SZAKDOLGOZAT

Debrecen
2009

Debreceni Egyetem
Informatikai Kar

A szoftvertesztelés módszerei

Témavezető: Fazekas Gábor
Egyetemi docens

Készítette: Panczák Ágnes
PTI MSc

Debrecen
2009

Tartalomjegyzék

1. Bevezetés

A szoftvertesztelés Magyarországon ma még gyerekcipőben jár. Az emberek egyik része úgy gondolja, a szoftvertesztelés nem több mint klikkelgetni – kattingatni, és hibát keresni a szoftverben, de akad olyan is, aki még soha nem hallott róla, a harmadik nézőpont szerint pedig kevés munka, sok pénz. A tesztelés lényegesen bonyolultabb, összetettebb voltát nem látják. Azt is mondhatnám, hogy lenézik ezt a munkát.[1]A cégvezetők, projektvezetők gyakran nem ismerik fel a tesztelés fontosságát. Számukra a hirtelen felmerülő problémák megoldása és a napi munkafolyamatok rutinszerű elvégzése fontosabb, így a szoftver(ek) minőségének javítása háttérbe szorul. Az is a tesztelés ellen szól, hogy ha minőségi szoftver kerül ki a piacra, akkor a support – ból kevesebb pénz folyik be.

[2]Valaha úgy hitték, hogy formális módszerekkel képesek kiváltani a tesztelés szükségességét. Ma már tudjuk, hogy a program hibáinak száma kevesebb egy biztos fejlesztői környezetben, de mivel ezek összeadódnak, így egy nagyobb programban több hiba van. Szükség van tehát tesztelésre és debuggolásra, mely a teljes ráfordítás akár 60% - át is jelentheti.

A dolgozat témája a fentebb említetteken kívül azért a tesztelés, mert tesztmérnökként dolgozom pár éve. Úgy gondolom, hogy tesztelésről nagyon keveset tudunk, és ahogy majd a felhasznált irodalomból is kitűnik, szinte csak angol nyelvű forrásokat találni. Magyar nyelvű forrás alig van, nekem egy magyar nyelvű könyvet sikerült felkutatnom, és persze mellette van még a tapasztalat. Kezdetben én sem tudtam, hogy ennyire összetett feladat lenne, hasonlókra gondoltam mint a többi ember, aki meghallja a szoftvertesztelés szót.

A dolgozatban ismertetem a szakmát, a módszereket, technikákat. Megvilágítom, miért olyan fontos a szoftvertesztelés, és miért kell ezt is tanulni.

[1] Forrás: http://teszteles.blog.hu/ Cégen belüli szemlélet hiánya című írás
[2] Forrás: Yashwant K. Malaiya: Automatic Test Software

2. Minőségbiztosítás és tesztelés

Minőség, Szoftverminőség és szoftverhibák

A minőség, a dolgok lényegét jellemző tulajdonságok összessége, filozófiai kategória, illetve valaminek az értékelését is magában foglaló jellege. A „jó" szoftver ismérveivel az 1970 – es évektől kezdtek el foglalkozni, mára pedig léteznek már szabványok, modellek, melyekkel meghatározható a szoftver minősége.

Minden szoftverben vannak hibák. Miért? Mennyi hibát tekintünk még elfogadhatónak? Humphery szerint, a tapasztalt programozók is ejtenek hibákat minden 7 – 10 forrássorban 1 – et. Természetesen, a hibák felét fordításkor javítják, a teszteléssel még több kerül elő, mégis 15% a szoftverben marad, amikor az már az ügyfélhez kerül. A szoftverminőség fogalmát nem lehet egyszerűen meghatározni, mint a minőség esetében. A fogalom az évek során jócskán megváltozott. Kezdetben, az minősült jónak, ami egyszer lefutott, és a várthoz hasonló eredményt adott, elsősorban fizikusok, matematikusok használták bonyolultabb egyenletrendszerek megoldására. Az 1970 – es, 1980 – as években megjelentek az assembler nyelvek, operációs rendszerek. Két program közül az volt jobb, amelyik gyorsabban valósította meg a feladatot, és kisebb kapacitással dolgozott, ún. mikrohatékonyság jellemezte. Később a mikrohatékonyság makrohatékonyságra váltott át. A számítógépeket ekkor már nem csak szakemberek használták. Fontossá vált a hordozhatóság, hogy a forráskód érthető legyen, követhető, és felhasználható. Ma már a helyesség a döntő. A szoftverminőség fogalmának megalkotását Garvin nevéhez fűzik (1984). 5 definíciót fogalmazott meg.

Garvin – féle szoftverminőségi definíciók:

1. Transzcendens definíció: a minőség a veleszületett kiválóságot jelenti, csakis tapasztalat útján ismerhető fel.
2. Felhasználás alapú definíció: a felhasználásra való alkalmasság. A definíció szubjektív, mivel a felhasználói igényekből indul ki.

3. Termék alapú definíció: a minőség precíz, és mérhető változó. A minőség tehát, a termék jellemzőiből fakad.
4. Folyamat alapú definíció: a specifikációknak való megfelelősség.
5. Érték alapú definíció: a költség függvényében határozza meg a minőséget. Jó minőség, alacsony áron alkalmas a feladat elvégzésére.

1977 – ben, majd 1978 – ban látott napvilágot Boehm, és McCall modellje (modelleket lásd: Függelékben). A modellek alkalmazásakor a felhasználónak ki kell választania a számára legfontosabb minőségügyi jellemzőket, és mérőszámokat. Mindkét modell rögzített, alkalmazásukkor feltételezzük, hogy az összes fontos minőségi jellemző, mérőszám, attribútum szerepel a modellben. A felhasználónak csak válogatnia kell közöttük.

Minőségbiztosítás kontra, tesztelés

Minőségbiztosításnak nevezhető – e a tesztelés? Olvastam egy cikket, mely szerint nem. Megkülönböztet minőségbiztosítást, és tesztelést, de elismeri, hogy létezik minőségbiztosítás a tesztelésben.

A tesztelés 2 okból nem minőségbiztosítás egyrészt, mert a fejlesztés része kellene, hogy legyen, másrészt pedig, tesztelni lehet jól és rosszul is. [3]A minőségbiztosítás egy olyan – jó esetben technikai apparátussal társuló – *értéknövelő* elv és koncepció, amit a fejlesztési folyamatra kell ráültetni, és annak a lehető legtöbb fázisában alkalmazni. A folyamat a követelmények elemzésével, tanulmányozásával kezdődik, és a támogatás végével ér véget. A minőségbiztosítás fogalmán értjük többek között a változás-, és kockázatkezelés, az ergonómia, a dokumentációs, a támogatási minőség, a belső és külső elvárások megfogalmazását, és az újrafelhasználhatóságot is.

Minőségbiztosítás a szoftvertesztelésben

Minőségi tesztelést kell végezni, ehhez szükség van egy tesztelési stratégiára, tervre. Ez tartalmazza az erőforrásokat, a feladatokat. Eszközök, egy másik feltétel, amennyiben nincsenek, nem lehet elvégezni a munkát. Próbáljunk

[3] Forrás: http://teszteles.blog.hu/Minőségbiztosítás vs Tesztelés című írás

meg minél költséghatékonyabbak lenni, ez szélesebb körű tesztelést tesz lehetővé. A vezetői szándék hiányában nem lehet minőségi terméket előállítani. Szakemberek, van olyan elmélet, mely szerint, ha valaki nem tud teljesíteni a saját területén, akkor tesztelni küldik. A tesztelésnek is vannak szakemberei, akik precízek, pontosak (kerek mondatokban képesek leírni például egy tesztesetet, hogy az ügyfél is képes legyen végig csinálni, ha hozzá kerül a szoftver), türelmesek (egy dolgot többször is el kell mondani, meg kell mutatni a fejlesztőknek), és tudásuk legjavát adva dolgoznak.

A tesztelés stratégiájaként módszereket, eszközöket, erőforrás-szükségleteket, ütemezést és terjedelmet kell meghatározni. Meg kell határozni továbbá, hogy milyen tesztekre lesz szükség, a dokumentációról részletesebben a 5. fejezetben írok.

A tesztelés végigköveti a szoftvert egy életcikluson át. Magában foglalja a validáció (jó rendszert építettünk – e fel?) és a verifikáció (jól építettük – e fel a rendszert?) fogalmát is.

A tesztelés és a tesztelő is fontos, mert: a tesztelő győződik meg arról, hogy tényleg működik a szoftver, hogy jól van felépítve, minden funkció meg lett – e valósítva melyeket a követelményekben leírtak, valamint, hogy a szoftver a felhasználó céljainak megfelel – e. A tesztelés folyamán kerülnek elő a hibák a szoftverben, melyeket jelezni kell, és ha javították ellenőrizni is. A debuggolás nem szoftvertesztelés! A tesztelés célja a hibák megtalálása, a debuggolás feladata a hibák pontos helyének és okának meghatározása.

Hibamentes szoftver nincs. A tesztelés csak a hibák meglétét bizonyítja. Vezérelvnek a kimerítő tesztelést állíthatnánk, de ez lehetetlen, általában az idő hiánya miatt. Tesztelhető területek például a menük funkciói, a funkciók kombinációi, helyes és helytelen adatokkal való tesztelés, ezek kipróbáláshoz idő kell, amiből nem sok van.

Mérések

Szeretünk mérni. Ez jó lehet, egy jó méréssel sok információt gyűjthetünk, már ha értjük mit is mérünk, jól választunk mérőszámot, össze illő dolgokat próbálunk mérni. Minél több összehasonlítási, műveleti lehetőség van egy skálán,

annál gazdagabb az a skála. A mérés az a folyamat, amely során a valós világ eleminek attribútumaihoz számokat vagy szimbólumokat rendelünk abból a célból, hogy jellemezzük vagy sorrendbe állítsuk őket. Néhány mérési skála és jellemzőik:

- Nominális skála: bizonyos kategóriákhoz való hozzátartozás kifejezésére szolgál, nem tartalmaz sorrendiséget. Műveletei: =, # (nem egyenlő).
- Rendezési skála: elemek közötti sorrendet fejezhetjük ki vele, mely valamilyen attribútum szerint alakul ki. Műveletei: =, #, <,>.
- Intervallumskála: a mért értéket ekvivalens intervallumok számával fejezzük ki. Műveletei: =, #, <, >, +, -.
- Arányskála: kifejez sorrendiséget, ha van egy nulla elem (az attribútum hiányára utal). Műveletei: =, #, <, >, +, -, /. Itt lehetőség van rá, hogy megállapítsuk például azt, hogy egy elem mekkora egy másikhoz képest.
- Abszolút skála: a mérés egy bizonyos entitás elemeinek megszámlálását jelenti, egyetlen leképezés lehetséges, az aktuálisan számolt szám. A mérés eredményével mindenfajta aritmetikai analízis értelmes.

Szoftverterméket ma, a 70 – es években megalkotott ciklomatikus számmal mérünk. Így, *n* számú szögpont, *e* számú élt tartalmazó, *p* komponensből álló, *G* gráf ciklomatikus száma: *e-n+p*, ha a program vagy modul 1 komponensből áll, akkor *e-n+1*.

