Tesztelési környezetek

Antóci Gábor

Készült: 2015

Terjedelem: 7 ív

Kézirat lezárva: 2015. Április 30.

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.*

TARTALOMJEGYZÉK

Bevezetés

A tesztkörnyezetek megfelelő definiálása, építés, előkészítése és karbantartása az egyik kulcs tényezője egy projekt sikeres tesztelésének. A tananyag elsősorban azoknak készült, akiknek már van valamennyi tesztelési, technikai alapjuk és jobban el kívánnak mélyedni a részletekben. A tesztelés alapjainak ismereteit a *Szoftvertesztelés a gyakorlatban* című tananyag tartalmazza ([70]).

A tesztkörnyezetek témaköre technológiai kérdés is egyben. A tananyag írásánál igyekeztem megtalálni az arany középutat az elméleti és gyakorlati elemek között, azaz megpróbáltam konkrét példákat felállítani, de a tananyagnak nem célja, hogy rendszeradminisztrációs tréninget is nyújtson, tekintettel a szinte korlátlan számú lehetséges technikai, technológiai megoldásra. Fontos célom volt az időt állóság a példákban, ami tekintettel az információs technológia gyors változására, szintén középútkeresést jelentett a nagyon konkrét és a nagyon elméleti definíciók között. Ugyanakkor célom volt, hogy a jelenkor újonnan előtérbe került technológiáit (felhők, mobil alkalmazások) is lefedjem.

A tananyag felépítése több részre tagozódik. Az elején a tesztkörnyezetek definiálásával foglalkozom, ahol inkább azok menedzsmentje, a velük szemben támasztott követelmények, biztonsági és adatkezelési kérdések, illetve a kapcsolódó szoftver fejlesztési folyamatok és azokkal való kapcsolatáról lesz szó.

A második része a tananyagnak konkrét architekturális felépítéseket boncolgat és jár körül. Itt feltérképezem a leggyakoribban előforduló eseteket, problémákat és kockázatokat, valamint az adott architektúrára, egyes esetekben, a leggyakoribb paraméterekkel rendelkező környezetre konkrét megoldási javaslatot is nyújtok.

Tesztkörnyezetek definiálása

A tesztkörnyezetek definiálása és karbantartása a tesztmenedzsment egyik legmeghatározóbb és legkomplexebb feladata, amiben több szakterület együttműködésére van szükség. Bár vannak kész receptek, bevett technikák egyes esetekre, általánosan igaz, hogy az adott környezeteket mindig testre kell szabni az adott körülményekre, és az adott paraméterekre kell megtalálni a legjobb kompromisszumot, mert egyszerre minden kívánalomnak – tekintettel arra, hogy a legtöbb követelmény olyan, hogy kielégítésükkor a többi paraméter optimuma ellen hatnak – nem lehet megfelelni a való életben.

A leggyakoribb paraméterek, amiket figyelembe kell venni a tervezéskor:

- Minőségi követelmények
- Teszt stratégia és módszertan
- Tesztelési kockázatok
- Architektúra és technikai megvalósítás
- Kiadás (Release) menedzsment
- Incidenskezelés
- Konfigurációkezelés
- Adatkezelés
- Biztonsági előírások
- Projektmenedzsment és módszertan
- Üzemeltetés
- Projekt büdzsé

Ezen területek és folyamatok a tesztkörnyezetek definiálása folyamán kölcsönösen hatással vannak egymásra, befolyásolják, hogy milyen és hány tesztkörnyezetet definiálunk, illetve a követelmények miatt változtatások lehetnek szükségesek az adott területen.

A későbbi fejezetekben részletesen is megvizsgáljuk, hogy az adott követelményrendszer miként tudja befolyásolni a definiált környezeteket.

Minőségi követelmények

A tesztelés tervezésének az első fázisában általában meghatározzuk a minőségi követelményeket (több más teendővel együtt). Az adott követelmények meghatározhatnak több paramétert a tesztkörnyezetből, amit teljesíteni kell ahhoz, hogy az adott követelményt ki tudja az adott projekt

elégíteni. Szemléltetésként álljon itt néhány példa ilyen követelményre, és hogy mindez hogyan befolyásolja a környezetek paramétereit, illetve amennyiben szükséges, egyben más területek folyamatait is.

