arra használjuk, hogy belássuk azt, hogy az újabb módosítások nem okozzák más funkcionalitás működésképtelenségét.

A szoftver megbízhatóság a rendszerekkel szemben támasztott elvárások közül az egyik legfontosabb és leginkább fejlődő ág. A szoftverfejlesztés minden része - folyamatai, eszközei, technológiai elemei - szerepet játszik abban, hogy mennyire megbízható a rendszer. Amikor a hibák gyökerét keressük mindig valamilyen emberi mulasztást találunk az igényektől kezdve, a fejlesztésen és használt, beépített eszközökön át, a tesztelésig és a használatig.

A megbízhatóság természetesen nem csak a szoftverfejlesztés területén jelentős tényező. Minden iparág számára fontos, hogy az előállított termék megbízhatóan működjön. Ezért a megbízhatóság körül nagy méretű elméleti és gyakorlati ismeretanyag épült fel. A szoftver megbízhatóság alapvetően három nagy területre osztható:

- Modellezés
- Mérés
- Tökéletesítés

A modellezés szerepe, hogy az adott problémakör, rendszer megbízhatósági tényezőit rendszerbe szedjük, és megtartsuk annak egyszerűségét, konzisztenciáját. Az egyes problémakörökhöz hasonló modelleket alkalmazhatunk, ha azok hasonló üzleti szakterülethez tartoznak. Az alkalmazott modellt általában testre kell szabni, ki kell terjeszteni. Nincs olyan megbízhatósági modell, mely általánosan alkalmazható.

A mérés sohasem egyértelmű a szoftverfejlesztés világában, nincs ez másképp a megbízhatósággal sem. Számokban elég nehéz arra a kérdésre válaszolni, hogy mennyire jó, megbízható egy rendszer. Ezért metrikákat kell bevezetni, melyek a rendszer különböző mutatóit mérik, és ezeket értékelve kapunk közvetett képet arról, hogy a rendszer megbízhatóság tekintetében milyen szinten áll. A metrikák származhatnak a fejlesztési folyamatból, a megtalált hibák és észlelt meghibásodások számából.

A megbízhatóság tökéletesítése nehéz feladat. Sok faktor játszik szerepet abban, hogy egy rendszert megbízhatónak vagy megbízhatatlannak nevezzünk. Természetesen a megbízhatóságon, csak úgy, mint

más műszaki ágazatokban, sokat javít a fejlesztési folyamat tökéletesítése. Azonban más ágazatokhoz viszonyítva még így is sokkal több emberi tényező marad a szoftver fejlesztés területén, mely túl összetetté teszi a megbízhatóság kérdéskörét. Jelenleg nincs jó módszer arra, hogy megoldjuk ezt az összetett problémát. Arra lenne szükség, hogy teljesen kimerítő teszteket készítsünk, továbbá tökéletesen működő és együtt működő hardver- és szoftverelemekből kellene összeállítani a teljes rendszert. Könnyen belátható, hogy ilyen hibamentes rendszer készítése nem biztosítható, azaz a megbízhatóság nem garantálható egy bizonyos szint felett. Az idő és erőforrás tényezők is negatív hatással vannak a tökéletesítés folyamatára.

Regressziós teszt (Regression Testing)

A regressziós tesztek célja, hogy belássák: egy hiba javítása nem járt újabb hibák létrejöttével a korábban jól működő funkciókban. Azon túl, hogy leteszteltük a javított funkciót, és belátjuk, hogy az elvárásoknak megfelelően működik, arról is meg kell győződni, hogy nem rontottunk-e el valamit - azaz nincsenek "nem várt mellékhatások" - egy korábban működő funkcióban. Arról szeretnénk meggyőződni, hogy a rendszer a javítás után is megfelel a specifikációnak.

Regressziós teszteket minden szinten tudunk végezni. Általában valamilyen tesztautomatizáló eszközt használunk, mivel sok teszt futtatására lehet szükség. Egység és modul integrációs teszteknél használhatjuk az xUnit eszközöket, rendszer teszt szinten pedig számos eszköz áll rendelkezésre különféle platformokon, melyek támogatják a magas szintű tesztek rögzítését és automatikus futtatását.

Strukturális teszt (Structural Testing)

Szinonimái: white-box, glass-box, clear-box teszt. Ahogy az elnevezésekből is látszik az a cél, hogy a rendszert annak ismeretében teszteljük, ahogy strukturálisan felépül, és közben lássuk a megvalósítás részleteit. A tesztelő ekkor tisztában van azzal, hogy hogyan valósul meg egy-egy funkcionalitás, osztály vagy metódus. Ezt a tudást használja arra, hogy a működésre, hatékonyságra, lefedettségre vonatkozóan következtetéseket vonjon le, és megfeleltesse azt az elvárásoknak.

Strukturális teszteket minden szinten végezhetünk. Egység és integrációs szinten általában a fejlesztők, míg rendszer és átvételi szinten a tesztelők és maga az ügyfél is tesztelhet. Alacsony szinten általában az egyik legfontosabb és legtöbbet vizsgált aspektus a kód tesztesetek általi lefedettsége, azaz a kódlefedettség (Code Coverage). A kódlefedettség vizsgálatának alapvető eszköze a ciklomatikus komplexitás, mint mérő szám. Ez a szám adja meg azt a szükséges teszteset mennyiséget, mely ahhoz kell, hogy minden utasításra legalább egyszer kerüljön vezérlés. Ez nem jelenti azt, hogy a tesztelés

kimerítő volt, csak azt, hogy a tesztesetek lefedik az utasításokat. További strukturális technikákkal a következő fejezetekben találkozhatunk.

Karbantartási teszt (Maintenance Testing)

A karbantartási teszt a rendszer evolúciójának képezi szerves részét. A rendszer élete során megváltozhat annak környezete, sőt általában maga a rendszer is folyamatosan változik. A karbantartási tesztek egyszerű kombinációi a funkcionális, teszteset alapú teszteknek és a regressziós teszteknek. A cél az, hogy belássuk a változások működőképességét, és helyességét, ha például új funkcionalitásról van szó, és utána belássuk azt, hogy a többi funkcionalitás továbbra is helyesen működik. Ezeket a teszteket bármilyen változtatás esetén érdemes végrehajtani, hogy meggyőződjünk a rendszer működőképességéről.

Hatáselemzés (Impact Analisys)

Ahhoz, hogy eldöntsük egy módosítás után, hogy milyen teszteket kell lefuttatni, szükség van arra, hogy lássuk, az adott módosítás milyen funkcionalitást érintett. A funkcionalitások összegyűjtését segíti a hatáselemzés, mely a változás hatásait deríti fel egy adott rendszerre vonatkoztatva.

SZÉCHENYI 2020

Tesztelési technikák

Statikus technikák

A statikus technikák (static techniques), más néven statikus elemzés, vagy statikus analízis, az egyik nagy csoportját képezi az elterjedt teszttervezési és tesztelési módszertanoknak (test planning and design methodologies). Ellentétben a másik nagy csoporttal, a dinamikus technikákkal (dynamic techniques), nem igényelnek futtatást, hiszen elemző módszerek, ezért jól alkalmazhatók nem csak programkódon, de különböző dokumentációkon, specifikációkon is. Segítségükkel korán találhatunk eltéréseket az előírásoktól (requirements), az előzetes tervektől, így kis költséggel sokat javíthatunk a termék minőségén (hiszen egy hiba javítása annál olcsóbb, minél korábban sikerül észrevennünk). Nem mellékesen a korai elemzésnek hála javulhatnak a jövőbeli tervezési és fejlesztési módszerek is, nem csak az aktuális termék. Másik nagy előnye, hogy eszközökkel jól támogatható, gondoljunk csak bele, már egy egyszerű helyesírás-ellenőrző is dinamikus tesztelést (dynamic testing) segítő eszköznek számít, nem beszélve az összetett forráskód-elemző megoldásokról, és a fordítóprogramok többszintű elemzőiről. A következőkben viszont nem ezekre fogunk fókuszálni, hanem megpróbáljuk hasznos gyakorlati tanácsokkal kiegészíteni az ISTQB bevezető tananyag ide kapcsolódó, de eléggé elméleti részét. Kezdjük is a sort a strukturált és kevésbé strukturált értékelésekkel (structured evaluations).

Csoportos értékelések (structured group evaluations)

Ezeknek a lényege, hogy az emberi elme elemző képességét használják ki. Bizonyos formái inkább dokumentumok elemzésére jók, mások például forráskód elemzésre. Az egyik típusú szereplő, akit érdemes bevonni az, aki mestere az elemzett területnek, tesztelő, fejlesztő, üzleti elemző (business analyst), technikai író (technical writer), adatbázis szakember (database manager). Minél több embert, minél több nézőpontot sikerül bevonnunk az elemzésbe, annál sikeresebb lesz a folyamat. Az adott iparterület szakemberei is hasznosak, akkor is, ha nem technikaiak (pl. banki szakértők, matematikusok), de ez nagyban függ a dokumentum témájától. Az egész folyamatból viszont sokat tanulhatnak a pályakezdők, illetve olyan szereplők, akiknek az adott dokumentumnak a típusa, célja, még ismeretlen, illetve többet kell megtudniuk az adott iparterületről.

A szerepkörök és határidők legyenek előre tisztázva, mindennek legyen gazdája. Ez nagyon fontos, mivel ezek a folyamatok lineárisak, ha elakadnak egy ponton, nem sikerülhet elérni a végére. Érdemes minden szereplőt megkérdeznünk rendelkezik-e mindennel, ami a részvételhez kell, és ezt előre biztosítani nekik. Gondoljunk csak bele, egy forráskód olvasásához szükség lehet fejlesztőkörnyezetre, vagy speciális dokumentumformátumoknál egzotikus programokra, amik képesek megnyitni, esetlegesen

SZÉCHENYI 2020

megjegyzéseket fűzni hozzájuk. Ez a tanács nevetségesnek tűnhet, pedig sok kellemetlenséget spórolhatunk meg, ha erre (is) odafigyelünk. Mindig közöljük előre hogy mi a pontos célja a folyamatnak, ne csak azt, hogy milyen dokumentummal dolgozunk, hanem azt is, hogy milyen szempontból szeretnénk jobbá, pontosabbá tenni.

Egy jó "trükk", hogy tesztelőként bejussunk sok ilyen értékelésre: vállaljuk el az írnok (scribe) szerepét minél többször. Erről a későbbiekben lesz szó, nagyon hasznos, de általában került szerep. Mindenesetre olyan helyekre is beülhetünk így, ahová esetleg kevésbé releváns a tudásunk, de hasznosnak érezzük, hogy jelen legyünk.

Átvizsgálás (review)

Az átvizsgálás az egyik legformálisabb értékeléstípus. Érdemes alkalmazni szerződésekre (contracts), követelményspecifikációkra (requirement specification), dizájnspecifikációkra (design specification), programkódra, teszt- és más tervekre (test plan), de akár kézikönyvekre (manual), leírásokra (know-how) is. Ajánlott azonnal elvégezni, mihelyst a dokumentum rendelkezésre áll. Továbbá, amikor a projekt a következő fázisba lép, érdemes lehet visszanézni mire nem gondoltunk az első átnézéskor. Harmadsorban pedig, amikor a projekt véget ér és átadásra kerül, hasznos lehet korrigálni a végső simítások fényében a korábbi dokumentációt.

Korai felismeréssel sok hiba kiszűrhető olcsón. Szinte minden esetben megéri legalább a legfontosabb terveket és követelményeket átvizsgálásnak alávetni. A teszt és javítás fázis (test and fix period/phase) lerövidül, így esetlegesen előrehozható a határidő (deadline), vagy több követelmény férhet bele a megadott időbe. A kevesebb hiba miatt a dinamikus tesztelés kevesebb hibát hozhat majd elő, aminek egyértelmű hozadéka a jobb kódminőség (code quality). Gondoljunk csak bele, minden utólagos kódmódosítás csökkenti a kód koherenciáját, és gyengíti az architektúrát.

Ha a megbízó (client/business owner) is képviselteti magát (ez nem ajánlott, csak a követelményekkel közvetlenül összefüggő dokumentumokra), az segíthet hamar észrevenni, ha rossz irányba kezdtünk volna el haladni. Ez szintén már előre segíti a helyes architekturális tervezést. A szigorúan informatikai jellegű átvizsgálásokba viszont ne vonjuk be az ügyfelet, azt hagyjuk meg a szakembereknek. A résztvevő emberek érzéke is fejlődhet, például a következő hasonló dokumentum írásánál a korábban talált típushibákat már el tudják kerülni.

Sajnos, ha nem megfelelően régi motorosok a résztvevők (de néha még akkor is), átcsaphat a szakmai kritika emberibe, ennek elkerülésére a megfelelő moderálás a jó megoldás. Arra mindenképpen

ügyeljünk, hogy a moderáló/levezető személy gyakorlott legyen, és ráadásul bírja a résztvevő személyek tiszteletét. A legjobb tanács ilyenkor hogy ne legyen más szerepe a moderáláson kívül (ezekről a későbbiekben lesz szó). A saját részünkről annyit tehetünk, hogy személyes véleményt nem viszünk a dologba, csak szakmailag megalapozott kritikákat teszünk, és ha lehet, pozitív kritikákat is mondunk, amikből mindenki tanulhat.

Amennyiben volt korábbi, hasonló jellegű projekt, elemezhetjük, hogy mennyi várható hibát szűrtünk ki korán, így kiszámolva, hogy mennyi időt/pénzt spóroltunk meg. Érdemes a helyesírást ellenőrizni, mielőtt kiküldjük a dokumentumot, így triviális hibákkal nem megy el senki ideje a megbeszélésen.

A folyamat fázisai

Ez a rész bőségesen ki van fejtve a tankönyvben, próbáljunk meg minél több hasznos tanácsot összegyűjteni a különböző fázisokhoz.

Tervezés (planning)

Semmiképpen ne becsüljünk időt addig, amíg nem láttuk milyen és milyen hosszú a dokumentum. Ne féljünk több időt becsülni a felkészülésre, mint magára a megbeszélésre. Legyünk tisztában azzal, hogy milyen szempontból kell elemeznünk a dokumentumot. Nem mindegy hogy tesztelői, vagy más egyéb szemmel nézzük.

Indító megbeszélés (kick-off)

Ez remek alkalom arra, hogy az esetlegesen hiányzó külső függőségeknek (external dependencies) utánakérdezzünk, illetve a megértéshez szükséges információkat gyorsan megszerezzük. Készüljünk kérdésekkel.

Egyéni felkészülés (individual preparaption)

Ha nem tudunk időre felkészülni, jelezzük, és kérjünk halasztást. Sokkal jobb késni, mint rosszul elvégezni a feladatunkat.

Átvizsgáló megbeszélés (review meeting)

A végső döntésnél legyenek érveink, hogy miért szavaztunk úgy, ahogy. A végső döntés lehet: elfogadás, javítás, teljes újraírás. Ne fogadjunk el olyan dokumentumot, amivel nem vagyunk maradéktalanul elégedettek, inkább küldjük javításra, szélsőséges esetben teljes újraírásra. Ha egy részhez megjegyzésünk van, ne engedjük a továbblépést, amíg el nem mondtuk a véleményünket. A végső

SZÉCHENYI 2020

summázó dokumentumot aláírás előtt mindenképpen olvassuk át, ne adjuk a nevünket olyanhoz, aminek nem ismerjük a tartalmát (bár ez inkább általános szabály, nem átvizsgálás-specifikus).

Javítás (rework)

Ide nem sok gyakorlati tanácsot lehet mondani, amire érdemes figyelni, hogy a hibák kritikussága szerint javítsuk a dokumentumot (amennyiben ez a mi feladatunk), hiszen ha jön egy magasabb prioritású feladat, lehet, hogy nem lesz időnk befejezni. Akkor pedig a rendelkezésre álló időt a fontosabb problémák javítására érdemes szánnunk. Sokan elölről haladnak, de nem az a helyes megközelítés.

Folytatás (follow up)

Mi magunk is kérjünk újabb átvizsgálási kört, ha úgy érezzük, hogy nagyon sokat módosítottunk. Általában nem szükséges, mert nem hoz annyi hasznot, mint az első kör. Ha vannak visszatérő hibák, azokat érdemes közölni a szerzővel, vagy önmagunknak figyelni rá, ha mi lennénk azok.

Szerepkörök (roles and responsibilities)

Menedzser (manager)

A menedzsment feladata ki kell, hogy merüljön abban, hogy meghatározza a prioritásokat, és az átnézendő dokumentumokat. Nem szerencsés, ha részt is akarnak venni a folyamatban. Nyugodtan közöljük a moderátorral, hogy nem szeretnénk, ha az megbeszéléseken a menedzsment képviseltetné magát.

Moderátor (moderator)

Sokszor a moderátor maga is bíráló, ám ez ellenjavallott. Mindenkinek könnyebb a dolga, ha a moderátor kívül marad a szakmai kérdéseken, és csak a formaiság megtartását felügyeli.

Szerző (author)

Szerzőként próbáljunk meg arra figyelni, hogy ne vegyük személyeskedésnek az érkező kritikákat, hiszen a folyamat elsődleges célja a dokumentum jobbá tétele. Csak akkor engedjük/kérjük a dokumentumunk átnézését, ha már mi elégedettek vagyunk a minőségével. Rossz stratégia félmunkát végezni, és arra számítani, hogy a bírálók (reviewer) majd elvégzik helyettünk ezt a munkát.

Bíráló (reviewer)

Mint ahogy a fentiekben, bírálóként figyeljünk oda, hogy csak szakmailag megalapozott kritikákat hozzunk fel, és ezeket utólag is érthető formában dokumentáltassuk. Bátran hozzunk fel mindenféle

kritikát akkor is, ha speciális szerepkörrel vagyunk felruházva. Tehát hiába lettünk mi például a megbízott biztonságtechnikai bírálók (security expert/reviewer), ettől függetlenül jelezzünk más jellegű problémákat is, ne hagyatkozzunk arra, hogy a többiek azt majd úgy is megtalálják.

Írnok

Mindig figyeljünk, hogy utólag is érthető jegyzeteket írjunk, ha nem tudnánk olyan gyorsan haladni, ahogy a megbeszélés zajlik, állítsuk meg egy-egy gondolati fonal végén, és szóljunk, ha folytatódhat tovább a megbeszélés. Ha valamilyen rövidítést, vagy szakkifejezést nem értünk, nyugodtan kérdezzünk rá, inkább, mint hogy haszontalan jegyzeteket gyártsunk.

Lehetséges problémák

Ha nem tudtunk rendesen felkészülni, kérjünk halasztást, illetve ha a becsült időnél kevesebbet kapunk csak a felkészülésre, akkor mindenképpen jelezzük, hogy mit nem tudunk vállalni a csökkentett időben (például hogy ki kell hagynunk szempontokat, amiknek utánanézni időigényes). Sose vegyük félvállról a felkéréseket, tartsuk szem előtt, hogy minden felkérés azt jelenti, hogy értékesnek tartják a szakmai véleményünket, és arra számítanak, hogy a közreműködésünk gyümölcse egy jobb, használhatóbb dokumentum lesz. Amennyiben tanuló célzattal veszünk részt, próbáljuk meg magunkévá tenni a bírálók szakmai nézetét, illetve tanuljuk a megbeszélések levezetésének módozatait a moderátortól. Az átvizsgálás adja a legjobb táptalajt a személyeskedésnek, ha ezt jól kezeli a moderátor, érdemes tanulnunk tőle.

Főbb típusok

A fenti leírás tartalmazza az összes elterjedt módszertan keretét a különböző típusú átvizsgálásokhoz, röviden nézzük a konkrét fajtákat, és gyűjtsünk hasznos ötleteket hozzájuk.

Átnézés (walkthrough)

Informális jellegéből adódóan nem igényel sok felkészülést, de egyszer érdemes átolvasnunk a dokumentumot a megbeszélés előtt. A megbeszélés fő célja, hogy a résztvevők megismerhessék a dokumentumot, melyet a szerző mutat be, de ettől függetlenül az észlelt hibákat jelezzük. Ezt utólag tegyük meg, lehetőleg írásban. Alapvetően ez egy nem moderált forma, ezért mi magunknak kell ügyelnünk rá, hogy a megbeszélés a helyes mederben maradjon.

SZÉCHENYI 2020

Inspekció (inspection)

Leginkább úgy jellemezhetjük, hogy egy az egyben megvalósítja az átvizsgálásnál írott összes pontot. Minden lépésről dokumentáció készül, és nagyon kötött az egész struktúra. Az egyetlen említésre méltó különbség, amire oda kell figyelnünk, hogy az inspekciónak kimondva vállalt célja, hogy javítson a folyamatokon is, ne csak a dokumentumon/kódon magán. Ezt úgy tudjuk a legjobban támogatni, hogy ha nem csak a dokumentumhoz adunk javaslatokat, hanem a folyamathoz is, illetve javaslatokat teszünk, hogy a jövőben hogyan lehetne hatékonyabb az inspekció, illetve a fejlesztés folyamata.

Technikai átvizsgálás (technical review)

Céljában különbözik a többitől, pusztán egy-egy célszempont szerinti értékelésre van szükség, gyakori, hogy a szerző nem is vesz részt a megbeszélésen magán. Természetesen továbbítsuk a talált hibákat is, de ne veszítsük szem elől a célt. Ha nem érezzük magunkat technikailag elég képzettnek, ezt jelezzük, mert ez a típus igényli a legnagyobb szaktudást.

Informális átvizsgálás (informal review)

Sok esetben nincsen megbeszélés, hanem a jegyzeteinket közvetlenül a szerzőhöz juttatjuk el. Ha van időnk, felkérés nélkül is végezzük bátran. Mindenféle átnézés segítség.

Tesztelők, mint bírálók

A szakma jellegéből fakad, hogy a tesztelő jó bíráló lehet, sok tesztelési módszer alkalmazható. Próbáljunk meg minél több átvizsgáláson részt venni, főleg a projektek elején, ezáltal hamar el tudjuk majd kezdeni a tesztek kidolgozását. Minden valószínűség szerint egy tesztelő tud egy követelményt tesztelhetőségi szempontból ellenőrizni, ami fontos szempont, ha az elején nincs ennek prioritása, az nagyban megnehezítheti a projekt későbbi, tesztelés-intenzív fázisait.

Tanácsok a sikeres átvizsgálásokhoz

Alapfeltétel, hogy a bizalom meglegyen a résztvevők között, és alapvető a szakmai tekintély. Ha ez nincs meg, nem igazán van értelme időt és erőforrást szánni a folyamatra. Ahogy tapasztalatot szerzünk, érdemes listákba szednünk az általános szempontjainkat, esetleg a típushibákat is, ezzel gyorsítva a folyamatot. Végül, de nem utolsósorban ne féljünk segítséget/tréninget kérni, ha nem világos a folyamat, vagy hiányzik szaktudásunk a megfelelő részvételhez.

Gyakorlás

A következőkben oldjunk meg néhány, csoportos értékelésekre vonatkozó feladatot.

SZÉCHENYI 2020

1. Az alábbiak közül melyek lehetnek átvizsgálások sikerességét leíró sikerességi mutatók (success metrics)?

- **i. Nem minden átvizsgálásnak van előre meghatározott célja (predefined goal)**
- **ii. Ha hibát találunk, örülünk, és objektívan nyilatkozunk róla**
- **iii. A vezetőségnek támogatnia kell az átvizsgálásokat**
- **iv. Nagy hangsúlyt kell fektetni a tanulásra, és a folyamatok fejlesztésére**

- **a) ii, iii és iv helyesek, i helytelen**
- **b) iii, i és iv helyes, ii helytelen**
- **c) i, iii, iv helyes, ii helytelen**
- **d) ii helyes, i, iii és iv helytelen**

A helyes válasz az "a". Még ha nem is formálisan, de minden átvizsgálás-típushoz tartozik egy jól meghatározott cél, különben azt sem tudnánk, hogy mikor vagyunk készen. Emellett nem tudnánk sem a szempontokat, sem a módszereket jól megválasztani. A hibakeresés az egyik legfontosabb célja bármiféle átvizsgálásnak, az az állítás helyes. A vezetőség sajnos gyakran nem támogatja az átvizsgálásokat, főleg nem a több embert és időt igénylő, formálisabb fajtáit. Azonban erre szánt idő, es akarat nélkül mi hiába vagyunk tisztában a hasznukkal, így a vezetőség támogatása elengedhetetlen. Az átvizsgálási folyamatok ugyan jól definiáltak, de mindenkinek, és minden csoportnak önmagához kell ezeket igazítania. Ugyanis ez egy olyan folyamat, amit ki kell kényszerítenünk, de ha nem tudjuk természetes módon művelni, el fogja venni az időt és az energiát az érdemi munkától.

SZÉCHENYI 2020

2. Az alábbiak közül melyik kifejezés vonatkozik a legjobban a megadott karakterisztikákra, vagy átvizsgálási folyamatokra?

1. **A szerző vezeti**
2. **Nem dokumentált**
3. **Nem vesz részt a vezetőség**
4. **Képzett moderátor, vagy képzett vezető vezeti**
5. **Jól definiált be- és kilépési feltételeket (entry and exit criteria) használ**

s) Inspekció

t) Technikai átvizsgálás

u) Informális átvizsgálás

v) Átnézés

a) s = 4, t = 3, u = 2 és 5, v = 1

b) s = 4 és 5, t = 3, u = 2, v = 1

c) s = 1 és 5, t = 3, u = 2, v = 4

d) s = 5, t = 4, u = 3, v = 1 és 2

e) s = 4 és 5, t = 1, u = 2, v = 3

A helyes válasz a "b". Ha visszaemlékszünk az inspekció leírására, láthatjuk, hogy az egyik legkomplexebb, és legformálisabb átvizsgálás-típusról van szó. Nem tudnánk a formalitás e szintjét garantálni, ha az egész folyamatot nem egy tapasztalt, és a szerzőtől független személy vezetné. Ugyanezen formalitási okokból, mivel a céljaink és módszereink is komplexek, nem engedhetjük meg, hogy ne legyenek szigorú be- és kilépési feltételeink, különben az eredményünk sem lehetne annyira átfogó. A technikai átvizsgáláson sokféle ember, sokféle szerepkör is részt vehet, viszont a céljai nagyon távol állnak a vezetőség látókörébe tartozó céloktól. Általában olyan kollégákat, szakembereket kérünk fel, akik elfogulatlanul, befolyásolás nélkül, szinte csak szakmai és technikai szempontokat néznek, ez pedig nem elvárás a vezetőséggel szemben. Az informális átvizsgálás talán a legkevésbé formális átvizsgálás-típus, nagyon gyakran egyedül végezzük, és eredményeinket szóban, vagy rövid, informális leírás keretében hozzuk a szerző tudtára. Éppen ezért már folyamat szinten sem megkövetelt, hogy az

időt formális, ámde jelen céljaink elérésében keveset segítő dokumentumok előállításával töltsük. Az átnézést, ami félúton van a formális és nem formális megközelítések között, minden esetben a szerző vezeti. Nem lenne hatékony egy, a dokumentumot kevésbé ismerő személyre bízni ezt a feladatot, hiszen az átnézés viszonylag megengedő a célokat illetően, és ha a szerző szabadabb kezet kap, sokkal jobban a saját, leghasznosabbnak ítélt céljaira tudja felhasználni ezt a típusú átvizsgálást.

3. Tekinthetjük-e az átvizsgálásokat és inspekciókat (reviews and inspections) a tesztelés részeinek?

a) **Nem, mert a dokumentáción végezzük őket**
b) **Nem, mert még tesztelés előtt végezzük őket**
c) **Nem, mert a teszt dokumentációra (test documentation) nem alkalmazhatóak**
d) **Igen, mert hibákat találnak, és javítják a minőséget**
e) **Igen, mert tesztelésnek szamit minden nem konstruktív tevékenység**

A helyes válasz egyedül a "d". A többi válaszról elég könnyen beláthatjuk hamisságukat. Különböző dokumentációkon végezzük őket, amik ugyanúgy a termék részei, mint maga az esetleges programkód. Nem elhanyagolhatóbb, és nem kevésbé fontos a dokumentáció minősége sem. Nem a tesztelés előtt végezzük az átvizsgálásokat, az már maga a tesztelés része. Egy jó tesztelési folyamat együtt indul a projekt összes kezdeti folyamatával, és kitart egészen az életciklus végéig (the end of the life cycle). Minden tevékenység, amit e két időbeli pont között végzünk, és a minőség emelését, vagy a bizalom (confidence) növelését, vagy esetleges hibák feltárását célozza, tesztelésnek nevezhetünk. A tesztdokumentációra, típustól, fajtától függetlenül pontosan ugyanúgy, és ugyanazon átvizsgálási módszereket használhatjuk, mint bármiféle egyéb, a projekttel kapcsolatos dokumentációra. A résztvevők, és más specifikus jellemzők változnak, de a folyamatok maguk nem különböznek semmiben, nem is hívjuk őket másként. A tesztelés, ha nem lennének konstruktív részei, elég nehezen elképzelhető, hogy működhetne. Gondoljunk csak bele, tervek és adatok nélkül semmilyen folyamat nem működhetne, kezdve a kidolgozáson át, a kód előállításán keresztül egészen az üzemben tartásig. Elég könnyen beláthatjuk, hogy komplex folyamatokat nem lehet pusztán nem konstruktív tevékenységgel működtetni.

SZÉCHENYI 2020

4. Mi a legfontosabb különbség az átnézés, és az inspekció között?

a. Az inspekciót a szerző, míg az átnézést képzett moderátor vezeti

b. Az inspekciónak képzett vezetője van, az átnézésnek pedig nincs vezetője

c. A szerzők nincsenek jelen az inspekcióknál, de jelen vannak az átnézésekkor

d. Az átnézést a szerző vezeti, míg az inspekciót egy képzett moderátor

A helyes válasz a "d". Az inspekciót nem vezetheti a szerző, egyrészt mert általában olyan szakemberről beszélünk, akinek nem feltétlenül van meg a kellő tapasztalata és képességei, hogy ilyen komplexitású átvizsgálást vezessen, más oldalról pedig az inspekció folyamata előírja a függetlenséget erre a szerepkörre. Az átnézést ellenben leghatékonyabb, ha maga a szerző, vagy több szerző esetén az egyik szerző vezeti. Az egyetlen folyamat, aminél nem azonosítunk szerzőt, az informális átvizsgálás, hiszen ezt általában egyedül végezzük, és informális jellegéből fakadóan nem szükséges, és nem különösebben előnyös, ha bevonunk egy moderátort is. A szerzők jelen lehetnek az inspekcióknál, sőt, ez kívánatos is, hiszen az első kézből származó visszajelzés a legjobb. Járulékos előny pedig a közvetlen reagálás, és az ezzel kialakuló párbeszéd hasznossága. Mindezeket elveszítenénk, ha kitiltanánk a szerzőket az inspekciókról.

5. Az alábbiak közül melyik állítás igaz az átvizsgálásra?

a) Nem hajtható végre a felhasználói követelményeket leíró dokumentumokon

b) A legkevésbé hatékony módja, hogy a kód tesztelésének

c) Nagyon kicsi az esélye, hogy átvizsgálással hibát találnánk a teszt tervben

A helyes válasz a "d". Az átvizsgálások elsődleges célpontjai a különböző statikus dokumentumok, legyenek azok akár követelmények, akár maga a programkód, tehát biztonsággal állíthatjuk, hogy a felhasználói követelményeket (user requirements) leíró dokumentumokon is biztos végezhetjük e folyamatokat. A kód tesztelésének egy elég hatékony módja a kód átvizsgálása, sokféle olyan hibára mutathat rá, amely más módszerekkel nem, vagy csak magasabb költséggel, nehezebben lenne kimutatható. Ha a teszttervben hibák találhatók, azokat a legkönnyebben a megfelelő szakemberek bevonásával, és egy jól megválasztott átvizsgálási folyamattal hozhatjuk felszínre. Ezen kívül gyakorlatilag nincs is módszerünk, hogy a teszttervben, vagy bármiféle más tervben esetlegesen előforduló hibákat még azelőtt észrevegyük, mielőtt a javítás költségei aránytalanul magasra szöknének.

SZÉCHENYI 2020

6. Egy átvizsgálási folyamatban a moderátor az a személy, aki:

- **a) Feljegyzést készít a találkozón elhangzottakról**
- **b) Közvetít a résztvevők között**
- **c) Felveszi a telefonhívásokat**
- **d) Megírja az átnézésére szánt dokumentumot**

A helyes válasz a "b". Aki a feljegyzéseket készíti, írnoknak nevezzük, előfordulhat, hogy kevés számú résztvevőnél a moderátorra hárul az írnoki szerepkör is, ez nem feltétlenül kerülendő, ám sok résztvevőnél kívánatos, hogy a moderátor teljes figyelmét a saját feladatának tudja szentelni. Ilyenkor a moderálás, és általában az írnoki teendők is teljes embert kívánnak. A telefonhívások, és úgy általában a külső megzavarások hátrányosak az átvizsgálásokra nézve. Ez nem csak a moderátorra igaz, hanem az összes résztvevőre, sőt, a moderátor dolga hogy felügyelje a résztvevők ezen irányú tevékenységét, és lépjen közbe, ha kell. Az átnézésre szánt dokumentum elkészítése, és előkészítése a szerző feladata. Van azonban olyan eset, amikor ez a két szerepkör általában egybe esik, ez pedig az átnézés. Ez nem jelenti azt, hogy elfogadhatjuk ezt a választ helyesnek, hiszen más esetben, például az inspekcióknál kifejezetten ellenjavallott, hogy a szerzőt válasszuk moderátornak.

7. Általában milyen sorrendben követik egymást a lépések egy formális átvizsgálásnál?