Hogyan végezzük a mérést? Az egyik javaslat, hogy felülről lefelé, mert így biztosítható a metrikák hasznos és releváns összegyűjtése. A másik lehetőség ennek épp a fordítottja, alulról felfelé irányuló mérés. A mérési programot így a szervezet gyakorlati szükségleteinek és a helyi környezetnek lehet megfeleltetni. A harmadik szempont, a kevert megközelítés. Eszerint mindkét fentebb említett megközelítésnek benne kell lennie a mérési programokban.

3. A tesztelés alapelvei

1. A tesztelés eredményeinek összehasonlíthatónak kell lennie a követelményekkel!
2. Nem lehet összevissza „nyomkodni” a szoftvert, készíteni kell egy tervet!
3. Nem az egésszel kell kezdeni a tesztelést, hanem a részletektől, a kicsitől haladni a nagy felé!
4. A tesztelés során megtalált hibák 70 – 80% – nak a 20% – a, a program egyes komponenseire vezethető vissza!

A tesztelés folyamata

A tesztelési folyamat a dokumentáció, a követelmények (TRS, SRS), a rendszerterv áttanulmányozásával indul. A tesztelő átolvassa, kérdéseket tesz fel, ha van olyan rész, amelyet nem ért. A dokumentációk után, vagy alapján ír teszteseteket, melyek alapján tud tesztelni, és ezek alapján meg is tudja mondani, hogy mi lett letesztelve. A teszt folyamatát is meg kell tervezni, és ez alapján kell a munkát elosztani, a folyamatot pedig nyomon kell követni.

A követelmények megismerése után minden tesztelési folyamat unit teszttel kezdődik, ez a legkisebb egységeket vizsgálja. A következő lépésben azt is vizsgálni kell, hogy a program egyes részei hogyan működnek együtt, egyáltalán működnek – e, erre való a modul teszt. Egy modulon belül végzi el az ellenőrzést. Ezeket együttvéve szokás komponenstesztnek is nevezni. Az integrációs teszt a következő lépés, mely a komponensek együttműködését ellenőrzi. A következő két lépés az alrendszer, illetve a rendszerteszt. Az alrendszer teszt a modulok közötti együttműködést vizsgálja, a rendszerteszt pedig az alrendszerek együttműködését. A végén elfogadási tesztre kerül a szoftver. Ekkor már meg kell felelnie az ügyfél igényeinek.

2 lényeges fázisra bontható a folyamat: alfa és béta teszt. Az alfa teszt ideje alatt kell a hibák nagy részét megtalálni és kijavítani. Ebben az időszakban (esetünkben ez 4 – 5 hét), főleg a tesztcsapat dolgozik a szoftverrel.

4. Szoftvertesztelési technikák

Black – box test

A black – box tesztelés esetében csak a felhasználói felület látható. A tesztelő nem látja a kódot, nem ismert számára a belső felépítés, működés. Egy fekete dobozként kezeli a szoftvert. A fekete – doboz tesztelés szinonimái: viselkedési teszt (behavioral test), funkcionalitás teszt (functional test), zárt – doboz teszt (closed – box), vagy nem átlátszó – doboz teszt (opaque-box). *Előnyei:* elfogulatlan, mert a tesztelő és a tervező nem ugyanaz a személy; a tesztelőnek nem szükséges programozási ismeretekkel rendelkeznie; a felhasználó szempontja érvényesül tesztelés során; a teszteseteket meg lehet tervezni, amint a specifikáció elkészült. *Hátrányai:* felesleges is lehet, ha a tervező már lefuttatta a testeseteket; nehéz a teszteseteket megtervezni; nem lehet mindent letesztelni az idő hiánya miatt. Néhány fekete doboz technika:

Equivalence Class Partitioning: Azon adat csoportokat, amelyekről tudjuk, vagy feltételezzük, hogy a tesztelt program azonos módon dolgozza fel, elég, ha egy reprezentálja. A helyes inputok mellett az inkorrekt inputokat is tesztelni kell. Lehetőség szerint minden inputra és outputra el kell végezni, esetenként több input eredménye egy output. A szélsőértékek mindig fontosak. Minden esetre nem tudunk tesztesetet írni, túl sok lenne, szelektálni kell. Szelektálás a használat, a szélsőértéket tartalmazó tesztesetek alapján.

Boundary Value Analysis: A határértéket és az alatti és feletti szomszédot kell tesztelni. (3 eset) Ha a szomszédos Equivalence Class – nak ugyanaz a határértéke, akkor a tesztesetek átfedik egymást. Ha nincs valós határérték, akkor csökkenhet 2-re a tesztesetek száma. Itt a valós határértékeket lehet kombinálni, az invalid értékeket nem érdemes.

State Transition Testing: A szoftver működése és az output-ja nem csak az inputtól, hanem a futási sorrendtől és az eseményektől is függ. A függőségek történetét állapot diagrammal illusztrálhatjuk. Az állapot változásokat események váltják ki valamilyen funkció hívásával. Minden állapotban minden funkciót tesztelni kell. A ciklikus állapotátmenet diagramot szokás átalakítani állapot fává.

Jó rendszertesztnek. A tesztesetekhez ismerni kell: a kiinduló állapotot, az inputokat, a végállapotot, az elvárt viselkedést, azt az eseményt, ami kiváltja a változást, és a várt reakciókat. Elsősorban objektum orientált rendszereknél használják.

Cause – Effect Graphing and Decision Table Technique: Az előzőekkel ellentétben ez összefüggést keres az input és az output közt. Meg kell találni az okokat és azok hatásait, amit diagramban lehet ábrázolni. Az okokat mint feltételeket fogalmazzuk meg. A hatásokat és okokat logikai feltételek kapcsolják össze. A diagramot át kell konvertálni döntési táblába, ahol minden feltétel és minden hatás szerepel. Oszlopok száma = tesztesetek számával.

Use Case Testing: Kívülről szemlélteti a rendszert. Jó rendszer, integrációs és különösen elfogadási teszthez. A teszteseteknél minden lehetőséget le kell fedni. A tesztnek akkor van vége, ha legalább egyszer minden lehetőség lefutott.

Syntax test: Ha formális feltételeknek kell megfelelnie az inputnak. Például fordító, értelmező vagy protokoll vizsgáló programok. A szabályok alapján kell a megfelelőségi és a negatív tesztesetet elkészíteni.

Random test: Az input adatokat véletlenszerűen választjuk ki a tesztesethez, akkor, ha adott az input adatok statisztikai eloszlása. Így az eredményt fel lehet használni statisztikai előrejelzésekhez és megbízhatóság vizsgálathoz.

Smoke test: Célja egy minimális megbízhatóság ellenőrzése. A fő funkciókra koncentrál. Nézi, hogy nincs – e összeomlás vagy komoly félreműködés. Jó annak az eldöntésére is, hogy érdemes-e további tesztet végezni. Szoftver update – k első és gyors tesztje is.

White – box test

A white – box tesztelés betekintést enged a kódba is. A tesztelő ismeri a szoftver belső működését, a logikai felépítését a szoftvernek. Szinonimák: üveg – doboz teszt (glass – box), tiszta – doboz teszt (clean – box), vagy szerkezeti teszt. *Előnyei:* tudjuk, hogy milyen adatokkal tesztelhető a szoftver; segít optimalizálni a kódot; segít a felesleges sorok eltávolításában, melyek hibát eredményeznének a működésben, tudjuk, hogy van implementálva a program.

Hátrányai: költségnövelő, mivel a programozási jártasság feltétel; lehetetlen a kódot bitről bitre elemezni és az összes hibát megtalálni és azonnal ki is javítani. Magában foglalja az adatok és az információ áramlását, az irányítást, a programozási gyakorlatot, a kivételek és a hibák kezelését. A kód megírása után bármikor alkalmazható, mégis előnyösebb, ha a unit teszt ideje alatt használjuk. Teszteléskor érdemes hackerként gondolkozni, ha tudni akarjuk mitől lesz a szoftver biztonságos, vagy feltörhető. Első lépésként értelmeznünk és elemeznünk kell a kódot, a biztonság kérdése alapfeltétel egy szoftver esetében. Második lépésben készítsünk olyan teszteket, melyek kiaknázzák a szoftver lehetőségeit. Harmadik lépésként, a tesztelőnek ismernie kell a különböző eszközöket és technikákat, melyek egy white – box teszt esetében használhatók. White box tesztek fajtái:

Statement Coverage: Az utasítások állnak a teszt középpontjában. Először a forrás kódot folyamatábrába konvertáljuk, majd ezt vizsgálva készítjük el és futtatjuk a teszteseteket. 100% - os a lefedetteség, de ez gyakran nem is érhető el. Elég gyenge teszt. Bizonyos ágak kimaradhatnak a tesztelés során.

Branch Coverage: Az elágazások állnak a teszt középpontjában. A döntések minden ágát ellenőrizni kell (if, case, loops). Hiányzó utasításokat is felderíthet. Branch Coverage teszteli a feltétel minden ágát, de ha a feltétel összetett, akkor célszerű ezt külön is vizsgálni. A cél, hogy minden atomi feltételt megvizsgáljunk úgy, hogy annak eredménye legyen igaz és hamis is. Gyengébb teszt, mint Statement Coverage!

Path Coverage: A 100% Branch Coverage sem garantálja, hogy minden változata az elágazásoknak tesztelve van. Branch Coverage egymástól függetlenül vizsgálja a Branch-okat. A PC ezek összefüggését is szem előtt tartja, ha loop – ok is vannak a kódban, akkor ez még fontosabbá válik.

Mit alkalmazzunk? Függ:

- Tesztelt objektum fajtájától
- A dokumentációk formalizáltságától (teszteset generálás)
- Standardoknak megfelelés
- Tesztelő tapasztalata
- Ügyfél kívánságai

- Kockázatelemzés
- Továbbiak: idő, költségvetés, dokumentáció, előző tapasztalatok, stb.

A lényeg: mindig több fajta technikát kell alkalmazni, hogy a többféle szempontot minél jobban figyelembe tudjuk venni.

Unit test

Fejlesztő végzi, és nem a tesztelő! Nem debuggolás! A fejlesztő így megbizonyosodik arról, hogy a program azon része teljesíti – e a követelményekben leírtakat, illetve tényleg úgy működik, ahogy az elvárható. Azokat a jellemzőket foglalja magában, melyeknek teljesülnie kell a teszt során. A fejlesztők számára ez bátorítást jelent arra, hogy változtassanak a kódon anélkül, hogy végiggondolnák milyen hatással lehet ez a teljes programra. Mit tesztelünk unit tesztekkel: függvényeket, alprogramokat, a program kisebb egységeit. Lényegében azt vizsgálja, hogy a megírt függvény, rutin önmagában működik – e. Ha objektum orientált programozásról beszélünk, egy unitnak egy metódus felel meg. A unit teszt célja, hogy szétválassza a program egyes részeit, és megmutassa, hogy ezek külön – külön működnek. A tesztelésnek ebben a szakaszában a fejlesztő még gyorsan ki tudja javítani az esetleges hibákat. A tesztesetek függetlenek egymástól. Általában automatizált tesztet végeznek, de előfordul a manuális úton való ellenőrzés is. A unit teszt unalmasnak és időpocsékolásnak tűnhet. Sok türelmet és alaposságot követel a teljes fejlesztői csapattól a projekt kezdetétől a végéig. Tudatosan kell végezni, nem lehet minden bemenetet letesztelni, amivel a való életben is bombázni fogják a szoftvert. Ellenőrzi a szoftver robusztusságát, hatékonyságát, karbantarthatóságát.

Előnye, hogy a problémák még a fejlesztés ideje alatt jelentkeznek, így korábban javíthatóak, kevesebbe kerülnek. Elősegíti a változást illetve a változtatást, mert a programozó később is változtathat a kódon, és a már előre megírt automatikus teszttel ellenőrizheti, hogy a változtatás után még mindig működik – e az a rész, ez hasznos, például egy regressziós teszt esetében. Ha a változások következtében valamilyen hiba lépne fel, az azonnal javítható. A unit

teszt tervezésnél figyelnek arra, hogy azon a legkisebb egységen belül lefedjenek minden lehetséges útvonalat, még az ismétlődést is számba veszik.

Egyszerűsíti az integrálás folyamatát, eloszlatja a bizonytalanságot és használható a lentről – felfelé teszteknél (bottom up). Könnyebbé teszi az integrációs tesztet, ha előbb a részeket teszteljük, majd a részek egészét. Általános vitatéma, hogy szükség van – e integrációs tesztre, a unit tesztek hierarchiája miatt úgy tűnik, mintha az már integrációs teszt volna, de az emberi tényező nem pótolható, bár némely álláspont szerint az emberi tesztelés teljesen szükségtelen. Valójában ez a programtól függ, hogyan lett megírva, mire akarják használni. Az emberi, vagy manuális tesztelés nagyban függ a szervezeten belül megtalálható források mennyiségétől.

A unit teszt feltételez létező dokumentációt a rendszerről. A fejlesztők és a tesztelők ebből tanulják meg, hogy egyes funkcióknak hogyan kell működniük. A unit tesztesetek testesítik meg azokat a jellemzőket, amelyek elengedhetetlenek a sikerhez. A másik oldalról, ha egy dokumentáció nagyon részletes, akkor hamar válhat használhatatlanná, például sok változás történt a UI – on.

A unit tesztelés 6 szabálya:

1. először a tesztet írd meg,
2. soha ne írj olyan tesztet, ami elsőre, hiba nélkül lefut,
3. kezdd olyannal, ami nem működik,
4. merj valami egyszerűt tenni, hogy működésre bírd a tesztet,
5. haladj lépésről lépésre,
6. használj maketteket.

Félreértések a unit teszttel kapcsolatban:

1. tesztelőnek kell megcsinálni: Alapvető félreértés, ugyanis a program egy részéről van szó. A programozónak a saját munkáját kell ellenőriznie.

2. másik fejlesztőnek kell megcsinálni: miért? Az a baj vele, hogy a saját munkájukat kell ellenőrizni.

3. kezdőknek kell megcsinálni: más gondolkodás, a tapasztalat hiánya. Azt hiszem utóbbi is elég ellenérv.

A sikeres befejezés tekinthető az integrációs teszt bementének is, ami a következő lépés.

Integration test (Integrációs teszt)

A komponensek együttműködését vizsgálja, ellenőrzi a funkcionalitást, a megbízhatóságot és a teljesítményt. A unit teszt és a rendszerteszt között van a helye. 2 fő módszer szerint lehet elvégezni: fentről – lefelé irányuló, és lentről – felfelé irányuló teszttel, de ide sorolják még big bang – et és a backbone – t is. Inputjának nevezhetnénk a unit tesztet, nagyobb egységeket képez belőle, és outputként megjelenik a teszt eredménye, mely készen áll a rendszertesztre. Minden komponenst lépésről – lépésre kell összekapcsolni. Célja, hogy ellenőrizze a funkcionalitást, a megbízhatóságot, és minden lényeges, a követelményekben szereplő adatot, és a különböző részeket. Ezeket a részeket black box teszttel ellenőrzik, különböző bemenő paraméterekkel. Tesztesteket készítenek, hogy minden komponens megfelelően legyen ellenőrizve. Korlátot jelent, hogy nincsenek feltételek szabva, eltekintve a tervezési elemek végrehajtásának megerősítését, melyet nem tesztelünk. Ha egy külső rendszer kapcsolatát vizsgáljuk, akkor azt hívjuk „higher integration test”-nek vagy „integration test in the large” – nak. Ilyenkor csak a kapcsolat fele áll a mi ellenőrzésünk alatt. A unit teszt drivereit használja.

Milyen sorrendben integráljuk a komponenseket?

1. A kockázatok és a rendszer architektúrának megfelelően.
2. A komponensek nem egyszerre készülnek el, de a tesztelők nem lehetnek munka nélkül.
3. Gyakorlatban, ahogy elkészül a komponens, úgy illesztjük a többihez és teszteljük.
4. Minél előbb kezdjük az integrációs tesztet, annál több komponenst kell egy vázzal helyettesíteni.

Fentről – lefelé (top down)

A rendszer magas szintű komponenseit a tervezés és az implementáció befejezése előtt integráljuk és teszteljük. Lényegében darabokra szedi a rendszert, hogy betekintést kaphasson az alrendszerekbe. Könnyebben megtalálható a hiányzó kapcsolat.

Lentről – felfelé (bottom up)

Először az alacsony szintű komponenseket integráljuk és teszteljük. A rendszer elemei nagyon részletesen vannak specifikálva. Később ezeket összekapcsolják, egészen addig folytatják, amíg teljes egészet nem alkotnak. Mag modellnek is nevezik, mivel kis darabokból indul, és nagyra nő. A tesztelési folyamat egészen addig tart, amíg a hierarchia legfelső fokán lévő elemig el nem jutunk. Előnye, hogy könnyebb a teszt eredményeit jelenteni.

Big bang

Mindent egy lépésben integrálunk.

Időt lehet vele spórolni. Ha nincs megfelelően megírva a teszteset, akkor egyrészt a teljes folyamat sokkal bonyolultabb lesz, mint eredetileg látszott, másrészt így a tesztcsapat sem éri el a célját.

A legtöbb modul úgy áll össze, hogy megformáljon egy rendszert, vagy annak egy alapvető részét, majd ezeket használja integrációs tesztre.

Egyik része a használhatósági modell teszt (usage model test). Szoftver és hardver esetében is használható integrációs tesztre. Alapja, hogy a felhasználó úgy indítja el a tesztet, mintha már élesben használná. Ennél a módszernél maga a környezet a teszt, míg az egyes komponenseket használatba véve próbálják ki. Feltételezi, hogy az egyes komponensek nem hibamentesek, ebben rejlik a sikere. A cél, hogy ne ismételjük meg azokat a teszteket, melyeket a fejlesztők már elvégeztek, hanem inkább azokkal a problémákkal foglalkozzunk, melyek a komponensek közötti együttműködésből származnak adott környezeten belül. Minél valóságosabb környezetet tudunk létrehozni, annál hatékonyabb és pontosabb lesz a teszt eredménye.

Backbone

A fokozatosan elkészülő komponensek egy vázba illeszkednek. Sorrend nem fontos, de kell egy erőteljes váz.

System test (Rendszerteszt)

Teljes, integrált teszt, mellyel megbecsülik, hogy a szoftver vagy hardver mennyire felel meg a követelményekben leírtaknak. Nem szükséges hozzá a program belső ismerete, fekete doboz teszttel végzik az ellenőrzést. Bemenetnek az integrációs teszt eredménye tekinthető, ha az sikeres volt, akkor kezdődhet a rendszerteszt. A rendszerben, mint egészben keresi a hibákat. Felfedező teszt fázis is egyben, ahol a lényeg a nem megfelelő viselkedés elérése, és a teszt nem csak a program külsejét ellenőrzi, hanem azt is, hogy megfelel – e az ügyfél elvárásainak. Az elvárásokat egy dokumentum tartalmazza: Functional Requirement Specification(s) (FRS), vagy System Requirement Specification(s) (SRS) tartalmazza. A rendszerteszt magában foglalja a terheléses tesztet is (stress test).

Néhány példa arra, hogy rendszer teszt ideje alatt milyen teszt típusokat érdemes használni:

Stress test (Terheléses teszt)

A program stabilitását, robosztusságát, használhatóságát, hibakezelését teszteli nagy megterhelés alatt. Cél, hogy a szoftver ne omoljon össze elégtelen körülmények között (például: kevés memória, vagy kevés hely van a merevlemezen). Gyakran hivatkozik olyan tesztekre, melyek nagy hangsúlyt fektetnek a terhelésre, a robosztusságra, a használhatóságra, a hibakezelésre különösen nagy megterhelés alatt. Nem várják el egy rendszertől, hogy túlterhelés alatt is megfelelően működjön, de elfogadhatóan igen (például ne vesszenek el az adatok).

Smoke teszt (Sanity test)

Füst tesztnek is nevezik, a főbb funkciók működését ellenőrzi. Mielőtt a tesztcsapat komolyabban is elkezdene foglalkozni a programmal, az előtt használják, hogy megállapítsák érdemes – e elkezdeni a munkát. Ha a program megbukik, akkor visszakerül a fejlesztőkhöz. A hibák felfedezésének leghatékonyabb módja. Gyakran automatizált tesztekkel ellenőrzik.

Elsősorban a piacra kerülő dobozos szoftvereket tesztelik vele. Általában a fejlesztők végzik, hogy lássák, a szoftver működik a fejlesztés különböző szintjein. Átfogónak, és aprólékosnak kell lennie, hogy a teljes szoftvert le lehessen vele tesztelni, és a „showstopper”, vagyis a legkomolyabb hibákat időben megtalálni. A teszt néhány ismérve:

- A „build” több kisebb programból áll, melyeket lekódoltak és integráltak. Minden adat, függvény, részmodul és minden, ami a működéshez szükséges benne van a buildben.
- A tesztek gyűjteménye rendszeren hajtódik végre, hogy a hibákat felfedezzék, és a működést ellenőrizzék.
- Több buildet integrálnak, és tesztelnek le smoke teszttel.

Usability test (Használhatósági teszt)

Felhasználó – barát tesztelés. Szubjektív módja a tesztelésnek, nagyban függ attól, hogy ki a szoftver végső felhasználója. A felhasználókkal készített interjúk, beszélgetések, a használatról készült felvételek, a felmérések eredményei használhatóak. A programozók, illetve a tesztelők nem tekintendők végső felhasználóknak.

Compatibility test (Kompatibilitási teszt)

Azt vizsgálja, hogy hogyan teljesít a szoftver különleges hardver vagy szoftver vagy operációs rendszeri vagy hálózati környezetben.

Hibakezelési teszt (error handling test)

A megérzésre, a nyomozásra, az alkalmazásra és a kommunikációs hibákra vonatkozik. Ha hiba történik log file készítése kötelező az alkalmazás leállításával, ha nem állítjuk le az alkalmazást, akkor javítsuk ki őket.

Jól informált emberek csoportja kell, hogy megjósolja, mi mehet tönkre egy alkalmazásnál. Szükséges, hogy ezek az emberek összegyűjtsék, és integrálják tudásukat, vizsgálódjanak, és nyomon kövessék a hibákat.