- **a) Tervezés, felkészülés, indító ülés, találkozó, javítás, követés**
- **b) Indító ülés, tervezés, felkészülés, találkozó, javítás, követés**
- **c) Felkészülés, tervezés, indító ülés, találkozó, javítás, követés**
- **d) Tervezés, indító ülés, felkészülés, találkozó, javítás, követés**

A helyes válasz a "d". Ennek a folyamatnak a formális jellegéből fakadóan tervezéssel kell kezdődnie, mely leírja majd a pontos lépéseket, a szereplőket, a célokat, felsorolja a ki- és belépési feltételeket, illetve nyilatkozik a szükséges dokumentumokról és eszközökről is. Ha ez megvan, csak akkor következhet az indító ülés, gondoljunk csak bele, még azt sem tudnánk tervezés nélkül, hogy kit hívjunk meg, ráadásul arról sem lenne fogalmunk, hogy mit mondjunk nekik! Ezután az indító ülés után már minden adott kell, hogy legyen a szereplőknek, hogy érdemben fel tudjanak készülni a folyamat további lépéseire. Ezt követi majd a közös találkozó. Ennek eredményéről értesül a szerző, vagy értesülnek a szerzők, majd ha szükséges, javításokat tesz, vagy tesznek. Ezt követheti, ha szükséges még egy felkészülés, es találkozó, de ezekkel általában nem számolunk, egy kör sok esetben elég. A folyamat

SZÉCHENYI 2020

lezárása a követés, amikor összegezzük a tapasztalatokat, hogy legközelebb még hatékonyabbak legyünk, illetve a külső partnereket is értesítjük erről, és az elvégzett munkáról (például a vezetőséget is).

8. Kinek a felelőssége, hogy egy közös találkozón ledokumentálja az elhangzott hibákat, problémákat, és nyitott kérdéseket, pontokat?

a) **Moderátor**
b) **Írnok**
c) **Bírálók**
d) **Szerző**

A helyes válasz a "b". A moderátor szerepe a közvetítés, és a résztvevők közötti feszültségek, problémák elhárítása, megakadályozása, így nem lenne kívánatos, ha ezeken kívül más feladata is lenne. A bírálók feladata, hogy előadják szakmai véleményüket, gondolataikat, illetve hogy aktívan és konstruktívan részt vegyenek a kialakuló beszélgetésekben. Ha kis létszámú a csoport, elképzelhető, hogy valamelyik bírálóra jut az írnoki szerepkör is, ám akkor tisztában kell lennünk azzal, hogy az ilyen módon túlterhelt bíráló nem fog tudni teljes értékűen részt venni a szakmai kérdések megvitatásában, hiszen pont ezek lesznek a találkozó azon időpillanatai, amikor egy írnok leginkább elfoglalt, hogy a beszélgetéssel párhuzamosan a jegyzetekből ne maradjon ki semmilyen fontos szakmai részlet. A szerző nem minden esetben jó írnok, vegyük sorra a különböző átvizsgálás-típusokat. Átnézésnél a szerző maga a moderátor, három szerepkört pedig erősen ellenjavallott betölteni. Inspekciónál a szerzőnek a szakmai beszélgetésre kell figyelnie, illetve illik aktívan részt is vennie benne, ami kizárja a hatékony írnok szerepét számára. Technikai átvizsgálásnál a szerző nem is szokott jelen lenni, illetve informális átvizsgálásnál sem.

9. Mi az informális átvizsgálás fő célkitűzése?

a) **Olcsón elérni használható eredményeket**
b) **Hibákat találni**
c) **Tanulni, megérteni, hibákat találni**
d) **Megbeszélni, döntéseket hozni, technikai problémákat megoldani**

A helyes válasz az "a". A többi válasz is helyes lenne, ha a kérdésünk nem a fő célkitűzésre, hanem általános célokra vonatkozott volna. A hangsúly itt az olcsóságon van, a szerző általában személyesen kéri fel a bírálót, vagy bírálókat, közvetítő nélkül. Maga a folyamat sem tartalmaz felkészülést, az átnézés

maga is informális, és a talált hibák, vagy ötletek közlése is gyakran csak szóban történik. Mindez afelé mutat, hogy rövid idő alatt, kevés ember bevonásával, olcsón szülessen használható eredmény. A hibakeresés egyértelmű cél, de ez minden átvizsgálásra jellemző. A tanulás és megértés jó járulékos célok, mint ahogy a megbeszélés is. A döntések meghozatala, és a technikai problémák viszont már nem tartoznak az informális átvizsgálás kifejezett céljai közé, sőt, tágabb értelemben nézve semmilyen típusú átvizsgálásnak nem céljai.

Statikus elemzés (static analysis)

A statikus elemzés célja hasonlatos az átvizsgálásokhoz, viszont lényeges különbség, hogy eszközökkel, programokkal végezzük. Hétköznapi példát hozva az egyik leggyakoribb statikus elemzés a helyesírás-ellenőrzés. Sajnos sok esetben egy projekten ezen kívül csak a forráskód elemezhető statikusan, viszont az nagyon jó alanya az ilyen típusú vizsgálatoknak. De bármi más is elemezhető, ami szabványos formában van megadva, például UML, HTML, XML dokumentumok.

Fontos, hogy a legjobb statikus technológiák sem pótolják a dinamikus elemzést. Néhány esetben megállapítható potenciális veszélyforrás, de számos esetben statikus módszerekkel nem lehet következtetni a hibákra. Ezért kell mind a kétfelé közelítést ismernünk, és használnunk.

A következőkben megpróbálunk magyarázatokat fűzni a tankönyvben leírtakhoz, nem lesz célunk e statikus módszerek beható ismerete.

Milyen típusú hibákat találhatunk meg?

- Szintaktikai hibák
- Eltérések a konvencióktól és szabványoktól (conventions and standards)
- Egyedi, cégre/projektre jellemző szabályoktól való eltérés (amiket mi definiálunk az eszköz által elfogadott formában)
- Biztonsági rések (security vulnerabilities), potenciális biztonsági problémák
- Vezérlési folyam jelenségek (control flow anomalies)
- Adatfolyam jelenségek (data flow anomalies)
- Típushelyesség ellenőrzése (type matching)
- Deklarálatlan/inicializálatlan változók (undeclared /un-initialized variables)
- Nem használt változók (unused variables)
- Elérhetetlen kód (unreachable code)
- Konzisztencia hibák (consistency faults)

SZÉCHENYI 2020

Adatfolyam jelenségek

A tankönyvben szereplő leírásokat nézzük meg bővebben, hogy jobban megértsük az ilyen típusú jelenségeket. Ezek a típusú hibák a változók értékeinek változásával, illetve a felhasználásukkal kapcsolatosak.

Egy változónak háromféle állapota lehet:

- Definiált (d) (defined): a változó inicializált, és van értéke.
- Referált (r) (referenced): a változó értékét olvassa a kód, vagy használja a benne tárolt referenciát.
- Nem definiált (u) (unreferenced): a változó első értékadása még nem történt meg.

A jelenségeknek három fő típusa van (bár jegyezzük meg, itt nem szükségszerűen hibákról beszélünk):

- UR-anomália: Akkor lép fel, amikor a kód megpróbál egy inicializálatlan (u) változót használni (r) a futás során. A legtöbb fordító már eleve szól figyelmeztetéssel, hogy inicializálás során határozzunk meg kezdeti értéket. Az pedig általában súlyos hibának számít, ha UR módon akarunk használni egy változót.
- DU-anomália: Akkor figyelhetjük meg, ha egy változó értéket kap (d), viszont az adott kódrészletben, blokkban már nincs olyan rész, ami használhatná ezt az értéket (u). Természetéből fakadóan ez a jelenség csak a lokális változóknál érdekes. Ott sem tekinthető súlyos hibának, viszont érdemes megvizsgálni, hiszen fölöslegesen használt erőforrást szabadíthatunk fel.
- DD-anomália: Szintén erőforrás-pazarlás, akkor figyelhetjük meg, ha egy változó úgy kap újra értéket (d), hogy az előzőt nem használtuk fel (d). Ennek nyilvánvalóan nincs sok haszna, ezért érdemes felülvizsgálni.

Jobban megismerve a jelenségek típusait már láthatjuk, hogy nem kell feltétlenül hibaként kezelni, viszont nagyon gyakran utalnak kimaradt kódrészletekre, tervezési hibákra, átgondolatlanságra.

Vezérlési folyam jelenségek

A módszer lényege, hogy az egyben futó kódrészleteket csomópontokkal jelöljük, az elágazásokat és hurkokat pedig élekként értelmezzük közöttük. Ezáltal sokkal átláthatóbban értelmezhetjük a program működését. Könnyen kiszűrhetünk kiugrásokat összetett hurkokból, amik inkonzisztenciához vezethetnek. A nagyon komplex kódot, sok elágazással, általában érdemes felülvizsgálni, nagyobb

SZÉCHENYI 2020

eséllyel lesz hibás, mint egy letisztultabb, logikusabban szervezett kód. Hasonló módszerekkel könnyen látható, hol található úgynevezett "halott kód" a programban. Ez olyan kódot jelent, aminek elméleti esélye sincsen arra, hogy meghívódjon. Ez nyilvánvalóan tervezési hiba, esetlegesen egy változtatás tette elérhetetlenné, de minden esetben felül kell vizsgálni a kódot.

Ciklomatikus komplexitás (cyclomatic complexity)

A statikus elemzés rengetegféle metrika megállapítására jó, nézzük meg ezek közül a legelterjedtebbet, a ciklomatikus komplexitást. Ahogy azt a tankönyvben láthatjuk, a vezérlési folyamat gráf csomópontjait és éleit megszámolva könnyen megkaphatjuk az adott program, kódrészlet komplexitását. Ez a szám tesztelési szempontból nagyon hasznos, mert megmondja nekünk, hogy összesen hány teszteset kell, hogy az adott kód összes lehetséges bejárási útvonalát be tudjuk járni. Minél kisebb ez a szám, annál hatékonyabban tudjuk tesztelni. Ez a 100% ág-lefedettség eléréséhez kell, amiről itt nem értekezünk.

Gyakorlás

Az alábbiakban oldjunk meg néhány, a statikus elemzés témakörébe tartozó gyakorló feladatot.

1. Fontos előnye a kód átvizsgálásának:

- **a) Támogatja a kód tesztelését már akkor is, amikor a futtató környezet még nem áll rendelkezésre**
- **b) Végezheti az a személy, aki a kódot írta**
- **c) Végezheti tapasztalatlan személy is**
- **d) Olcsó végrehajtani**

A helyes válasz az "a". Nyilvánvaló lehet számunkra, hogy tesztelésnek számítanak a statikus módszerek is, és elég sok módszert ismerünk már, ami hibák megtalálásával, és más előnyökkel, például a minőség javulásával kecsegtetnek. Ezen módszerek másik nagy előnye, hogy nagyon korán elkezdhetjük, amikor még a futtatás nem megoldható, viszont a hibák kijavításának költsége még relatíve kicsi. Az átvizsgálást végezheti ugyan az a személy, aki a kódot írta, de semmiképpen nem fontos előny, vagy kívánatos cél ez. Sokkal hatékonyabb, ha más személyre bízzuk ezt, ez új nézőpontot, és más megközelítést hozhat. Sajnos elég jellemző trend, hogy amit elsőre nem veszünk észre, legyen az hiba, vagy olyan pont, amin lehetne javítani, azt később sokkal nehezebben vesszük észre, szemben egy olyan személlyel, aki még egyszer sem siklott át felette. Tapasztalatlan személyre semmiképp nem érdemes sem ilyen, sem más típusú átvizsgálást bízni, amennyiben a minőség javítása, vagy hibakeresés a cél. Ettől független van haszna, egy tapasztalatlan személy sokat tanulhat, ha átnézi és megérti egy tapasztaltabb kolléga kódját, ám ez nem

SZÉCHENYI 2020

a legfontosabb előnye az ilyen típusú tevékenységnek. Hogy olcsó-e végrehajtani, az egy relatív kérdés, azonban általánosan kimondhatjuk, hogy nem olcsóbb, mint más olyan tevékenységeket végezni, amelyek ugyanezen célt szolgálják. Egy alapos átnézéshez fel kell fognunk nem csak az átnézni kívánt kódot, hanem annak környezetét is, ami hosszadalmas lehet. Ezen felül visszajelzést kell adnunk, esetlegesen szóban is egyeztetnünk kell a kód szerzőjével, mindezek pedig a hasznosságot tartják szem előtt, és nem cél, hogy a minőség rovására faragjunk a költségeken.

2. Milyen defektusokat találhatunk statikus elemzés segítségével?

- **i. Nem használt változók**
- **ii. Biztonsági sebezhetőségek**
- **iii. Kódolási előírásoktól való eltérés (deviation from the coding standards)**
- **iv. Nem hívott függvények és eljárások**

- **a) i, ii, iii és iv is helyes**
- **b) iii helyes, i, ii, iv helytelen**
- **c) i, ii, iii és iv is helytelen**
- **d) iv és ii helyes, i és iii helytelen**

A helyes válasz az "a". A statikus elemzés egy nagyon erős eszköz lehet, ha jól használjuk. Ha megnézzük, a nem használt programelemek gond nélkül kiszűrhetőek futtatás nélkül is. A biztonsági kockázatok is jól körül határolhatóak a kódban, legalábbis az esetleges sebezhetőségek potenciális előfordulása a kód elemzésével is megállapítható. Az előírt kódolási szabályokat pedig nem is tudnánk futtatással ellenőrizni, hiszen (technológiától és nyelvtől függően) nem feltétlen áll már akkor rendelkezésre a forráskód, illetve a futtatás egyébként sem ad hozzá új információt a kód formátumának vizsgálatához.

SZÉCHENYI 2020

3. Az alábbiak közül melyiket NEM találhatjuk meg statikus elemzés segítségével?

a) Ha egy változóra korábban hivatkozunk, mint ahogy használnánk

b) Elérhetetlen ("halott") kód

c) Memóriaszivárgás (memory leak)

d) Ha újradefiniálunk egy változót, mielőtt először használnánk

A helyes válasz a "c". A kód elemzésével könnyen megállapíthatjuk, ha egy változó nem lett inicializálva z adott blokkban. Ha a kód egy részére nem történik (még elméletileg sem) hivatkozás, ezt is könnyen detektálhatjuk statikus módszerekkel. Persze több mint valószínű, hogy egy-egy futás alkalmával nem fut le egy kód összes része, ám ettől az aktuálisan nem futtatott részéket nem tekintjük halott kódnak, csak azokat, amelyeket ha akarnánk, sem tudnánk elérni a kód módosítása nélkül. A memóriaszivárgás tipikus futásidejű (runtime) probléma, pusztán statikus módszerekkel nagyon csekély esélyeink vannak ellene, érdemesebb futásidejű elemzőket (debugger) használnunk. A változók definiálása és újradefiniálása viszont statikus módszerekkel jól lekövethető probléma.

SZÉCHENYI 2020

Dinamikus technikák (dynamic analysis)

Nagyon egyszerűen definiálhatjuk, hogy miben különböznek a dinamikus módszerek a statikusoktól. Ha a futó rendszert elemezzük működés közben, akkor dinamikus technikákat használva tehetjük ezt meg, futtatás nélkül lehetetlen ilyen technikákhoz folyamodnunk. Ne tévesszen meg minket, ha a komponens, vagy részegység önmagában nem fut, és nekünk kell előállítani futtató környezetet, ettől ez még teljes értékű dinamikus elemzésnek számít. A következőkben nem fogunk értekezni a kód ilyen formában történő kiegészítéséről, figyelmünket inkább a teszttervezés (test design) felé fogjuk fordítani. Célunk, hogy megismerjünk sokféle módszert, amikkel megvalósíthatjuk a szisztematikus megközelítést, és belássuk, miért is hasznos ez a megközelítés.

A helyes megközelítés az alábbi fázisokra bontható:

- Meghatározni az előfeltételeket (entry criteria), amelyek teljesülése esetén kezdhetünk neki a végrehajtásnak. Ide érdemes a szükséges és elégséges feltételeket felsorolnunk, csak azokat, amik nélkülözhetetlenek.
- Teszteseteket definiálni (define test cases), erre veszünk át módszereket, és nézünk példákat a következőkben. Először átvesszük az alapvető módszereket példákkal, aztán kombinálva használjuk őket a jobb lefedettség elérésének érdekében.
- Meghatározni a tesztesetek végrehajtásának a módját. Erre nem szánunk sok időt, lévén a futtatás mikéntjét legcélszerűbb valós körülmények között gyakorolni.

Az előfeltételeket és feltételeket szigorúan kell vennünk, amennyiben nem teljesülnek, nem kezdhetjük el, illetve nem érdemes befejeznünk a tesztelést. Például, ha a feladatunk, hogy 10 felhasználó adatainak ellenőrizzük a migrációját, nem érdemes belekezdenünk, amíg nem került minden adat migrálásra, és nem érdemes befejeznünk, amíg minden adat átnézésre nem került. Kérdezhetnénk, miért ne kezdhetnénk bele, amikor 5 felhasználó adatai készen állnak, de vegyük alapvetésnek, hogy csak akkor kezdjünk neki a tesztelésnek, ha már nem várunk változtatást a rendszeren. Ugyanígy, fals pozitív eredményt kaphatunk, ha 8 adat hibátlan átvizsgálása után feltételezzük, hogy az utolsó 2 felhasználó adatai sem lesznek hibásak, és emiatt elmulasztjuk a megvizsgálásukat. Jelen feladathoz akkor járunk el körültekintően, ha belépési előfeltételnek megadjuk, hogy a teljes migráció legyen készen, kilépési végfeltételnek pedig hogy minden felhasználói migrált adat egyenlő és egyforma alapossággal le legyen tesztelve, és minden ellenőrzött adat egyezzen meg a forrással. Természetesen a valós feladatok nem mindig ennyire bőkezűek a felhasználható idővel, de a meghozható és szükséges kompromisszumokról egy következő fejezetben ejtünk majd szót csak.

SZÉCHENYI 2020

Ha megvannak a be- és kilépési feltételek, érdemes kitalálnunk egy olyan teszteset-struktúrát, amely hatékonyan segít bennünket a kitűzött célok elérésében. Jelen esetben érdemes a legkisebb, már egyben megfogható egységet azonosítani egy tesztesetnek, tehát például egy adott felhasználó adatai kerüljenek egy tesztesetbe. Itt jön be a képbe a követhetőség (traceability) fontossága: minden esetben tároljuk le, hogy melyik teszteset melyik követelményt fedi le. Enélkül, amennyiben változás következne be a követelményekben (például egy vagy több felhasználó forrásadatai megváltoznának, ami magával hozza, hogy a migrációt újra kell futtatni, és ezáltal ellenőrizni), egy pontos követhetőségi mátrix (traceability matrix) hiányában át kellene néznünk az összes tesztesetet. Ez jelentős, és ráadásul felesleges plusz időbefektetéssel járna. Viszont ha rendelkezünk egy teljes követhetőségi mátrixszal, könnyen és gyorsan azonosíthatjuk azt a pár tesztesetet, amelyek frissítésre szorulnak.

Az alábbiakban nem fogunk kitérni a tesztesetek formális leírásának szabványaira, mindössze a megtervezésükhöz szükséges módszereket, valamint a teszt beviteli adatokat és elvárt eredményeiket fogjuk definiálni. A leírás módszertanát egy következő fejezetben vesszük át, illetve az érdeklődőknek érdemes megtekinteni az IEEE 829 szabványcsomagot, amely bőséges leírásokat tartalmaz sokféle, tesztelést segítő dokumentumformátumról.

A végrehajtás mikéntjéről röviden annyit, hogy legtöbbször nem hatékony önmagában álló teszteseteket végrehajtani, érdemesebb inkább valamilyen szempont szerint csoportosítani őket. Erről általában valamilyen teszt végrehajtási terv értekezik. Amennyiben a teszteseteinket berendeztük e csoportokba, érdemes már konkrét időzítéseket és felelősöket rendelünk a különböző csoportokhoz a végrehajtás megkezdése előtt. Amennyiben egy-egy ilyen teszteset automatizálásra kerül, általában tesztszkriptként (test script) hivatkozunk rá a későbbiekben.

A tesztesetek megtervezéséhez felhasználható technikákat két nagy csoportra szoktunk osztani, az úgynevezett fekete doboz és fehér doboz technikákra (black box and white box techniques).

A fekete doboz technikák esetén a tesztelt rendszert vagy komponenst egy átlátszatlan doboznak tekintjük, aminek csak a bemenetét és a kimenetét tudjuk kezelni, olvasni. A kimeneteket gyűjtőnéven megfigyelési pontnak (observation point), míg a bemeneteket vezérlési pontnak (control point) is hívhatjuk. Dióhéjban egy fekete doboz típusú technika alkalmazása nem más, mint az ismert rendszerspecifikációk (system specification) alapján történő modellalkotás, majd e modell segítségével a szükséges bementi adatok és a belőlük fakadó kimeneti adatok összességének meghatározása.

Ezzel ellentétben a fehér doboz technikák magának a rendszernek és a komponenseknek a működésére koncentrálnak, és ezt vizsgálva állapítják meg az elvárt lefedettséghez (coverage) szükséges teszteket. Ez a megközelítés nagyon hasznos olyan tesztek megtervezésére, amelyek nehezen vagy egyáltalán nem érhetőek el akaratlagosan pusztán fekete doboz megközelítést használva. Ez nem csak a releváns teszteredmények megszerzése végett fontos, hanem a megfelelő lefedettség elérésében is.

Ezen technikák sok más néven is ismeretesek, lévén sokféle módszertan és terminológia használja ugyanezt a megközelítést az ISTQB módszertanon kívül is. A fekete doboz technikákat szoktuk még funkcionális, specifikáció alapú, viselkedési, illetve zárt doboz technikáknak is hívni (functional, specification-based, behaviorial, closed box). A fehér doboz technikákat pedig strukturális, ritkábban üvegdoboz technikáknak (structural, glass-box) is hívhatjuk, illetve típus alapján beszélhetünk külön komponens hierarchia (component hierarchy), vezérlési folyam és adatfolyam technikákról is.

Általában alacsony szintű teszteknél (egység, integrációs és rendszer) érdemes a fehér doboz technikákat részesítenünk előnyben, míg a magasabb szintű (felületi, alkalmazás szintű, funkcionális, átvételi) tesztekhez pedig a fekete doboz technikák illenek jobban. Ugyan a legtöbb technikáról egyértelműen eldönthető, hogy melyik nagy csoportba tartozik, de néhány esetben, ahol a két megközelítés fedi egymást ott szürke doboz (grey box) technikáról beszélhetünk.

Ugyan mind a két csoport formális technikák alkalmazását írja elő, de a fekete doboz technikákhoz szoktunk sorolni két speciális fajtát, az intuitív és a tapasztalat alapú teszteléseket (intuitive and experience-based) is. Ezek a technikák leginkább informálisak és a végrehajtójuk tapasztalatára alapozva érnek el eredményeket.

Fekete doboz technikák

Ezen technikák alapötlete, hogy a specifikációból következő összes értelmes bemeneti adat és hozzátartozó kimeneti adat egy valódi részhalmazát határozza meg oly módon, hogy az így kapott szűkebb halmaz is várhatóan ugyanolyan hatékonysággal állapítsa meg a helyes működést, illetve tárja fel a hibákat, mint az összes esetet tartalmazó halmaz.

Ekvivalencia particionálás (equivalence partitioning)

A technika alapötlete, hogy a különböző határértékek (boundaries) mentén egyenrangúként kezelt, úgynevezett ekvivalencia osztályokra (equivalence classes) bontja fel a bemeneti adatok értelmezési tartományát (partition). Nagyon fontos megjegyeznünk, hogy egy osztályba tartozó minden értékhez ugyanazt a kimeneti értéket várjuk, amennyiben ez nem teljesül, rosszul állapítottuk meg az osztályokat, és minden valószínűség szerint további osztályokra szükséges bontanunk őket. Vegyünk át egy példát:

SZÉCHENYI 2020

Célunk, hogy leteszteljünk egy kedvezmény kiszámító programot. A számunkra érdekes előírások az alábbiak: amennyiben az eladási ár 15000 dollár, vagy kevesebb, semmilyen kedvezményt nem kap a vásárló, 20000 dollárig 5%-os kedvezményt adhatunk, 25000 dollárig a kedvezmény 7%, efölött pedig 8,5%. Ezen követelményekből az alábbi táblázat vehető fel az ekvivalencia osztályok azonosítására:

Ekvivalencia osztály		**Példa**	**Kedvezmény (%)**	**Új ár**
1	X < 0	-13500	-	-
2	0 <= X <= 15000	9400	0	9400
3	15000 < X <=20000	19850	5	18857,5
4	20000 < X <=25000	23560	7	21910,8
5	X > 25000	31890	8,5	29179,35

A fenti táblázat minden osztályából elegendő egyetlen értéket választanunk, hiszen az elvárások alapján egyazon osztály tetszőlegesen sok tagjától sem várunk el különböző viselkedést. Azonban az így felvett osztályok még nem fedik le a teljes tartományt, meg kell tehát állapítanunk olyan osztályokat is, amelyek a nem szokásos és nem előírt működéseket tesztelik. A fenti példában ezt az egyes sor tartalmazza, lévén alapesetben a vételárat nem értelmezzük a negatív számok tartományára.

Ha jól csináltuk, ezen a ponton definiált osztályainknak le kell fedniük a teljes tartományt. Ezt könnyen ellenőrizhetjük, ha megvizsgáljuk milyen típusú adattal is dolgozunk. Az egyszerűség kedvéért a centektől most eltekintve megállapíthatjuk, hogy a tartományunk az egész számok halmaza, akkor végeztünk jó munkát, ha nem tudunk olyan elemet mondani a tartományból, amely nem tartozik bele valamelyik osztályba. Ahhoz, hogy ezt jelen példában elérjük, két félig nyitott osztályt is használnunk kell, mégpedig az egyik a negatív számok, a másik pedig a 25000 fölötti pozitív számok osztálya. Még utolsó ellenőrzésként vizsgáljuk meg, hogy biztosan kapott-e saját osztályt minden önmagában is azonosítható, megfogható követelmény.

A technika egyszerűnek tűnhet, bár van pár megközelítés, amit mindenképpen figyelembe kell vennünk. Az első a határok pontos megállapítása, hiszen rosszul megállapított határokkal az osztályok is pontatlan eredményekhez vezetnek. Éppen ezért ezt a technikát karöltve szokták alkalmazni a határérték

SZÉCHENYI 2020

analízissel (boundary value analysis), melyről a következőekben külön is szót ejtünk. A második sokkal inkább a megvalósításhoz köthető, lévén a számítógép memóriája és adattípusai is végesek, ezért érdemes e határpontokról is tájékozódnunk és felhasználnunk őket tesztjeinkhez. A fenti példát alapul véve például nehézkes lehet negatív számokkal dolgozni, amennyiben a megfelelő beviteli mező nem ad erre lehetőséget. Hasonló logika alapján a használt adattípus felső korlátjánál nagyobb számot választva tesztesetnek valószínűleg előre nem várt hibába fogunk ütközni. Ezen fizikai megkötésekre ne úgy tekintsünk, mint hátráltató tényezőkre, hanem az ezek által megszabott korlátokat is kezeljük hasonlóan a specifikációból jövő határokkal.

A valós tesztelési helyzetekben általában nincs ilyen egyszerű dolgunk, hogy a kimenetet egy vagy több, de független bemenet szabályozza. Több, nem független bemenet esetén kombinálnunk kell a különböző bemenetek osztályait, mert csak így érhetjük el a megfelelő lefedettséget. A fenti példát kiegészítve a bolt, ha árul egy terméket háromféle színben és minden szín elérhető mind a négy érvényes partícióban, akkor ideális esetben 12 kombinációt kell letesztelnünk. Sajnos sok esetben a dolgunk még ennél is nehezebb, ha három vagy annál is több bemeneti dimenzióval kell számolnunk, mert a szükséges tesztesetek száma könnyen irreális mennyiségűre nőhet. Ezen esetek kezelésére most nem térünk ki, bővebb tárgyalásuk a tesztminimalizálással (test minimization) foglalkozó fejezetben lesz fellelhető.

A technikához, mint minden más formális technikához számolható lefedettség egy egyszerű képlettel:

$$\frac{lefedett\ partíciók\ száma}{összes\ partíció\ száma} \times 100\%$$

Határérték-analízis

Az ekvivalencia particionálás módszere kiindulási alapnak jó, de mint sejthető, van gyenge pontja. Ez a gyenge pont az osztályokat elválasztó határértékek kezelésében keresendő. Amennyiben az osztályok diszkrétek, akkor használhatjuk ezt a technikát, és érdemes is használnunk, mert mint az sejthető, a határokon és közelükben általában nagyobb eséllyel akadhatunk hibára, mint például a tartományok közepén.

A megközelítés viszonylag egyszerű, egy határ három tesztesetet határoz meg, a határt magát, illetve mind a két legközelebbi szomszédját (itt emlékezzünk, hogy egy tesztadat nem pusztán a bemeneti értékből áll, hanem meg kell határoznunk hozzá az elvárt kimenetet is). Ez utóbbi kettő meghatározásához ismernünk kell a minimális növekményt. Egész számoknál a lépték 1, viszont lebegőpontos számoknál nekünk kell meghatároznunk egy tűrést, amin belül az értékek már különbözőnek tekinthetők. Általában elegendő, hogy tizedesekben gondolkodunk, nem mindig érdemes

precízen figyelembe venni az aktuális fizikai számábrázolást az adott futtatási környezetben. Hogy érthetőbb legyen, például ha a határunk 15, és lebegőpontos a tartomány, akkor a 14.9; 15; 15.1 jó tesztesetek lehetnek. Érdemes külön kezelni a 15 és a 15.0 eseteket, amik nekünk logikailag ugyan egyenértékűek, de konvertálástól függően a tesztelt rendszerben számíthatnak két külön értéknek.

Ha jobban belegondolunk, valójában a két szomszédos érték már része az ekvivalencia osztályoknak, de kiemelt szerepük miatt külön teszteljük őket. Van olyan elv, mely szerint a > és < típusú szabályoknál elegendőnek tekinti 2 eset alkalmazását, hiszen például < esetén a határ, és a felső szomszédja definíció szerint ugyanazt az eredményt kellene, hogy produkálja. Mi használjuk nyugodtan mindig mind a két szomszédot minden esetben, sokkal jobb lefedettséget ad, és gyakrabban tár fel hibákat, mint amennyi plusz időbe kerül az alkalmazásuk. Sajnos a sokszor a specifikáció pongyola, mint például: "A kedvezmény 25000 dollár alatt 7%, afölött pedig 8,5%." Folyó szövegként olvasva fel sem tűnik, viszont teszteset tervezés közben már szemet kell, hogy szúrjon, hogy ez a megfogalmazás nyitva hagyja a kérdést, hogy kereken 25000 dollárnál mennyi az elvárt kedvezmény. Lássuk a fentebbi példához tartozó határérték táblázatot:

Alsó szomszéd	**Határérték**	**Felső szomszéd**
-1	**0**	1
14999	**15000**	15001
19999	**20000**	20001
24999	**25000**	25001

Az azonosított tesztesetek száma 12, mint az a táblázatból jól látható.

Természetesen a végtelenben végződő osztályoknak csak az egyik oldali határa tesztelhető, mint ahogy a nem diszkrét tartományokra sem értelmezhető ez a módszer. Ebben a tekintetben különbözik az ekvivalencia particionálás módszerétől. Hogy mit is értünk ez alatt? Például ha különböző szemszínek alapján van definiálva a működés, akkor egyszerűen megadhatunk osztályokat, minden lehetséges színtartományt egy osztálynak véve, de nagyon nehéz lenne pontos határokat keresnünk ehhez a megközelítéshez.

Ennél a technikánál is vegyük figyelembe a fizikai megvalósításból (physical representation) fakadó határokat a specifikációból következők mellett, csak immár új, extra határértékként értelmezve őket. Természetesen itt is értelmezhetünk lefedettséget, ami nagyon hasonlatos az előzőhöz:

*letesztelt tesztesetek száma (beleértve a szomszédokat is)/összes tesztesetek száma*100%*

Zárógondolatként jegyezzük meg hogy a particionálás és határérték-elemzés együtt alkalmazandó, még akkor is, ha az egyszerűbb megértés érdekében külön tárgyaltuk őket.

Állapotátmenet tesztelés (state transition testing)

Az előbbiekben nagyon hatékony példákat láthattunk azokra az egyszerű esetekre, amikor nem kellett figyelembe vennünk a végrehajtás történetét, tehát a tesztesetek tetszőleges sorrendben ellenőrizhetőek voltak, sorrendjük, illetve a tény, hogy egyáltalán végrehajtottuk-e őket nem befolyásolták az eredményeket. Amennyiben a helyzet ennél bonyolultabb, a véges automaták elméletét (theory of finite automatas) érdemes segítségül hívnunk.

A fogalmak ismertetésétől most eltekintünk, ismeretüket alapvetésnek tekintjük. Ezen teszteléshez a rendszert állapotokkal írjuk le, az átmeneteket pedig végrehajtható tesztlépéseknek tekintjük. Fontos, hogy a különböző állapotokból elérhető összes lépést bejárjuk, amennyiben a specifikáció alapján ez nem lehetséges, jelezzük az illetékesnek, hogy a specifikáció kiegészítésre szorul. A végállapotnak, vagy végállapotoknak kiemelt szerepe van, egy teszteset akkor teljes, ha végállapotban ér véget, és akkor van elegendő tesztesetünk, ha minden lehetséges utat bejártunk, ugyanis egy bejárási út pontosan egy tesztesetet ír le.

Nézzünk egy szemléletes példát, amelyen keresztül megnézünk egy hatékony teszteset leírási módot is.

Feladatunk egy webáruház kosarának tesztelése. A kosár üresen indul, maximum 3 termék helyezhető bele, és ha a kosár nem üres, továbbléphet a felhasználó a fizetéshez, ami egyúttal a kosarat is üríti (a fizetést most nem teszteljük). Ezen kívül elérhető egy kosár ürítése funkció, amely tartalomtól függetlenül üríti a kosarat. A gráfot az átláthatóság kedvéért most táblázattal reprezentáljuk, de érdemes gyakorlásként felrajzolnunk belőle a gráfot is. A sorokban a leírásból kiolvasható összes állapotok szerepelnek, az oszlopok fejlécei pedig az azonosított átmenetek szerepelnek. Vizsgáljuk meg, hogy ki tudunk-e tölteni minden mezőt a leírás alapján, mert ha nem, kiegészítést kell kérnünk a specifikációhoz.

Kiindulási állapot	Elem hozzáadása gomb	Fizetés gomb	Kosár ürítése gomb
Üres kosár (S1)	S2	-	-
1 elem a kosárban (S2)	S3	S5	S1
2 elem a kosárban (S3)	S4	S5	S1
3 elem a kosárban (S4)	-	S5	S1
Fizetési képernyő (S5)	-	-	-

Egy fentihez hasonlóan megtervezett ábra sokat segíthet nem csak az elvárt, de a hibás/nem támogatott esetek azonosításában is. Hiszen a legtöbb esetben az elvárás nem tér ki a nem elvárt akció végrehajtására (például megnyomni a "fizetés" gombot egy webshopban, amikor a kosár üres), viszont a rendszernek erre is fel kell lennie készülve (például a "fizetés" gombot ilyen esetekben elérhetetlenné kell tenni, esetleg a felhasználót is lehet figyelmeztetni, hogy a kosár üres). A fenti példában a "-" értéket tartalmazó cellák mutatják a nem értelmes átmeneteket, ezáltal remek negatív teszteseteket biztosítva nekünk.

Immár egy példán keresztül megismerve a technikát, vegyük sorra, mikre lehet szükségünk, hogy teszteseteket alkossunk a segítségével:

- A kiindulási állapotra (starting state)(például üres a kosár, a kezdőképernyőn állunk)
- A szükséges lépésekre (például terméket kiválaszt, másik terméket kiválaszt, "fizetés"re kattint)
- A lépések elvárt eredményeire (például a kosárban egy termék lesz, a kosárban két termék lesz, a kosár kiürül, majd az egyenlegünk csökken)
- Az elvárt végállapot leírására (end state) (például a rendszer két termék sikeres megrendelését rögzítette, az egyenlegünk pedig a megfelelő összeggel csökkent)

A teszteseteken belüli átmenetekhez pedig az alábbiak szükségesek:

- A kiindulási állapot (például a kosár üres)
- Az esemény (trigger), amely kiváltja az állapotváltást (például egy termék kosárba helyezése)
- Az esemény hatására elvárt átmenet (transition) (például a kosár növelése egy termékkel)
- Az elvárt következő állapot (például a kosárban egy termék van)

Nem minden esetben ilyen egyszerű a rendszer összes a kód lehetséges állapotát leírnunk, az előzőekben megismert módszerekhez hasonlatosan itt is használhatunk több dimenziós eseteket. Azonban a legtöbb esetben ez aránytalanul nagy időbefektetést igényelne, ezért e technikák általában különálló működések tesztelésére használhatóak hatékonyan, például a fenti példában sem foglalkoztunk a vásárló egyenlegével, sem más egyéb aspektusával a használt rendszernek, viszont azokat külön megvizsgálhatnánk, akár ezzel a módszerrel is. Ennek a technikának a fő ereje az objektumorientáltan megvalósított üzleti logikák (business logic) tesztelésében rejlik.

Logika alapú technikák (logic based techniques)

A korábbiakban megismert technikák alapjául csak a bemeneti és kimeneti adatokra vonatkozó specifikációk szolgáltak. A következőkben olyan döntési módszereket nézünk meg, amelyek logikai összefüggéseket keresnek a bemeneti és kimeneti adatok között, és ezáltal szűkítik le az összes tesztesetek halmazát a releránsakra és szükségesekre. A legtöbb ilyen technika alapja az ok-okozati gráf, melyet részleteiben nem tárgyalunk.

Döntési tábla tesztelés (decision table testing)

Az ok-okozati gráf (cause-effect graph) megközelítésre épülő technikák közül a legelterjedtebb talán a döntési tábla. Az alábbiakban csak ezzel fogunk részletesen foglalkozni. Általános reprezentációja egy mátrix, melynek oszlopai definiálják a teszteseteket, sorai pedig a teszteseteket leíró feltételeket, illetve lehetséges lépéseket. Az alapötlet, hogy a táblázat könnyen kezelhető maradjon, hogy a rácspontokba (cell) kizárólag logikai értékek kerülhetnek. Amennyiben mégis többelemű rácsra lenne szükségünk, vizsgáljuk meg újra a feltételeket és lehetséges lépéseket, és addig bontsuk, ameddig már minden sorban foglalt logikai feltétel reprezentálható binárisan. Az alábbi példán keresztül megérthetjük miért is praktikus ez a megközelítés.

Feladatunk, hogy pénzt szerezzünk egy bankjegy kiadó automatából, ehhez az alábbi feltételek teljesülése szükséges:

- A bankkártyánk érvényes
- Helyes PIN kódot adtunk meg
- Legfeljebb háromszor próbálkoztunk a PIN kód megadásával
- Van elegendő pénz a gépben is, és a számlánkon is

A gép az alábbi műveletekre képes:

- Kártya visszautasítása

SZÉCHENYI 2020

- PIN újbóli bekérése
- Kártya elnyelése
- Új kívánt összeg bekérése
- Kifizetés

		Eset 1	Eset 2	Eset 3	Eset 4	Eset 5
Feltételek	**A bankkártyánk érvényes?**	N	I	I	I	I
	Helyes PIN kódot adtunk meg?	-	N	N	I	I
	Legfeljebb háromszor próbálkoztunk a PIN kód megadásával?	-	N	I	-	-
	Van elegendő pénz a gépben is, és a számlánkon is?	-	-	-	N	I
Műveletek	**Kártya visszautasítása**	I	N	N	N	N
	PIN újbóli bekérése	N	I	N	N	N
	Kártya elnyelése	N	N	I	N	N
	Új kívánt összeg bekérése	N	N	N	I	N
	Kifizetés	N	N	N	N	I

A táblázatunk az I és N értelemszerűen a logikai Igent és Nemet jelöli, az alábbiak szerint:

- Feltétel Nem esetén az adott feltétel nem teljesül
- Feltétel Igen esetén az adott feltétel teljesül
- Művelet Nem esetén az adott művelet nem hajtható végre/nem érhető el
- Művelet Igen esetén az adott művelet végrehajtható/elérhető

Mint láthatjuk, valójában nem szükséges minden tesztesetnek minden feltételre érvényes logikai megállapítást tartalmaznia, hiszen elképzelhető, hogy az a feltétel csak a vizsgált folyamat egy későbbi

pontján válik értelmezhetővé (például ha a bankkártya elbukik az érvényességi vizsgálaton, nem tudjuk értelmezni, hogy áll-e rendelkezésre pozitív egyenleg, hiszen ehhez a tesztesethez nem tudjuk meghatározni a használni kívánt számlát). Amennyiben nem használjuk még ezt a technikát magabiztosan, egyszerűbb ezeket az eseteket is logikai nemként kezelni, ezáltal könnyebben leellenőrizhetjük, hogy a teszteseteink lefedik-e az összes lehetséges kombinációt, melyeknek a száma 2^n, ahol "n" az összes feltételek száma. Ha jól végeztük el a feltételek megállapítását, és a specifikáció is teljes, akkor minden akcióhoz egyértelműen meg kell tudnunk mondani, hogy az adott tesztesethez tartozó egyedi feltétel-együttállás (combination of conditions) esetén melyik akció elérhető és melyik nem (például a fenti példa kettes számú tesztesetében egyértelműen el tudjuk dönteni, hogy csak a pin újbóli bekérése kell, hogy elérhető legyen, a többi pedig egyértelműen nem).

Mint azt láttuk, ha megvizsgáljuk az összes lehetséges sort, az azzal a nem elhanyagolható előnnyel is jár, hogy fel tudjuk deríteni a specifikációból esetlegesen hiányzó részeket. A technika másik nagy előnye, hogy gyakorlatilag bármilyen más formális fekete doboz technika reprezentálható a döntési táblákkal (emlékezzünk, hogy döntési gráfot is megadhatunk meg hasonló táblázattal).

Használati eset alapú tesztek (use-case-based testing)

A hasonlóság alapján fekete doboz tesztnek minősített, ám valójában nem teljesen formális technika talán a legfontosabb mind közül, hiszen az alapját nem a felhasználói elvárásokból származtatott formális követelmények adják, hanem maguk az eredeti felhasználói elvárások. A legtöbb esetben nem technikai emberek a felhasználók, így elvárásaik sem formálisan megfogalmazottak. Éppen ezért a mi feladatunk, hogy mégis formális teszteseteket tudjunk alkotni az elvárásaikból. Az esetek döntő többségében ehhez speciális döntési gráfokat veszünk fel, amelyben egy-egy állapot nem feltétlenül értelmezhető a rendszernek ebben a formában, csak több állapotként, viszont a felhasználó szempontjából mégis egy lépésnek számít. Hogy jobban megértsük, például az előbbi bankautomatás esetben, amikor a megrendelő elmondja nekünk, hogy hogyan szeretné, hogy működjön az automata, valószínűleg egy informális mondattal ("a kártya behelyezés után kérje be a pin kódot a gép") el fogja intézni azt az egész működést, amit nekünk többféle technika használatával egyébként majd alaposan meg kell terveznünk és le kell tesztelnünk.

Speciális még ez a döntési gráf abból a szempontból is, hogy szinte minden esetben a felhasználó szemszögéből figyeli meg a rendszert. Éppen ezért ezek az esetek szinte sosem tartalmaznak olyan bemeneteket, amelyek nem a felhasználótól jönnek (például a banki központi rendszer visszajelzése a rendelkezésre álló pontos összegről), illetve azon kimenetek is hiányoznak, amelyeket a rendszer nem

közvetlenül a felhasználónak nyújt (például a formális naplóbejegyzés a banki központi rendszernek a tranzakció részletes végrehajtásáról).

A technika félig informális, felhasználó-központú jellegéből fakadóan csak nagyon magas szintű be- illetve kimeneti feltételeket tudunk meghatározni (például a megrendelőt nem fogja érdekelni, hogy nekünk összesen több tucat bankkártya és bankszámla kombinációra van szükségünk az összes lehetséges eset lefedéséhez). Ebből megállapíthatjuk, hogy az így elkészült tesztesetek szinte minden esetben pozitív esetek, a maradékuk pedig kevés számú tipikus negatív eset, de szinte semmilyen esetben nem tartalmaznak leírásokat szélsőséges helyzetekre, illetve szándékos károkozásra.

Ezen technika értéke leginkább magasabb szintű rendszerteszteknél, és tipikusan az átvételi teszteknél mutatkozik meg. Akkor mondhatjuk, hogy lefedtük az összes elvárást, amennyiben a kezdetben meghatározott formális rendszerkövetelmények minden mondatához (szélsőséges esetben tagmondatához) egyértelműen meghatározható egy (ritka esetben több) teszteset, amelynek egyedüli célja ennek az egy feltételnek a lefedése, és semmi más.

Összefoglalás

Miután megismertünk több fekete doboz technikát, vegyük sorra az előnyeiket és hátrányaikat. Az ide tartozó formális módszerek legfőbb hátránya, hogy nem tudják azonosítani a rosszul azonosított specifikációkat, hiszen kivétel nélkül mind a specifikációt veszik alapul. Ezáltal lyukakat találhatnak, de képtelenek lesznek megállapítani azt, hogy a termék nem a megfelelő elvárásokat teljesíti. Viszont ezek mellett is talán a legfontosabb tesztelési formának számítanak olyan rendszerek esetén, melyeknek a felhasználói emberek.

Gyakorlás

Az alábbiakban oldjunk meg néhány, a fekete doboz technikák témakörébe tartozó feladatot.

1. Egy űrlapon az egyik mező az abc kis- és nagybetűit fogadja el. Válasszuk ki, hogy az alábbi tesztesetek közül melyik tartozik érvénytelen partícióba (invalid partition):

a) CLASS
b) cLASS
c) CLass
d) CLa01ss

A helyes válasz a "d". A többi válaszról elég könnyű belátni, hogy minden elemében csak megengedett értékeket tartalmaz. Ellenben, ha egy tesztadatban bármi szerepel a nem megengedett értékek közül, akkor mindenképpen az érvénytelen partícióba tartozik, akkor is, ha részben érvényes elemeket tartalmaz.

2. Ismert az alábbi adórendszer: Minden fizetés első 40.000 forintja adómentes. A következő 15.000 Ft 10%-os adóterhet vonz magával. Az ezt követő 280.000 Ft 22%-kal terhelődik, és minden ezt meghaladó összeg 40%-kal. Az alábbi tesztértékek közül melyek tartoznak egy ekvivalencia osztályba?

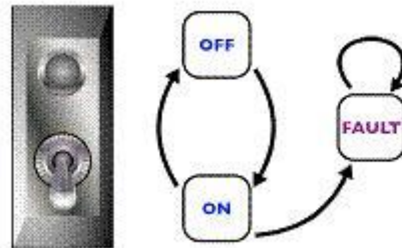
- **a) 48.000, 140.000, 280.000**
- **b) 52.000, 55.000, 280.000**
- **c) 280.001, 320.000, 350.000**
- **d) 58.000, 280.000, 320.000**

A helyes válasz ismét a "d". A megadott adatokból gyorsan kiszűrhetjük, hogy az adó mértéke nem lesz érdekes a feladat megoldása szempontjából, viszont a partíciókat meghatározó határértékeket meg kell határoznunk, ezek pedig rendre: 40.000, 55.000, 335.000. Ezek meghatároznak 4 érvényes partíciót:

1) >0 és <40.000
2) >40.000 és <55.000
3) >55.000 és <335.000
4) >335.000, ennek a partíciónak felső határa nincsen.

Innen már csak egy lépés van hátra, minden megadott értéket besorolunk a neki megfelelő partícióba. Ez alapján pedig kimondhatjuk, hogy a "d" válasznál megadott mindhárom érték a 3-as partícióba tartozik, ellentétben a többi válaszlehetőséggel, ahol az értékeke vegyesen, legalább két partícióba sorolhatóak be.

3. Az alábbi ábra egy kapcsoló működését írja le. A megadottak közül melyik számít nem megengedett állapotátmenetnek?



a) **OFF -> ON**
b) **ON -> OFF**
c) **FAULT -> ON**

A helyes válasz a "c". A vizsgálat ebben a példában egyszerű, amennyiben a gráf áll rendelkezésre, meg kell vizsgálnunk, hogy a két megadott állapotot összeköti-e átmenet nyíl (transition marker/arrow), mert ha nem, akkor az egy közvetlenül nem megvalósítható átmenet, tehát nem megengedett. Ha a feladat egy döntési táblával lenne megadva (ne feledjük, hogy minden állapotátmenet reprezentálható döntési táblával is), elég lenne leolvasnunk a megfelelő mezőből, hogy az átmenet logikai igaz vagy hamis.

4. Egy megrendelés-kezelő rendszerben egy megrendelés egy termékből minimum 10.000, és maximum 99.999 darab lehet. Az alábbiak közül melyik teszthalmaz fogadható el, ha csak érvényes partíciókra, és érvényes határértékekre vonatkozó értékeket veszünk figyelembe?

a) **1.000, 5.000, 99.999**
b) **9.999, 50.000, 100.000**
c) **10.000, 50.000, 99.999**
d) **10.000, 99.999**
e) **9.999, 10.000, 50.000, 99.999, 100.000**

SZÉCHENYI 2020

A helyes válasz a "c". A helytelen válaszokat kiszűrnünk itt sem nehéz, a tartományunk adott, így az alábbi értékeket tartalmazó válaszokat egyből kiesnek: 1.000, 5.000, 9.999, 100.000. Hiába lenne nagyon jó teszteset mind a 9.999, és a 100.000 is (hiszen egy határérték közvetlen szomszédjai), de már érvénytelen tartományba esnek. A "d" válasz is elfogadható lehetne, viszont nem tartalmaz ekvivalencia particionálás segítségével megállapított értéket, hiszen mindkét megadott érték határérték, a feladat pedig “és” kapcsolatot definiál: “érvényes partíciókra, és érvényes határértékekre”. Mindezeket megvizsgálva, csak a "c" válasz lesz minden értékében helyes.

5. Ha adott az alábbi döntési táblázat, mely egy biztosító üzleti logikáját írja le:

Feltételek	**1**	**2**	**3**	**4**
Magyar állampolgár?	Nem	Igen	Igen	Igen
Életkora 18-55?	Nem számít	Nem	Igen	Igen
Dohányzik?	Nem számít	Nem számít	Nem	Igen
Döntések				
Biztosítjuk?	Nem	Nem	Igen	Igen
Jár-e 10% kedvezmény?	Nem	Nem	Igen	Nem

Illetve ismert az alábbi két teszteset:

Teszteset A: Sándor, 32 éves, dohányos, Budapest

Teszteset B: Jean-Michel, 65 éves, nem dohányos, Párizs

Melyik válasz a helyes az alábbiak közül?

a) **A – Biztosítjuk, 10% kedvezménnyel, B – Biztosítjuk, nincs kedvezmény**
b) **A – Nem biztosítjuk, nincs kedvezmény, B – Nem biztosítjuk, nincs kedvezmény**
c) **A – Biztosítjuk, nincs kedvezmény, B – Nem biztosítjuk, nincs kedvezmény**
d) **A – Biztosítjuk, nincs kedvezmény, B – Biztosítjuk, 10% kedvezménnyel**

SZÉCHENYI 2020

A helyes válasz a "c". A megoldás menete nem bonyolult, mindössze ki kell keresnünk az adatoknak megfelelő oszlopot a táblázatból. Amire érdemes figyelnünk, az a "nem számít érték". Ha nem lenne ez az érték, akkor a táblázatunknak nem 4, hanem 8 értékeket tartalmazó oszlopot kellene tartalmaznia. Például a kettes sorszámú oszlopnál nem számít, hogy a potenciális ügyfél dohányzik-e. Ha ki akarnánk fejteni, ezzel eltüntetve a "nem számít" értéket (not relevant value), akkor fel kellene vennünk a mostani kettes oszlop helyett két másikat, rendre az alábbi értékekkel:

Feltételek	**2A**	**2B**
Magyar állampolgár?	Igen	Igen
Életkora 18-55?	Nem	Nem
Dohányzik?	Igen	Nem
Döntések		
Biztosítjuk?	Nem	Nem
Jár-e 10% kedvezmény?	Nem	Nem

Mint láthatjuk, a két oszlop csak a dohányzást vizsgáló sorban különbözik, azonban a döntésben nem. Így, ha rövidíteni akarunk a táblázaton, használhatjuk az eredetileg megadott leírási módszert, ami a "nem számít" segítségével rövidebben és átláthatóbban írja le ugyanazokat a szabályokat. Hasonló logikával láthatjuk, hogy az egyes oszlopot is ki lehetne fejteni, viszont itt bonyolultabb a helyzet, négy új oszlopot kellene bevezetnünk, az alábbiak szerint:

SZÉCHENYI 2020

Feltételek	1A	1B	1C	1D
Magyar állampolgár?	Nem	Nem	Nem	Nem
Életkora 18-55?	Nem	Igen	Nem	Igen
Dohányzik?	Nem	Nem	Igen	Igen
Döntések				
Biztosítjuk?	Nem	Nem	Nem	Nem
Jár-e 10% kedvezmény?	Nem	Nem	Nem	Nem

A kulcs az összes lehetséges kombináció kigenerálása, tehát a logikai igen-nem értékek összes lehetséges kombinációját kell megállapítanunk. Amennyiben a döntési táblánk nem bináris (ez erősen ellenjavallott, viszonylag kevés munkával minden döntési tábla megadható binárisan, aminek a kezelése sokkal egyszerűbb), akkor az összes lehetséges értéket kell sorra kombinálnunk, például ha a példabeli egyes oszlopnál a dohányzást 3 érték írná le (nem, ritkán, sűrűn), akkor 6 (3*2) oszloppal lehetne csak helyettesíteni.

SZÉCHENYI 2020

6. **Egy tesztelés megtervezése az alábbi tevékenységekből áll:**

i. **Megtervezni, és részletesen ledokumentálni a teszteseteket, az ismert teszteset tervezési módszereket használva**
ii. **Meghatározni a tesztesetek futtatási sorrendjét**
iii. **Elemezni a követelményeket és specifikációkat, hogy megállapíthassuk a tesztfeltételeket**
iv. **Meghatározni az elvárt eredményeket**

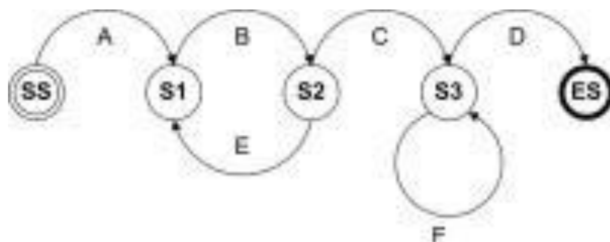
A tesztek azonosítását és tervezését leíró folyamat szerint mi a helyes sorrendje a fentebb felsorolt tevékenységeknek?

a) **iii, i, iv, ii**
b) **iii, iv, i, ii**
c) **iii, ii, i, iv**
d) **ii, iii, i, iv**

A helyes válasz az "a". A folyamatnak mindenképpen elemzéssel kell kezdődnie, meg kell vizsgálnunk az összes rendelkezésre álló követelményt. Ennek célja, hogy meg tudjuk állapítani, milyen tesztfeltételeket kell kielégítenünk a teszteseteinkkel. Ezt követi a tesztek konkrét kidolgozása, egészen addig ameddig minden tesztfeltételt elégségesen le nem fedtünk. Ha mindezzel megvagyunk, meg kell határoznunk az aktuális tesztelés célját, hogy az elkészült teszteseteink egy ezt kielégítő részhalmazát állapíthassuk meg a futtatáshoz. Ezután, ahogy megvan a futtatni kívánt teszthalmaz, meg kell határoznunk egy futtatási sorrendet.

SZÉCHENYI 2020

7. Adott az alábbi állapotátmenet:



Milyen bejárást kell alkalmaznunk, hogy elérjük a teljes él-lefedettséget? (0-switch coverage)

a) **A, B, E, B, C, F, D**

b) **A, B, E, B, C, F, F**

c) **A, B, E, B, C, D**

d) **A, B, C, F, F, D**

A helyes válasz az "a". Ennek a lefedettségnek a lényege, hogy minden élt legalább egyszer érintsünk. Jelen példában ez könnyen megoldható, hiszen egyetlen bejárással minden él érinthető. Elképzelhető, hogy más esetben több út is szükséges. Ami viszont nem kizáró tényező a megoldásnál, az az érintett állapotok száma. Mint láthatjuk, van három olyan állapotunk is, amit kétszer kell érintenünk, hogy elérjük a szükséges él-lefedettséget, ám ez teljesen rendben van, ennél a lefedettségnél bármelyik állapotot bármennyiszer érinthetjük.

8. Ha adott az alábbi döntési táblázat, akkor a felsorolt tesztesetek és elvárt eredmények közül melyik érvényes?

a) **23 éves, A osztályba tartozó férfi, a prémium 90, a pótdíj értéke 2500**

b) **51 éves, C osztályba tartozó nő, a prémium 100, a pótdíj értéke 2500**

c) **31 éves, B osztályba tartozó férfi, a prémium 90, a pótdíj értéke 500**

d) **43 éves, A osztályba tartozó nő, a prémium 100, a pótdíj értéke 2500**

Feltételek	1	2	3	4
Életkor	<21 év	21-29 év	30-50 év	>50 év
Biztosítási osztály	A	A vagy B	B, C, vagy D	C vagy D
Döntések				
Prémium	100	90	70	70
Pótdíj	2500	2500	500	1000

A helyes válasz az "a". Mint láthatjuk, itt nem egy bináris döntési táblával van dolgunk, hanem az életkornál egy egész tartomány is lehet megadva, a biztosítási osztálynál pedig egy, vagy több osztály is szerepelhet. Ez lassíthat a feladat értelmezésén, viszont gondoljunk bele, ha évenként kellene lebontanunk, és mellé kifejtenénk még az osztályt is, akkor 8*2, összesen 16 oszloppal kellene pótolnunk csak a kettes oszlopot! Ez nyilvánvalóan nem eléggé hatékony, mind a táblázat felvételét, mind a használatát feleslegesen lassítaná. Ezen túl viszont a feladat megoldása nem kellene, hogy problémákba ütközzön, mindössze meg kell néznünk, hogy a megadott értékek egy, vagy több oszlopra illeszthetőek rá, ha csak egyre, akkor az a teszteset helyes, ha kettő, vagy több oszlop is szükséges, akkor pedig az a teszteset helytelen, nem elfogadható.

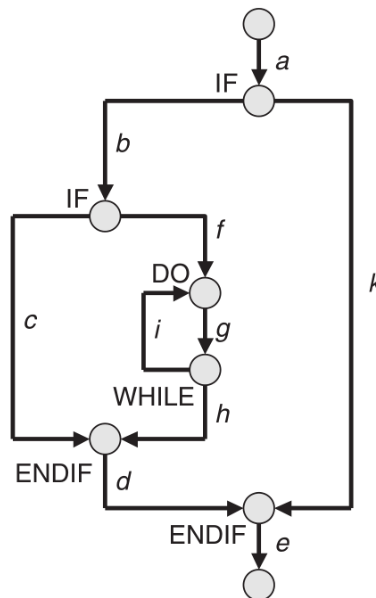
Fehér doboz technikák

Ezen technikák közvetlenül a forráskódját tesztelik a megvizsgálni kívánt rendszernek. Az alapötlet, hogy a forráskód struktúrájából kiindulva olyan eseteket állapítson meg, melyek különböző megközelítések alapján be tudják járni a kód egészét. Önmagában ez kevés lenne sokféle hibás működés megállapítására, ezért e technikák alkalmazásakor is a specifikációból kell tesztfeltételeket meghatároznunk. Az alábbiakban megpróbáljuk röviden összefoglalni a különböző típusú vizsgálatokat, azok céljait, és az elérhető lefedettséget.

Utasítás szintű tesztelés (statement testing)

Talán a legalapvetőbb fehér doboz technika, célja, hogy minél kevesebb tesztesettel minden kifejezés lefusson. Az ilyen típusú tesztek tervezéséhez nagyon hasznosak a már korábban is használt gráfok, ahol az olyan utasítások, melyek mindenképpen együtt futnak le, egy csomópontnak számítanak. A feltételes

utasítások nyilvánvalóan elágazásokat generálnak a kódban, melyeket úgy reprezentálunk, hogy az adott kódblokkot tartalmazó gráf elemet több kimeneti éllel ruházzuk fel. Lássunk egy példát, egy viszonylag egyszerű kód reprezentációját:



A fenti ábrát megvizsgálva megállapíthatjuk, hogy az összes csomópont bejárható egy tesztesetben, amennyiben bejárási útnak az a, b, f, g, h, d, e utat választjuk. Ezzel elértük a 100%-os lefedettséget, hiszen az erre vonatkozó képlet itt is hasonlatos a fekete doboz technikáknál megismertekhez:

$$\frac{bejárt\ csomópontok\ száma}{összes\ csomópont\ száma} \times 100\%$$

Mint a fentiekből látszik, ez a technika sajnos nem biztosít túl releváns tesztelést viszont rendkívül hatásos az elérhetetlen kód ("halott kód") felismerésére.

Döntés alapú tesztelés (decision/branch testing)

Az előző technika hibáinak kiküszöbölésére született meg ez a módszer, amely teljesen hasonlatos gráfon dolgozik, ám minden logikai elágazás egy-egy újabb tesztesetet definiál.

Mint az sejthető ez a technika 100%-os utasítás szintű alapú lefedettséget produkál, legtöbb esetben redundánsan. Ennek az ellenkezője általában nem teljesül. Egy nagyon egyszerű példa, hogy ha egy IF feltételhez tartozó ELSE ág nem tartalmaz további kifejezést, ez az ág nem lesz bejárva utasítás szintű megközelítéssel, döntési alapúval viszont igen.

Ezzel a módszerrel már a program fő vezérlési elemei is megvizsgálhatóak. Lefedettség képlete:

SZÉCHENYI 2020

$$\frac{bejárt\ logikai\ döntések}{logikai\ döntések\ összes\ száma} \times 100\%$$

Ez a technika alkalmas arra, hogy detektáljunk olyan logikai döntéseket, melyek mögül hiányoznak szükséges végrehajtó utasítások.

Feltételek tesztelése (test of conditions)

Sajnos még az előbbiekben megismert módszerek sem elegendőek, hogy a program helyességét minden esetben belássuk. A legéletszerűbb példa az összes olyan eset, amikor egy logikai döntés nem egyszerű, hanem több logikai döntés kombinációja (összekapcsolva például AND, OR és NOT operátorokkal).

Többszörös feltételek tesztelése (multiple condition testing)

Ilyen esetekben hasonlóan kell eljárnunk, mint a döntési alapú tesztekben, viszont azt összes tesztesetek számát a fekete doboz technikáknál megismert döntési tábla sorainak számához hasonlatosan határozhatjuk meg. Ez a gyakorlatban azt jelenti, hogy az ilyen típusú lefedettséghez abban az esetben, ha két logikai értékünk adja meg a kombinált feltételt (tetszőleges logikai operátorral összekapcsolva), összesen négy esetet kell vizsgálnunk (igaz-igaz, igaz-hamis, hamis-igaz, hamis-hamis). Természetesen az ilyen alapossággal előállított tesztesetek teljesítik a 100%-ot mind kifejezés, mind pedig döntési tesztelési szempontból.

Feltétel meghatározásos tesztelés (condition determination testing/minimal multiple condition testing)

Az előzőekben ismertetett módszer legnagyobb hátránya, hogy nagyon könnyen kezelhetetlen mennyiségűre duzzasztja a szükséges tesztesetek számát. Ezt úgy küszöbölhetjük ki, hogy csak olyan atomi érték kombinációkat veszünk figyelembe, amelyek megváltoztatása befolyásolná az egész összetett feltétel kimeneti logikai értékét. Ez attól a nyilvánvaló előnytől eltekintve, hogy kevesebb tesztesetet eredményez, sajnos rendelkezik hátránnyal is, például ha rövidzár-kiértékelés miatt teljesen kihagyjuk egy feltételben szereplő logikai kifejezés kiértékelését, megváltozhat a program működése.

Bejárási út alapú tesztelés (traversal based testing)

Az eddigi módszerek hasznossága ott ér véget, ha a kódban hurkok vannak. Tesztünket ezáltal kiegészíthetjük olyan tesztesetekkel, amelyek bizonyos feltételeknek megfelelően többször is végigmennek az olyan kódrészeken, amelyek ismétlődhetnek. Ezt a bemeneti paraméterek helyes megválasztásával érhetjük el (például a teszteset megfelelő módon változtatja meg a ciklus lépésszabályzó változójának értékét). Attól függően, hogy mi a célunk ezzel a teszteléssel, az általunk meghatározott számú ismétlések alapján fog növekedni a szükséges tesztesetek száma.

Összefoglalás

Habár e technikákat nem tárgyaltuk olyan alapossággal, mint a fekete doboz technikákat, viszont a tárgyaltakhoz hozzáolvasva a statikus tesztelésnél meghatározott, de leginkább ide vonatkozó tudnivalókat, megállapíthatjuk az ilyen típusú tesztelés előnyeit és hátrányait. Nagy előnye e technikáknak, hogy sokféle célt határozhatunk meg és mindre van olyan módszer, amely pontosan ezt az eredményt adja nekünk. Mint az láthattuk leginkább alacsony szintű tesztelést tudunk támogatni ilyen módszerekkel (egység és integrációs tesztek) viszont a jól meghatározott lefedettség számítással akár fekete doboz technikák valós, megmozgatott kódbeli lefedettségét is mérhetjük. Számos előnyük ellenére ezek a módszerek sem hibátlanok, például hiányzó kódot nem tudnak megtalálni, csak a létező kód hibáit tudják ellenőrizni. Azonban, ha ezen előnyök és hátrányok mellé odarakjuk a fekete doboz technikák előnyeit és hátrányait, megállapíthatjuk, hogy e két család remekül kiegészíti egymást, és ha ideális tesztelést szeretnénk folytatni, egyik sem hagyható el, illetve nem helyettesíthető a másikkal.

Megvalósítás

Ahhoz, hogy a fenti technikákat vizsgálni tudjuk, a forráskód kiegészítésre szorul, például a különböző kódrészletekhez számlálókat kell rendelni, melyek nulláról indulnak és tárolják, hogy az adott feltétel hányszor lett kiértékelve, vagy éppen az adott kifejezés végrehajtva. Ezt már nagyon rövid kódoknál is nagyon nehéz lenne kézzel megtennünk, ezért fehér doboz tesztelést szinte kizárólag erre specializált eszközökkel végzünk, például a legtöbb IDE támogatja az ilyen típusú tesztelést.

Egységtesztelés

Amikor új funkcionalitást fejlesztünk ki, érdemes egy kódtöredéket (snippet of code) írnunk, ami leteszteli. Ez a korábban bevezetett definíciók szerint egységtesztnek számít, illetve ha több, logikailag különálló részét is érinti a kódnak, akkor integrációs tesztnek. Egy jól felépített egységteszt-rendszer sok ilyen kisebb kódrészletből áll. Minden alkalommal futnak, futhatnak, amikor a kód fordításra kerül. Ettől azt várjuk, hogy a regressziós hibákat hamar kiderítsük, hiszen ha egy korábban már működő teszt a soron következő fordítás után nem fut le, viszonylag könnyen behatárolhatjuk, hogy melyik újonnan elkészült kódrészletben keresendő a hiba. Mint a nevük is sugallja, ezek a kódtöredékek rövidek, és a kódnak egy jól behatárolt részét tesztelik csak.