Volume test (Térfogat teszt)

A nem funkcionális tesztek csoportjába tartozik, melyeket gyakran felcserélnek és/vagy félreértenek. Azt a tesztelést jelenti, amikor az alkalmazást bizonyos mennyiségű adattal bombázzuk. Ez az adatmennyiség lehet adatbázis méretű, vagy annak a file – nak a mérete, mely a teszt tárgyát képezi. Például: ha egy adatbázis mérettel szeretnénk tesztelni a szoftverünk, akkor először kiterjesztjük az adatbázisunk méretét, majd azzal végezzük el a tesztet. A másik lehetőség, hogy csinálunk egy megfelelő méretű file – t (.xml, .dat, mindegy), és ezzel teszteljük a szoftvert.

Load test

Az alkalmazást szokatlanul nagy leterhelésnek tesszük ki, például egy weboldal esetében azt vizsgáljuk, hogy mennyi idő után nem válaszol, vagy egy több felhasználós szoftvernél mennyi felhasználót képes még kiszolgálni. Használhatunk szimulátorokat, melyeknek megadható a felhasználók száma, avagy ha ismerjük a felső határt, akkor annak a 1,5 – 2 – szeresével tesztelhetünk.

Egy rövid ideig teszteltem egy regisztrációs szoftvert. Nem volt más feladat, csak a látogató nevét, és a látogatott nevét bekérni, havonta több tízezer regisztrációra számoltak, a lényeg, hogy kaptam hozzá egy tool – t is, melynek csak azt kellett megadni, hogy hány regisztrációt készítsen.

A Stress test – hez hasonló, de ott inkább a szélsőértékeket nézzük, itt viszont a konkrét terhelésről van szó. Load test esetében nagy szerephez jut a felhasználók száma, míg stress test – nél a tárhely, az adatok.

Ad hoc tests (Exploratory test)

Az ad hoc egyik nagy előnye, hogy kitűnően használható felfedezésre. A követelményeket hiába olvassuk el, nem adnak semmiféle támpontot arról, hogy is működik a program. Az ad hoc – al megtalálhatóak a lyukak a stratégiában, és az alrendszerek közötti kapcsolatokat is más megvilágításba helyezi, melyek egyébként lehet, hogy nem válnának láthatóvá. A tesztelés teljességét vizsgálja,

így ha van még lefedetlen terület, hozzáadható, és arra is lesz tesztesetünk. A tesztelési ciklus során bármikor végezhető.

Nem érdemes használni, ha ismételni kell a teszteket, ebben az esetben inkább dokumentáljuk, és automatizáljuk a teszteseteinket. Nem javasolt a release teszt ad hoc módszerrel. Ez alapos dokumentációt és futtatást igényel.

Nem tervezett folyamat, inkább a tapasztalat vezérli. Mielőtt bármit is tennénk, vegyünk elő papírt és ceruzát, írjuk le, mi az, ami érdekel a program futása során. Legyünk részletesek! Mit szeretnénk tesztelni, milyen eredményeket várunk. Azt is írjuk le, hogy hol jelentkezhetnek hibák, és ezeket hogyan szeretnénk előhozni. Egy jó ad hoc tesztelő képes megérteni a tervezés céljait, és a különböző funkciók követelményeit is.

Regression test

Az a teszttípus, amely olyan visszafejlődések után kutat a szoftverben, melyeket még nem fedtek le. Elterjedt módszer, hogy a korábban futtatott teszteseteket újra lefuttatjuk, és ellenőrizzük, hogy a korábban javított hibák nem jönnek – e elő újra. Ez nem azt jelenti, hogy ha van például 5 lezárt projektünk, és 3000 javított hibánk, hogy kezdjünk neki és nézzük át mindet, teszteljük újra. A tapasztalat azt mutatja, hogy a szoftver fejlődésével együtt párhuzamosan jelennek meg a már korábban javított hibák. Az egyik ok az lehet, hogy a javítás eltűnik a hanyag ellenőrzés miatt, a másik ok a javítás „törékenysége”, azaz, ott lett megjavítva, ahol előjött, de általában a szoftverben nem történt változtatás, így ugyanaz a hiba újra jelentkezhet a szoftver bármelyik életciklusában. Használható manuális és automatikus tesztként is. Gyakoriság terén, ez változó, van, ahol hetente futtatják a teszteseteket, máshol havonta, vagy minden alfa teszt végén.

Maintenance test

Azonosítja a felszerelés, az eszköz hibáit, vagy megerősítést ad afelől, hogy a javítás tényleg elkészült. 3 szinten végezhető el, vagy a komponens szinten, vagy a szerviz szinten, vagy a rendszer szinten.

Szerviztípusok, Hol? Használjuk:

- Megelőző szerviz – a már létező rendszerben változtatunk azért, hogy elkerüljük a működés közbeni meghibásodásokat.
- Javítási szerviz – olyan problémák javítása, melyek a használat során lépnek fel.
- Tökéletesítő szerviz – olyan módosítások, melyek javítanak a megbízhatóságon, a biztonságon, a hatékonyságon, és költséghatákonnyá teszik a műveletet.
- Alkalmazkodó szerviz – olyan módosítások, melyek a környezetváltozással merülnek fel az elvárásokban is.

Acceptance test (Elfogadási teszt)

A legutolsó fázis, amikor az ügyfél teszteli, használja a programot. Ő ellenőrzi, hogy teljesülnek – e a követelmények, hogy a saját környezetében is működik – e a szoftver. A cél már nem a hibák megtalálása! Ez időre már elvárható, hogy a szoftver hibáit javították. Ebben a fázisban már nem változtatnak a szoftveren, nincs Change Request (CR), ezért a fejlesztés során érdemes prototípusokat bemutatni. Magasabb szintű regressziós teszteket is igényel. Lehetőségünk van felülvizsgálni, ellenőrizni a rendszer hatékonyságát és integritását. Ha bármilyen hézagot, vagy problémát találnak, azt javítják.

Nem minden nézőpont szerint tartozik a rendszertesztek közé. Az egyik álláspont, hogy ez is rendszerteszt, hiszen tényleg a teljes szoftvert ellenőrzik, valós környezetében, ahol használni is fogják, egy másik felfogás szerint, azonban ez már túlmutat rajta. Kikerült a fejlesztők, és tesztelők keze közül. Nevezik UAT – nek (User Acceptance Test), vagy CAT – nak (Customer Acceptance Test) is.

Automatikus tesztek

Szükség van rájuk? Mindenképpen, időt és energiát spórolhatunk meg velük. Elsősorban smoke tesztekre használjuk, vagy kisebb egységeket automatizálunk. Előnye, hogy a monoton, hosszadalmas, sok ismétlést tartartalmazó munkát nem manuálisan kell elvégezzük. Hátránya, hogy hiányzik belőle az emberi képzelőerő, a kreativitás. Nem a teszt lefuttatása a sok, hanem a kódot megírni hozzá.

A tesztelés nagy része még mindig manuálisan történik, és ráérzés vezérelt. A teszteléshez elvárható lenne valamilyen eszköz használata. A tesztelést segítő eszközök az 1970 – es években kezdtek megjelenni.

Az automatikus teszt nem más, mint tapasztalatok, tesztesetek és forgatókönyvek halmaza, melyek célja, hogy eldöntsék, hogy az implementáció megfelelően működik – e. Ideális estben a teszt a specifikációk megadása után teljesen automatikus.

Agile testing

[4]Az agilis teszt eltér a hagyományos teszteléstől. Az emberek annyit tudnak róla, hogy a tesztelés és a fejlesztés párhuzamosan folyik. Nem csak ebben különbözik a hagyományos szoftverteszteléstől, lássuk részletesebben.

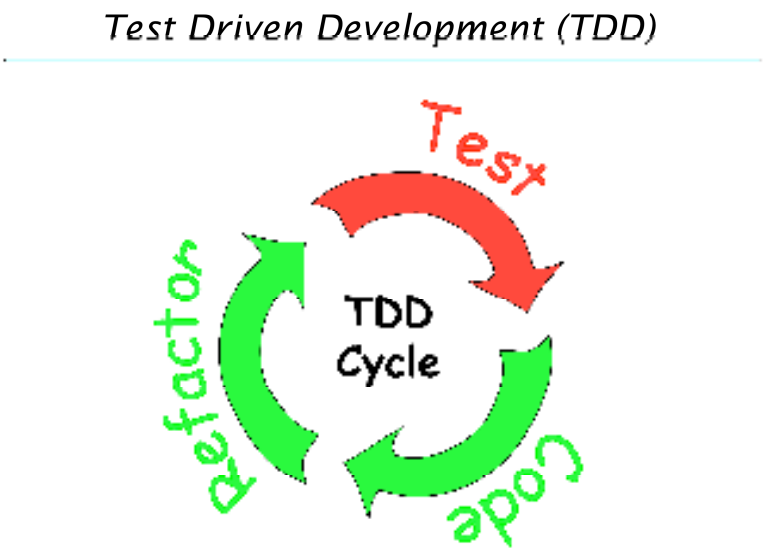
Amilyen hamar csak lehet az ügyfél szempontjai szerint tesztel. Hagyományos tesztelés minőségi gátnak felel meg, míg az agilis tesztelés a minőség „kapusa". Felelőssége, hogy rossz szoftver ne kerüljön a piacra. Az agilis csapatok a kezdetektől úgy építik a szoftvert, hogy a teszteléstől megkapják a szükséges visszacsatolást. Gyakoriak a meetingek, melyeken a talált hibák prioritását, fontosságát beszélik meg. Az agilis teszt folyamatos, nincs beleintegrálva a folyamatba. A folyamatos tesztelés az egyetlen út, hogy biztosítsuk a folyamatos fejlődést. Agilis tesztelésnél nem csak a tesztelőn van a tesztelés felelőssége, hanem az egész csapaton. Mindenki érdeke, hogy a tesztelés elkészüljön. Rövidebb a visszacsatolási idő. Hagyományos tesztelésnél egy visszacsatolási idő akár hónapokban is mérhető (a kód megírásától a teszt lefuttatásáig tartó időt foglalja magában), agilis teszt esetében ez egy hetet, vagy néhány napot tehet ki. A cél, hogy megtaláljuk az arany középutat a belső elvárások és a saját magunk által támasztott követelmények között. Például, nem biztos, hogy egy Commodore 64 – es gépen, vagy Windows 2000 operációs rendszeren futnia kell adott szoftvernek. A hibákkal teli szoftvert nehezebb tesztelni. Ne használjunk túl sok dokumentációt, például a részletes tesztesetek helyett használjunk rövid, újrahasznosítható listát. Foglalkozzunk a lényeggel, ne

[4] Forrás: Elisabeth Hendrickson: Agile Testing Nine Principles and Six Concrete Practices for Testing on Agile Teams, 2008

vesszünk el a részletekben. Próbáljunk minél kisebb méretű dokumentációt készíteni, ne a Word – öt részesítsük előnyben, inkább használjunk valamilyen tool – t vagy keretrendszert, például FIT/Fitness. Agilis tesztelésnél ha azt mondjuk valamire, hogy kész van, az azt jelenti, hogy implementáltuk ÉS leteszteltük. Egyszerre definiálja a tesztet és a követelményeket így tisztábbak a feltételek, és a cél megosztottá válik.

A gyakorlat:

- Automatizált a Unit és az Integration teszt: gyorsan lefut, izolálja a különböző elemeket, gyakran futtatják, programozók írják.
- Tesztvezérelt fejlesztés (Test Driven Development TDD): tesztelés -> kódolás -> újragondolás, majd újra indul a kör.

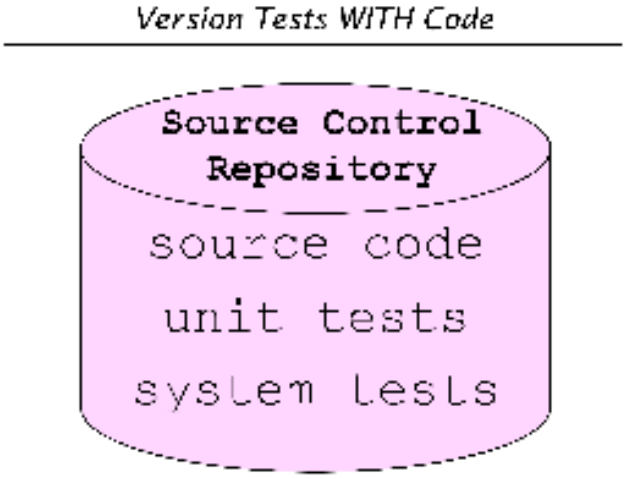


1. ábra: A tesztvezérelt fejlesztés

- Automatizált rendszer szintű regressziós teszt: a csapat tagjai együttműködnek, elejétől a végéig teszt, futtatható követelményeket reprezentál, és ellenőrizhető viselkedést.
- Elfogadott tesztvezérelt fejlesztés: megbeszélés a részvényesekkel, az igények felmérése (nem funkcionális tulajdonságokra is vonatkozik, úgy mint, megbízhatóság, biztonság, stabilitás), szűrés arra, hogy mi alkossa az elfogadási tesztet, tesztvezérelt fejlesztés

alapján történik, így íródik meg a kód, mutassuk meg a részvényeseknek a kész feature – t, majd várjuk a visszacsatolást.

- Felfedező tesztelés: egyszerűen tanulni a szoftverről, megtervezni a teszteket, és lefuttatni ezeket az előző eredményeit felhasználva.
- Verziótesztek kóddal (Version Tests With Code):



2. ábra: Verziótesztek kóddal

Statikus és Dinamikus tesztelés

Statikus teszt esetében a munka, amit elvégzünk sztenderdek halmaza. Vonatkozhatnak a kódolásra, az integrációra, a fejlesztésre. Nem futtatjuk a szoftvert, nem részletes teszt! Ez a verifikációs folyamat. Általában a kódot ellenőrizzük szintaktikai hibák után kutatva, vagy a dokumentációt olvassuk át. A fejlesztő dolga, hogy ezeket megtegye. Automatizálható folyamat.

A dinamikus tesztelés egyet jelent dolgozni a szoftverrel, inputokat megadni, ellenőrizni, hogy a helyes eredményt kaptuk – e. Ez a validációs folyamat. Dinamikus tesztre használható a unit, az integrációs, a rendszerteszt, a regressziós, az elfogadási teszt.

Minél hamarabb derül ki egy hiba, annál kevesebbe kerül kijavítani.

Párhuzamos tesztelés

Azt jelenti, hogy egy időben tesztelünk több szoftvert, illetve egy szoftver több részét. Párhuzamos tesztelés nem jelent dupla annyi ráfordítást, sem időben, sem pedig pénzben, de az eredmény sokkal jobb lehet.

Objektum orientált tesztelés

A rendszernek nincs teteje, az alrendszerekbe integrált objektumok lazán kapcsolódnak egymáshoz. Törekedni kell arra, hogy a megoldandó feladatot olyan kisebb egységekre osszuk fel, melyeket könnyű megoldani. Integrációs és rendszerteszt esetén is jól alkalmazható. Lehetőség van csonkok definiálására is, melyek helyettesítik a nem tesztelt program részeit. Objektum orientált technika a wrapper osztályok használata, melynek oka, hogy megkönnyítik a dokumentálást is. Egy wrapper osztályt létrehozunk minden osztályhoz, így a deklarációk megadásával elérhetők lesznek a tesztelt osztály belső változói. A dokumentáció esetében pedig a wrapper osztályt a tesztelt osztállyal azonos interfésszel hozzuk létre. Teszteléskor mindig a wrapper osztály függvényeit hívjuk majd meg. Ez az a tesztelési eljárás, ahol a fehér doboz tesztelést ki kell terjeszteni (hogy lefedje a nagyobb objektumokat), és alternatív megközelítési módokat kell találni az integrációs teszteléshez. Az Objektum orientált tesztelés szintjei:

- az objektumokhoz kapcsolódó egyedi műveletek tesztelése (fekete/fehér doboz módszerrel is tesztelhetünk),
- objektumosztályok tesztelése (fekete doboz módszer, de ki kell terjeszteni a műveletsorozatokra is az ekvivalencia osztályozást),
- objektumok egy csoportjának tesztelése (fentről lefelé módszer nem alkalmazható),
- objektum orientált rendszer tesztelése (verifikáció és validáció hasonlóan történik, mint más rendszereknél).

A tesztnek tartalmaznia kell:

- Az objektumokhoz kapcsolódó összes művelet különálló letesztelését.
- Az objektumokhoz kapcsolódó összes attribútum beállítását és vizsgálatát.

- Az objektum összes lehetséges állapotának vizsgálatát: az összes olyan eseményt szimulálni kell, ami állapotváltozást okoz az objektumban.

Objektum orientált tesztelésen belül is megkülönböztetünk 3 különböző módszert. Az egyik *a forgatókönyv* vagy használati eset alapú tesztelés. Lényege, hogy a tesztek az esetek leírásain és objektumcsoportokon alapulnak, melyek kiegészíthetők együttműködési és szekvencia diagramokkal. Ez gyakran a leghatékonyabb, először a leggyakoribb forgatókönyv kerül tesztelésre, így a legtöbb tesztelési munkát a leggyakrabban használt részekre lehet fordítani. A diagramok által meghatározhatók a kimenetek. Másik lehetőség a *száltesztelés*. A rendszer egy sajátos bementre adott válaszának tesztelésén alapszik. Az objektum orientált rendszerek általában eseményvezéreltek, azaz meg kell határozni, hogy az események feldolgozása hogyan halad keresztül a rendszeren. A harmadik az *objektum együttműködési* teszt. Együttműködő objektumcsoportok esetén módszer – üzenet útvonalak azonosítása.

5. Dokumentáció

Mint a tesztelés más részére, erre is van szabvány, ez az IEEE 829. Szó lesz tesztesetekről, teszttervről, vagy koncepcióról. Általában a dokumentumokról elmondható, hogy sablon használatával egységesíthetőek. A legrosszabb eset, ha egyáltalán nem készül dokumentáció.

A koncepció során megtervezzük a tesztelés folyamatát. Ez magában foglalja, hogy mit tesztelünk, hány ember áll rendelkezésre, mi az, amin tesztelünk, mi az, amit nem tesztelünk, milyen teszteket hajtunk végre (regressziós, unit, stb.). A tervnek az is fontos része, hogy mikor mondhatjuk egy termékről, hogy működik, melyek a teszt vége feltételek, illetve kik végezték a teszteket, kik a felelősök.

Minden tesztesetet, eredményt, folyamatot dokumentálni kell!

A tesztelő döntése, hogy mi kerüljön bele a dokumentációba. Ha kérdésekre akarunk válaszokat adni könnyebb megírni a dokumentációt. Ilyen kérdések lehetnek: mi a csapat küldetése, mi a célunk a termék tesztelésével? Ha a dokumentáció nem tud segítséget nyújtani, akkor nem ér semmit. További kérdés lehet, hogy a dokumentáció, amit elkészítesz egy termék, vagy eszköz? A termékért fizetnek, az eszköz házon belül használható, kevésbé kell nyomon követni, frissíteni. Milyen gyorsan változik a design? Ha gyorsan, akkor ne írjunk részleteket. A dokumentációnak ellenőrizni kell a tesztelendő projektet? Például, most milyen lépés következik? Mikor készüljön el a dokumentáció? A követelményekkel együtt, amikor még nem ismerjük a szoftvert, vagy majd később, a tapasztalatok alapján? Ez lényeges szempont például tesztesetek írásánál. Megírhatjuk őket a követelmények alapján is, de ekkor még nem tudunk túl sokat, így ha elkészül is, valószínű, hogy változtatni kell majd a teszteseteken. Kik olvassák a dokumentációt? A tesztcsapat, az ügyfél? Ettől függ ugyanis, hogy mennyire fontosak, de semmiképpen se legyen túl részletes, legyen egyszerű és átlátható. Ki, vagy kik felügyelik a dokumentációt? Milyen mértékben kellene a teszt dokumentációnak támogatnia a nyomon követhetőséget és a projektek állapotáról szóló jelentést, valamint a teszt folyamatát? Kell-e dokumentálnunk mindent? Mire használható a dokumentáció? Mindenképpen dokumentálni kell a

projekt menetét, a felvett hibákat. Találhatunk – e hibákat, ha csak a teszteseteket használjuk? Szerintem nem, főleg ha ezek lépésről – lépésre, nagy részletességgel vannak megírva, és a csapat tagjai már oda – vissza ismerik a lépéseket. Használni kell a képzelőerőnket, és ha találunk egy hibát, akkor azt pontosan leírni, hogy hol, mit csináltunk, milyen környezetben. Ha elkészült egy teszteset meg lehet mutatni a fejlesztőknek. Ez két okból is hasznos lehet, egyrészt, mivel ők jobban ismerik a szoftvert, így hozzá tehetnek még ha úgy gondolják hiányzik valami, másrészt találhatnak benne ésszerűtlen, vagy felesleges dolgot, amit nem érdemes tesztelni, vagy a teszt eredménye nagyon nyilvánvaló.

A dokumentáció és az új tesztelők: olvastassuk el a dokumentációkat, a követelményeket, a user manual – kat, a teszteseteket. Gyakoroltassuk az írást. Írjanak minél több bug riportot, persze az elején felügyelettel, olvassák el a régieket, vegyenek részt a tesztesetek írásában is.

A teszt dokumentációk frissítése: tudjuk – e biztosítani, hogy ha változik a szoftver, akkor vele változik például a teszteset és a terv is? Tudjuk, de ez többlet idő, ami a teszteléstől vonódik el. Minden tesztesetnek legyen meg a gazdája, a felelőse, így megelőzhető, hogy ha változás következik be, akkor a dokumentációt nem frissíti senki.

Segít a teszt dokumentáció abban, hogy észrevegyél egy félkész részt a programban? Egy olyan területen, ahol van néhány hiba, ott több hiba is lesz, vagyis, alaposabban kell ezeket a részeket tesztelni. Ha itt már megfixálták a hibákat, lehet, hogy a program másik része lesz megbízhatatlan.