Az egységtesztelés világában elég gyakori megközelítés a tesztvezérelt fejlesztés (test driven development), melynek lényege dióhéjban, hogy először a teszteket készítjük el, es csak azután az alkalmazást magát. Nézzünk az alábbiakban erre egy szemléletes példát. Tételezzük fel, hogy egy játékprogramot készítünk, és írnunk kell egy osztályt (a hordozó nyelvünk jelen példában a C# lesz), mely

SZÉCHENYI 2020

a játékost reprezentálja. A játékosnak rendelkeznie kell életerővel, mely létrehozáskor szükségszerűen magasabb, mint nulla. Az előbbiekben említett tesztvezérelt fejlesztést alapul véve kezdjük egy teszt írásával, mely ezt az ismert feltételt teszteli:

```
class PlayerTests {
	bool TestPlayerIsAliveWhenBorn() {
		Player p = new Player();
		if (p.Health > 0) {
			return true; // pass the test
		}
		return false; // fail the text
	}
}
```

Ez a kódrészlet egyből fordítási hibát okoz, hiszen meg maga a játékost leíró osztály sem létezik, ezáltal nem is hivatkozható, se nem példányosítható. Ettől függetlenül mi tudjuk értelmezni a tesztet, ha létrejön egy játékos, kívánatos, hogy nullát meghaladó életereje legyen, máskülönben a tesztünk nem fog sikeresen lefutni. Most, hogy ezzel megvagyunk, hozzuk létre a játékost leíró osztályt:

```
class Player
{
	public int Health { get; set; }
}
```

Most már futtathatjuk az összes elkészült (jelen pillanatban egy darab) tesztünket. Sajnálattal tapasztalhatjuk, hogy a tesztünk futtatása sikeres, ám eredménye sikertelen, hiszen az egész típusú változók nullát kapnak kezdőértékként, ha nem határozunk meg mást. Ahhoz, hogy a tesztünk lefusson, változtatnunk kell a játékost leíró osztályon (fontos, hogy nem a teszten):

```
class Player
{
        public int Health { get; set; }
        Player()
        {
                Health = 10;
        }
}
```

Ha újrafordítunk, majd segítségül hívjuk az egységtesztek futtatását végző szoftverünket, akkor a tesztünk sikeresen lefut ezen a javított játékos osztályon. Most, hogy a játékos létrehozása jól működik, írjunk meg két további tesztet:

```
class PlayerTests
{
        bool TestPlayerIsHurtWhenHit()
        {
                Player p = new Player();
                int oldHealth = p.Health;
                p.OnHit();
                if (p.Health < oldHealth)
                {
                        return true;
                }
                return false;
        }
        bool TestPlayerIsDeadWhenHasNoHealth()
        {
                Player p = new Player();
                p.Health = 0;
                if ( p.IsDead() )
                {
                        return true;
                }
                return false;
        }
}
```

A második és harmadik tesztünk sem túl bonyolult, nézzük meg, és próbáljuk kitalálni, mire vonatkozhatnak. A második azt tesztelné, hogy amennyiben a játékos találatot kap, csökken-e az életereje. A harmadik megvizsgálja, hogy amennyiben a játékos életereje 0-ra csökken, halottnak fog-e számítani (legalábbis a játék szempontjából). Ha jobban megvizsgáljuk, láthatjuk, hogy a definíciónk nem pontos, hiszen nem csökkentettük a játékos életerejét, hanem explicit felülírtuk. Bezárással ez a probléma elkerülhető, számunkra ennek még a fejezet egy későbbi részén lesz jelentősége. Mivel a tesztek futásához szükség van két új tagfüggvényre a játékost reprezentáló osztályban, az ismét kiegészítésre szorul:

```
class Player
{
        public int Health { get; set; }
        public bool IsDead()
        {
                return false;
        }
        public void OnHit()
        {
        }
        public Player()
        {
                Health = 10;
        }
}
```

Az egységtesztelés egy jó megközelítése, hogy megírjuk a tesztet, kiegészítjük a kódot, lefuttatjuk a tesztet, majd ha sikertelen (nem a futtatás, hanem a teszt futtatásának eredménye sikertelen), akkor addig javítjuk a kódot, ameddig a teszt sikeresen le nem fut. A fentiekben tett kiegészítéseink sajnos ilyen formában nem elegendőek a sikeres futtatáshoz. Ami jó hír, hogy a futtatáshoz elegendőek, azonban mi azt is szeretnénk, ha nem csak futnának, hanem sikeresen futnának, ennek szellemében tegyünk további kiegészítéseket a kódunkon:

SZÉCHENYI 2020

```
class Player
{
	public int Health { get; set; }
	public bool IsDead()
	{
		return Health == 0;
	}
	public void OnHit()
	{
		Health--;
	}
	public Player()
	{
		Health = 10;
	}
}
```

Ne aggódjunk, ha egyből hibát vélnénk felfedezni a kódban, pusztán demonstrációs szándékkal került oda. Ha az így kiegészített és javított kódon futtatjuk az elkészült három tesztünket, mindegyiktől sikeres futást várunk, ami a gyakorlatban azt jelenti, hogy "true" értékkel térnek vissza a hívó környezethez. A kódot leellenőriztük ugyan, de nem teszteltük le elég kimerítően, például létrehoztunk egy `TestPlayerIsDeadWhenHasNoHealth()` tagfüggvényt, de nem hoztuk létre a párját, ami azt vizsgálná, hogy ha több élete van a játékosnak, mint nulla, akkor viszont nem halott. Ez egy nagyon fontos dologra mutat rá, ami egyben az ilyen típusú tesztelés általános szemlélete is, hogy nem elég egyetlen szempontból megvizsgálnunk egy atomi dolgot, hanem sorra kell vennünk az összes logikailag különböző állapotát. Jelen esetben a játékos halotti státuszát megállapító függvény visszatérési értéke logikai érték, ami azt jelenti, hogy a jó lefedettséghez (legalább) kettő tesztre van szükség. Ujjgyakorlatnak létrehozhatjuk ezt a hiányzó függvényt.

Tovább folytatva kísérletünket, tételezzük fel, hogy tovább folytatjuk a játék megírását addig, amíg egy teljesen működő verziónk nem lesz. Majd a végső teszteléseknél valami furcsa történik, például lesz olyan eset, amikor a játékosnak el kellene haláloznia, de mégsem teszi, sőt, azután halhatatlanná válik. Ez nyilván nem kívánatos működés. Mivel minden játékosra vonatkozó tesztünk lefut, ez azt jelenti, hogy nincs olyan tesztünk, ami ki tudná mutatni ezt a hibát. Tételezzük fel, hogy alapos hibakereséssel megállapítjuk, hogy a hiba akkor lép fel, amikor a játékosnak kevés életereje van, például két egység, vagy kevesebb, és sok találatot kap rövid idő alatt egy, vagy több ellenféltől.

SZÉCHENYI 2020

Egységtesztelési szemlélettel az első dolgunk, hogy írunk egy olyan tesztet, ami speciálisan ezt az esetet teszteli:

```
class PlayerTests
{
        bool TestPlayerShouldBeDead()
        {
                Player p = new Player();
                p.Health = 2; // give him low health
                // Now hit him a lot, to reproduce the bug description
                p.OnHit();
                p.OnHit();
                p.OnHit();
                p.OnHit();
                p.OnHit();
                if ( p.IsDead() )
                {
                        return true;
                }
                return false;
        }
}
```

Ez a teszt lefut, viszont eredménye sikertelen lesz. Most már rendelkezünk egy teszttel, amely ismételhetően tudja reprodukálni a hibát. Most következhet a kódunk kijavítása, hogy ne csak a korábbi tesztek, hanem ez az újonnan elkészült is lefusson. A játékos halálát vizsgáló tagfüggvény jó hely lehet a hiba kezelésére:

```
class Player
{
        bool IsDead()
        {
                return Health <= 0;
        }
        ...
}
```

Így módosítva a kódot, a nulla életerővel, vagy kevesebbel rendelkező játékos halottnak fog számítani. Ezzel ismét minden tesztünk sikeresen lefut, és köszönhetően az elkészült új tesztnek, azonnal észre fogjuk venni, ha bármilyen okból kifolyólag újra megjelenne.

Gondolkozzunk el, hogyan lehetett volna még kijavítani a kódot? Egy másik lehetséges megoldás lenne az életerő csökkenést kezelő függvényben vizsgálni, hogy ha a találat nulla alá csökkentené a játékos életerejét, akkor az új érték mindenképpen nulla legyen. A példát még számtalan módon lenne érdemes folytatni, de egyelőre elégedjünk meg ennyivel, hiszen célunk jelen esetben nem a játékírás, hanem az egységtesztelés mikéntének megértése, és hasznosságuk belátása.

Gyakorlás

Az alábbiakban oldjunk meg néhány, fehér doboz technikákra vonatkozó feladatot.

1. Vizsgáljuk meg az alábbi állítások igazságtartalmát:

- **i. 100% kifejezés-lefedettség 100% ág-lefedettséget is garantál**
- **ii. 100% ág-lefedettség 100% utasítás-lefedettséget is garantál**
- **iii. 100% ág-lefedettség 100% döntési-lefedettséget is garantál**
- **iv. 100% döntési-lefedettség 100% ág-lefedettséget is garantál**
- **v. 100% kifejezés-lefedettség 100% döntési-lefedettséget is garantál**

- **a) ii igaz; i, iii, iv & v pedig hamisak**
- **b) i & v igazak; ii, iii & iv pedig hamisak**
- **c) ii & iii igazak; i, iv & v pedig hamisak**
- **d) ii, iii & iv igazak; i & v pedig hamisak**

A helyes válasz a "d". Ezt könnyen beláthatjuk, ha elgondolkodunk, hogy mikre nem vonatkozik a utasítás lefedettség. Egy nagyon jellemző példa az üres/hiányzó ELSE ágak bejárása, melyek nem relevánsak pusztán az utasítások futását vizsgálva, viszont ág- és döntési lefedettség szempontjából igencsak fontosak lehetnek. Amennyiben ilyen részek nincsenek a kódban, akkor a 100%-os utasítás lefedettség meg fog egyezni a másik kettő 11%-os lefedettségével, viszont azt nem mondhatjuk, hogy ez minden esetben garantált.

SZÉCHENYI 2020

2. Mennyi teszt szükséges minimálisan az alábbi pszeudókód részlet teljes utasítás- és ág-lefedettséghez?

```
Read P
Read Q
IF p+q > 100 THEN
	Print "Large"
ENDIF
IF p > 50 THEN
	Print "pLarge"
ENDIF
```

a) **Utasítás-lefedettséghez 2, és ág-lefedettséghez is 2**
b) **Utasítás-lefedettséghez 3, ág-lefedettséghez pedig 2**
c) **Utasítás-lefedettséghez 1, ág-lefedettséghez pedig 2**
d) **Utasítás-lefedettséghez 4, ág-lefedettséghez pedig 2**

A helyes válasz a "c". P es Q értékeit jól megválasztva mindkét feltétel igaz lesz, így például egy olyan teszt ami P értékének 100-at használ, illetve Q értékének szintén 100-at vesz, a megadott kód minden során végig fog haladni. AZ ág-lefedettséghez ez nem elég, pont a fentebb említett rejtett ELSE ágak miatt, lássunk két érték-párt, amik már elegendőek:

P = 75 és Q = 0, így az első feltételnél az ELSE, a másodiknál az IF ágba futunk bele.

P = 25 es Q = 100, így az első feltételnél az IF, a másodiknál az ELSE ágba futunk bele.

Így mind a két feltételnek bejárjuk az IF, és az ELSE ágát is, elérve ezzel a 100%-os ág lefedettséget.

Alternatívaként lássunk két másik értékpárt, amik együtt már szintén elegendőek a feladat megoldásához:

P = 100 és Q = 100, így mind az első, mind a második feltételnél az IF ágba futunk bele. Észrevehetjük, hogy ezek az értékek pontosan megegyeznek a kifejezés-lefedettséghez használt értékekkel.

P = 10 és Q = 10, így mind az első, mind a második feltételnél az ELSE ágba futunk bele.

Ez a két megoldás ekvivalens, és tovább nem szűkíthető úgy, hogy megmaradjon a kívánt lefedettség.

SZÉCHENYI 2020

3. Mennyi teszt szükséges minimálisan az alábbi pszeudókód részlet teljes utasítás-lefedettséghez?

```
Disc = 0
Order-qty = 0
Read Order-qty
      IF Order-qty >=20 THEN
            Disc = 0.05
            IF Order-qty >=100 THEN
                  Disc =0.1
            ENDIF
      ENDIF
```

a) Utasítás-lefedettséghez 4

b) Utasítás-lefedettséghez 1

c) Utasítás-lefedettséghez 3

d) Utasítás-lefedettséghez 2

A helyes válasz a "b". Nem nehéz belátnunk, hogy van olyan "Order-qty" érték, ami mindkét feltételt igazzá teszi, például a 110. Mivel az ELSE ágak minkét esetben hiányoznak, ezen értéket választva el is értük a 100%-os utasítás-lefedettséget.

SZÉCHENYI 2020

4. Mi jellemző a fehér doboz technikákra?

i. **A tesztelés alatt álló elem belső működésének ismeretét felhasználva választjuk ki a tesztadatokat.**

ii. **A programkód ismeretét használja, hogy megvizsgálja a kimeneti értékeket, és feltételezi, hogy a tesztelést végző személy ismeri a program logikáját.**

iii. **Az alkalmazás teljesítményét méri.**

iv. **Funkcionalitást is ellenőriz.**

a) **i, ii igazak, míg iii és iv hamisak**

b) **iii igaz, míg i, ii, és iv hamisak**

c) **ii és iii igazak, míg i és iv hamisak**

d) **iii és iv igazak, míg i és ii hamisak**

A helyes válasz az "a". A két igaz állítás egy az egyben következik a fentebb megismert definíciókból. Az alkalmazás teljesítmények mérésére külön módszerek léteznek, de az egységtesztelés szűken vett értelmezésébe nem tartoznak bele. A funkcionalitást sem ellenőrizzük, arra főként a fekete doboz technikák valóak, a funkcionális követelmények ismerete nem szükséges a kielégítő fehér doboz teszteléshez.

5. Mennyi teszt szükséges minimálisan az alábbi működés teljes utasítás- és ág-lefedettséghez?

```
Kapcsold be a számítógépet
Indítsd el az "Outlook"-ot
HA az "Outlook" elindul AKKOR
      Küldj egy emailt
Zárd be az "Outlook"-ot
```

a) **Utasítás-lefedettséghez 1, és ág-lefedettséghez is 1**

b) **Utasítás-lefedettséghez 1, ág-lefedettséghez pedig 2**

c) **Utasítás-lefedettséghez 1, ág-lefedettséghez pedig 3**

d) **Utasítás-lefedettséghez 2, és ág-lefedettséghez is 2**

e) **Utasítás-lefedettséghez 2, ág-lefedettséghez pedig 3**

A helyes válasz a "b". Amennyiben azt a feltételt tesszük, hogy az "Outlook" elindul, láthatjuk, hogy a leírt utasítás minden során sikerült végighaladnunk. Viszont nem teszteltük ez esetben a hiányzó EGYÉBKÉNT feltételt, így szükségünk van egy második tesztesetre is, amely feltételezi, hogy az "Outlook" nem indul el. Ebben az esetben nem is küldünk levelet, viszont a két teszt csak együttesen elégíti ki egyszerre a mind a 100%-os utasítás, mind a 100%-os ág-lefedettséget.

6. Mennyi teszt szükséges minimálisan az alábbi működés teljes utasítás- és ág-lefedettséghez?

```
IF A > B THEN
      C = A - B
ELSE
      C = A + B
ENDIF
Read D
IF C = D THEN
      Print "Error"
ENDIF
```

a) **Utasítás-lefedettséghez 1, ág-lefedettséghez pedig 3**
b) **Utasítás-lefedettséghez 2, és ág-lefedettséghez is 2**
c) **Utasítás-lefedettséghez 2, ág-lefedettséghez pedig 3**
d) **Utasítás-lefedettséghez 3, és ág-lefedettséghez is 3**
e) **Utasítás-lefedettséghez 3, ág-lefedettséghez pedig 2**

A helyes válasz ismét a "b". Az előző példákkal ellentétben az ELSE ág most nem üres, így nem is lehetséges mindössze egy tesztesettel elérnünk a 100%-os utasítás-lefedettséget. Példa a két esetre:

A = 20 és B = 10 és D = 10, ebben az esetben bejárjuk az első feltételes kifejezés IF ágát, illetve a második feltételes kifejezést (ez utóbbit teljes egészében, hiszen nincsen ELSE ága).

A = 20 és B = 30 és D = 900, ebben az esetben bejárjuk az első feltételes kifejezés ELSE ágát, a második feltételes kifejezést pedig nem (D értékét szándékosan választottuk így, hogy demonstráljuk, nem szükséges még egyszer bejárnunk a kód azon részét).

Ez a két érték-hármas pontosan megfelel az ág-lefedettséghez is, hiszen sikerült érintenünk mind a két feltételes kifejezés IF, és ELSE ágait is.

7. Az alábbiak közül melyik állítás NEM igaz a komponens tesztelésre vonatkozóan?

a) A fekete doboz technikáknak mind megvan a hozzájuk tartozó, megfelelő mérési technikájuk

b) A fehér doboz technikáknak mind megvan a hozzájuk tartozó, megfelelő mérési technikájuk

c) A ciklomatikus komplexitás nem egy tesztmérési technika

d) A fekete doboz technikáknak mind megvan a hozzájuk tartozó, megfelelő teszttervezési technikájuk

e) A fehér doboz technikáknak mind megvan a hozzájuk tartozó, megfelelő teszttervezési technikájuk

A helyes válasz az "a". Ha sorra vesszük a többi válaszokat:

- A fehér doboz technikáknak mind megvan a hozzájuk tartozó, megfelelő mérési technikájuk: ez az állítás igaz, hiszen a fentiekben láthattuk, hogy minden, ebbe a kategóriába tartozó technikának megvan a megfelelő számolási módszere a lefedettségre, ami gyakorlatilag az egyetlen fontos metrika a különböző technikákra vonatkozóan.
- A ciklomatikus komplexitás nem egy tesztmérési technika: Ez a módszer megad egy mérőszámot, mely a kód bonyolultságát jellemzi, es mivel egzakt, tökéletesen megfelel mérőszámnak
- A fekete doboz technikáknak mind megvan a hozzájuk tartozó, megfelelő teszttervezési technikájuk: ez az állítás is igaz, hiszen mind a fekete, mind a fehér doboz technikák tervezett, szisztematikus módszerek, ha nem így lenne, intuitív, illetve tapasztalati alapú, informális technikának számítanának.
- A fehér doboz technikáknak mind megvan a hozzájuk tartozó, megfelelő teszttervezési technikájuk: Ezen technikákra még fokozottabban jellemző a tervezettség, hiszen még a külső elvárások ismerete sem szükséges a precíz teszttervezéshez (ellentétben a fekete doboz technikákkal)

Viszont az "a" válasz kilóg a sorból, nagyon sok esetben nem lehet precíz lefedettséget, vagy más ismételhető, ellenőrizhető metrikát rendelnünk a tesztekhez, hiszen a követelmények nincsenek az ehhez szükséges formalizált alakban, illetve nem is teljesek (matematikai értelemben).

Intuitív és tapasztalat alapú módszerek

A cím nem véletlenül ír módszereket és nem technikákat, ugyanis ezek az alábbiakban tárgyalt megközelítések szinte teljes mértékben informálisak. Fő céljuk, hogy olyan hibákat keressenek, amelyeket nehezen, vagy sehogy nem lehetne megtalálni szisztematikus módszerekkel.

A leggyakoribb alap a tesztelő, vagy más, vizsgálatot végző személy saját tapasztalata korábbi hasonlatos rendszerekkel és szituációkkal. Ezt a megközelítést gyakran nevezik hibasejtéses tesztelésnek (error guessing). Nagy erőssége a módszernek, hogy megtalálhat nem a specifikációból következő, például a rendszertervezési megközelítésből, a használt nyelvből, a beépített keretrendszerekből, vagy akár a futási körülményekből fakadó hibákat is.

Egy másik, szintén ide tartozó módszer a felderítő tesztelés (exploratory testing), ez abban az igen gyakori esetben használandó, amikor csak a rendszerhez van hozzáférésünk, de a dokumentumok hiányosak, vagy teljesen hiányoznak. Mivel az ilyen esetekben nem támaszkodhatunk szisztematikus módszerekre, a rendszer működését mi magunk próbáljuk meg felfedezni és dokumentálni, majd rutinunkat és tapasztalatunkat felhasználva hibákat keresni. Azonban egy komplex rendszer esetén ez bajos lehet, ezért kezdetben előnyös, ha kisebb részekre bontjuk az elvégzendő feladatot. Minden ilyen részhez kreálunk egy úgynevezett "teszt charter"t (test charter), amely vezet minket az adott rész elemzésében. Egy ilyen dokumentum az alábbi kérdésekre kell választ adjon:

- Miért? Mi a célja ennek a tesztelésnek?
- Mit tesztelünk? Melyik részére vonatkozik pontosan a rendszernek?
- Hogyan? Milyen formális, fél-formális és informális módszerek illenek ehhez a részhez?
- Mit várunk? Milyen típusú hibákat keresünk?

Ha ilyen módon formalizáljuk a tesztelést, megalapozzuk a későbbi formálisabb módszerek alkalmazását, és egy sokkal pontosabb kép lesz a rendszer működéséről a fejünkben. Részletesebben nem is tárgyaljuk e módszerek mikéntjét, egyet azonban fontos megjegyeznünk: nagyon veszélyes pusztán informális módszereket használva tesztelni, próbáljuk meg minél inkább csak kiegészíteni vele szisztematikus, megtervezett, alapos tesztjeinket.

SZÉCHENYI 2020

Összefoglalás

Most már sokféle tesztelési technikát és módszert ismerünk, azonban gyakorlat nélkül nem mindig egyszerű feladat annak eldöntése, hogy melyeket érdemes használnunk. Az első és legfontosabb maga a rendszer felépítése, ugyanis nem minden esetben áll rendelkezésre minden szükséges be- és kimeneti csatorna, például ha a rendszerünk egy komplex rendszer része, és a többi komponenssel felület nélkül, API-n keresztül kommunikál, máris kizárhatjuk a legtöbb magas szintű fekete doboz technikát. Ezen kívül az elérhető vagy megszerezhető dokumentumok is behatárolhatják a lehetőségeinket, például ha nincsenek meg a magas szintű specifikációk, akkor a használói eseteket legjobb esetben is csak sejthetjük. Fontos tényező lehet a különböző szabályoknak való megfelelés, például törvényi, ipari, szabványi, esetleg csak az adott megrendelőre vagy projektre jellemző előírásoknak (regulation) való megfelelés. Erre rengeteg példát lehetne hozni, ugyanis ezek az egyszerű esetektől kezdve (például mennyi az adott országban a felnőttkor határa), a nagyon bonyolultakig (például az autóipar egészére jellemző nagyon magas, szinte minden általunk ismert lefedettségi képlet alapján elért 100%-os fehér doboz teszt lefedettség) terjednek. Tényező lehet még a tesztelést végző szakemberek tapasztalata is, mint azt láthattuk, bizonyos informális módszerek kevés tapasztalatnál nem túl hatékonyak. Ritka eset, de előfordul, hogy a megrendelőnek saját, speciális metrikái vannak, melyeknek szintén meg kell felelnünk, akár már az átvételi tesztek előtt is. Most már széles körben elfogadottnak mondható a kockázat elemzés alapú (risk based) megközelítés is, melynek lényege, hogy az üzleti, biztonsági, vagy más egyéb szempontból kockázatosabb területekre nagyobb hangsúlyt fektet a tesztelésben is. Sajnálatos módon a legtöbb esetben az időhiány, illetve az elégtelen tesztkörnyezet is befolyásolja az elérhető eredményeket. Általános szabályként elmondhatjuk, hogy ilyen esetekben kezdjük mindig a legrelevánsabb tesztekkel, illetve hívjuk fel a figyelmet azon hiányzó tesztekre, melyeket nem áll módunkban ilyen körülmények között elvégezni.

A teszttervezési technikákhoz sem határozhatóak meg egyértelmű aranyszabályok, de azért tekintsünk át pár hasznos tanácsot. Mindig a funkcionalitás tesztelését vegyük előre, és győződjünk meg róla, hogy az ezekhez tartozó elvárt eredmények megfelelnek az eredeti céloknak. Az ekvivalencia osztályokat mindig használjuk együtt a határérték elemzéssel, illetve ha lehetőségünk van rá, számoljunk ezekhez fehér doboz lefedettséget is. Amennyiben egy teszt eredménye maradandóan befolyásolja a rendszer működését, érdemes bizonyos teszteket többször végrehajtanunk, és az így keletkezett új teszteseteket állapotátmenet gráfban ábrázolni. Hogyha többféle bemeneti adat között logikai összefüggést gyanítunk, érdemes egy külön döntési táblában felderíteni, hogy igazunk van-e. Amennyiben lehetőségünk adódik az összes felhasznált fekete doboz technikát lefedettség vizsgálatoknak is alávetni, próbáljuk meg

SZÉCHENYI 2020

azonosítani a gyenge pontokat, hogy a rendszer mely részeit lenne érdemes további, fehér doboz technikákkal is elemezni a nagyobb lefedettség érdekében. Ha belevágunk a fehér doboz technikák alkalmazásába, ne adjuk alább 100%-os döntési lefedettségnél, illetve a hurkokat járjuk be többször. Akkor járunk el helyesen, ha a fehér doboz technikákat az alacsonyabb szinten használjuk, a fekete doboz technikákat pedig minden szinten, például a fehér doboz tesztek megtervezéséhez is. Amennyiben a tapasztalatunk elegendő, ne féljünk az informális módszerekre sem időt szánni, remek kiegészítésnek bizonyulhatnak a szisztematikus módszerekhez. Záró gondolatként jegyezzük meg, hogy önmagában a legjobban kitalált tesztelési módszer sem elegendő, tartozzon akár a fekete, akár a fehér doboz családba.

SZÉCHENYI 2020

Tesztmenedzsment

Bevezetés

Ebben a fejezetben a teszt menedzsmentet két aspektusból tárgyaljuk. Egyrészről sorra vesszük a teszt menedzsment elemeit, mint teszttervezés, teszt stratégia, tesztbecslés, monitorozás, kockázatok, konfiguráció menedzsment, incidens menedzsment. Mindezek során a következő kérdésekre keressük – és remélhetőleg adjuk meg - a válaszokat:

- Mit jelent a teszt menedzsment?
- Milyen elemekből, épül fel?
- A gyakorlatban milyen folyamatokon keresztül valósulnak meg a teszt menedzsment elemei?
- Hogyan kapcsolódnak egymáshoz?
- Hogyan támogatják egymást?

Másrészről a teszt menedzser és a tesztelő tipikus feladatait gyűjtjük össze. A definíciókon túl megvizsgáljuk, hogy a gyakorlatban milyen szituációkban kell helytállnia a teszt menedzsernek és a tesztelőnek, milyen szerepeket kell játszaniuk és ehhez milyen képességekre, készségekre van szükség.

Ha a szoftver fejlesztés teljes életciklusát projektként értelmezzük, akkor a tesztelést is lehet projektnek értelmezni – projekt a projektben –, aminek a projekt menedzsere a teszt menedzser. Ezért a teszt menedzsernek hasonló feladatai vannak, mint általában egy projekt menedzsernek viszont szoftvertesztelésre specializáltan és azon belül is a minőségre, a tesztelésre koncentrálva. Az adott projektben meghatározza a tesztelés célját/céljait, annak eléréséhez szükséges feltételeket, technikákat, erőforrásokat.

Az adott fejlesztési projektben értelmezni kell a tesztelési céljait, feladatát, részfeladatokká bontani azt úgy, hogy a végrehajtó személy is megértse, átlássa a feladatát. Fontos eleme még a tervezés – ha nem a legfontosabb -, ami magában foglalja az erőforrások, eszközök, feladatok, idő-egységek tervezését, a monitorozást, hibajelentések módját, eszközét, változások esetén azok válaszidejét, reagálási folyamatát. Egyszóval: mindent, ami a projekttel – azaz teszteléssel – kapcsolatos. A tesztelés eredményeiről a tesztelési ciklus befejezésével – beszámolót kell készíteni. Ezzel segítséget nyújt, javaslatot tesz a projectet érintő döntések meghozatalához.

Ezen túl, mint minden menedzsertől, a teszt menedzsertől is elvárt, hogy a rendelkezésre álló erőforrásokat a lehető leghatékonyabban használja fel. A tesztelő elsősorban a tesztelés végrehajtására

és azt megelőző tervezésre koncentrál. A teszt menedzsertől és másoktól kapott információ alapján teszt tervet készít, teszteseteket generál. Kiválasztja a megfelelő teszt technikákat, összegyűjti a szükséges eszközöket, teszt adatot. Végrehajtja a tesztelést, hibákat jelent.

A következő alfejezetben a teszt menedzsment elemeit tárgyaljuk, abból a szempontból, hogy ezek hogyan segítik a teszt menedzsert a feladatai ellátásában.

Hogyan lesz valaki Teszt Menedzser? – Magyarországon Teszt Menedzser képzés nincs – leginkább tapasztalati úton lehet egy tesztelőből teszt menedzser. Ahhoz, hogy teszt menedzser legyen valaki legalább 4-5 éves komoly tesztelési tapasztalat szükséges. Nem kizárólag egy üzleti környezetben, esetleg kizárólag tesztelési területen tevékenykedő cég kötelékében. Ami fontos, hogy minél többféle tesztelési technológiát, azok előnyeit és hátrányait ismerje és alkalmazza is azokat.

Teszt menedzsment elemei

Teszt stratégia

Definíció szerint a teszt stratégia egy magas szintű dokumentum, ami leírja, hogy milyen teszt szinteket hajtunk végre és ezek részleteit, a teljes fejlesztő szervezetre vagy programra (jellemezően több, kapcsolódó projekt) vonatkozóan. Vegyük sorra a legismertebb teszt stratégia fajtáit:

Analitikus: például a kockázat-központú stratégia, amely a tesztelés alapjául a különböző tesztelendő elemek kockázati tényezőinek tükrében állítja fel a követendő stratégiát. Ennek alapja egy igen erős kockázat-elemzés, mely a megrendelővel és a programozóval szoros együttműködéssel jön létre. Egy másik analitikus stratégiai megközelítés az ún. követelmény-központú stratégia, amelynél az analízis a megrendelő által támasztott követelményeket célozza meg.

Modell-alapú: ennek alapja lehet például egy matematikai modell, ami meghatározza egy adott szerver válaszidejét egy adott hosszúságú kérésre, vagy adott mennyiségű felhasználó terhelés melletti válaszidő-változást. Amennyiben az eredmény nem éri el a modell által megadott értéket a teszt elbukott. De a modell nem feltétlenül kell, hogy matematikai legyen, lehet egy adott séma vagy dizájn is az alapja ennek a típusnak.

Szisztematikus (Methodical): Leginkább tapasztalati alapú stratégia. Az évek során összegyűjtött elemek (egyfajta lista), amik a legtöbb problémát okozzák a rendszerben, mindezt hozzárendelve egy nemzetközi szabványnak (pl.: ISO 9126), ami megadja a tesztelés körvonalát, főbb területeit és fókuszát. Leggyakrabban egy házon belüli sablont ad, amelyet folyamatosan bővítenek és használnak az érintett területen dolgozók.

SZÉCHENYI 2020

Folyamat- vagy szabvány-kompatibilis: például amikor a teszteléshez az IEEE 829-es szabványt használjuk valamely szakirodalom (pl.: Craig-Jaskiel: Systematic Software Testing – Artech House, 2002.) kiegészítve. Esetleg a szabványon kívül, annak kiegészítéseként valamely más metodológiát is használva, mint például az Agile Extreme Programming, vagy az Agile Scrum.

Dinamikus: ez a típusú stratégia inkább irányvonalakat ad a teszteléshez, mint határozott szabályokat. Többnyire szintén tapasztalati alapokon, a termék/program ismert hiányosságaira helyezve a hangsúlyt. Ilyen például a felfedező tesztelés (exploratory testing), ami a lehető legtöbb hibának a felfedésére fókuszál a tesztelési ciklus alatt.

Konzultációs vagy irányított: ez a megközelítés vonja be leginkább a fejlesztőket/programozókat és a rendszer felhasználóit egyrészt, hogy a tesztelendő egységeket meghatározzák, másrészt, hogy a tesztelés folyamatában aktívan részt vegyenek.

Regresszió-elkerülő (Regression-averse): Például amikor minden funkcionális tesztesetet automatikusan lehet futtatni, így bármilyen módosítás történik a rendszeren minden teszteset automatikusan újra futtatható, így ellenőrizve, hogy sehol nem okozott hibát az új kódrészlet. A regresszió-elkerülő stratégia egyfajta készlete a különböző eljárásoknak – többnyire automatizált folyamatoknak – ami lehetőséget biztosít arra, hogy felfedezzük a regressziós programhibákat.

A teszt stratégiát jellemezően teszt menedzser vagy a tesztelő csapat vezetője alakítja ki. Ez az a dokumentum ami, összefoglalja, hogy hogyan dolgozik a tesztelő csapat, mi várható el tőlük. Számtalan tényező (külső, illetve belső) befolyásolja a teszt stratégiát. Külső tényezők közé sorolhatjuk például azt, hogy mennyire függetlenül dolgozik a tesztelő csapat a fejlesztő csapattól. Ha a tesztelő csapat egy független cég, akkor a stratégiát saját magának fogalmazza meg, szerződésein, megbízásain keresztül érvényesül. Viszont ha egy szervezeten belül működik, mindenképpen közvetlenül illeszkedni kell a teszt stratégiának a szervezet egyéb stratégiai céljaihoz. Hosszú időt vehet igénybe, mire egy ilyen stratégia kialakul, és a szervezet elfogadja. Esetlegesen sok vezetővel kell egyeztetni. Belső tényező az, hogy milyen célokat fogalmaz meg magának a tesztelő csapat. Például érdemes eldönteni azt, hogy a tesztelő csapat egy szervezeten belül kiszolgálni akarja a fejlesztő csapatot teszteléssel, vagy saját maga fejlesztésével hatással akar lenni a teljes fejlesztési folyamatra.

Mit tartalmaz a teszt stratégia?

1. Tesztelés célját és hatáskörét
2. Lefedésre kerülő üzleti követelmények körét
3. A tesztelés szabályait és felelősségi körét

SZÉCHENYI 2020

4. Jelentések tartalmát és előállítási módját
5. Tesztelésre alkalmas állapot feltételeit
6. Tesztelés előfeltételeinek megteremtési módját
7. Tesztadatok előállításának módját
8. Teszteléshez használt eszközöket
9. Alkalmazandó teszt típusokat és tesztelési technikákat
10. Tesztek és futáseredmények követésének módját
11. Kockázat csökkentési módokat
12. Hibajelentés és nyomon követés módját
13. Ügyféligényben történt változások követésének módját
14. Rendszer konfigurációban történt módosítások kezelésének módját
15. Rendszer oktatási tervét

Egy alkalommal megkeresett egy cég, hogy segítsek letesztelni a terméküket. (A termék jelen esetben egy internetes áruház volt igen kis termékpalettával, de komoly árakkal és komoly technikai és banki háttérrel.) Az első kérdésem az volt, hogy mikorra kellene a tesztelést befejezni? A válasz sajnos lehangoló volt:

- A tervek szerint a következő hónapban indul az egyik akciónk, amiket csak ezen a webáruházas felületen vehetnek igénybe leendő vevőink. Azaz egy hónap múlva indulnunk kell!

- De a termék, azaz maga a virtuális áruház már tesztelhető állapotban van, igaz? – jött a következő kérdésem.

- Nos, sajnos még nincs… - ismételten nem a leginkább várt válasz. Szó szót követett és kiderült, hogy az előző programozói gárda abbahagyta a munkát és egy új csapat vette át, akik ráadásul teljesen elölről kellett, hogy kezdjék, mivel nem volt a fejlesztésekről semmi dokumentáció. Kértem két nap gondolkodási időt. Ezt a két napot arra használtam fel, hogy egyrészről átnéztem a piacon már működő, hasonló szektorban tevékenykedő internetes áruházakat, azok esetleges hibáit próbáltam felderíteni. Végignéztem az ilyen típusú honlapok tesztelési technikáinak legújabb vívmányait. Másrészről konzultáltam szakmabeli barátaimmal, ismerőseimmel, akik

már foglalkoztak hasonlóval. Ez utóbbiaktól az időbeliségre, a tesztelési idő szükségleteiről próbáltam információt gyűjteni.

A két nap elteltével újra találkoztunk, s további kérdésekkel bombáztam a megrendelőt: hány tesztelési ciklust terveztek/enged a projekt? Hány fővel számolhatok a teszt végrehajtás során? Van-e tényleges teszt környezet? Mennyire integrált a rendszer? Lesz-e más, például SAP vagy egyéb vállalat-irányítási rendszerrel összekötve? Stb. A fél órára tervezett megbeszélésből egy egész estés diskurzus lett, aminek az eredménye egy majdnem teljesen elkészült teszt stratégia volt – hozzá kell, hogy tegyem, hogy sikerült egy plusz hónapot kicsalni a megrendelőből.

Egy héttel később – ennyi időt kértem a teszt stratégia elkészítésére – újra találkoztunk és a kész stratégiát átadtam a megrendelőnek, aki elégedett volt a munkámmal. Természetesen itt még nem ért véget az én feladatom, de mivel mindent előkészítettünk maga a tesztelés már nem hozott semmi meglepetést. Köszönhető mindez annak, hogy a stratégiát a fejlesztő csapattal karöltve készítettem el támaszkodva az ő rendszerbeli tudásukra és a folyamatos, a fejlesztés során előjött gyenge pontokra. Végül sikerült a tervezettnél 2 héttel korábban – tehát a projekt tervei szerint 1 hónap helyett csak 2 hetes csúszással – a terméket átadni és a webáruházat elindítani. Ez alkalommal a Dinamikus és a Konzultációs stratégiai modellek ötvözetét alkalmaztam – bátran mondhatom, hogy sikerrel.

De – mint minden elmélet – a gyakorlatban ez szinte sosem alakul ki ilyen tisztán. A legtöbbször egy belső tesztelői csapat dolgozik együtt a programozókkal/fejlesztőkkel, nem ritkán ez utóbbiak közül válik ki, alakul ki a tesztelői gárda. Éppen ezért a teszt stratégia a legtöbb esetben nincs megfogalmazva, vagy csak elméleti síkon létezik.

Szerencsésebb a helyzet, ha ennek a belső tesztelői csapatnak a vezetője egy gyakorlott, profi tesztmenedzser. Ilyenkor csak a kezdeti szakaszban, az első 4-5 projektnél lesz nehézkes a teszt stratégia megfogalmazása, majd annak betartása, ahhoz való alakulás, idomulás. A teljesen rutinszerű működéshez egy újonnan formálódott belső tesztelői csapatnál – egy profi vezetővel – átlagosan 10 projekt teljes végrehajtása szükséges. (Persze csak abban az esetben, ha a csapatösszetétel minimális mértékben változik a projektek alatt.)

SZÉCHENYI 2020

Amennyiben a tesztelést egy külső, teljesen független cég képviseli, a helyzet – és a teszt stratégia – még jobb. Független tesztelői csapat esetén szükségszerű és szinte létfontosságú a teszt stratégia írásba foglalása és mindkét/három fél által elfogadása, majd annak folyamatos és pontos betartása. Ennek hiányában nehéz lenne a tesztelői csapatnak bebizonyítani, hogy a megfelelő munkát végezte el a megfelelő módon. Még nehezebb lenne a végrehajtott munka anyagi vonzatát rendezni…

Teszttervezés és becslés

Ahogy azt már korábban megjegyeztük, egy fejlesztési projekten belül a tesztelést is értelmezhetjük projektként. A teszt tervet pedig értelmezhetjük a tesztelési projekt tervének. Ha innen közelítjük meg a teszt tervet, könnyebben érthetőbbé válik a tartalma és a célja is.

Definíció szerint (ISTQB) a tesztterv a következőket tartalmazza:

- Projekt, feladat rövid ismertetése
- Felhasznált dokumentumok
- Leszállítandó dokumentumok
- Tesztelés hatóköre (Érintett rendszerek, Tesztelendő funkciók, Nem tesztelendő funkciók)
- Teszt stratégia
- Teszt eredmények kiértékelésének folyamata
- Teszt fázisok, időzítések
- Döntéshozatali fórumok
- Elfogadási kritériumok (Elfogadási kritériumok az egyes fázisokban, Tesztelés felfüggesztésének kritériumai, Tesztelés újraindításának kritériumai)
- Erőforrások (Teszt eszközök, Személyi erőforrás, Teszt környezetek)
- Szerepkörök, felelősök és feladatok
- Tesztadatok
- Kockázatok
- Betanítás, oktatás

Ezt a dokumentumot jellemzően a teszt menedzser készíti el, de esetleg egy gyakorlott tesztelőt is megbízhatnak vele. Vannak részei azonban, amit egyedül tipikusan nem írhat meg senki – nem kizárólag egy teszt menedzser, de ténylegesen egyedül senki nem hozhatja létre. Legalábbis releváns módon nem. Ilyen például a kockázati rész. Ez tipikusan egy olyan terület, amit többen kell, hogy megírják.

(Megjegyzem: maga a kockázat-elemzés folyamata külön irodalommal rendelkezik, amit most itt nem fejtenék ki teljes mélységében a helyszűke miatt.) A gyakorlatban sokszor a projekt menedzser és a teszt menedzser közötti kommunikációt rögzíti a tesztelés tervezéséről. Mások is adhatnak információt, mint például rendszergazdák, adatbázis adminisztrátorok, akik a teszt környezetért felelnek. Ezt a dokumentumot célszerű a projekt vezetőivel és a projekt többi tagjával is elfogadtatni és akár aláíratni is.

Pár hónappal ezelőtt megkeresett az egyik projekt menedzser, akinek az aktuális projektjéről érintőlegesen már hallottam. Beszélgetésünk az első alkalommal nagyjából így hangzott:

PM: Arra szeretnélek kérni, hogy készíts egy teszttervet a jelenlegi projektemhez.

ÉN: Igen, hallottam már erről a projektről. Hol is tartotok most?

PM: Már elkezdődött az implementálás.

ÉN: Ez nem éppen szerencsés. Tudod, ideális esetben hamarabb érdemes bevonni a teszt menedzsert. Mikorra lesznek kész a kódolással? Hány programozó dolgozik rajta?

PM: Az egységtesztet a fejlesztők csinálják. Azt gondoltam, hogy elég, ha csak most szólok. Még egy hétig kódolnak, 4 fejlesztő dolgozik rajta, mindegyik 50%-ban.

ÉN: Ha jól értem, akkor integrációs és rendszer teszthez szeretnéd, ha tervet készítenék?

PM: Igen.

ÉN: Jó lett volna, ha már a projekt elején szólsz, esetleg akkor, amikor az elvárások összegyűjtése elkezdődött. A tervezés még akkor is sok időt vehet igénybe, sok információt kell begyűjteni. Esetleg átnézhettem volna az elvárásokat is, még mielőtt elkezdik az implementálást. Lehet, hogy néhány hibát már a dokumentációból ki tudtam volna szűrni.

PM: Mire van szükséged?

ÉN: Szeretnék hozzáférést a projekt teljes dokumentációhoz. Remélem, hogy benne vannak a projekt céljai, ütemezése (pl.: mikor megy élesbe), módszertana (V-

shape vagy Agile?), menedzsment eszköz, kockázatok, minőségi elvárások. Ezen túl szeretném tudni, hogy kik fogják előkészíteni a teszt környezetet. Honnan, melyik menedzsertől kérhetek tesztelői erőforrást? Van-e elérhető teszt menedzsment eszköz?

PM: Sajnos a projekt dokumentáció nem teljes, de igyekszem minden információt beszerezni. Azt gondolom, hogy még szükségünk lesz néhány beszélgetésre…

A további beszélgetéseink során minden szükséges információt megkaptam ahhoz, hogy elkészítsem a tervet. (Ennek folyománya volt, hogy a projekt dokumentáció is elkészült.) Mivel időközben megváltoztak bizonyos jellemzői a projektnek, nekem is számtalanszor kellett módosítanom a Teszttervet – még annak elfogadása után is! De ez már egy másik történet…

A teszt terv egy élő dokumentum. A fejlesztési projekt folyamán változik alakul, formálódik. Nem is feltétlenül maga a dokumentum a legfontosabb, hanem az hogy a megfelelő kérdéseket feltegyük és megtaláljuk a válaszokat, megszülessenek a döntések. A tesztterv mindezek rögzítése. Annak, aki felelős a teszt tervért, kell megfogalmazni és feltenni ezeket e kérdéseket a megfelelő személynek. Néha ezek a kérdések kellemetlenek lehetnek a másik fél számára – Pl.: nincs megfelelő projekt dokumentáció, nincs részletezve a pontos követelmény, megfelelősége az adott projektnek –, de ennek ellenére fel kell tenni ezeket a kérdéseket is, mivel a megfelelő információ nélkül a teszt tervezése legalábbis kétségessé válik. Viszont érdemes megőrizni a partneri viszonyt a projekt résztvevőivel. Ehhez jó emberismeret, eredményes konfliktuskezelés szükséges. Ne feledjük: kérdéseinkkel, kéréseinkkel befolyásoljuk a teljes fejlesztési folyamatot. Figyelnünk kell arra, hogy ez ne legyen túlzott mértékű. Ha ezekre a tényezőkre nem tudunk koncentrálni, azzal járhat, hogy nem szívesen fognak együtt dolgozni velünk a projekt résztvevői.

Fontos alkotóeleme a teszttervnek a különböző erőforrások, idő-intervallumok meghatározása. Mivel sok módosító tényező merülhet fel a projekt, illetve a tesztelési folyamat során pontos adatokat nem lehet megadni ezekre vonatkozólag. Csak becsülni lehet – persze ennek is van tudományosnak mondható módszere(i), modellje(i). (Csak annak mondható, de nem tudományos. Akárhogyan is nevezik, mégiscsak egyfajta becslés.)

Az egyik ilyen módszer a *mérőszám alapú becslés.* Ez a módszer az előző, hasonló projektek tapasztalatai, mérőszámai alapján becsüli meg az adott projekt erőforrás- és időigényét. Ez tapasztalati

eredményekre támaszkodik, ami elég jó megközelítést ad a tényleges szükségletek kielégítésére, viszont feltétele, hogy már több, hasonló típusú, méretű és bonyolultságú projekt tesztelésének dokumentált anyaga legyen a becslés hátterében. Ezen dokumentumok a legtöbbször azonban – sajnálatos módon – megbízhatatlanok. Ennek több oka lehetséges, de az egyik leggyakoribb az, hogy a megrendelők nem szeretik azt látni, hogy a tesztelés valójában mennyi időt és pénzt emészt fel. Ennek érdekében gyakran az időtényező adatait módosítani szokták…

A másik a *szakértő alapú becslés*. Ez akkor is működik, ha nincs hasonló projekttel tapasztalata teszt menedzsernek, viszont a részfeladatok, azok idő- és pénzbeli vonzata tekintetében rendelkezik szakértő csapattaggal. Amennyiben a menedzser olyan szerencsés helyzetben van, hogy a projekthez kapcsolódó összes részterületen rendelkezik szakértő kollégával, úgy azok tapasztalatait és véleményét alapul véve elég pontos közelítésben tudja meghatározni az adott projekt tesztelési ciklusának erőforrás igényét.

Más módszerek is léteznek, mint a WAG vagy a SWAG, Súlyozott átlag (Weighted Average), Delphi, Összehasonlító becslés (Comparative vagy Historical Estimating). Némelyik valójában nem valós módszer, csak arra találták ki, hogy a megrendelő felé valami tudományos néven eladják a véletlenszerű becsléseiket… Mások azonban ténylegesen egy módszeresen kidolgozott képletet használnak. Lássuk ezeket részleteiben:

WAG és SWAG[1]

Sajnos elég gyakran használt módszerek annak ellenére, hogy az általuk kapott értékek igen ritkán találkoznak a valósággal. A legtöbbször akkor használják, amikor igen rövid időn belül kell számokat és igényeket meghatározni egy-egy adott tesztelési folyamat kapcsán – mindenféle előzetes felmérést, tapasztalatot, tényszerű adatot nélkülözve. Éppen ezért nagy a kockázata annak, hogy ezzel a módszerrel rosszul becsüljük meg a szükséges erőforrásokat. *Ezek a módszerek erősen kerülendőek!*

Súlyozott átlag (Weighted Average)

Hárompontos technikának vagy statisztikai technikának is hívják ezt a módszert. Minden feladat esetében három szempontot kell megvizsgálnunk:

1. Legjobb esetben (Optimista)
2. Normál (elfogadható) esetben (Normál)
3. Legrosszabb esetben (Pesszimista)

[1] WAG = Wild-Ass Guess, azaz „Hasraütés"; SWAG = Scientific Wild-Ass Guess, „Tudományos hasraütés"

milyen és mennyi erőforrás (idő, ember, eszköz, pénz) szükséges a végrehajtásához. A képlet a becsléshez pedig így néz ki:

$$E = \frac{Optimista + (4 \times Normál) + Pesszimista}{6}$$

Ahol

- E = a becsült érték, azaz a kérdésre adandó válasz maga;
- Optimista = a legjobb esetben, amikor minden megy, mint a "karikacsapás";
- Pesszimista = ami elromolhat, az el is fog romlani;
- Normál = amikor minden működik kisebb-nagyobb nehézségekkel, de még elfogadható;

A tapasztalatok alapján ez a képlet adja a legjobb becslést minden projekt esetében. Használatát erősen ajánlom – tapasztalataim alapján szinte tökéletes adatokat ad vissza. Természetesen ez a módszer sem tökéletes, hiszen a három tényező (optimista – normál – pesszimista) megállapítása szintén csak becslés. Ezekhez is kell némi múltbeli tapasztalat, tehát ez a módszer egy kicsit bevonja a Delphi (lásd következő módszer) vagy az Összehasonlító becslés módszerét is. Esetleg mindkettőt.

Delphi

A csapatban dolgozó specialisták, szakemberek szaktudására épít. Ha minden területen rendelkezik a teszt menedzser egy ilyen emberrel, akkor érdemes ezt a technikát választania, hiszen a szaktudás és a tapasztalat igen jó megközelítésben pontos becslést ad. Erősen hasonlít a szakértő alapú becslésre, annyi a különbség, hogy itt a csapat ténylegesen adott.

Összehasonlító becslés (Comparative vagy Historical Estimating)

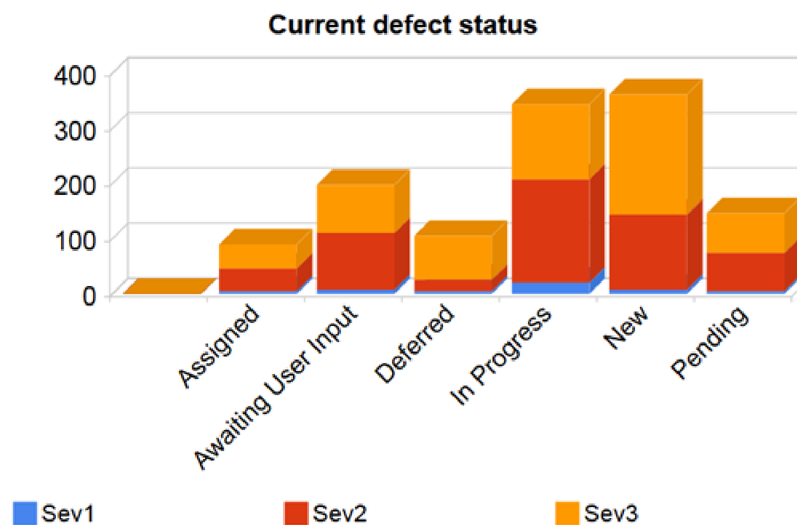
Ez a technika az előző projektek adataira épít. Alapjául ugyanolyan, vagy nagyon nagymértékben hasonlító projekt lebonyolítása során keletkezett adatokat használ. Feltétele: legyenek korábbi, jól dokumentált projektek. Csak olyan közegben használható, ahol ez létezik, pontosan vezetett és nem "kozmetikázott". Leginkább cégektől független tesztelők, tesztelői cégek esetén lehet életképes ez a technika.

A felsoroltak közül a legtöbb projekt esetén a legpontosabb adatot a súlyozott átlag adja. Ennél egy kicsit jobb, ha ténylegesen szakértőkkel dolgozhat a teszt menedzser, hiszen a szakértők tapasztalatai alapján még pontosabb adatot kaphatunk a tesztelési idő és az ahhoz szükséges erőforrás-mennyiség meghatározásához. Mivel ilyen szerencsés helyzetben – szakértői csapat nem áll mindenkinek rendelkezésére – igen kevés tesztelési vezető van, ezért én a súlyozott átlagú meghatározást javaslom

SZÉCHENYI 2020

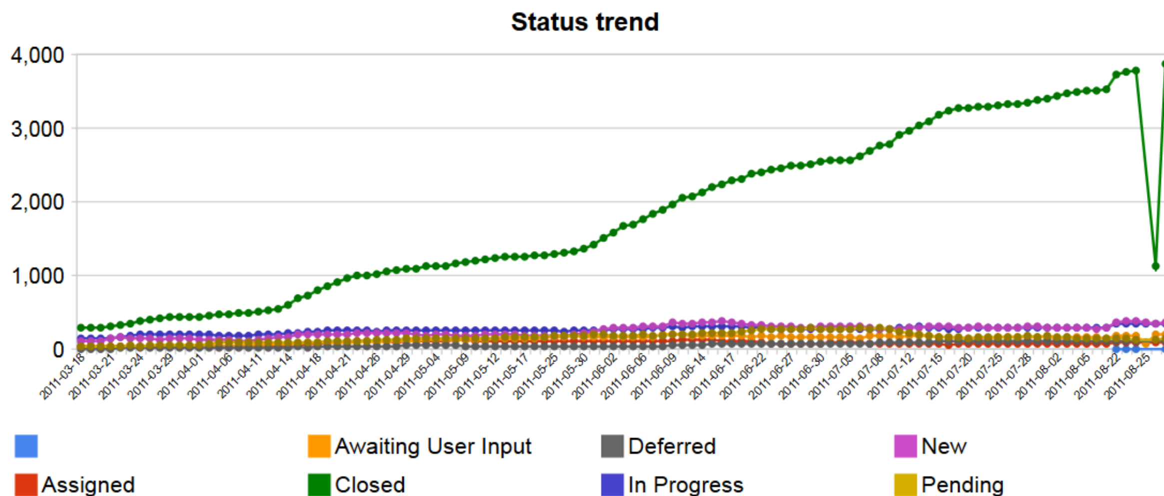
Teszt monitorozás

A teszt monitorozás során azt figyeljük, hogy a teszttervhez képest mi történik a teszt végrehajtása során. Státusz és trend adatokat használunk azon metrikák alapján, amit a teszttervben meghatároztunk. Státusz adat egy metrika adott időpontra vonatkozó értéke. Pl.: hány teszt esetet futtatunk le, hány hibát találtuk, hány órát dolgoztunk egy adott időpontig, mennyi pénzt költöttünk eddig.

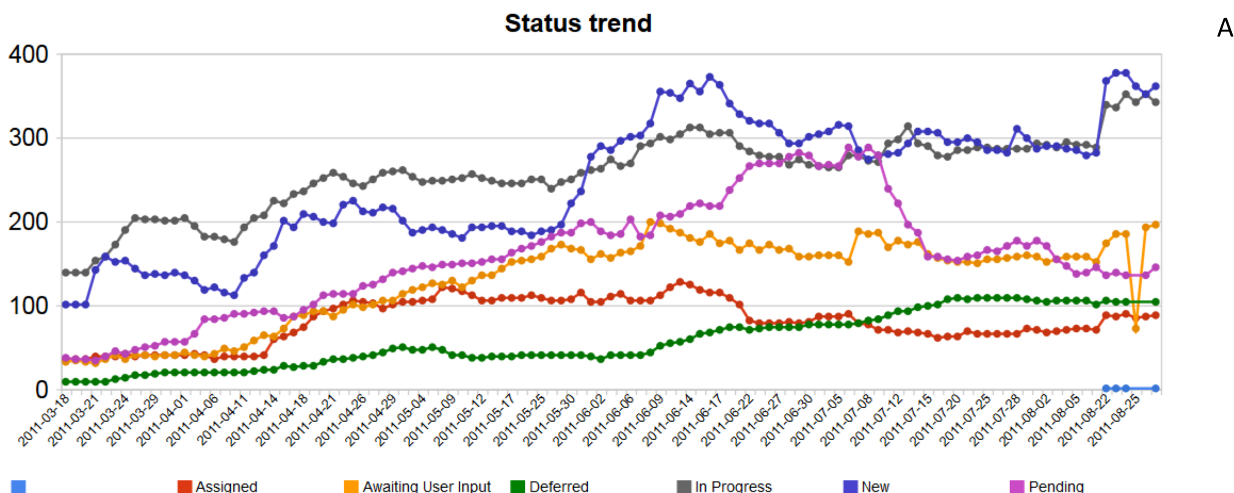


A fenti ábrán tipikus státusz adatokat láthatunk arról, hogy mennyi, milyen státuszú és súlyosságú hiba van rögzítve egy adott pillanatban. Fontos lehet ez az adat abból a szempontból, hogy találtunk-e kritikus hibát (Sev1) vagy sem. Ezek a hibák milyen státuszban vannak pl. feldolgozás alatt (In Progress). Lehet, hogy abban állapodtunk meg, hogy a rendszert csak úgy adhatjuk át, ha nincs benne kritikus hiba. Ha ezt teljesíteni akarjuk, nem csak az a fontos, hogy egy adott pillanatban hány kritikus hibáról tudunk, hanem az is hogy hogyan alakul ezeknek a hibáknak a száma. Tehát arra vagyunk kíváncsiak, hogy az idő függvényében hány kritikus hibát találunk, rögzítünk, mennyin kezdtünk el dolgozni, mennyit sikerül kijavítani. Ez egy trendadat.

A lenti ábrán trend adatokat láthatunk. Az egyik kategóriában (Closed) lényegesen több adat van a többihez képest, ezért ezek időbeni változása nem látszik. Következő ábráról töröltük ezt a kategóriát.



Az ábrán látszik egy kiugró adat. Még mielőtt bárki megpróbálná valamilyen teszteléshez kapcsolódó eseményhez kötni, elárulom, hogy az adatok feldolgozása közben történt egy hiba – emiatt a kiugró adat. Érdemes tisztában lennünk azzal, hogy hogyan történik a metrikákhoz az adatgyűjtés és a feldolgozás. A grafikonok, ábrák elemzésénél fontos lehet.



A fenti ábrán a hibák számának változását látjuk az idő függvényében. A különböző színekkel a különböző státuszokat jelöltük. Érdemes például azt megfigyelni, hogy az új (new) hibák száma hogyan alakul. Bizonyos időszakokban a hibák száma növekszik, majd csökken. Viszont összességében a teljes időszakra vonatkozóan növekszik. Mit jelenthet ez? A periodikus növekedés, csökkenés lehet annak az eredménye, hogy egymást követő teszteléses időszakokban rögzítik a hibákat a tesztelők, a közbenső időszakokban a

fejlesztők javítják azokat. Viszont az is látszik, hogy a fejlesztők nem tudnak lépést tartani. Összességében több hibát találnak a tesztelők, mint amennyit ki tudnak javítani a fejlesztők. Ebben a konkrét esetben még súlyosabb a helyzet. Ugyanis nem csak két státusz között oszlanak el a hibák. A különböző státuszokba az új státuszúakból kerülnek át a hibák. Az újonnan rögzítet hibákból nem csak lezárt, javított (closed) hibák lesznek, hanem minden más státuszú. Tehát nem lett minden olyan hiba kijavítva, ami már nem új státuszú. Ez azt jelzi, hogy esetleg még nem értük el azt a minőségi szintet, amit már el lehet fogadni. Már az is pozitív jelnek tekinthető ha az új hibák száma csökkenő tendenciát mutat.

Konfiguráció menedzsment

A konfiguráció menedzsment olyan tevékenység, ami során az informatikai infrastruktúra (konfiguráció) elemeit azonosítjuk, a közöttük lévő kapcsolatot, változásokat rögzítjük. Célja, hogy csak egy központilag jóváhagyott elem kerüljön a rendszerbe, így minden, a konfigurációban végbement változás egy központi helyen, kronologikusan tárolva legyen, így bármikor visszakereshető a teljes konfiguráció egy adott dátumra levetített állapota.

Konfiguráció menedzsment alá tartozó elemek (a teljesség igénye nélkül):

- szolgáltatások
- felhasználók
- szoftver és hardver elemek
- kapcsolódó dokumentációk
- az egyes elemek közötti kapcsolatok

A konfiguráció menedzsment felel azért, hogy a szoftver fejlesztés teljes életciklusában a megfelelő hardveres és szoftveres elemek készen álljanak az adott fázis végrehajtására. Például a konfiguráció menedzsment felel azért, hogy a fejlesztőknek meglegyen a fejlesztési környezet, azaz ha egy már meglévő rendszerre fejlesztenek egy új feature-t (erre nincs jó magyar szó), akkor a fejlesztői környezet az éles rendszer másolata legyen a fejlesztés kezdetekor. De ugyanígy a teszt környezet is. Természetesen lehetnek eltérések (hardveresen) az éles rendszer és a teszt rendszer között, de a tesztelést ennek tudatában kell megtervezni, a hardver-függő tesztelést (stressz-teszt, teljesítmény-teszt, stb.) ehhez mérten kell előirányozni.

A közelmúltban az egyik kollégám panaszkodott az éppen aktuális projektjére. Először nem is értettem, miért, hiszen a státusz-jelentések alapján – amit volt szerencsém én is megkapni – a projekt a tervezett ütemben haladt. Minden

kód készen lett határidőre, áttették a teszt rendszerre, a tesztelést a megfelelő időben el lehetett kezdeni, az erőforrások készenlétben, stb. Mindezek alapján mindig csodálkozva hallgattam az óvatos panaszkodását. Mivel nem értettem, egy idő után inkább rákérdeztem, hogy mi is a gond. Így utólag úgy gondolom, hogy a saját lelki nyugalmam és egészségi állapotom javítandó talán nem kellett volna…

Kollégám ugyanis elmesélte, hogy valóban minden kód készen lett, mindet feltették a teszt rendszerekre – de persze nem teljesen időben és nem teljesen egyszerre. (Ez a többes szám már gyanús volt…) És itt jött a baj: hogy nem egy rendszerre… Némely kódot az elsődleges teszt rendszeren, másokat a másodlagos teszt rendszeren kellett tesztelni. Volt olyan, hogy a tesztelés alatt derült ki, hogy bizonyos kódrészletek szükségesek a többi fejlesztés működéséhez, így nem írhatjuk felül az adott projekt kódjával, ami újabb fejlesztést – és ezzel együtt még több tesztelést – indukált. "És akkor mi a baj? Mindegyik rendszer az éles rendszer másolata, nem?" – merülhet fel sokakban a kérdés. Igen, valóban az – illetve annak kellene lennie. Viszont az egyes modulok – amiket két rendszerre tettek fel – egymásra is hatással voltak, de mivel nem kerültek egy rendszerre, ezen hatásokat nem lehetett tesztelni. De ez még nem minden: a tesztelési ablak alatt a jelentett hibákat a fejlesztők azonnal javították – ami végeredményben igen jó – és fel is tették a megfelelő teszt rendszerre – ami viszont tesztelés szempontjából nagyon nem jó. (Tesztelési ciklus alatt nagy kockázatot rejt magában, ha a kódot a teszt-rendszeren folyamatosan változtatják/frissítik. Főleg olyankor, ha ezt nem is közlik a tesztelői csapattal… A teszteredmények ugyanis mindennek tükrében legalábbis megkérdőjelezhetővé válnak.) Arról nem is szólva, hogy a tesztelési idő is jelentősen megnövekedett, hiszen két konfigurációt kellett használni, a teszttervet módosítani kellett, akárcsak a teszteset-gyűjteményt. Némely tesztesetet pedig – a sűrű kódváltoztatás miatt – többször is végre kellett hajtani. Ennél a projektnél a konfiguráció menedzsment egyetlen elemét sem használták megfelelően: a teszt-rendszerek nem voltak előkészítve, a kapcsolódó alkalmazások nem voltak felkészítve a tesztelésre, a szerver leállások és kódfrissítések nem voltak időben kommunikálva… Lehetne még sorolni, de azt hiszem, látszik a lényeg. (Mindennek ellenére a kollégám heroikus erőfeszítéssel és jelentős túlórával, de mindent

letesztelt, minden fellelt és vélt hibát jelentett, melynek eredményeképpen egy minőségi termék kerülhetett az éles rendszerre.)

Kockázatok

Az egyik legnehezebb feladat a kockázatok feltárása és megfelelő kezelése. Definíció szerint a kockázat egy olyan, valamilyen valószínűséggel bekövetkező esemény, ami negatív hatással van a projektre vagy a projekt eredményére (legyen az szoftver, esemény vagy bármi más egyedi produktum). A nehézség részben abban rejlik, hogy nagyon sokféle kockázat létezik. Másrészt nem előre megjósolhatóan következik be. Könnyen előfordulhat, hogy teszt menedzser azonosít kockázatokat, lépéseket tesz, pénzt költ arra, hogy ha bekövetkeznek, könnyen el lehessen hárítani a következményeket, de mégsem következnek be. Később pedig már nehezen tudja elfogadtatni, hogy pénzt költsenek erre.

A kockázatokat három kategóriába sorolhatjuk:

Project kockázat: Ez a kategória magára a projectre vonatkozik. Azokat a kockázatokat gyűjtjük ide, amik a project működését veszélyeztetik, a költségekre, a tervezett időkeret vagy a szükséges erőforrásokra lehet negatív hatással. Pl: a betervezett erőforrások mégsem lesznek elérhetőek (esetleg betegség miatt).

Termék kockázat: Ezek a kockázatok a termék tervezett funkcióit, tulajdonságait, minőségét érinthetik. Pl: bizonyos funkciók mégsem fognak működni.

Üzleti kockázat: Ezt a kategóriát ritkábban említi a szakirodalom, ennek ellenére fontos lehet ha külső beszállítóval dolgozunk együtt, aki termékeivel vagy szolgáltatásaival hozzájárul a project eredményeihez. Az ide sorolt kockázatok a külső beszállítón keresztül lehetnek hatással a projectre. Pl: a beszállító mégsem teljesít időben, vagy a project alatt jobb megoldást találunk a piacon, mint amit a beszállító ajánl.

Ahogyan már említettem, a kockázat-analízis, vagy kockázat-elemzés (risk analysis) nem lehet csak a teszt menedzser feladata. Ami az ő szemében nem kockázat, lehet, hogy a fejlesztő vagy a végfelhasználó szemében már kockázatot jelent. Éppen ezért egy kockázati elemzést érdemes a projekt menedzserrel, a fejlesztővel és a végfelhasználóval közösen végrehajtani.