Az *IEEE 829* – es szabvány:

- alapszabvány tesztelés dokumentálásához,
- semmi sem kötelező ebben a szabványban, ez egy irányadás,
- a szabvány szerkezetet, keretet, és definíciókat biztosít,
- alapjául szolgál azon dokumentációknak, melyek cég és cég között „vándorolnak”,
- a szabvány eléggé rugalmas,
- „Guns don’t kill people, people kill people.” – Ne fogd a szabványra, hogy miatta rontottad el a tesztet, nem a szabvány tesztel, hanem te,

- néhány jellemző:
  - szolgálatkész mentalitás: „csináld, amit a terv mond”, másra ne figyelj,
  - nincsenek benne útmutatók, javaslatok, hogy mikor mit tételezzünk fel,
  - nincs benne nyilvánvaló tudatosság, és a költségekről sincs szó, ami időt a dokumentációval töltünk, azt inkább tesztelésre kellene fordítani,
  - a szabvány kihangsúlyozza a dokumentáció mélységét, minél mélyebb, annál jobb,
  - nem lehet eldönteni, hogy egy teszteset az jó vagy rossz,
  - a dokumentációk fenntartásának költségei óriásiak: nem csak egy helyen kell változtatni a dokumentáción, ha változik a szoftver,
  - automatikus teszt paradigmák, melyek magában foglalnak generált és véletlenszerűen kombinált teszt adatokat idegennek tűnnek a szabvány számára,
  - egy túl ambiciózus terv többet árthat, mint használ, pl.: cég megállapodik, hogy dokumentálják a teszt folyamatát, illetve erre egy sablont használnak, majd feladják ezt azért, hogy a valódi munkát el tudják végezni időben, eredménye: szerződésszegés ÉS hibás termék.

Összegezve: akkor használjuk a szabványt, ha a tesztcsoporton kívül más is használja a dokumentációkat; érthetőbb és elfogadottabb. Ha csak a tesztcsapat használja a dokumentációkat, akkor elég egy sablon, amit mindannyian használnak.

6. Szabványok, modellek

IEEE829

Dokumentálásra vonatkozó szabvány. Részletesebben lásd 5. fejezet Dokumentáció.

ISO 9126

Nemzetközi szoftver minőségügyi szabvány. Egyrészt azon emberek miatt jött létre, akik negatívan hathatnak a szoftver fejlesztési folyamatára, másrészt a szakemberek sokáig hiányoltak egy olyan szabványt, mely egyedi, egyértelműen határozza meg a szoftvertermék minőségi jellemzőit. Miután letisztáztuk a prioritásokat és átváltottuk ezeket mérhető mennyiségekre, a szabvány megpróbálja fejleszteni a projekt céljait. 4 részre osztható:

1. minőségi modell
2. külső metrikák
3. belső metrikák
4. minőség a használatban metrikák.

Minőségi modell tartalmazza:

Funkcionalitás: a feladatok halmazának létezése, és ezek tulajdonságai, részei: alkalmasság, hitelesség, együttműködés, szolgálatkészség, biztonság.

Megbízhatóság: a szoftver azon képessége, hogy képes fenntartani működésének szintjét, részei: érés, visszaállítás, hibatűrés.

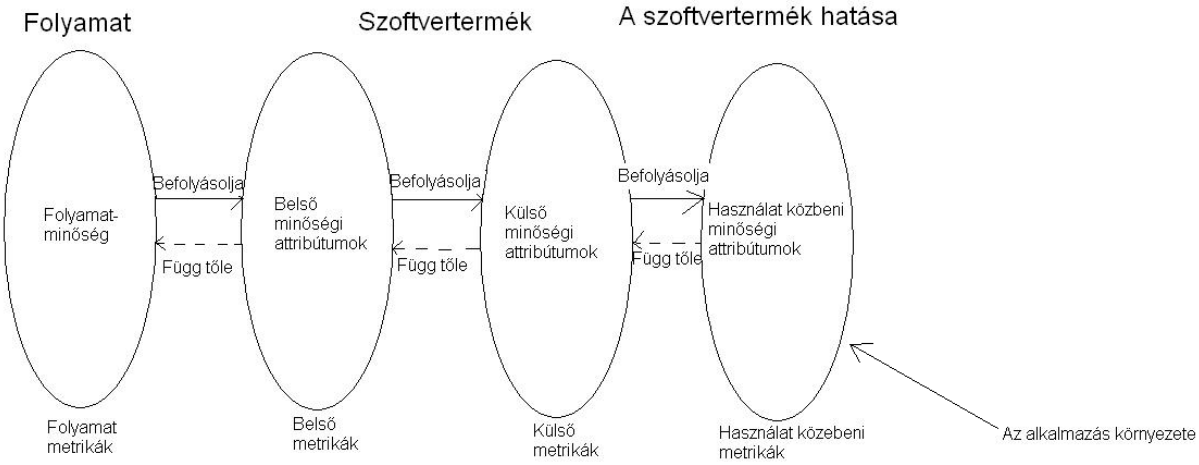
Használhatóság: a működéshez szükséges erőfeszítés, részei: tanulhatóság, megérthetőség, működőképesség.

Alkalmasság, hatékonyság: a működések közötti kapcsolat, részei: időbeli és forrásbeli viselkedés.

Fenntarthatóság: az erőfeszítés, mely ahhoz kell, hogy részletes módosításokat hajtsunk végre, részei: stabilitás, elemezhetőség, változtathatóság, tesztelhetőség.

Hordozhatóság: a szoftver átvihető legyen egyik környezetből a másikba, részei: telepíthetőség, áthelyezhetőség, alkalmazkodóképesség, alkalmazhatóság, megfelelősség.

Mindegyik tulajdonságnak vannak attribútumai, ezek azért nincsenek konkrétabban definiálva, mert mindegyik szoftver más és más, így mindegyikre más attribútum értékek érvényesek. A szabvány keretet biztosít a szoftver minőségi modelljének meghatározására, de minden vállalat pontosan meghatározhatja a saját modelljét. A külső és belső metrikák közötti különbség, hogy a belső metrikákhoz nem kell futtatni a szoftvert, a külső metrikákhoz pedig igen. A szabvány megkülönbözteti a meg nem egyezésből adódó hiányosságokat és a nem tervezett használati követelmények be nem teljesülését, ahol a meg nem egyezésből adódó hiányosságok a nem tervezett használati követelmények része, hasonlóan a validáció és verifikáció problémájához.



3. ábra: ISO 9126

A szoftvertermékre vonatkozó általános elvárások:

- Mindig fontos, és elvárható, hogy a szoftver stabilan működjön, és „elfogadható” válaszidőket produkáljon, relatív, hogy mit tekintünk elfogadhatónak.
- A rövid válaszidők és a módosíthatóság gyengítik egymást.
- Tapasztalat szerint, bizonyos rendszereknél fontos a folyamatos rendelkezésre állás és a visszaállíthatóság, például bankok esetében.

Általános elvárásokat, megállapításokat tapasztalatok alapján tesznek, ezeket egészítik ki a konkrét esetek, rendszerek, megfigyelések. Ebben az esetben a jellemzők módszeres vizsgálata segíthet, ekkor figyelembe

vesszük az üzleti folyamat jellemzőit, kik a felhasználók, melyek a szoftver jellemzői.

IEEE1028

Áttekintés

Ez a szabvány 5 szoftvertesztelési típust határoz meg, a végrehajtáshoz szükséges folyamatokkal együtt. Nem határoz meg az ellenőrzés szükségességének meghatározásához tartozó folyamatokat, és nem határozza meg az ellenőrzés eredményeinek elrendezését. Ezek a típusok magukban foglalják a menedzsment, a technikai, a vizsgálati, a felfedező és az audit ellenőrzéseket. A szabvány alkalmas arra, hogy az IEEE többi szoftverfejlesztésre vonatkozó szabványával együtt, illetve önállóan is alkalmas legyen szoftvertesztelési szabványnak. Utóbbi esetben a menedzsmentnek meg kell határoznia azokat az eseményeket, melyek megelőzik és követik a szoftvertesztelést. A szabvány alkalmas a projekt és konfigurációs menedzsmentre, a minőség biztosítására, a verifikáció és a validáció támogatására. Elfogadható követelményeket tartalmaz szisztematikus szoftvertesztelésre, melyek a következők:

1. csapat részvétele,
2. az ellenőrzés eredményeinek dokumentálása,
3. dokumentált folyamatok az ellenőrzés levezetésére.

Azok az ellenőrzések, melyek nem tesznek eleget a követelményeknek, nem szisztematikus ellenőrzések. A szabvány célja nem az, hogy megtiltsa a nem szisztematikus ellenőrzéseket. Minden típus tartalmaz kikötéseket, melyek a következők:

- Bevezetés: leírja az ellenőrzés, tesztelés tárgyát, vagy tárgyait, és áttekintést biztosít a folyamatokról.
- Felelősségek: meghatározza a szerepeket és felelősségeket.
- Input/Bemenet: a szükséges követelmények.
- Belépési feltételek: azon követelmények, melyeket a tesztelés előtt meg kell vizsgálni, ezek a jóváhagyások, és az indítási események.

- Folyamatok: tervezés, folyamatok áttekintése, előkészítés, vizsgálódás, vagy értékelés, vagy az eredmények felvétele, újradolgozás, nyomon követés.
- Kilépési feltételek: azon követelmények, melyeket a tesztelés után kell megvizsgálni, de az előtt, mielőtt teljesnek nyilvánítanánk a tesztet.
- Output/Kimenet: a teljesítések minimális halmaza, melyeket az ellenőrzés során produkálni kell.

Széles körben használt szabvány, magában foglalja például az anomáliák, az auditok jelentését, a buildelés folyamatait, a szerződéseket, az ügyfél panaszait, a kockázati feljegyzéseket, a release terveket, a forráskódot, a menedzsmentek terveit.

Menedzsment ellenőrzés

Célja, hogy ellenőrizze a folyamatot, meghatározza a terveket, az időbeosztásokat, megerősítse a követelményeket, és ezek kiosztását, megbecsüli a menedzsment megközelítésének hatékonyságát. Technikai tudás szükséges hozzá. A következő szerepeket kell létrehozni:

- döntéshozó: az a személy, aki levezeti az ellenőrzés folyamatát.
- az ellenőrzés vezetője: az adminisztratív dolgokért felel, úgy mint, előkészítés, tervezés.
- a felvevő: dokumentálja az anomáliákat, a döntéseket, javaslatokat.
- a menedzsment csapat: a rendszer egészéért felelősek.
- technikai csapat: gondoskodik a szükséges információkról a menedzsment számára, hogy el tudják végezni a feladatukat.

Technikai ellenőrzés

Célja, hogy valamiféle becslést adjon a termékről, miszerint, alkalmas – e arra, amire tervezték, valamint azonosítja a specifikációktól való eltéréseket. Megerősíti vajon:

- a szoftver alkalmazkodik – e a specifikációkhoz,
- hűséges – e a szabályokhoz, tervekhez, útmutatókhoz, és a projekten alkalmazott folyamatokhoz,

- a szoftverbeli változásokat implementálták.

A kioszthatő szerepek:

- döntéshozó: meghatározza, hogy az ellenőrzés tényleg teljesült – e.
- az ellenőrzés vezetője: az ellenőrzésért felelős, az adminisztratív dolgokért is ő felel.
- a felvevő: dokumentálja az anomáliákat, a döntéseket, javaslatokat, melyeket a csapat tagjai tesznek.
- technikai csapat: gondoskodik a szükséges információkról a menedzsment számára, hogy el tudják végezni a feladatukat.

Nyomozás

Cél, hogy megtaláljuk, és azonosítsuk az anomáliákat. Ez egy szisztematikus vizsgálat:

- ellenőrzi, hogy a szoftver kielégíti – e a specifikációkat,
- ellenőrzi, hogy a szoftver kielégíti – e a speciális attribútumokat,
- ellenőrzi, hogy a szoftver alkalmazkodik – e az alkalmazható szabályokhoz, szabványokhoz, útmutatókhoz, tervekhez és folyamatokhoz,
- azonosítja a szabványoktól és specifikációktól való eltéréseket,
- összegyűjti az adatokat,
- az adatokkal javítja a nyomozási folyamatot és támogatja a dokumentációt.