Különböző kockázat-elemző technikák léteznek, mint a minőségi kockázat elemzés (Qualitative Ris Analysis) vagy a mennyiségi kockázati elemzés (Quantitative Risk Analysis). Személy szerint én egy egyszerű táblázatot használok:

Kockázati tényező	Előfordulási valószínűség	Hatáspont	Kockázati pontérték	Reagálás
Emberi erőforrás hiánya	5	7	35	Biztosítani a megfelelő utánpótlást valamint a dokumentációt folyamatosan frissíteni kell.
Áramszünet a projekt ideje alatt	2	6	12	A fontos szervereket, központokat szünetmentes tápegységgel kell ellátni. A fejlesztők illetve más, számítógépen dolgozók használjanak saját teleppel rendelkező számítógépet, például laptopot, notebook-ot.

Ahol a

- Kockázati tényező = rövid megfogalmazása a kockázatnak
- Előfordulási valószínűség = 0-10-ig terjedő pont, ahol a 0 a legkisebb valószínűség az előfordulásra
- Hatáspont = 0-10-ig terjedő pont, ahol a 0 a legkisebb hatással bíró esemény
- Kockázati pontérték = az Előfordulási valószínűség és a Hatáspont szorzata
- Reagálás = a kockázat megjelenésekor a tervezett reakció.

Természetesen a kockázati tényezőket a Kockázati Pontérték alapján kell rangsorolni, de el kell, hogy mondjam, hogy néha el lehet – sőt, kell – ettől térni. Mire is gondolok? Például ha a projekt maga egy uszodai úszóverseny, akkor a kockázatok között megjelenhet az, hogy beomlik az uszoda. Ennek az előfordulási valószínűsége szinte 0, de legyen mégis 1-es értékű az Előfordulási Valószínűsége. Viszont a hatása a rendezvényre katasztrofális, tehát a Hatáspont 10. Ugyanakkor megjelenik az is, mint kockázat, hogy némely versenyző nem érkezik meg az adott napra. Ennek előfordulási valószínűsége nem túl magas (5), de hatása csekély mértékű magára a teljes versenyre (3). (Esetleg 1-2 versenyszámban nem lesz 3 induló, csak 2.)

Így a Beomló tető KPÉ[2]-je 10 (1x10=10), míg a hiányzó versenyző(k)é 15 (5x3=15). Ezen számítás alapján a hiányzó versenyzők kockázata nagyobb, mint a beomló tetőé…

Incidens menedzsment

Az ISTQB definíciója szerint az incidens menedzsment az incidensek felismerésének, vizsgálatának, a különböző intézkedések és rendelkezések szervezésének folyamata. Magába foglalja az incidens rögzítését (log), osztályozását (priority) és hatásának vizsgálatát (severity).

Tulajdonképpen szoftvertesztelés szempontjából nem is incidens-, mint inkább hiba-menedzsmentről (defect management) lehet beszélni. Az oka ennek az, hogy az incidens lehet olyan esemény, amelynek kijavításához nem szükséges a rendszer egy részén vagy egészén módosítani, csak egyszerűen valamit manuálisan kell beírni / átállítani / kijavítani. (Az ilyen incidensek rövidítése a programozói szlengben: PEBKAC, PBKAC[3], POBCAK[4], esetleg ID-10-T Error, vagy ID107 Error[5].) Tipikusan ilyen incidens, amikor egy felhasználó a termék regisztrációja során hibásan adja meg az e-mail címét és ezt kell javítani a rendszerben.

Miért is van szükség hiba-menedzsmentre? Ennek oka nagyon egyszerű: egy-egy projekten – főleg, ha időben elég terjedelmes, vagy technikailag komplex – nem csak egy tesztelő dolgozik, esetleg még csak nem is egy helyen (irodában) és/vagy időzónában, de ennek ellenére biztosítani kell azt, hogy az egymás által feltárt hibák a többi tesztelő számára is láthatóak legyenek, elkerülve ezzel a többszörös hiba-regisztrációt, valamint segítve a teszttervezés és hibajavítás folyamatát. Ezen kívül a hiba-menedzsment hathatós segítséget nyújt a minőség méréséhez szükséges metrikák elkészítésében, valamint a szoftver fejlesztőinek a fejlesztés további szakaszainak megtervezésében.

> *"Még kezdő tesztelő korromban történt meg velem a következő eset: egy szép tavaszi reggelen éppen leültem az asztalomhoz a munkahelyemen, amikor észrevettem, hogy egy fontosnak titulált üzenetet villogtat a levelezésem. Nosza, ugrottam és meg is nyitottam – hiszen "fontos".*
>
> *Az üzenetet elolvasva nem tudtam, hogy sírjak, vagy nevessek. Ugyanis mindössze ennyi volt a tartalma:*
>
> *'A rendszer nem működik, segítsetek!'*

---

[2] KPÉ = Kockázati Pontérték
[3] Problem Exists Between Keyboard And Chair
[4] Problem Occurs Between Chair And Keyboard – az amerikai kormányzati és katonai rövidítések listájáról
[5] ID-10-T vagy ID107 = IDIOT (IDIÓTA)

SZÉCHENYI 2020

Próbáltam behatárolni, hogy melyik rendszer lehet, amelyik 'nem működik'. A feladó alapján eléggé le tudtam szűkíteni a kört, mindössze 12 szóba jöhető alkalmazásra. Mivel mindegyiket ismertem nekiláttam kideríteni a 'nem működik' varázsszavak jelentését…

Hosszas, órákon át tartó ellenőrizgetés után még mindig nem voltam biztos a dolgomban, hiszen mindegyik alkalmazás működött. Úgy, ahogy annak kell. Minden szerver élt, a munkafolyamatok elkezdődtek és be is fejeződtek – egyszóval: nem értettem. Sajnos nem tudtam azonnal rákérdezni a levél feladójánál, hogy mi is a hiba, mert elég nagy volt az időzóna-eltérés közöttünk. Mindenesetre próbáltam megtalálni, mi is a gond, de egészen addig, amíg nem tudtam elérni a kollégát – egyébként ez 4 napba tellett, mert szabadságra ment, miután a levelet elküldte – nem tudtam feltárni a hibát. Amikor azonban végre telefonon elértem, akkor megvilágosodtam…

Nyers fordításban 'A rendszer nem működik…' jelentése:

1. *Teszt rendszeren jelentkezett a hiba*
2. *A munkafolyamat egy bizonyos, igen speciális pontján egy nagyon ritka ország-nyelv-termék együttállásban egy olyan ágra fut, ami nincs lekezelve a rendszerben*
3. *A pontos adatok már nincsenek meg a kollégának sem – elfelejtette, milyen adatokat használt (ország-nyelv-termék)*

Ekkor döntöttem el, hogy a kollégáknak, akiknek szükséges tréninget fogok tartani arról, hogy hogyan is kell egy hibát lejelenteni. Hosszas előkészületek után – és nem kevés ambícióval – el is érkezett a nagy alkalom: bár sokan ellenezték, én megtartottam a tréninget. Eredményes volt-e? Határozottan igen."

A hiba-menedzsment fő eleme maga a hibajelentés, vagy hibajegy. Ennek rögzítése általában egy hiba-kezelő rendszerben történik, ami lehet egy szoftver, adatbázissal összekötve és megfelelő felhasználói felülettel kiegészítve, de lehet akár egy Excel tábla is – minden attól függ, hogy a projekt milyen eszközöket engedhet meg magának. (Az ingyenes, web-alapú hiba-kezelő eszközök elterjedésével az Excel alapú hiba-kezelés szinte "kihalt".) Az a legfontosabb, hogy minden, a projektben résztvevő tudja kezelni az adott hiba-kezelő rendszert.

A hibajegynek tartalmaznia kell a következő elemeket:

SZÉCHENYI 2020

- **Egyedi azonosító** (többnyire a rendszer automatikusan generálja)
- **Cím vagy rövid leírás** – pár szóban összefoglalva az adott hibajelenség lényegét. Nem szabad túl hosszúnak lennie - 100 karakternél többnek nem érdemes lennie.
- **A hiba részletes leírása**, amiben a hiba előidézéséhez szükséges folyamat lépésről-lépésre rögzítve van, valamint mindaz, amit a hiba okozhat a rendszerben/termékben.
- **Környezet**, ahol a hiba jelen van, azaz ahol a hibát reprodukálni lehet. Fontos, hogy a környezet leírásánál – szerver-kliens kapcsolat esetén – nem csak a helyi vagy aa szerver gép leírása kell, hanem mind a kettő nagyon fontos. (pl.: helyi környezet: Windows 7 x64 Enterprise ENG SP1, Firefox v37, etc., Szerver: Stage környezet, amennyiben a szabványos környezeti rétegeket (tier) használunk)
- **A hibafeltárás pontos ideje** – általában napra pontosan elég megadni, de szintén projektfüggő. Akár percre pontos hivatkozás is fontos lehet, ha pl.: Agile XP metódust használ a projekt.

Az alábbi információk nem kötelező jellegűek, vagy nem a tesztelő hivatott beállítani, viszont hasznos információval szolgál a fejlesztő vagy az üzleti elemző, esetleg a megrendelő felé, valamint a hibajegyek közötti keresést vagy rendszerezést segítik elő:

- Hibajegy nyitásának dátuma
- Utolsó módosítás dátuma
- Projektazonosító (amennyiben van ilyen)
- Al-projekt azonosító (komplex projekteknél lehet ilyen)
- Kapcsolódó hibajegy(ek) (ha van ilyen)
- Fontosság/Sürgősség (priority)
- Súlyosság (severity)
- Hibajegy tulajdonosa (a rendszer általában automatikusan hozzárendeli a rögzítő felhasználóhoz) – aki a hibát rögzítette a hibakezelő-rendszerben
- Felelős személy – aki az adott modulért/folyamatért felel
- Tesztelő személy – aki az adott modult/folyamatot teszteli
- Értesítendő személyek – akiknek még tudniuk kell az észlelt hibáról
- Határidő – a hiba javításának határideje
- Érintett verzió
- Megrendelő – aki az adott modult, folyamatot megrendelte

SZÉCHENYI 2020

- Csatolt anyagok (pl.: log állomány, ha van, vagy képernyőkép)
- Kapcsolódó specifikáció
- Hibajegy státusza
- Egyéb információ(k), megjegyzés(ek)

Vannak esetek, amikor ezeket a részleteket, vagy ezek egy részét maga a tesztelő is ki tudja tölteni. Tapasztaltabb tesztelők – vagy programozókból, fejlesztőkből kikerült tesztelők – a hiba leírása mellett sokszor már a lehetséges megoldást is felvázolják, esetlegesen javaslatokat tesznek az adott hiba forrásának (root cause) megjelölésével együtt.

A tesztelő szinte minden nap találkozik a hiba-menedzsment rendszerrel, néha akár többel is. Sokszor már rutinszerűen ír meg egy-egy hibajegyet – de ezt a rutint nem egyszerű megszerezni. Érdemes megvizsgálni többféle hiba-menedzsment eszközt, hogy ne érje a tesztelőt meglepetés. Egy pár ingyenes, nyílt forráskódú hiba-menedzsment megoldás:

1. BugZilla - http://www.bugzilla.org/ -
2. Mantis - http://www.mantisbt.org
3. Trac - http://trac.edgewall.org/
4. Redmine - http://www.redmine.org/
5. Request Tracker - http://bestpractical.com/rt/
6. OTRS (Open Technology Real Services)- http://otrs.org/
7. EventNum - http://forge.mysql.com/wiki/Eventum/
8. Fossil - http://www.fossil-scm.org
9. The Bud Genie - http://www.thebuggenie.com/
10. WebIssues - http://webissues.mimec.org/

Tesztelő csapat

Egy alkalommal csörgött a telefonom, s a vonal végén egy régi, főiskolai évfolyamtársam köszöntött. Kellemesen elbeszélgettünk úgy 4-5 percen keresztül, nosztalgiáztunk, kikérdeztük egymást az elmúlt évekről. Mikor megemlítettem, hogy mivel foglalkozom jelenleg hirtelen elcsendesedett a vonal.

- *Tudom – szólalt meg végre – ezért is kerestelek meg. Szükségem lenne egy tesztelő csapatra az új szoftveremhez.*

- *Nocsak – lepődtem meg – és mi lenne a feladat? Mekkora csapatra gondoltál? Mi ez az új szoftver? Mikorra kell letesztelni? Milyen szinten kell tesztelni? Mennyi időnk van rá? Milyen környezetet használ?*
- *Nos… Ezek jó kérdések, de a választ egyelőre nem tudom mindegyikre. Fontosak ezek a kérdések mind?*
- *Persze, hogy fontosak. Minden ezen alapszik – legfőképpen a költségek. Ezek nélkül nem tudom, hogy tudok-e segíteni…*

A fenti kis történetben elhangzanak azok az alapvető kérdések, amik meghatározzák egy tesztelő csapat felépítését. Sajnos az is megjelenik, amit általában egy projektvezető reagál a kérdésekre… "Fontosak ezek?..." (Vagy a másik kedvencem: "Majd kialakul, csak segíts! Nem nagy dolog…") Igen, fontosak ezek a kérdések. A legfőképpen a kalkulált költségek miatt, de az időbeliséget is nagymértékben módosítják a válaszok, a tesztelés hosszát, mélységét, stb.

A legkisebb csapat létszáma: 1 fő. Maximuma nem megadható, hiszen akár több száz vagy ezer tesztelő is dolgozhat egy adott projekten (pl. a Windows 8 OS tesztelésén világszerte dolgoztak tesztelők annak ellenére, hogy önmaguk nem is tudták, hogy éppen akkor tesztelőként dolgoztak...).

Két fős tesztelői csapat esetén már beszélhetünk – igaz, kissé túlzó módon – teszt menedzserről és tesztelőről. Nem valószínű ugyanis, hogy a két tesztelő egymástól teljesen függetlenül tesztelnének. Illetve mégis előfordulhat, de az a projekt hihetetlen meggondolatlanságról és erőforrás-pazarlásról tesz tanúbizonyságot. De térjünk vissza csak a tesztmenedzser-tesztelő modellhez: ahogy említettem, már két főnél is megjelennek ezek a szerepkörök (NEM munkakörök, hanem **szerepkörök**. Ez nagyon fontos!), hiszen a tesztelés folyamatának tervezését – bár közösen készítik el esetleg – csak az egyik személy fogja képviselni a megrendelő felé.

Ha a tesztelő csapat nagyobb létszámú, ezek a szerepkörök még élesebben elkülönülnek, feladataik jobban elhatárolódnak. A menedzser-tesztelő modell kiegészül – egy bizonyos létszám felett – egy harmadik szereppel, a koordinátor, vagy tervező szerepkörével. Így a tesztelői "Triumvirátus" a következő lesz: Tesztmenedzser–Teszttervező–Tesztelő.

Még bonyolultabb a helyzet, ha nem projektről, hanem programról (összefüggő projektek egysége), esetleg portfólióról (projektek, programok összessége, amik akár egymástól függetlenek is lehetnek, de egy közös vezetői csapathoz tartoznak) beszélünk. Ilyen esetekben leehet akár több tesztmenedzser is,

akárcsak teszttervező, valamint egyre nagyobb mértékben jelenik meg a tesztautomatizálás is – még több terhet róva a tesztmenedzserre illetve teszttervezőre.

Tesztelő (Teszt végrehajtó)

A tesztelői szerepkör a legegyszerűbb szerepkör a tesztelés folyamatában. (Ez nem azt jelenti, hogy bárkiből lehet azonnal jó tesztelő. Ez is egy szakma, amit tanulni kell és gyakorolni ahhoz, hogy igazán kitűnő lehessen a szakterületén.) Kizárólag az egyes tesztesetek végrehajtásáért, a talált hibák jelentéséért és a teszt jelentés elkészítéséért felel. A gyakorlottabb tesztelőkre már rá szoktak bízni némi tervezési vagy adatgyűjtési/adatgenerálási feladatot is. Egyszerű szerepkör, de néha nagyon unalmas tud lenni – gondoljunk csak a regressziós tesztelésre. De megvan az a szépsége, hogy egy új projekt esetében először a teszt végrehajtó látja működés közben teljes egészében a tesztelendő szoftvert – olyan érzés, mint ha egy új világot fedezne fel az ember.

Teszttervező

Gyakorlott tesztelőkből (teszt végrehajtó) lesz teszttervező. Nem különálló szerepkör, gyakran összekapcsolódik a teszt végrehajtó vagy a teszt menedzseri szerepkörrel. (Esetleg mindkettővel.)

Fő feladata a projekt dokumentáció (Projekt terv, elvárások, rendszer-terv, technikai terv, funkcionális terv, stb.) alapján megtervezni a tesztelési ciklus minden elemét, létrehozni (megírni) a teszt tervet, a teszteset-gyűjteményt (vagy esetleg többet is). Meghatározni a tesztelési ablakok hosszát, idejét és mennyiségét.

Fontos, hogy a teszttervező csak projektekre értelmezhető. Amennyiben már egy programon dolgozik, úgy már teszt menedzserről beszélünk.

Teszt Menedzser (Teszt Vezető)

Több éves tesztelői és teszttervezői tapasztalat után válik – gyakran észrevétlenül – a tesztelő teszt menedzserré. Feladatai nem sokban különböznek a teszttervezőétől, csak a feladatok komplexitása növekedik, mégpedig nagymértékben. Teszt menedzser esetében többet nem egy projektről, hanem több, összefüggő projekt (program) tesztelési ciklusának megtervezéséről beszélünk. Ebben a folyamatban ugyanúgy elkészül egy tesztterv – a Mester Tesztterv – ami alapján a projektek, al-projektek teszttervezői létrehozhatják saját, a projektjükre érvényes teszttervet.

A teszt menedzser jellemzően nem készít teszteset gyűjteményt, mivel program szinten tevékenykedik. Egy teszteset gyűjtemény program esetén nem értelmezhető. Természetes kivételt képez az, amikor a

SZÉCHENYI 2020

program egy hosszadalmas folyamat egyes moduljainak, mint projekteknek az összessége, mert ebben az esetben a teljes folyamatra értelmezhető a teszteset gyűjtemény létrehozása.

Fontos feladata még a teszt menedzsernek a tesztelés folyamatának monitorozása: a hibajegyek számának nyomon követése, a teszttervek, teszt jelentések ellenőrzése és felügyelete, kapcsolattartás és jelentések készítése a program/projekt vezetői számára, metrikák készítése, elemzése. Bármilyen, a tervhez viszonyított – különösen a negatív hatással bíró – eltérés kezelése, problémák megoldása.

Tesztautomatizálás

Szoftvertesztelés során arra törekszünk, hogy a lehető legtöbb hibát megtaláljuk a tesztelt alkalmazásban a szoftver fejlesztési életciklus lehető legkorábbi fázisában. Manuális tesztelők az alkalmazás különböző pontjain hajtanak végre előre definiált teszteseteket, különböző adatokkal. A tesztesetek száma nagyon nagy lehet, amik között előfordulhat, hogy sokszor ugyanazokat a lépéseket hajtja végre egy tesztelő, csak különböző adatokat használva.

Attól függően, hogy mennyi tesztelési ciklusról beszélünk, a tesztek végrehajtása is változhat. Az alkalmazásban történt gyakori kódváltoztatás, új funkciók implementálása, különböző környezetben történő futtatás mind indokolhatja a tesztek újbóli végrehajtását.

Automatizált tesztek segítségével arra törekszünk, hogy kiváltsuk a manuális regressziós teszteket, fókuszálva a hatékonyságnövelésre, teszt lefedettségre valamint az erőforrás-igény csökkentésre.

Automatizált teszt

Szoftver tesztelésére használt szoftver, ami képes kontrollálni, futtatni, riportálni a különböző tesztek végrehajtását. Képes a várt és az aktuális eredmények kiértékelésére, be és kimenő adatok kezelésére, valamint a futás során felmerülő váratlan hibák kezelésére.

Nagyon sok fajta automatizálásról beszélhetünk. Ebben a részben a funkcionális tesztek automatizálásáról lesz szó. Olyan módszerek és példák lesznek bemutatva, amik a felhasználói felületet használják. Ennek a résznek nem célja, hogy bemutassa és részletesen tárgyalja a különböző teljesítmény, terheléses és modul tesztelés lényegeit és folyamatait.

Tények:

Automatizált teszt nem hajt végre több lépést annál, mint amit lekódoltunk.

Nem helyettesíti teljes egészében a manuális tesztelést.

Nem biztos, hogy gyorsabban hajtja végre a teszteket, mint a manuális tesztelő.

Kezdetben költséges.

24/7 – ben lehet teszteket végrehajtani.

SZÉCHENYI 2020

Célja:

Felmérni/felderíteni azokat a tesztelési feladatokat, folyamatokat, amiket a szokásos regressziós tesztelés során hajtunk végre, ezáltal manuális erőforrásokat szabadítunk fel. Előfordulhat, hogy olyan feladatokat hajtunk végre, amik sokkal gyorsabban végezhetőek el gép által, mint manuálisan (teszt előkészítések, tesztadat generálás, eredmény validálás, nagy mennyiségű adat feldolgoztatása, stb.). Automatizált szoftverteszteléssel nagymértékben támogatjuk a tesztelési folyamatokat. Folyamatosan törekszünk a hatékonyság növelésre, ami megmutatkozhat teszt végrehajtási idők csökkentésében, manuális tesztelési erőforrások felszabadításában.

Érvek az automatizált szoftvertesztelés mellett

Számos olyan dolog van, amik indokolják az automatikus tesztek használatát. A következő lista azokat az érveket sorakoztatja fel, amik a tesztautomatizálás mellett szólnak:

- Hatékonyság növelése
- Tesztelési költségek csökkentése
- Erőforrás csökkentése
- Manuális tesztelési hibák megszüntetése
- Unalmas feladatok végrehajtása
- Pontos eredmények
- Bármikori végrehajthatóság
- Újrahasznosíthatóság

Miért nem egyszerű az automatizált szoftvertesztelés?

Speciális szoftver és hardver szükségeltetik a tesztek elkészítésére és futtatására. Elengedhetetlen, hogy a tesztautomatizálással foglalkozó tesztelőknek valamilyen szinten legyen programozási jártasságuk, hiszen elkerülhetetlen, hogy a különböző modulok, egységek, funkciók megfelelően fel legyenek mérve és implementálva. Fontos szem előtt tartani, hogy nem váltja ki teljesen a manuális tesztelést, viszont lehet olyan részeket automatizálni, ami a manuális tesztelési folyamatot helyettesíti. Kezdetben nagy hangsúlyt fektetünk a tesztek kiválasztására és implementálására, ami költséggel jár. Ennek a megtérüléséhez idő kell.

SZÉCHENYI 2020

Mikor nem javasolt a tesztautomatizálás használata?

Olyan esetekben, amikor nincsen a tesztelési folyamat dokumentálva, nehéz lehet pontosan definiálni, milyen lépéseket kell tartalmaznia egy tesztnek. Nem lehetünk biztosak benne, hogy a megfelelő folyamat lesz implementálva, ezáltal a pontos folyamat tesztelve. Gyakori funkcionalitás (alkalmazásban történt kódváltoztatás) változás okozhat olyan működést, amire nem lett az automatikus teszt felkészítve. Ebben az esetben elég gyakran kellene újra aktualizálni tesztjeinket. Abban az esetben is találkozhatunk ilyen gondokkal, ha az alkalmazás funkcionalitása nem stabil. Szükséges vizsgálnunk, hogy mennyi idő és erőforrás szükségeltetik a teszt elkészüléséhez, valamint hogy mennyi lehet a megtérülés mértéke. Abban az esetben, ha minimálisnak véljük az automatizálás hatékonyságát, megtérülését, nem érdemes elkezdeni implementálni. Nem indokolt az automatizálás elkezdése akkor sem, ha kevés időnk van a teszt implementálására.

Automatizált szoftvertesztelés – Generációk

Az elmúlt években hatalmas léptékben fejlődtek a tesztautomatizálást támogató eszközök. Kereskedelmi forgalomban számos cég kínál nagyon jó megoldásokat, viszont létezik pár ingyenes, nyílt forráskódú jól használható alkalmazás is.

Érdemes kiemelni három "generációt".

Lineáris automatizálás

Szoftverteszteléssel foglalkozó csoportok, szervezetek előbb utóbb felismerik, hogy bizonyos teszteket célszerű lenne gép által végrehajtatni, hiszen minden egyes tesztelési ciklusban ugyanazokat a lépéseket hajtjuk végre. Legegyszerűbbnek az a megoldás kínálkozik, hogy amit a manuális tesztelő végrehajt, azt egy az egyben rögzítsük valahogy, mint egy makrót, és a gép ezt majd végrehajtja helyettünk. Ez a fajta automatizálás egyszerű, gyors, nem kíván nagy fokú programozási jártasságot, viszonylag gyorsan megtanulható és egy az egyben megfeleltethető a manuális tesztnek. Hátránya sajnos, hogy nem tartjuk szem előtt a fenntarthatóságot, nem lehet paraméterezni, valamint a hibakezelés is nehézkesen megoldható, ha egyáltalán lehetséges. Éppen ezért a tesztek futását felügyelni kell, ami nem túl rugalmas és kényelmes megoldás.

Képzeljük el, hogy van egy webáruház, ahol könyveket lehet rendelni. A weboldalunk teszteléséhez felhasználói fiókokat kell létrehoznunk különböző címekkel, valamint a különböző felhasználói fiókokhoz különböző rendeléseket, tranzakciókat kell végrehajtani. Tegyük fel, hogy minden egyes tesztelési ciklusban végre kell hajtanunk ezeket a teszteket. A teszttervezés során ötven darab tesztesetet

definiáltunk. A tesztek alkalmasak automatizálásra, hiszen minden egyes tesztnél majdnem ugyanazokat a lépéseket kell végrehajtani: bejelentkezés, könyv(ek) kiválasztása, rendelés feladása, kijelentkezés.

Lineáris automatizálás használatával ötvenszer kell rögzítenünk ezeket a teszteket, hiszen az automatikus teszt egy az egyben megfeleltethető a hozzá tartozó manuális tesztnek. A tesztek futtatása során egyértelműen meg tudjuk mondani, hogy melyik teszt futott hibára, milyen adattal. A probléma ott kezdődik, ha megváltozik a rögzített folyamatok bármelyik része. Tegyük fel, hogy a fejlesztők a bejelentkezési felületen egy új, kötelezően kitöltendő mezőt terveznek implementálni. Ebben az esetben az összes automatizált tesztben aktualizálni kell a bejelentkezési lépéseket. Ez elég időigényes, sok idő megy el minden egyes teszt frissítésére.

A lineáris automatizálás akkor használható jól, ha gyorsan, egyszerűen el tudjuk készíteni a teszteket, a karbantarthatóságra továbbfejlesztésre pedig nem, vagy kevésbé figyelünk oda.

Példa:

A következő példában egy bejelentkezési folyamatot fogunk leszkriptelni. A felületen egy Username és egy Password szövegmező, valamint egy Login gomb van megjelenítve. Miután a "Name" és "Pass" sztringet beíratjuk a megfelelő szövegmezőbe rákattintunk a Login feliratú gombra. Ezután egy ellenőrzés történik, hogy a "Hello Name" szöveg létezik vagy sem a felhasználói felületen. A feltétel teljesülése esetén a Testünk sikeres, ellenkező esetben hibás.

```
Input “Name" into Username textbox
Input “Pass" into Password textbox
Click Login button
If “Hello Name" exists then
      Test Passed
Else
      Test Failed
End If
```

Előnyök:

- Gyors fejlesztés
- Rövid eszköz betanulási idő

SZÉCHENYI 2020

- Információhoz juthatunk a tesztelt alkalmazás felépítéséről, amennyiben létezik olyan funkciója az automatizáló eszközünknek, amivel rögzíteni tudjuk a felhasználói felületen végzett lépéseinket
- Független szkriptek
- Egyszerű hibadetektálás
- Kevés tervezés szükséges

Hátrányok:

- Futtatás néha helytelen
- Egy dimenziós szkript
- Nehéz olvashatóság
- Szükséges a tesztelt alkalmazás funkcionalitásának mély, alapos ismerete
- Újrafelhasználhatóság hiánya
- Paraméterezés hiánya
- Költséges fenntarthatóság

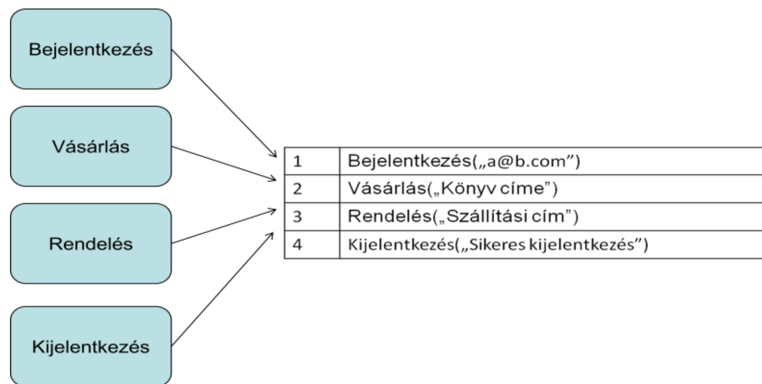
Adatvezérelt automatizálás

A következő generáció/szint az adatvezérelt automatizálás. Tapasztalat alapján ez az a szint, amit nagyon jól lehet karbantartani, befektetés/megtérülés szem előtt tartása mellett nagyban hozzájárul az automatizálás sikerességéhez. Lineáris automatizáláson alapul, viszont itt már előtérbe kerül az egyes funkciók modularitása, a tesztadatok külön állományban való tárolása. A különböző, egymástól független modulok paraméterezhetőek, így a tesztadatok nincsenek "hard" kódólva s tesztekben, külön tárolhatóak és ezáltal külön lehet őket karbantartani.

Vizsgáljuk meg az előző esetet és nézzük meg, hogy lehet a webáruházat adatvezérelt módon automatizálni.

Elsőként próbáljuk meg beazonosítani, hogy melyek azok az egységek amik külön funkcióval bírnak. Ezek alapján készítsünk egy bejelentkezés modult, egy vásárlás modult, egy rendelés modult, valamint egy kijelentkezés modult. A manuális teszteseteinknél használt tesztadatokat tároljuk el egy külön állományban, mondjuk egy Excel dokumentumban. A különböző modulokban használjunk paramétereket, amik az Excel dokumentumban szereplő adatokra hivatkoznak. Így elértük, hogy nem

fixen szerepelnek a tesztadataink az automatikus tesztekben, hanem mint paraméterek. Ha különböző tesztadatokat szeretnénk használni a végrehajtás során elég csak a tesztadatokat tartalmazó dokumentumot aktualizálnunk. A különböző modulok így ugyanazokat a lépéseket fogják végrehajtani minden egyes tesztadatra.



Paraméterezés:

```
Open Data Table

Input <NameParameter> into Username textbox

Input <PassParameter> into Password textbox

Click Login button

If "Hello <NameParameter>" exists then

      Test Passed

Else

      Test Failed

End If

Close Data Table
```

NameParameter	PassParameter
Eszter	PassEszter
Balázs	PassBalázs
Péter	PassPéter

Abban az esetben, ha a fejlesztők egy új funkciót terveznek implementálni valamelyik részén az alkalmazásnak, elég, ha csak a hozzá tartozó modulban aktualizáljuk a lépéseket. A teszt végrehajtás során csak a modulokat hívjuk meg, így nem kell minden egyes tesztnél javítanunk a tesztjeinket.

Előnyök:

- Újrafelhasználhatóság
- Korai szkript készítés
- Könnyű olvashatóság
- Standardizálás
- Hibakezelés használata

Hátrányok:

- Technikai szakemberekre van szükség
- Komplexebb fenntarthatóság
- Alapos tervezést igényel

Kulcsszó vezérelt automatizálás

A tesztautomatizálás talán legkomplexebb típusa a kulcsszó vezérelt automatizálás. Tartalmazza az adatvezérelt automatizálás minden tulajdonságát (modularitás, paraméterezhetőség, hibakezelés stb.) viszont itt az elkészült kódrészletek nem a szkript törzsét képezik, hanem kulcsszavaknak feleltetjük meg a funkciókat. Ezt a módszert szokták táblavezérelt automatizálásnak is hívni. A különböző egységeket kulcsszavakhoz rendeljük, majd egy táblában rögzítjük, hogy milyen lépéseket szeretnénk végrehajtani, milyen adattal. Ennél a módszernél a befektetett idő a legtöbb. Amennyiben ilyen automatizálási módszert választunk, számolnunk kell azzal, hogy nagyon sok időt kell fordítanunk a különböző modulok kidolgozására. Ez a fajta automatizálás a leginkább alkalmazás független. Jól megtervezett, megfelelő munkafolyamatok mentén felügyelt automatizálás során el tudjuk érni azt, hogy a tesztelők, üzleti elemzők már a tesztelési ciklusok előtt képesek összeállítani a szükséges adattáblát a kulcsszavakkal és adatokkal. A kulcsszavak és a hozzájuk társított funkcionalitások dokumentálása elengedhetetlen. Az automatizálás során nagy mennyiségű kulcsszó lista, kulcsszó "szótár" fog létrejönni, amiből fel tudjuk építeni a tesztelési folyamatot.

A kulcsszavak (Action) alkalmazásspecifikus és alkalmazásfüggetlen funkcionalitással bíró függvényekkel vagy szkriptekkel vannak kapcsolatban

A korábban említett webáruházunk belépés részének tesztelésére próbáljunk meg kulcsszó vezérelt tesztet létrehozni.

Legyen a tesztlépésünk a következő: Miután beírjuk a felhasználónevünket és a jelszavunkat, kattintsunk a bejelentkezés gombra és ellenőrizzük le, hogy a képernyőn megjelent-e a belépés sikerességét jelző szöveg.

A táblában tároljuk, hogy a Belépés képernyőn szeretnénk a korábban lekódolt funkciókat végrehajtani.