Átnézés, átfutás

Cél, a szoftver becslése, felmérése. Legfőbb nézőpontok:

- anomáliák megtalálása,
- a szoftver javítása,
- alternatív implementációk átgondolása,
- szabványoknak és specifikációknak való megfelelősség ellenőrzése.

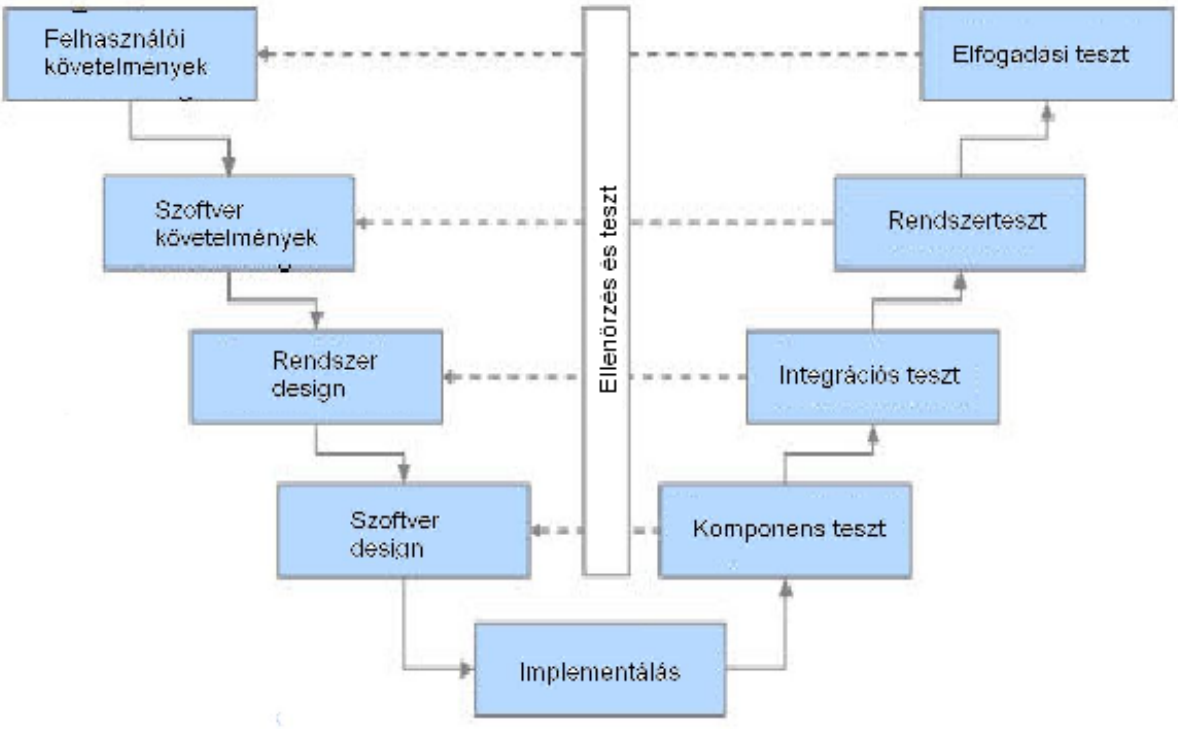
Auditok

Cél, hogy biztosítsa a szoftver megfelelősségének, és az alkalmazható szabályok, szabványok, útmutatók, és egyéb dokumentációk független ellenőrzését. A vizsgálatnak egy áttekintéssel kell kezdődnie.

Az auditor az a személy, aki megvizsgálja a terméket, az audit terv alapján. Dokumentálniuk kell a megfigyeléseiket, és javaslataikat. Minden auditornak befolyásoktól és külső hatásoktól mentesnek kell lennie, melyek miatt nem tudnának független döntéseket hozni.

V – modell

A V – modell a szoftverfejlesztés műszaki folyamatait részletesen leíró modell. Magában foglalja a verifikációs és validációs folyamatot a szoftverfejlesztés teljes életciklusa alatt. 1997 – ben tették közzé, mint az informatikai rendszerek fejlesztésének életciklusát leíró szabványt.



4. ábra: V - modell

Tehát, a V – modell a „rendszerfejlesztés, módosítás és karbantartás folyamán szereplő összes tevékenységet, terméket, valamint kapcsolatukat szabályozza”.

7. A gyakorlat

A gyakorlatban nem gondoljuk végig igazán, hogy most, abban a pillanatban milyen technikát, módszert alkalmazunk. A külső szemlélő számára csak kattintgatunk és aztán „bepötyögünk” valamit egy Excel táblázatba, vagy arra alkalmas egyéb szoftverbe, és ezekből lesz néhány ábra, amit nem mindenki ért. A régi feature – ök már ismertek mindannyiunk számára, így az ezekhez tartozó teszteseteket mindegy ki tartja rendben, belső szabályaink szerint az, akinek kiosztják, mint feladatot. Az új dolgokat pedig meg kell ismerni. Ezen ok miatt kapunk egy korai build – et, össze tudjuk hasonlítani a követelményekkel, kicsit előre dolgozni. A feladatkiosztás itt sem maradhat el, de ha már készült valamihez teszteset, akkor meg tudjuk kérdezni az illetőt, tudunk segítséget kérni tőle, ha nekünk kell frissíteni, vagy használni.

A tesztelés sincs benne a kisujjában senkinek, ezt is meg kell tanulni. A kezdetekkor nagyon sokat lehet fejlődni, de egy idő után bele is lehet fáradni. Nem maga a tesztelés válik fárasztóvá, netalán unalmassá, hanem maga a termék. Pár évvel ezelőtt elképzelésem sem volt arról, hogy tesztelni ennyi féleképpen lehet. Amit akkor el tudtam képzelni, az ad hoc tesztelést fedte le. Majd később úgy döntött, az egész csapat, hogy képezzük magunkat, és „felnövünk” a munkánkhoz. Előzményekről annyit kell tudni, hogy ez mai napig egy kis cég, 11 tesztelő van, és a tesztcsapat egyfajta véletlennek köszönhető. Nevezetesen egy fejlesztő elkezdte kicsit alaposabban átnézni az addig elkészült munkát, és talált néhány hibát, ekkor rábízták, hogy ezután csinálja a tesztelést, kinevezték tesztelőnek. Az elején ez az egy ember még egyáltalán nem alkalmazott semmilyen módszert.

Néhány szó a hibákról, és követésükről. Használtunk Bugzillát, és vele párhuzamosan SQL – t is. Ma már csak utóbbit, néhány hónapja leépítettük Bugzillát, nem volt értelme 2 helyen tárolni a hibákat. A Microsoft termékbe vesszük fel a hibákat, abban állítjuk át az állapotát, azon keresztül (is) kommunikálunk a fejlesztőkkel. A hibákat angolul vesszük fel, ha bármilyen megjegyzést még akarunk írni hozzá, akkor azt szintén angolul. A folyamat úgy néz ki, hogy mi felveszünk egy hibát, csatoljuk a screenshot – kat, a logokat,

leírjuk a lépéseket, magát a hibát, beállítunk priority – t, target – et, környezetet, címet adunk neki. A cím akkor jó, ha magában foglalja a lényeget. Nem elég annyit írni, hogy „Error occured”, vagy „Does not work”, ezek elég tág megfogalmazások, ez alapján nem tudják eldönteni, hogy érdemes – e egyáltalán foglalkozni adott hibával. Minden hiba a vezetőfejlesztőhöz kerül első körben. Ő elosztja a munkát a modulok felelőseihez. Ha tudják reprodukálni, akkor jó, ha nem, akkor visszaküldik, és kezdődik a nyomozás, a bizonyítás. Maradjunk a reprodukálható ágnál, utána elkezdik javítani, ha végeztek, akkor Fixed állapotot kap a hiba, és akkor ellenőrizzük egy új build – ben. Ezután két lehetőség áll fenn, vagy tényleg javították, és akkor Pass állapotot kap, vagy nem, ekkor lesz Fail. Utóbbi esetben szintén le kell írni a megfigyeléseket, az új tapasztalatokat. Ami viszont mindig igaz, minél hamarabb derül ki egy hiba, annál olcsóbb javítani, újratesztelni. Ha Pass, akkor regressziós tesztben még részt vehet, akkor ugyanis visszakeressük ezeket is, és újrateszteljük.

A dokumentáció szintén nagyon fontos. Nem össze – vissza írogatunk, és nem értelmetlenek a grafikonok, kimutatások sem, miután már értjük, hogy mi van mögötte. Van egy terv, ami szerint haladunk, és ismerjük a követelményeket, ezek alapján írjuk meg a teszteseteket, vagy ha már vannak korábbról, akkor csak frissítjük őket. Amit a dolgozatban említettem, hogy minden dokumentumnak legyen felelőse, aki átírja, frissíti, tehát használhatóvá teszi, nos, ez nem ilyen egyszerű. Megeshet, hogy az illető átkerül másik projektre, és nem lesz ideje foglalkozni a dokumentációs munkával, ekkor beugrik valaki, és átveszi, ha sokat kell javítani, akkor új tulajdonost kap adott teszteset, avagy követelmény.

A terv részét nem ismerem részletesen, de azt tudom, hogy valamennyire igyekszünk szabványhoz igazodni. Ha nem is szó szerint követjük, de mindenképp le van írva, hogy milyen operációs rendszeren tesztelünk, milyen eszközöket használunk, mennyi erőforrás van, mennyi ideig tart a tesztidőszak. Az időszak végén pedig megvannak a kimutatások, összesítések is, például mennyi hiba lett felvéve, akár hetekre lebontva, mennyit vettünk fel mi, és mennyit a másik csapat.

A tesztkörnyezet felállítása is a mi dolgunk, a tesztelés folyamatán kívül is. Környezetre azért van szükség, hogy ellenőrizni tudjuk, például a javítást, el

tudjuk végezni a tényleges tesztelést, segítséget jelent, ha a program fut is, és annak alapján tudjuk a dokumentációt megírni. Mindenképp szükségünk van szerverekre, és kliensekre. Szerverek az adatbázisszervereink, az FTP, a levelezéshez szükséges, a web szerver, sőt, a saját domain – ünk, és mindehhez rendszergazdánk, aki általában közülünk kerül ki hosszabb – rövidebb időre.

Az automatikus tesztek megírására külön csapatunk van. Ugyan a teljes szoftver nincs lefedve, nem lehet az egészet automatizálni, nem is volt cél, de egy smoke tesztet le tudunk futtatni. Az addigi 8 órás munkából, így 4 órás lett, eddig manuálisan csak 1 gépen, 1 operációs rendszeren tudtuk ezt megtenni, automatikus tesztekkel pedig lényegesen több gépen. A tesztek kódolásával tavaly (2008) év elején kezdtünk el foglalkozni, az év végére kitűzött célt pedig sikerült teljesíteni, és mindenképpen folytatjuk a munkát.

Tanúság a múltból: Nemrég voltunk egy workshop – on, cég rendezte, csak a mi ágazatunknak, hogy ismerjük meg jobban egymás munkáját, és tudjunk beszélni felmerülő problémákról. Voltak előadások, természetesen volt némi történelem, hogy a termék honnan indult, és ma hol tart, milyen módszerek szerint dolgoznak a fejlesztők, megosztották a tapasztalatokat, volt természetesen tesztelésről is szó, és később kiderült, ennek volt a legnagyobb visszhangja. Az idő véges volta miatt nem lehetett minden kérdésre válaszolni, mindent részletesen elmondani, de az előadások után a csoportvezetők még maradtak, és együtt beszélgettek tesztelésről. Szinte hihetetlen volt látni, hogy hiába dolgoznak velünk, nem tudják, hogy mit csinálunk, milyen problémáink vannak, pedig meg szoktuk osztani velük, és hogy pont a tesztelés kelti fel a legnagyobb érdeklődést.

8. Szószedet

Anomália: olyan feltétel, mely eltér a követelményekben, design – ban, dokumentációkban, szabványokban megfogalmazottaktól. Ellenőrzések, tesztelés, elemzés során találhatók meg.

Audit: bizonyítékok nyerése, és ezek objektív kiértékelésére irányuló rendszeres, független és dokumentált folyamat, annak meghatározására, hogy az auditkritériumok milyen mértékben teljesülnek.

Nyomozás:

A szoftver termék vizuális ellenőrzése, hogy felderítsük és azonosítsuk az anomáliákat.

Menedzsment ellenőrzés:

Szoftver beszerzés, támogatás, fejlesztés, működtetés vagy szervizelés szisztematikus ellenőrzése a menedzsment által.

Ellenőrzés:

Egy folyamat, vagy találkozó, melyen a szoftver bemutatásra kerül a menedzsereknek, felhasználóknak, és minden érdekelt félnek.

Szoftver termék:

Programok, dokumentációk és adatok összessége.

Technikai ellenőrzés:

Egy módszeres ellenőrzés a szoftveren, mely magában foglalja a szoftver alkalmasságát és azonosítja a szabványoktól és követelményektől való eltérését. Tartalmazhat ajánlásokat, alternatívákat.

Átnézés, átfutás:

Statikus elemzés, melyben a tervező vagy programozó kérdéseket tesz fel, és véleményt alkot a lehetséges hibákról, fejlesztési szabványokról és problémákról.

Minőség: egy termék, rendszer vagy folyamat saját jellemzői összességének képessége, hogy kielégítse a vevő igényeit.

Termék: a folyamat eredménye.

Folyamat: erőforrásokat használ, hogy a bemeneteket kimenetekké alakítsa.

Követelmény: kinyilvánított elvárások.

Vevői megelégedettség: a vevő véleménye arról, hogy a termék milyen mértékben elégítette ki az igényeit, elvárásait.

Rendszer: egymással kapcsolatban vagy kölcsönhatásban lévő elemek összessége.

Szoftver: szellemi termék, amely egy hordozó médiumon levő információkból áll.

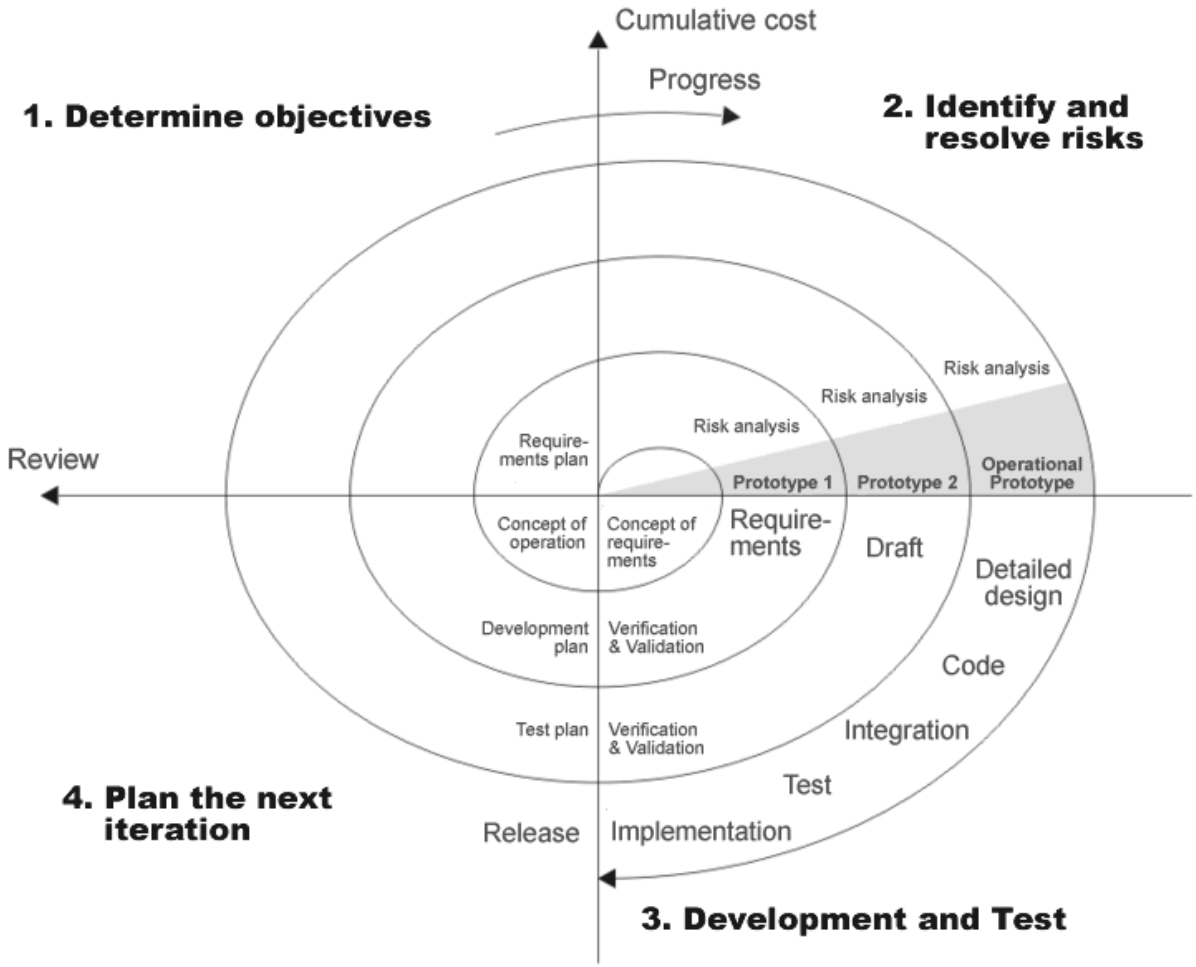
Hiba: egy követelmény vagy elvárás nem teljesülése, vagy az elvárt működéstől való eltérés.

9. Irodalomjegyzék

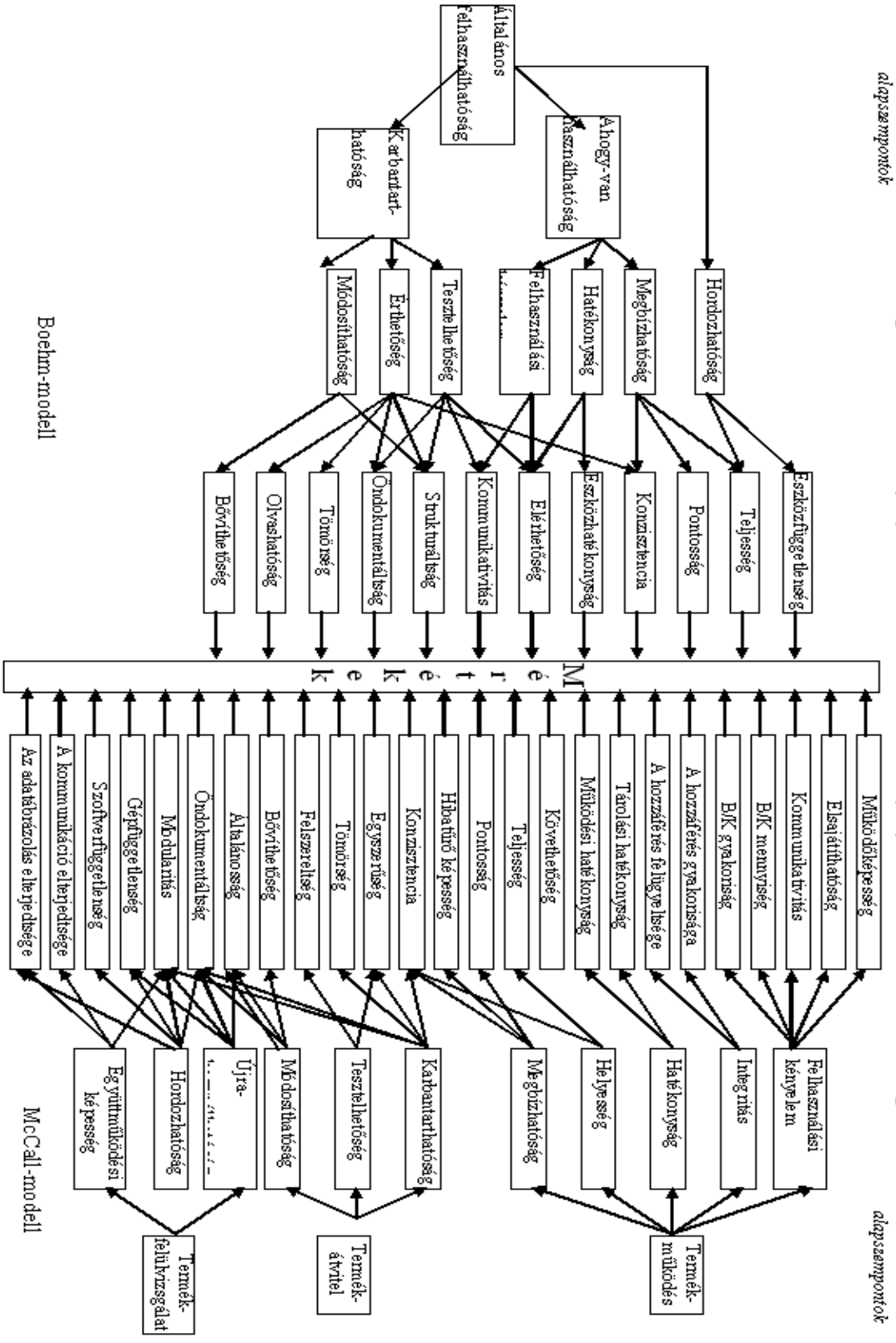
1. Balla Katalin: Minőségmenedzsment a szoftverfejlesztésben, Panem Könyvkiadó, 2007
2. Yashwant K. Malaiya: Automatic Test Software
3. Elisabeth Hendrickson: Agile Testing Nine Principles and Six Concrete Practices for Testing on Agile Teams, 2008
4. Cem Kaner, James Bach, Bret Pettichord: Lessons learned in Software Testing, 2002
5. Andreas Spillner, Tilo Linz, Hans Schaefer: Software Testing Foundations 2nd edition, 2007
6. http://teszteles.blog.hu

10. Függelék

Boehm modell:



Boehm és McCall modell:



Köszönetnyilvánítás

Ezúton szeretném megköszönni Dr Fazekas Gábor tanár úrnak, és külső konzulensemnek a támogatást.