Képernyő	Objektum	Action	Érték	Recovery	Megjegyzés
Belépés	Username	Beírás	"Balázs"		
Belépés	Password	Beírás	"PassBalázs"		
Belépés	Login	Kattintás			
Helló		Ellenőrzés		Stop_Test	

```
Open Keyword File
Execute Data rows to end of file (EOF)
  If <Action> == Beírás Then
      Call Beírás(Screen, Object, Value)
If <Action> == Kattintás Then
      Call Kattintás(Screen, Object)
 If <Action> == Ellenőrzés Then
       Execute Ellenőrzés(Screen)
Implement Exception Handling Routine Based on Pass/FailStatus
End Loop
```

Előnyök:

- Újrafelhasználhatóság
- Korai szkript készítés
- Könnyű olvashatóság

- Standardizálás
- Hibakezelés használata
- Más tesztelők is tudnak teszteseteket készíteni

Hátrányok:

- Technikai szakemberekre van szükség
- Komplexebb fenntarthatóság
- Alapos tervezést igényel
- Felsővezetői támogatás szükséges (költség, idő, erőforrás)

Összegzés

Nem feltétlenül lehet azt mondani, hogy egyik fajta módszer jobb mint a másik. Mindegyiknek megvan a maga előnye/hátránya. Az biztos, ha tesztautomatizálással szeretnénk foglalkozni, akkor elengedhetetlen, hogy felmérjük mire, és hogyan szeretnénk használni. Korábban említettük, hogy léteznek kereskedelmi forgalomban kapható eszközök, valamint ingyenes nyílt forráskódú alkalmazások is. A megfelelő eszköz kiválasztásához nélkülözhetetlen egy kísérletet tennünk az eszközökkel. Fel kell mérnünk, hogy melyik megoldás az, ami a legjobban illeszkedik a meglévő üzleti/tesztelési folyamatainkra.

Nem szabad figyelmen kívül hagyni azt, hogy hosszú távon elengedhetetlen, hogy olyan szakemberek foglalkozzanak tesztautomatizálással, akiknek technikai tudásuk van, megfelelő tapasztalattal rendelkeznek a különböző automatizálási lehetőségek terén, képesek felmérni, eldönteni, hogy milyen teszteket érdemes automatikus tesztre átalakítani, valamint hogy olyan standardokat fogalmazzanak meg, amik mentén a tesztautomatizálás sikeres és működőképes tud lenni.

Ezek után jogosan merülhet fel a kérdés: A tesztautomatizálás alkalmazásfejlesztés?

A válasz: Igen. Megfelelő automatikus tesztek implementálása előtt szükséges a megfelelő design, és tervezés. A kódolás standardok mentén kell, hogy történjen, a dokumentálás nélkülözhetetlen, a karbantarthatóság szükséges ahhoz, hogy időről időre használhatóak legyenek a tesztjeink. Az elkészült teszteket tesztelni kell, hogy azt hajtják-e végre, amit szeretnénk, úgy viselkednek-e, ahogy elvárjuk tőlük.

Eszközválasztás

Jogosan vetődhet fel a kérdés, hogy szükséges tesztautomatizálásra eszközt használnunk? Nem egyszerűbb, ha elkezdünk magunk programkódokat készíteni és azok mentén felépíteni a tesztelést? Hiszen a tesztelendő alkalmazás tulajdonságait, viselkedését, működését jobb esetben ismerjük, így egy általunk készített teszt bármilyen programnyelvben adottnak tűnik. Igazából ez nem mindig van így. Léteznek kereskedelmi forgalomban kapható automatizált tesztelést támogató eszközök. Az árukat tekintve is igen nagy eltéréseket tapasztalhatunk. Meg kell vizsgálnunk, hogy egy eszközbe fektetett pénz és energia, hosszútávon megéri-e a tesztelőknek, illetve a vállalatnak, cégnek. A bekerülési költség magas tud lenni egy megvásárolandó eszköznél, de figyeljük meg, hogy mi minden egyéb pozitívum is jelenik meg. Az eszköznek lehet van egy saját programozási nyelve, esetleg általunk ismert valamely programozási nyelvet támogatja. Elképzelhető, hogy számos olyan funkcionalitást tartalmaz, amit nagy időbefektetéssel lehetne egy tesztelőnek lefejlesztenie, esetleg megterveznie. Támogatást kapunk, ami azzal jár, hogy bármikor rendelkezésünkre tudnak állni és segítséget kaphatunk a gyártó cég szakembereitől.

Elképzelhető, hogy az igényeinknek egy nyílt forráskódú automatizáló eszköz is tökéletesen megfelel. Ebben az esetben is, és a fenti esetben is érdemes betartani és megvizsgálni az alábbiakat:

Mindenképpen fontos, hogy eszközválasztás előtt szánjunk időt a potenciális eszközök kipróbálására. Szerezzünk be minél több információt az eszközről, próbaverziót telepítsünk és próbáljuk meg megismerni, hogy milyen módszer szerint működik az eszköz. Mérjük fel, hogy támogatja-e azokat az alkalmazásokat, amiket tesztelni szeretnénk. Elérhetőek interneten nyilvános fórumok, dokumentumok, közösségek, akik ugyanazt az eszközt használják.

Nem szabad türelmetlennek lennünk. A tesztautomatizálás egy igen komplex ugyanakkor kihívásokkal teli terület. Egy rosszul kiválasztott eszköz, a rosszul felépített folyamatok kockáztatják az automatizálás sikerességét.

Vizsgáljunk meg egy konkrét esetet tesz automatizálás bevezetésére. Az eszköz kiválasztása nem ideális módon történt. Megvásároltunk egy eszközt, amiről nem tudtuk, hogy hogyan és miként kellene megfelelően használni. Nem volt ismertek az eszközben lévő lehetőségek. Miután elkezdtünk ismerkedni az eszközzel, akkor vált világossá, hogy szerencsére a lehető legjobb kereskedelmi forgalomban lévő, piacvezető terméket használjuk. Annyit tudtunk, hogy jó az eszköz, de hogy miként akarjuk használni az nem volt egyértelmű, és sajnos a vezetők sem volt tisztában teljesen a benne rejlő lehetőségekben.

Kezdetben a hagyományos tesztelési ciklusok során próbálkoztunk meg automatizálni, majd láttuk, hogy ha nem fordítunk elég figyelmet és erőforrást a feltérképezésre, akkor nem tudunk minőségben előrehaladni. Indítottunk egy egy éves próba projektet, aminek az volt a célja, hogy felmérjük hogyan alkalmazható a tesztautomatizálás a rendelések rendszerben való rögzítésére. Miért a rendelések rögzítését választottuk? Felmértük, hogy a tesztelési folyamatainkban hol vannak kritikus területek. Azonosítottunk három kritikus területet, viszont a rendelések felvitelénél azt tapasztaltuk, hogy más tesztestekhez, adatgenerálás miatt is kell rendeléseket létrehozni a rendszerben. Ez egy megfelelő indok volt. Miért manuális tesztelői erőforrást használjunk, mikor ugyanazokat a lépéseket egy programmal is meg tudjuk csinálni?

Az első próbálkozások során a fentebb említett lineáris automatizálást használtuk, viszont a meglévő eszköz támogatta a paraméterezhetőséget, így tudtunk Excel dokumentumokból tesztadatokat használni.

Ahogy telt az idő, úgy tapasztaltuk, hogy valamilyen szinten érdemes lenne modularizálni a különböző képernyők szerint a tesztjeinket. Elkezdtünk adatvezérelt automatizálás mentén dolgozni. Kisebb modulokat hoztunk létre, függvényekben tároltuk el a lépéseket, és így könnyebben tudtuk karbantartani a tesztjeinket.

Szerencsére az eszköznek volt egy jól használható riportálási modulja. Ennek köszönhetően könnyedén tudtuk detektálni, hogy mikor és mi miatt futott hibára a tesztünk.

A próba projekt leteltével sikerült egy kerek képet kapnunk arról, hogy a meglévő eszköz tökéletesen használható. Azt tudja, amire nekünk szükségünk volt. Sikerült megtapasztalnunk, hogy teszteket elkészíteni nem csak annyiból áll, hogy rögzítjük a manuális lépéseket. Teljesen más gondolkozásmód szükséges, hogy felépítsük, és felkészítsük a tesztjeinket, akár nem várt eseményekre, vagy például arra, hogy különböző adatokkal hogy bánjon.

Meggyőződtünk arról, hogy sikeresen lehetett erőforrást csökkenteni (manuális tesztelést váltottunk ki vele), ugyanakkor bebizonyosodott számunkra az is, hogy jól definiált folyamatok nélkül nem lehet hatékonyan automatikus teszteket létrehozni.

SZÉCHENYI 2020

Eszköz tulajdonságok

Eszközválasztás során tartsuk szem előtt, hogy rendelkezik-e az eszköz az alábbi lehetőségekkel, tulajdonságokkal:

Felvétel és visszajátszás

Nagyban tudja segíteni a fejlesztő csapat munkáját, ha lehetőség van a tesztlépések rögzítésére, ezáltal már egy automatikus váz létre tud jönni. Az eszköz pontosan azokat lépéseket rögzíti, amit a felhasználó végez a tesztelendő alkalmazáson. Ilyen például a különböző mezőkbe történő értékadások, adat begépelések, gombokra, linkekre történő kattintás. Az eszköz ezt rögzíti mint egy makrót, és már vissza is játszható, futtatható! Ez természetesen nem azt jelenti, hogy el is készült a tesztünk, hiszen ezzel nem készítettünk modulokat, nem paramétereztünk, nem történik hibakezelés, eredményvizsgálat.

Objektumok tárolása

A karbantarthatóság szempontjából nagyon jó, ha az eszközünk tud objektumokat tárolni. Ez nem más, mint eltárolni azokat az objektumokat és a hozzájuk tartozó tulajdonságokat, amiket korábban érintettünk a felvétel során. Így ha később módosítás történik egy objektumon a felhasználói felületen, akkor elég, ha a tesztünkben csak az objektumok frissítjük, a tesztünket végre tudjuk hajtani. Futás során az eszköz az objektumokat próbálja beazonosítani a felületen, úgy tudja végrehajtani az adott lépést.

Naplózás

Fontos lehet a tesztfuttatás során a végrehajtott lépéseket külön egy napló állományban rögzíteni. A tesztfutások után egyszerűbb a kiértékelés, ha megfelelő információk vannak benne.

Be és kimenő adatok tárolása

Mint azt korábban említettük, nagyban segíti a munkánkat, ha a tesztfuttatások során egy külön állományba tárolni tudjuk a tesztadatokat. Ezzel elkerüljük azt, hogy külön írt függvényekkel próbálunk egy dokumentumot feldolgozni a tesztadatok használatához. Nagyon hasznos, ha ezt a dokumentumot nemcsak adatbevitelre használjuk, hanem különböző értékeket ki is tudunk benne menteni. Például ha létrehozunk egy új felhasználót a felületen, majd az alkalmazásunk egy egyedi azonosítót állít elő ehhez a felhasználóhoz, akkor azt az értéket ki tudjuk menteni a felületről és beírni a dokumentumba.

Futási idők mérése

Tesztfuttatások után, az eredmények kiértékelésekor, ha például sok hibás eredményünk van, akkor segíthet, ha futási időket is rögzítünk. Ekkor kiderülhet, hogy a tesztelt alkalmazás teljesítménye nem megfelelő, nem tudott a tesztünk szinkronizálva végrehajtódni, mert lassú volt az alkalmazás. Futási idők rögzítésével azt is meg tudjuk állapítani, hogy a manuális teszthez képest milyen gyors vagy lassú volt a tesztvégrehajtás.

Szinkronizálás a tesztelt alkalmazással

Egy automatikus teszt sorról sorra hajtódik végre. Általában a gép gyorsabban hajtja végre a teszteket mint az ember, ezért jó ha képes az eszközünk szinkronizáltan teszt lépéseket végrehajtani az alkalmazáson. Ezzel elkerüljük azt, hogy bizonyos lépések gyorsabban hajtódnának végre, mint ahogy indokolt lenne. Elcsúszik a teszt az alkalmazás felett.

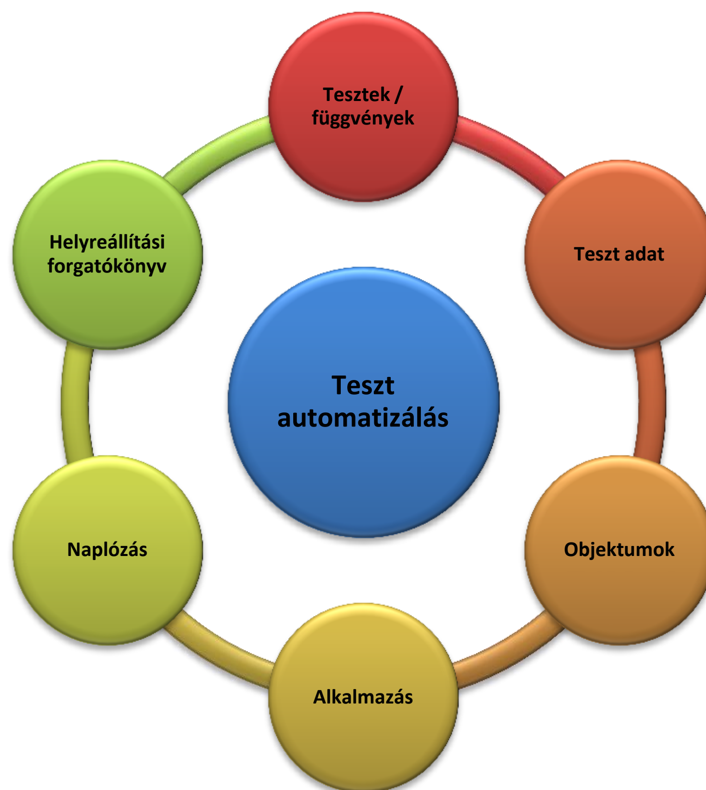
Kapcsolat más eszközzel

Egyre több olyan tesztautomatizálást támogató eszköz létezik, amelyik képes más tesztelést támogató eszközzel kommunikálni. Képzeljük el, hogy van egy tesztmenedzsment eszközünk is. Ebben az eszközben tároljuk a manuális és automatikus tesztjeinket, tesztadatainkat. Tesztelési ciklusokban a tesztmenedzsment eszközben hajtjuk végre a teszteket, így például képesek vagyunk meghívni egy automatikus tesztet és az automatikus teszteszközünk képes a tesztmenedzsment eszközbe visszariportálni az eredményt. Ha létezik egy inicidenskezelő rendszerünk, akár az automatikus teszt lefutása után egyből tudunk hibákat riportálni ebbe a rendszerbe.

Helyreállítási forgatókönyv

Tesztek létrehozása során gondolnunk kell arra is, hogy a tesztünk nem tud végrehajtódni bizonyos nem várt események miatt. A tesztek futhatnak éjjel is, így ha az alkalmazásunk valamilyen oknál fogva nem elérhető, akkor a tesztünk ezt automatikusan nem tudja detektálni. Vagy ha olyan tesztadatot használunk, ami egy olyan nem várt ablakot dob fel, amit nem kezelünk a tesztünkben, akkor a futás elakad. Ezekben az esetekben nagy segítség, ha a teszt automatizált eszközünk egy helyreállító funkcióval bír. A tesztelő csapat különböző akciókat tud megadni ilyen esetekben. Például: álljon le a tesztfutás, minden ablakot zárjunk be és lépjünk be újra, majd folytassuk a tesztvégrehajtást a következő iterációtól, indítsuk újra a gépet. Ezeket az eseteket hívjuk helyreállítási forgatókönyvnek, amikor előre nem ismert, váratlan esetek fordulnak elő.

SZÉCHENYI 2020



Tesztautomatizálás bevezetése

Elsőként talán a legfontosabb kérdés az, hogy készen állunk-e a változásra? Amennyiben tesztautomatizálást szeretnénk használni, bevezetni, úgy elkerülhetetlen, hogy a meglévő folyamatainkon változtassunk. Automatikus tesztek használatával a tesztelési folyamatok módosulhatnak, végrehajtásuk akár eltérő lehet, mint korábban. Tesztek automatizálása, elkészítése más folyamatokon keresztül történik, mint egy manuális tesztelés, így ezeket az új folyamatokat be kell tudnunk építeni a meglévő folyamataink közé.

Fel kell deríteni, hogy mik azok a területek ahol célszerű, és érdemes automatikus teszteket létrehozni. Hol tudunk manuális teszteléssel foglalkozó erőforrásokat felszabadítani annak érdekében, hogy a tesztelők más, érdekesebb tesztelési feladatokat lássanak el. Sok, ugyanolyan lépést tartalmazó tesztek nagyon jó jelöltek automatizálásra. Ezekben az esetekben akár kockázat is felmerülhet, hiszen a tesztelők elfáradhatnak, lassabban tudnak teszteket futtatni. A gép által végrehajtott tesztek esetében nem tudunk hibázni. Pontosan azokat a lépéseket hajtja végre, amiket kell, és ugyan olyan gyorsan századszor is.

Tartsuk szem előtt azt is, hogy egy meglévő teszt akár adatgenerálásra is használható más egyéb teszteléseknél. Ebben az esetben is nélkülözni tudunk manuális erőforrást, hiszen adatgenerálás során is ugyanolyan lépéseket használunk, mint egyéb manuális tesztelés során.

Célszerű, ha a tesztautomatizálási csapatnak saját folyamataik vannak. A csapat tudja megmondani, hogy mi az a munkamenet, ami alapján hatékonyan tudnak teszteket implementálni. Standardok felépítése szükséges (függvények, változók elnevezése, tesztek tárolása, verziókövetés, adatok tárolása, eredmények tárolása, tesztfuttatások menedzselése). Folyamatosan szem előtt kell tartani a befektetés/megtérülés vizsgálatát. Tudni kell jól becsülni az időket, kiértékelni a futási időket és hatékonyságot. Amennyiben egy tesztnek a fenntarthatósága nem éri meg, úgy nem érdemes a tesztet fenntartani.

Amennyiben olyan eszközt használunk, aminek korlátos az elérhetősége (pár licenszt vásároltunk meg) akkor tudni kell menedzselni ezeket. Vizsgálni kell, hogy mikor mire használjuk az eszközt. Tesztek készítésére, vagy teszt végrehajtásra. A különböző üzleti területek eltérő priorizációval bírnak, így tudni kell a tesztek végrehajtását ütemezni.

Be kell látni, hogy a tesztautomatizálás nem egyszerű probléma. Megértéséhez, használatához a környezetet (üzleti elemzők, programozók stb.) folyamatosan képezni kell. Mivel nem specialistái a területnek, így tudatosítani kell, hogy mikor, mire lehet használni. Meg kell értetni azt, hogy nem lehet irreális elvárásokat támasztani az eszközökkel, valamint a megoldásokkal szemben. Sokszor tekintik az automatikus teszteket egyfajta varázslatnak. Komplexitása miatt sokszor azt feltételezi a környezet, hogy mindent képes megoldani, és gyakran találkozhatunk olyannal is, amikor azt feltételezik, hogy akár mást is csinál, mint amit "megmondtunk" neki. Ezek elkerülése végett szükséges időszakosan tanítani a környezetet.

Nem feledkezhetünk meg arról sem, hogy a tesztautomatizálás bizonyos költségekkel jár. Szükségünk van hardver és szoftver erőforrásra. Nem mindegy, hogy a kiválasztott automatizálási eszköz milyen hardveren használjuk. A tesztek készítési idejére kihatással lehet az, hogy milyen gyors processzorunk van, milyen perifériáink stb. Ugyanez a helyzet a tesztek végrehajtásánál. Egy gyenge konfiguráció esetében fenn áll a veszélye, hogy különböző szinkronizációs hibákba ütközünk, ezáltal a teszt eredményeink nem feltétlenül a megfelelő eredményeket mutatják.

Előfordulhat, hogy különböző okok miatt, az automatikus tesztek fejlesztése nem az általában használt teszt környezetben történik. Tisztában kell lenni azzal, hogy milyen különbségek vannak és ezek milyen hatással lehetnek az automatikus tesztek futására.

Az alábbi párbeszédek egy tesztelő és a menedzsere között zajlanak le. Figyeljük meg, hogy miként kezd az automatizálás megjelenni, és az érdeklődést felkelteni mind a tesztelőben, mind a menedzserben.

Menedzser: Szia. Láttam, hogy szeretnél velem beszélni az automatizált szoftvertesztelésről.

Tesztelő: Szia. Igen. Tudod van az az eszköz amit a cég 2 éve vett meg. Igazából nagyon senki sem használja. Lehetséges, hogy amennyiben az időm engedi belepillantsak, hogyan működik az eszköz?

Menedzser: Igen, tudom hogy létezik az eszköz. Kettő kollégánk foglalkozik a megismerésével. Esetleg ha érdekel, kérdezd meg őket, hogy mit tudunk megcsinálni vele.

Tesztelő: Tudom, beszéltem is velük. Érdeklődtem, hogy mikor használjuk a teszteket. Véleményem szerint nem igazán haladnak jó úton a folyamattal.

Menedzser: Hogy érted ezt?

Tesztelő: Igazából nem tudjuk, hogy mire tudjuk használni teljesen az eszközt. Nekem van olyan érzésem, hogy ennél sokkal hatékonyabban is tudnánk teszteket létrehozni és használni is.

Menedzser: Sajnos a nagy projektek miatt nem nagyon van időnk, hogy másnak is lehetőséget nyújtsunk ezzel foglalkozni.

Tesztelő: Tudom, meg is értem, de nekem 20%-ban van szabad kapacitásom. Gondoltam, ha időm engedi, belemélyedek ebbe a területbe.

Menedzser: Amennyiben nem fogja hátráltatni a szokásos teljesítményedet akkor nyugodtan. Kérlek tartsd szem előtt, hogy a projektek sikeressége most kritikus.

Tesztelő: Természetesen. Köszönöm. A projekt végeztével tudunk beszélni ismét? Addigra lehet tudok valamilyen információval szolgálni.

Menedzser: Rendben.

Tesztelő: Köszönöm

Menedzser: Én köszönöm

Pár hónappal később a következő párbeszéd zajlódott le:

Menedzser: Szia. Mire jutottál az automatikus teszteléssel kapcsolatban,

Tesztelő: Szia. Örülök, hogy újra tudunk erről beszélni. Nos, sikerült kipróbálnom az eszközt és nagyon sok hasznos információt is találtam róla.

Menedzser: Meg tudsz osztani pár fontos infót?

Tesztelő: Természetesen. Elsőként, azt tapasztaltam, hogy a vállalatirányítási rendszerünket tökéletesen lehet vele tesztelni. Viszonylag könnyen lehet alap szkripteket létrehozni és egy külön adattáblából tudunk teszt adatokat beadni a futtatás során. Szóval lehet paraméterezni, hogy könnyen tudjunk különböző adatokra tesztelni.

Menedzser: Jól hangzik. Mi a helyzet a WEB-es alkalmazásainkkal?

Tesztelő: Sikeresen tudtam szkriptet rögzíteni vele. Picit trükkösebb használni a weben, de szerintem meg tudjuk oldani, hogy ott is működjön.

Menedzser: Nagyszerű, akkor a következő tesztelési ciklusban már tudjuk is használni pár tesztelésre?

Tesztelő: Véleményem szerint még lehet, hogy több tapasztalatot kellene szereznünk magáról az eszközről. Nem volt annyi időm foglalkozni ezzel amennyit szerettem volna.

Menedzser: Értelek. Látom, hogy érdekel a tesztautomatizálás, és jó eredményeket is értél el. Szerintem indítsunk el egy Pilot projektet a tesztautomatizálásra. Dokumentáljuk le, hogy mit szeretnénk elérni az elkövetkező 9 hónapban. Pont lesz egy tesztelési ciklusunk is, abban esetleg már ki is lehet próbálni.

Tesztelő: Jól hangzik.

Menedzser: Legközelebbi alkalommal tárgyaljuk meg, hogy mit szeretnénk elérni, és mi alapján döntünk úgy, hogy a tesztautomatizálást alkalmazzuk vagy sem. Kérlek gyűjtsd össze, hogy mire lesz szükség.

Tesztelő: Rendben, kidolgozzuk, hogy mi alapján vizsgáljuk a folyamatot.

Körülbelül három – négy hét elteltével egy újabb beszélgetés zajlódott le:

Tesztelő: Szia, örülök hogy újra tudunk beszélni a következő tervekről.

SZÉCHENYI 2020

<u>Menedzser:</u> Üdvözöllek. Kíváncsi vagyok, milyen fejleményekkel tudsz szolgálni.

<u>Tesztelő:</u> Az elmúlt időszakban sikerült átolvasnom és megfelelően belelátnom még jobban az automatizálásra használt eszközünkre. A vállalatirányítási rendszerünkben tényleg nagyszerűen lehet használni a felvétel opciót. Egy az egyben rögzíteni képes azokat a lépéseket, amiket a felhasználói felületen véghezviszek. Nem mondom, hogy valamikor nem okozott fennakadást pár objektum felismertetése, de sikerült. A felvétel funkció nagy segítség, hiszen nem kell minden egyes lépést kézzel lekódolnunk. Miután a nyers váza megvan a tesztnek, utána lehetséges egyéb lépéseket, ciklusokat, feltételeket hozzáadni, ezáltal a különböző igényeinkre szabni a teszteket.

<u>Menedzser:</u> Hm. Értem, érdekesen hangzik. Akkor gondolom, hogy ez az egyben azokat a lépéseket hajtja végre amiket felvettél?

<u>Tesztelő:</u> Igen. Akár már az első felvétel után vissza lehet játszani.

<u>Menedzser:</u> Lenne egy kérdésem. Gyakran fordul elő, és tudom hogy a jelenlegi teszt eseteink között is van sok hasonló, csak más adatokkal hajtjuk végre ugyan azokat a lépéseket. Akkor minden egyes teszt adatra külön scriptet kell készíteni? Le kell másolnunk és más adattal kitölteni ezeket a lépéseket?

<u>Tesztelő:</u> Szerencsére nem. Értem a kérdésedet és gondoltam én is erre. Szerencsére az eszközünk nagyon jól paraméterezhető. Miután rögzítünk egy teszt esetet, igazából belefejlesztjük a teszt adatot is. Ez jó addig, amíg kész nem lesz egy teszt. Ezután lehetőségünk van beparaméterezni a tesztet. Akár futtatás előtt mi is meg tudunk adni egy fix értéket, vagy egy sima Excel adattáblában tudjuk eltárolni az adatokat. Minden egyes sor az Excel állományban egy teszt iterációt jelent. Szóval az eszköz annyiszor hajtja végre a tesztet amennyi teszt adatunk van.

<u>Menedzser:</u> Ez jó hír. Mi a helyzet az eredmények kiértékelésével?

<u>Tesztelő:</u> Igen pont erre akartam kitérni. Nagyon jó beépített riportálási felülete van az eszköznek. Minden egyes iterációról kapunk egy tiszta képet, hogy milyen lépés futott le, és milyen eredménnyel. Lehetőségünk van kézzel is beilleszteni pár extra riportálási lépést.

<u>Menedzser:</u> Ez jól hangzik. Ezek alapján meg kellene próbálni használni az eszközt valamilyen stratégia alapján. Azt látom, hogy ez nem kevés befektetéssel jár. Idő kell, hogy felmérjük mit lehet automatizálni, és a tesztek elkészítéséhez is nem egy egyszerű felvétel szükséges.

Tesztelő: Jól látod igen. Véleményem szerint egy olyan területen kellene elkezdeni felmérni az igényeket és a teszteket ahol nagy időmegtakarítást tudunk elérni.

Menedzser: A vállalatirányítási rendszerünket nézve a kritikus folyamatot kellene megvizsgálni. Megrendelések, kiszállítás és számlázás. Ha a területeken működő folyamatok nem működnek akkor az nagy gondot tud okozni. Jelen pillanatban a rendelések felvitele az ahol nagy terhelés van. Minden egyes tesztelési ciklusban a regressziós tesztjeink száma igen nagy, sok időbe telik kézzel az összes rendelést felvinni. Holott a rendelések felvitelének módja majdnem minden egyes esetben ugyan az. Volt arra is példa, hogy egy bizonyos project még további rendelésekre is igényt tartott. Úgy gondolom, hogyha ezen a területen tudnánk elérni jó eredményeket, akkor sok esetet tudnánk automatikusan lefedni. Azt javaslom, hogy a rendelések rögzítésére próbálj valamilyen megoldást találni. Kérlek, hogy jegyezd fel az eredményeket, mert szeretném majd ismertetni a vezetőséget az eredményeinkről.

Tesztelő: Rendben. Amint lesznek eredményeim szeretném, ha újra tudnánk beszélni.

Menedzser: Természetesen. Szerettem volna kérni is ezt. Köszönöm

A folyamatos, rendszeres megbeszélések és fejlesztések, próbálkozások után, a próba időszak végeztével egy összetettebb megbeszélés zajlódik le:

Tesztelő: Szia, szeretném megosztani veled az eredményeket a hosszú próbálkozási időszak után.

Menedzser: Szia. Rendben hallgatlak.

Tesztelő: Kezdetben belefutottunk pár problémába, hiszen új volt az eszköz számunkra. Ahogy láttad sikeresen vettük az akadályokat, és a rendelések felvitelét sikerült leautomatizálnunk. A tesztelési ciklusban igaz még nem kommunikáltuk, hogy automatikusan is próbálunk teszteket futtatni, de pár rendelést a manuális tesztelés mellett automatikusan is felvittünk a rendszerbe. Az eredmények azok voltak, amiket vártunk. Ugyan azt sikerült reprodukálnunk, mint a manuális tesztelőnek. Közösen ellenőriztük is. Véleményem szerint ezzel a módszerrel a következő tesztelési időszakban 40%-ot a meglévő teszt eseteinkből le tudunk fedni automatikusan. A másik dolog, hogy ezzel körülbelül egy napi munkát tudunk spórolni a manuális tesztelőnek. Úgy gondolom, hogyha ilyen tempóban haladunk, és jól mérjük fel a teszteket, akkor egy fél éven belül már megtérül a befektetett energia.

Menedzser: Ez nagyszerű. A karbantarthatóságot tekintve mennyi időt kell rászánnunk tesztelési ciklusonként a tesztjeink frissítésére?

Tesztelő: Igen ez az egyik dolog, amiben változtatni kell. Úgy vettem észre, hogy modulárisan kellene felépíteni a tesztet, hiszen más üzleti területnél is vannak kapcsolódási pontok. A jövőben így sokkal gyorsabban tudunk majd teszteket elkészíteni, viszont most erre is egy kicsivel több időt kell fordítanunk. Úgy gondolom, hogy ezt jókor fedeztük fel, így később ez nem okoz akkora fejfájást.

Menedzser: Rendben. Az eredményeket továbbviszem, és megmutatom a többi Menedzsernek is. Bízzunk benne, hogy ők is látni fogják ezután az automatizálás előnyeit. Köszönöm!

Tesztelő: Köszönöm. Kíváncsian várom az eredményt.

A fenti példákon látszik, hogy egy automatizálási eszköz és annak használata nagyon sok időt igényel. Felsővezetői oldalról van egyfajta bizonytalanság. Látszik, hogy extra költségekkel jár a tesztek elkészítése és maga az eszköz használata is, amennyiben az kereskedelmi forgalomban kapható. Megfelelő stratégia mellett viszont meggyőződhetünk arról, hogy a vállalatnak, a tesztelői csoportnak érdemes, vagy sem tesztautomatizálást használni.

A tesztelőnek fel kell mérni, hogy milyen teszt dokumentáció áll rendelkezésre. A fenti példából a döntés a rendelések felvitelére esett. Ebben az esetben az automatizálással foglalkozó tesztelő nagy valószínűséggel felkeresi a rendelések teszteléséért felelős kollégát, és próbálja beszerezni a szükséges információkat. Ezek lehetnek: teszt dokumentációk, tesztesetek leírása, teszt adatok, ha a vállalatirányítási rendszerben speciális jogosultságok szükségesek a rendelések felvitelére, akkor azt is meg kell tudakolnunk. Jobb esetben elképzelhető, hogy a begyűjtött információk alapján az automatizálással foglalkozó tesztelő rögzíteni tudja a tesztlépéseket. Tapasztalatom alapján ez még sincs így. Egy teszt automatizálónak sokszor nincs meg az az üzleti tudása az adott területhez, ami szükséges lenne. A tesztesetek leírásából ugyan végre lehet hajtani a teszteket, de ha automatizált folyamatot szeretnénk kiépíteni, akkor sok esetben elkerülhetetlen, hogy a meglévő tesztelési folyamatot ne módosítsuk. Egy jó módszer tud lenni az is, ha az üzleti területhez értő tesztelővel, akár üzleti elemzővel megbeszélések során derítjük fel azt a területet, vagy alkalmazást, amire érdemes figyelmet fordítani. Minél jobban beletanulunk az automatizálásba és szembesülünk a kudarcokkal, annál több hasznos kérdést tudunk feltenni majd tesztkészítés előtt.

Ahogy haladunk előre a Pilot projektben folyamatosan szembesülünk olyan dolgokkal, amik kicsit nehezítik a tesztek implementálását. Ilyenek lehetnek: nem fut le többször a teszt, valamiért elakad egy bizonyos pont után. Nem a megfelelő lépés hajtódik végre az alkalmazáson egy bizonyos fázisában az automatikus tesztnek. Valamilyen nem várt esemény bekövetkezésekor nem tud továbblépni a tesztünk.

Sok ehhez hasonló probléma léphet fel. Idővel olyan megoldásokat tudunk találni ezekre a problémákra, amik a hosszú távú tesztimplementálásban nagy segítséget fognak jelenteni. Felismerjük, hogy bizonyos esetekben, ugyanakkor más teszt eseteknél sokszor ugyan azokat a tesztlépéseket kell végrehajtanunk. Lehetőség nyílik arra, hogy különböző modulokat definiáljunk, így elkerüljük a duplikált kódkészítést. Ezzel közelebb kerülünk az újrafelhasználhatóság megvalósításához. A szinkronizáció segítségével kiküszöbölhető az a probléma, hogy a tesztünk gyorsabban hajtsa végre a lépéseket, mint ahogy a tesztelt alkalmazás tudná. Különösen WEB-es alkalmazásoknál tapasztalunk gyors végrehajtást, és ebből adódhatnak elcsúszási hibák. A tesztjeinkben bizonyos fázisoknál bele kell fejlesztenünk olyan lépéseket, amik megvárják, míg az adott oldal betöltődik, és csak utána engedjük tovább a tesztfutást. Ilyen hibákból származik az a probléma is, hogy a tesztünk nem képes továbbfutni, leáll. Ezeket a nem várt eseményeket (például egy felugró ablak lekezelését, amire előre nem számítottunk, így nem is kezeltük a tesztünkben) is képesnek kell lennünk lekezelni a tesztekben. Képzeljük el, hogy mekkora időveszteséget tud okozni egy ilyen hiba, ha több száz teszt esetet kell végrehajtanunk, és már a tízedik tesztnél leáll a futtatás. Mivel teszt végrehajtások történhetnek este is, így elengedhetetlen, hogy a teszteknek végre kell tudniuk hajtódni ezekben az időpontokban is.

Érdemes megfontolni ezeket a dolgokra és szem előtt tartani, ha tesztautomatizálást szeretnénk bevezetni. A hosszú távú sikeres használhatósághoz ezek nélkülözhetetlenek.

A pilot projekt végével egy átfogó dokumentációt célszerű elkészíteni. Ennek a dokumentumnak tartalmaznia kell a kitűzött célokat, azok megvalósításának módját, és a végleges eredményeket. Az eredmények között ne felejtsük el kimutatni, hogy az automatizálással mennyi időt töltöttünk, és mennyi időt tudtunk spórolni vele. Ezekből a számadatokból látszódni fog, hogy rövidtávon, vagy hosszútávon milyen sikereink lehetnek az automatizált tesztekkel. Ezen eredmények alapján szükséges a környezetünknek is bemutatni a tesztautomatizálás szükségességét. A különböző üzleti területeken elért sikerek után egyre több igény fog mutatkozni a tesztek automatikus végrehajtása iránt.

A megfelelő folyamat kiépítése

Minél szélesebb körben kezdjük a tesztautomatizálást használni, annál inkább tapasztaljuk, hogy szükségünk van definiált, szigorúan kontrollált folyamatok mentén dolgozni. Nélkülözhetetlen egy olyan folyamat, amin belül kezeljük, menedzseljük a tesztek létrehozását és karbantartását. Folyamatosan változó üzleti igények miatt folyamatosan aktualizálni kell a teszteket, új igények esetén meg kell vizsgálnunk, hogy mi éri meg és mi nem, hogy tesztet implementáljunk.

SZÉCHENYI 2020

Létezik számos folyamat, ami mentén működőképes lehet a tesztautomatizálás. Az egyik ilyen módszer a SCRUM (agilis módszertan) ami leginkább fejlesztői csapatoknak kínál előnyöket. Mint korábban említettük, a tesztautomatizálás nem más, mint szoftverfejlesztés. A SCRUM lehetővé teszi, hogy bizonyos fázisokon keresztül, felügyelve tudjunk megoldásokat készíteni. A SCRUM mint módszertan bemutatása nem célja ennek a dokumentumnak. Az, hogy egyes csapatok milyen módszertant követnek befolyásolhatja a környezet is.

Fontos megjegyezni, hogy egy létező módszertan nem fogja garantálni a folyamatunk tökéletes működését. Pontosan ki kell dolgozni, hogy milyen szempontok szerint választunk ki teszteket automatizálás céljából, milyen ütemterv szerint fogjuk lefejleszteni az adott automatikus tesztet, valamint hogyan fog beépülni az a meglévő tesztelési folyamatainkba.

Ezek után jön talán a legnehezebb feladat: Ismertetni, megértetni és alkalmazni a környezettel. Tudnunk kell megfelelően kommunikálni, hogy miért választottuk az adott módszert, mik azok a szabályok, amik mentén működtetni akarjuk a tesztautomatizálást. Be kell vonnunk a manuális tesztelőket, üzleti elemzőket, csoportvezetőket, menedzsereket, hiszen így lesz látható az egész folyamat mindenki számára. Előfordulhat, hogy a megnövekedett igényeket nem tudja az automatizálással foglalkozó csapat a megadott időkereten belül elkészíteni, így a felsővezetői döntés elkerülhetetlen a tesztek kiválasztásában.

A tesztimplementálási ciklusban próbáljunk meg rendszeresen kapcsolatban lenni a manuális tesztelőkkel, üzleti elemzőkkel, hiszen számukra is fontos, hogy sikeresen elkészüljenek a tesztek. Az üzleti terület működéséről tőlük tudunk megfelelő információt begyűjteni. Amennyiben elakadtunk egy teszt készítése során, velük együtt kell döntést hoznunk, hogy folytassuk a teszt elkészítését, vagy ne töltsünk több időt a fejlesztéssel.

Befektetés / megtérülés nyomon követése

Automatizált szoftverteszteléssel foglalkozó csapatok könnyen abba a hibába eshetnek, hogy nem követik a tesztek elkészítésére fordított időt, így nincs egy tiszta kép, hogy bizonyos teszteket érdemes-e elkészíteni, fenntartani, és használni. Különböző tesztek elkészítése kihívásokat rejt, amikből az idők során hasznos tapasztalatokat lehet gyűjteni ahhoz, hogy milyen típusú teszteket érdemes elkészíteni és használni. Beleeshetünk abba a hibába, hogy nem mérjük fel az elkészítésre szánt szükséges emberi erőforrásokat, tesztkészítési időket, jövőbeni fenntarthatóságot, valamint a futtatások gyakoriságát,

mennyiségét. Ezek azok a dolgok, amik alapján a tesztautomatizálás hatékonyságát követni és szabályozni tudjuk.

Fontos, hogy felmérjük a befektetéshez szükséges időt/költséget, valamint megpróbáljuk felmérni és a jövőben követni a futtatások, teszt használatok során nyert időt. Ezzel tudjuk becsülni, majd később pontosan megmondani a megtérülés mértékét.

Azt az időintervallumot, amin belül szeretnénk a befektetett energiát visszanyerni, befolyásolhatja a tesztelési ciklusok száma, a tesztfuttatási periódusok, a projektek, vagy akár egyéb tényezők is.

Sokféleképpen lehet figyelni és rögzíteni a tesztek elkészítésére fordított időt. A követező lista segíthet milyen faktorokat célszerű figyelembe venni:

- Teszttervezésre fordított idő
- Teszt elkészítésére fordított idő
- Tesztadat készítési idő
- Teszt karbantartására fordított idő
- Hányszor hajtódik végre egy tesztelési ciklusban
- Mennyi tesztelési ciklus van
- Szükséges-e betanulás
- Szükség van-e a tesztelendő folyamatról információra
- Van-e elérhető tesztelői környezet
- Elérhető tesztdokumentáció
- Van-e egyéb olyan hardver vagy szoftver, ami költséggel jár és szükséges az alkalmazás teszteléséhez

Amikor elkészül a teszt, és készen állunk a rendszeres használathoz, futtatáshoz akkor a későbbiekben ezeket a faktorokat célszerű figyelni és követni.

- Teszt végrehajtási idő
- Hányszor hajtjuk végre a tesztet
- Teszt futtatás előkészítésére fordított idő
- Tesztadatok karbantartási ideje
- Mennyi időt fordítunk tesztkarbantartásra

Ha jobban belegondolunk, egy tesztvégrehajtás, futtatás során nem mindig csak a futási idő szabja meg, hogy mennyi időnyereségünk volt. Tudunk olyan részeket azonosítani egy automatikus tesztben, ami manuálisan sokkal több időbe telne végrehajtani.

Például:

Tegyük fel, hogy egy tesztelés során végrehajtunk bizonyos számú manuális lépést. Automatikus teszt implementálása előtt sikerül felmérnünk pontosabban a folyamatot, és azt tapasztaljuk, hogy egy-egy fázist másmilyen úton is meg tudunk csinálni, végre tudunk hajtani a géppel. Tegyük fel, hogy a tesztelés folyamatának a közepén, miután létrehoztunk egy teszt felhasználót, egy SQL lekérdezéssel, amit az automatikus tesztünk hajt végre, másodpercek töredéke alatt visszakaphatjuk a létrehozott felhasználó egyedi azonosítóját a rendszerből. Ezzel elértük azt, hogy a manuális tesztelőnek nem kell elindítani egy SQL lekérdezést futtató eszközt, nem kell manuálisan előkészíteni a szükséges SQL parancsot, majd az eredmény után visszatérni a tesztelendő felületre és folytatni a folyamatot. Látszik, hogy különböző fázisokat máshogyan is meg tudunk oldani, ugyanazzal az eredménnyel, csak hatékonyabban.

A fenti példa alapján, meg tudunk határozni egy bizonyos szorzót, hogy a tesztelés közepén végrehajtott SQL lekérdezés mennyivel volt gyorsabb, mint ugyanannak az eredménynek a kinyerését elvégzendő manuális lépések. Itt nem csak a tesztek futása a mérvadó. Ilyen és ehhez hasonló mérési eredmények tisztább képet adnak a hatékonyságról.

Természetesen ez nem azt jelenti, hogy ebben az esetben is egyértelmű és egyszerű a méréseinket meghatározni. Rosszul felállított megtérülést követő folyamatok pontatlan és nem releváns eredményeket adhatnak.

A jobb szemléltetés érdekében elemezzük ki az alábbi egyszerű táblázatot:

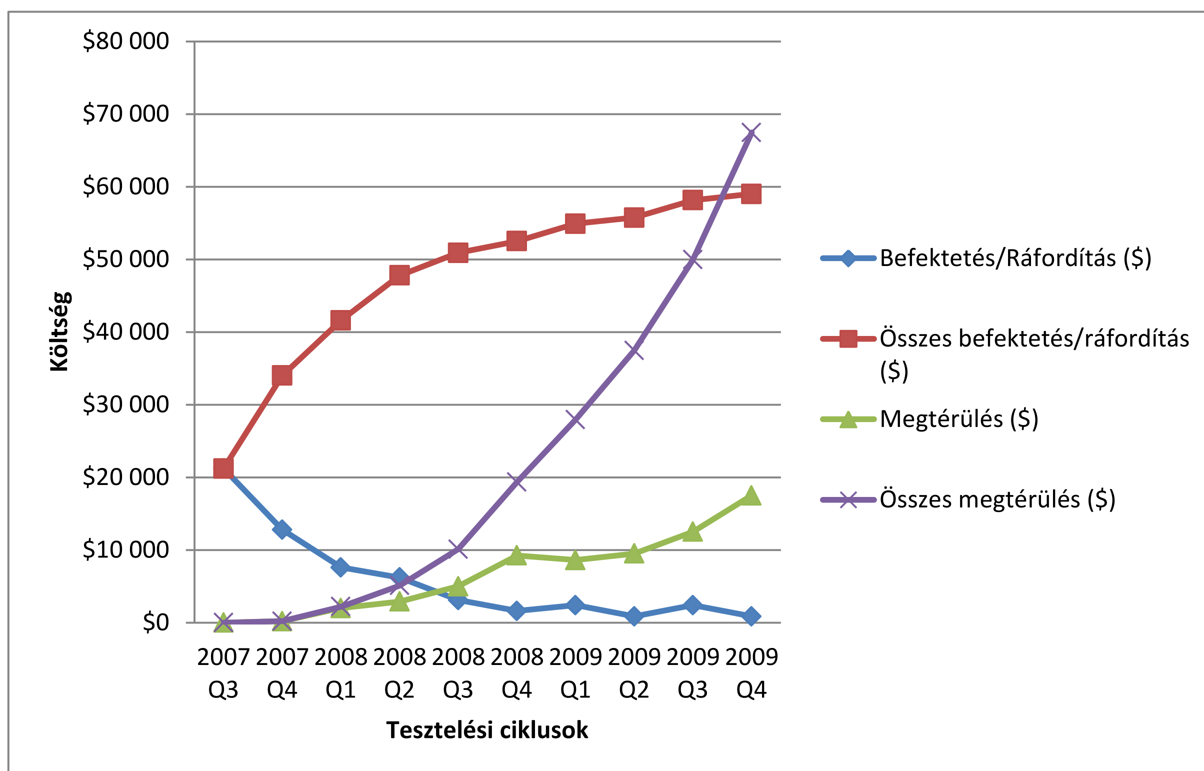
	Manuális	Automatikus
Teszt végrehajtás (perc)	8	5
Tesztek száma	10	10
Egyéb idő (perc)	30	7
Összesen (perc)	110	57

A táblázatból jó látszik, hogy tesztvégrehajtási időt mérünk, tesztek számát határoztuk meg, valamint egyéb időt is rögzítünk, ami például tesztadat előkészítési időt jelölhet, vagy teszteredmények rögzítésének idejét stb.

A számítás itt igen egyszerű. A teszt végrehajtási időt megszorozzuk a tesztek számával, és hozzáadjuk az egyéb időt. Így megkapjuk, hogy mennyi időbe telik a 10 tesztet manuálisan, valamint automatikusan végrehajtani. Látszik a két számadatból, hogy mennyi időt spóroltunk meg automatikus tesztvégrehajtással.

Természetesen ezeket az értékeket tesztvégrehajtáskor rögzítjük. Sokkal komplexebb a dolog akkor, ha a befektetési időket is megpróbáljuk szemléltetni ezek mellett az adatok mellett.

A következő diagram sokkal komplexebb, ugyanakkor pontosabb rálátást biztosító befektetés/megtérülés követést mutat:



A jobb megértés érdekében elemezzük ki az ábrát.

Az ábra függőleges tengelye a költségeket, a vízszintes tengely pedig a tesztelési ciklusokat mutatja. A tesztelési ciklusok ebben az esetben negyedévente szerepelnek. Minden negyedévben vizsgáljuk, hogy adott automatikus tesztekre mennyi időt szántunk készítés terén, valamint mennyi időt fordítottunk karbantartásra. Ezeket rendszeresen vizsgálva kapjuk a fenti ábrát.

Tegyük fel, hogy egy teljesen új automatikus teszt implementálását kezdtük el az ábra szerint 2007 Q3-as tesztelési ciklusában.

A befektetés/ráfordítás görbe azt mutatja, hogy mennyi időt szántunk az adott teszt elkészítésére. Ez a költség származhat a teszt elkészítéséből, a különböző konzultációs időkből, amit a fejlesztőkkel, vagy az üzleti elemzőkkel közösen fordítottunk az adott teszt elkészítésére. Ide tartoznak a teszt adatok elkészítésének ideje, a próbafuttatások, a hibajavítások, módosítások is.

Figyeljük meg, hogy ez a görbe a további tesztciklusokban csökkenő értékeket mutat. Ez abból adódik, hogy az a tesztelési folyamat nem változott a tesztelési ciklusokban, nem voltak új funkciók implementálva üzleti területen, így nem kellett módosításokat, javításokat eszközölni a teszten. A karbantartás ideje fokozatosan csökken.

Az összes befektetés/ráfordítás görbe azt mutatja, hogy összesen, időről időre mennyi időt, erőforrást fektettünk bele az adott automatikus tesztbe. Ez nem más, mint a befektetett költségek összege.

A megtérülés görbe azt mutatja, hogy a tesztelési ciklusokban mennyi időt nyertünk az automatikus teszt futtatásával.

Az összes megtérülés pedig a tesztelési ciklusokban történt megtérülésének összegét ábrázolja.

Ezekből a görbékből nagyon sok hasznos információt tudunk megállapítani. A negyedévente befektetett költségek mutathatják azt például, hogy nincs elég információnk az adott tesztelési területen, és azért kell folyamatosan időről időre karbantartani a tesztet, vagy akár azért mert mindig valamilyen változás történik egy adott területen.

A legfontosabb itt az, hogy keressük azt a pontot, amikor az összes befektetés és az összes megtérülés metszi egymást. Ekkor tudjuk azt mondani, hogy a befektetett költségek megtérültek, és attól a ponttól kezdve csak hatékonyságról beszélhetünk.

Összefoglalás

Automatizált szoftvertesztelés egy nagyon érdekes terület. Fontos, hogy megismerjük a tesztelendő alkalmazást, felmérjük, hogy tényleg alkalmasak a meglévő tesztek az automatizálásra vagy sem. Ha tesztautomatizálási folyamatokat szeretnénk bevezetni, akkor célszerű egy próba projektet indítani, aminek keretein belül megvizsgáljuk, hogy hol és milyen módon tudjuk beilleszteni a tesztelési folyamatainkba az automatikus teszteket. Mindenképpen kövessük, hogy mikor, mire, mennyi erőforrást használtunk, és azt is, hogy a tesztjeink futtatása milyen nyereséggel bír. Ekkor tudunk megfelelő képet kapni, hogy a folyamataink és tesztjeink rendben vannak vagy sem. Szükséges a meglévő tesztjeinken, vagy folyamatainkon változtatni, vagy sem.

Mindezek mellett talán az egyik legfontosabb dolog: Ne próbáljunk meg mindent automatizálni, ugyanis a manuális tesztelést nem helyettesíthetjük vele teljes mértékben.

Függelék

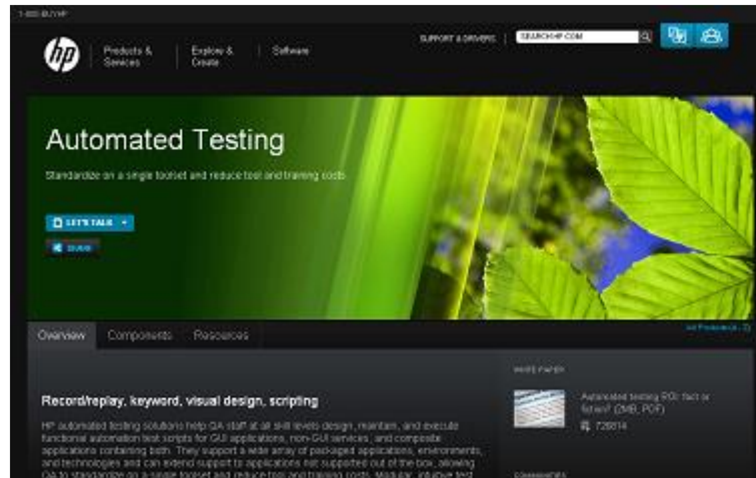
Piacvezető tesztautomatizálási eszközök

A következőkben áttekintjük azokat a tesztautomatizáló eszközöket, amelyek pillanatnyilag a legnépszerűbbek, leginkább használtak az iparban.

TELERIK TESTSTUDIO

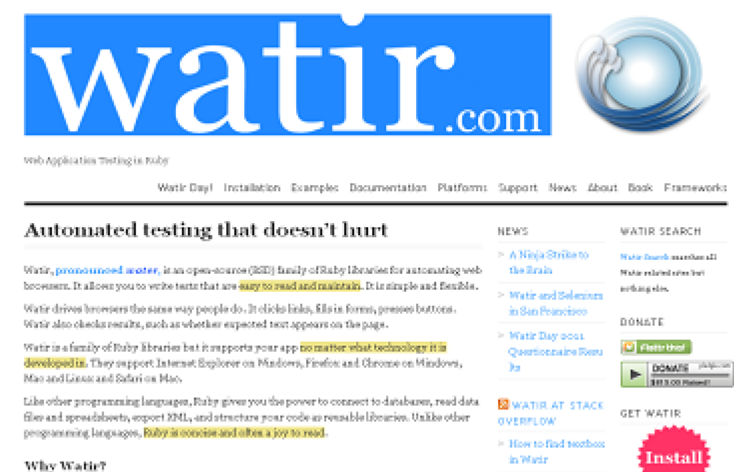
A Telerik TestStudio egy tipikus “minden egyben" (all in one) tesztelő szoftver, amely funkcionális, load, teljesítmény és mobil applikáció tesztelésére egyaránt alkalmas. A mélységi funkcionális tesztelés magában foglalja a HTML5, AJAX, Silverlight, WPF alapokon fejlesztett natív webes és asztali alkalmazások, továbbá a mobil és tablet alkalmazások tesztelését is. Ezen túlmenően a tesztelő csapatok realizálni tudnak többek között JavaScript hívás, dinamikus oldal szinkronizáció, kliens oldali viselkedés, UI virtualizáció és XAML animáció teszteket is.

QTP



A HP-QuickTest Professional szoftver funkcionális és regressziós tesztautomatizálási lehetőségeket nyújt különböző szoftverkörnyezetek számára. Használható kulcsszó-alapú, szkript-alapú és grafikus felhasználói interfészeken keresztül is. A legfontosabb eszközei a következők: egy kaszkádolt optimizáló rendszer, üzleti folyamatok által vezérelt tesztelés, IT-ellenőrzött üzleti vagyon kezelés.

Watir



A Watir egy nyílt forrású Ruby könyvtár család, amely web-böngészők automatikus tesztelésére szolgál. Nagyon könnyen olvasható és módosítható tesztek írhatók benne. Egyszerű és flexibilis. Segítségével lehet rá lehet kattintani egy linkre, ki lehet tölteni formanyomtatványt, meg lehet nyomni egy gombot. A

Watir ellenőrzi az eredményeket, például, hogy egy elvárt szöveg megjelenik-e az oldal megfelelő helyén. Legfontosabb eszközei: csatlakozás adatbázishoz, adatállomány és táblázat beolvasása, XML exportálása, a kódjaink újrafelhasználható könyvtárakba való rendezése.

TOSCA Testsuite

A TOSCA Testsuite funkcionális és regressziós tesztek automatikus végrehajtását biztosítja. A tesztautomatizálási függvényeken túlmenően tartalmaz még integrált tesztmenedzselő eszközrendszert, grafikus felhasználói interfészt (GUI), egy parancssoros interfészt (CLI) és egy alkalmazás programozói interfészt (API), egy dinamikus és szintetikus tesztadat generátort, egy dinamikus üzleti alapú magasan automatizált teszteset generátort és a manuális és automatizált tesztek egységes kezelését függetlenül attól, hogy azok GUI vagy nem GUI tesztek.

Selenium

A Selenium a webes alkalmazások egy hordozható tesztelési keretrendszere. A Selenium egy tipikus "record/playback" típusú eszköz, amelyben egy tesztelési szkriptnyelv megtanulása nélkül tudunk írni teszteket. Főbb eszközei: rögzítés és lejátszás, intelligens mező kiválasztás, Xpath kezelés, selenium parancsok automatikus kiegészítése, tesztek közötti böngészés, ellenőrzőpontok beállítása, Ruby szkriptek kezelése, felhasználói selenium állományok kezelése, oldal fejlécek együttes kezelése.

VISUAL STUDIO TEST PROFESSIONAL

SZÉCHENYI 2020

A Visual Studio Test Professional a Microsoft által kifejlesztett integrált tesztelői eszközrendszer, amely egy teljes tervezés-tesztelés-értékelés munkafolyamatot lefed a fejlesztők és tesztelők együttműködése során, miközben növeli a tesztelők láthatóságát a projekt teljes menete során. Eszközei között megtalálható az állományaktivizálás hiba esetén, a manuális tesztelés, az újrafelhasználható manuális teszt rögzítése, az alkalmazás életciklus kezelése.

RATIONAL FUNCTIONAL TESTER

A Rational Functional Tester egy automatizált funkcionális és regressziós tesztelési szoftver. A tesztelőknek automatikus tesztelési lehetőséget biztosít funkcionális, regressziós, GUI és adatvezérelt teszteléshez. Fontosabb lehetőségei: egyszerűsíti a teszt létrehozás és vizualizáció lehetőségét a felhasználói történet alapú tesztelés kezelésével, életciklus követési lehetőséget ad, validálja a dinamikus adatokat egy dinamikus adatvalidáló varázsló segítségével, folyamatos tesztelés lehetőséget biztosít kulcsszó-alapú teszteléssel, SDK proxy van, a párhuzamos fejlesztéshez tesztszkript- verzió ellenőrzés áll rendelkezésre.

TESTCOMPLETE

A TestComplete egy olyan tesztautomatizálási eszköz, amely lehetőséget teremt tesztek létrehozására, kezelésére, futtatására bármely ablakban webes szoftverek és vastag kliensek esetén. Bárki számára könnyen és gyorsan átlátható és használható. Álljon itt néhány lehetősége: nyílt alkalmazás programozói interfész, könnyű kiterjeszthetőség, könnyen érthető dokumentáció, szkripttel történő teszt létrehozás, alkalmazás támogatás. A könnyű használhatóság miatt bárki (minden különösebb előképzettség és tapasztalat nélkül) néhány perc alatt tud jó teszteket összehozni. Ára alacsony, a támogatása egészen kiváló.

TESTPARTNER

A Testpartner egy olyan tesztautomatizálási eszköz, amely elsősorban a funkcionális tesztelést segíti, különös tekintettel az üzletkritikus alkalmazásokra. A fejlesztési projekt résztvevői egy olyan rétegezett megközelítésben dolgoznak együtt, amely lehetővé teszi a fejlesztők, minőségbiztosítók és a nem-technikai alkalmazás felhasználók mindenkori együttműködését a tesztelőkkel, elérve ezáltal azt, hogy a rendelkezésre álló időkeretben a lehető legtöbb tesztelési tevékenység legyen végrehajtva. Eszközei: vizuális felhasználói történet kezelés, automatikus regressziós tesztelés, automatikus objektumorientált szkript generálás, integrált VBA.

SOATEST

SZÉCHENYI 2020

A Parasoft által fejlesztett SOAtest a webes alkalmazások, az üzenetek, a protokollok, a biztonság és a felhő alkalmazások tesztelését automatizálja. A SOAtest biztonságos, megbízható, a jogszabályoknak megfelelő üzleti folyamatokat garantál és a Parasoft nyelvi termékekkel (pl. Jtest) integrálva a szoftverfejlesztés életciklusának kezdetétől segít a csapatnak megelőzni, vagy detektálni az alkalmazás hiányosságait. Néhány eszköze: kliens-szerver emuláció, többszintű verifikáció, teszteset koordináció, regressziós tesztelés, automatikus teszteset generálás, kódolási standardok kikényszerítése, SOAP-alapú vállalati rendszer kezelése mind SOAP kliens, mind SOAP szerver oldalon. Mindezek a lehetőségek egy igen korai modul tesztelési lehetőséget biztosítanak.

TESTDRIVE

A TestDrive egy hatékony tesztautomatizálási megoldást biztosít grafikus felhasználói felületek és böngésző alkalmazások távoli teszteléséhez. Segítségével jelentős időráfordítás csökkentés és minőségnövelés érhető el anélkül, hogy a tesztelés maga jóval bonyolultabbá válna. A tesztelési szkriptek automatikusan állnak elő a manuális tesztek eredményeiből, a tesztek eredménye egyidejűleg elemezhető a mögöttes adatbázis segítségével.

Squish

A Squish a vezető platformokon és technológiákon átnyúló Gui tesztautomatizálási eszköz funkcionális és regressziós GUI tesztek számára. A Squish drámaian lecsökkenti a szoftver verziók Gui teszteléséhez szükséges időt és emellett növeli az alkalmazások minőségét. A Squish tetszőleges szkriptnyelven (pl. JavaScript, Perl. Python, Tcl) megírt tesztet tud kezelni, megadva így a tesztelőknek azt a szabadságot, hogy kedvenc nyelvüket alkalmazhassák. Főbb eszközei: elosztott és multi-alkalmazásos tesztelés, testreszabott vezérlés, GUI objektum térkép, adatvezérelt tesztelés, naplózás, kötegelt végrehajtás.

SZÉCHENYI 2020

TÁRGYMUTATÓ

adatsérülés, 18
adatvezérelt automatizálás, 196
alfa-teszt, 97
állapotátmenet tesztelés, 138
átnézés, 119
átvételi teszt, 57
átvizsgálás, 116
automatizálás, 56
Automatizálás, 192
Automatizált teszt, 192

bejárási út alapú tesztelés, 153
béta-teszt, 98
bíráló, 118
biztonsági teszt, 105
black-box, 101

ciklomatikus komplexitás, 129

Delphi, 178
dinamikus technikák, 132
döntés alapú tesztelés, 152
döntési tábla tesztelés, 140

egységtesztek, 49
ekvivalencia particionálás, 134
emberi tévedés, 18

fehér doboz technika, 151
fejlesztő, 10
fekete doboz technikák, 134
felhasználó, 10
felhasználói élmény, 23
feltétel meghatározásos tesztelés, 153
feltételek tesztelése, 153
funkcionális hibák, 22
funkcionális teszt, 97, 111
függetlenség szintje, 37
függőségek emulálása, 53

gray-box, 89

hálózati hiba, 15
hardver hiba, 14
használati eset alapú tesztek, 142
használhatóság, 23
használhatósági teszt, 101
határérték-analízis, 136
hatás, 25
hatáselemzés, 114
hatékonyság, 23
hatékonysági teszt, 110
hibakeresés, 27
hordozhatóság, 24

Incidens menedzsment, 185
informális átvizsgálás, 120
inspekció, 120
integrációs teszt, 51
írnok, 119
ismerethiány, 19
issue tracking, 24
ISTQB, 9, 28, 40

játékszoftverek, 13
jUnit, 63

karbantartási teszt, 114
karbantarthatóság, 24
kezelői hiba, 17
kitartási teszt, 100
Kockázatok, 183
kompatibilitási teszt, 110
komponensteszt, 49
Konfiguráció menedzsment, 181
konfigurációs hiba, 16
kulcsszó vezérelt automatizálás, 199

lineáris automatizálás, 194
linearitás vizsgálat, 111
logika alapú technikák, 140

megbízhatóság, 23
megbízhatósági teszt, 111
megfelelési teszt, 110
menedzser, 118
mennyiségi teszt, 101
minőség, 21
minőségbiztosítás, 21
minőségirányítás, 21
mocking, 54, 86
moderátor, 118
műszaki teszttervezés, 30

nem funkcionális teszt, 98

nemzetköziesség teszt, 104

Project kockázat, 183

regressziós teszt, 112, 113
rendszer szoftverek, 12
rendszerintegrációs tesztek, 52
rendszerteszt, 55

sanity-teszt, 57
skálázhatósági teszt, 108
smoke-teszt, 56
static techniques, 115
statikus elemzés, 127
statikus technikák, 115
stresszteszt, 100
strukturális teszt, 113
stubbing, 54, 82
Súlyozott átlag, 177
SWAG, 177

szabályossági teszt, 110
szervezési hiba, 20
szerző, 118
szoftverfejlesztési modellek, 41

technikai átvizsgálá, 120
teljesítmény teszt, 98
terheléses teszt, 99, 111
Termék kockázat, 183
Teszt Menedzser, 190
Teszt monitorozás, 179
Teszt stratégia, 170
tesztautomatizálás, 56, 113, 192
tesztbázis, 30
tesztelemzés, 30
tesztelés, 25
tesztelő, 10, 36, 37, 38, 44, 45, 113
Tesztelő, 190
Tesztelő csapat, 188
teszteset, 30, 32, 33, 49, 50
tesztfeltételek, 30
tesztszintek, 49
tesztszkript, 56
teszttervezés, 30
Teszttervező, 190
teszttípusok, 96
többszörös feltételek tesztelése, 153

utasítás szintű tesztelés, 151

Üzleti kockázat, 183
üzleti szoftverek, 11

vezérlő szoftverek, 12
visszaállíthatósági teszt, 108
V-Modell, 41

WAG, 177

xUnit, 60

SZÉCHENYI 2020

FELHASZNÁLT SZAKIRODALOM

[1] HSTQB. (2014. 07 01). hstqb.org. Letöltés dátuma: 2014. 07 01, forrás: HSTQB.org: http://www.hstqb.org/index.php?title=Headlines

[2] HTB Glossary. (2013. 07 01). hstqb.org. Letöltés dátuma: 2014. 07 01, forrás: hstqb.org: http://www.hstqb.com/images/5/5f/HTB-Glossary-3_13.pdf

[3] IEEE. (2014. 07 01). IEEE.org. Letöltés dátuma: 2014. 07 01, forrás: IEEE organisation: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6835311&refinements%3D4294965216%26queryText%3Dsoftware+quality

[4] IEEE. (2014. 06 10). SCRIBD - Real Unlimited Book. Letöltés dátuma: 2014. 07 01, forrás: http://www.scribd.com/: http://www.scribd.com/doc/101838061/IEEE-610-Standard-Computer-Dictionary

[5] IEEE 829. (2008. 01 11). IEEE Standard for software and system test documentation. online.

[6] ISO. (2014. 07 01). www.iso.org. Letöltés dátuma: 2014. 07 01, forrás: www.iso.org.: http://www.iso.org/iso/catalogue_detail.htm?csnumber=20115

[7] http://qtp.blogspot.hu/2008/10/generations-of-test-automation.html

[8] http://www.tutorialspoint.com/software_testing/testing_types.htm

[9] http://books.google.hu/books/about/Implementing_Automated_Software_Testing.html?id=UVlgr2TTwCAC&redir_esc=y

[10]http://www.softwarequalitymethods.com/papers/star99%20model%20paper.pdf

[11]http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm

[12]Graham, Rex Black, Erik van Veenendaal, Dorothy (2012). *Foundations of Software Testing: ISTQB Certification.* (3. ed.). London: Cengage Learning EMEA.

[13]Craig-Jaskiel (2002). Systematic Software Testing – Artech House.

[14]Bob van de Burgt, Dennis Janssen, Erik van Veenendaal (1998). Successful Test Management: An Integral Approach.

[15]Peter Farrell-Vinay (2008). Manage Software Testing.

[16]Rex Black (2009). Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software testing.

[17]http://automated-testing.com/test-management-resources.html

[18]http://www.qaforums.com/