Példa1: *Tesztelni kell a végleges rendszernek pontosan megfelelő rendszerrel a terméket az élesbe állítás előtt.*

Ez a példa tipikusan megjelenő követelmény szokott lenni, de az esetek nagy többségében nem teljesíthető. Gondoljunk bele, hogy egy szerver-kliens alkalmazásnál ez mennyi lehetséges kombinációt jelenthet kliens oldalon. A követelményeknek pontosnak és konkrétnak kell lennie, ha megfelelő módon akarjuk kezelni. Ilyen esetekben érdemes az adott követelményt pontosítani, illetve több darabra bontani. Ilyen lehet például a következő bontás:

- *Tesztelni kell az adott szoftvert olyan módon, hogy az infrastrukturális felépítés, a szerverekben használt hardverelemek, az operációs rendszer és a telepített egyéb szoftverek verziószáma és a kapcsolódó egyéb rendszerek, és az azokon telepített szoftverek és azok verziószáma és az adatok mennyisége és minősége pontosan megegyezzen az élesbe állás állapotával.*
- *Tesztelni kell a rendszert az alábbi kliens oldali konfigurációkkal:*
    - *Operációs rendszerek (Windows 7, Windows 8, Windows 8.1, Mac OSX legfrissebb, iOS legfrissebb, Android legfrissebb)*
    - *Böngészők: Internet Explorer legfrissebb, Firefox legfrissebb, Google Chrome legfrissebb, Safari legfrissebb, Opera legfrissebb változatával*
    - *Operációs rendszer nyelvek: angol, magyar, koreai*

Láthatóan a követelmények sokkal pontosabbak. Szerver oldalon láthatóan kell lennie egy a jövendőbeli élő környezettel minden szempontból azonos környezetnek a teszteléshez. Mivel az ilyen környezet felépítése és fenntartása általában drága, viszont több tesztkörnyezetre is szükség lehet, ezért csak egy, esetleg két ilyen rendszert szoktak építeni. A többi rendszer gyakran jelentősen alacsonyabb teljesítményű, és a különböző tesztelési tevékenységek szervezésével érjük el, hogy a szükséges tesztelési tevékenységeket (amikor teljes rendszerre van szükség) el tudjuk végezni.

A kliens oldalon, ha minden kombinációt, amit megkövetelünk, tesztelni szeretnénk 180 különböző rendszert definiálunk. Ilyen esetben segíthet, ha csak páronként garantáljuk a megfelelő környezeteket lásd páronkénti tesztelés módszere (Orthogonal Array, Pairwise method)[2]. Ilyen esetben az alábbi párosításokat kapjuk:

#	OS	Nyelv	Böngésző
1	Windows 8.1	Angol	Firefox
2	Windows 8	Angol	Internet Explorer
3	Android	Angol	Safari
4	Windows 7	Angol	Opera

5	iOS	Magyar	Opera
6	Windows 8.1	Magyar	Chrome
7	Windows 8.1	Koreai	Safari
8	OSX	Koreai	Firefox
9	Windows 8	Koreai	Chrome
10	Android	Magyar	Internet Explorer
11	iOS	Koreai	Internet Explorer
12	Windows 8.1	Angol	Internet Explorer
13	OSX	Koreai	Opera
14	Windows 7	Koreai	Chrome
15	OSX	Magyar	Safari
16	Windows 7	Magyar	Firefox
17	Android	Koreai	Opera
18	Windows 7	Angol	Internet Explorer
19	Windows 8	Magyar	Safari
20	Windows 8.1	Angol	Opera
21	Windows 8	Koreai	Opera
22	iOS	Angol	Safari
23	OSX	Angol	Internet Explorer
24	Windows 8	Koreai	Firefox
25	iOS	Magyar	Firefox
26	iOS	Angol	Chrome
27	Android	Koreai	Firefox
28	Android	Angol	Chrome
29	OSX	Angol	Chrome
30	Windows 7	Magyar	Safari

Táblázat 1 - Kliensvariációk szűkítési példája

Látható, hogy az eredeti 180 kombinációt 30-ra csökkentettük, ami már kezelhető mennyiség. Ezt is tovább optimalizálhatjuk, hiszen nem feltétlenül kell minden böngészőt külön operációs rendszerre tenni, viszont ha több tesztelőnk van, és egyszerre kell azonos operációs rendszerrel különböző böngészőn tesztelni, az problémát okozhat. Ugyanígy az iOS és Android rendszerek nyelve már átállítható (sőt egyes Windows változatoké is), ami további optimalizálást tehet lehetővé.

Bár a páronkénti tesztelés egy erős optimalizációs technika, megvannak a korlátai, és nem segít azon, ha rossz értékeket választunk. Egyes esetekben más technikák (például ekvivalencia particionálás), illetve a tesztelői tapasztalat jobb eredményeket hozhat adott esetben.[3]

Példa2: *Az adott rendszeren tesztelni kell, hogy az elvárt terhelést teljesíteni tudja, illetve meg kell határozni azokat a paramétereket, amelyek változása az üzemeltetésben olyan körülmények kialakulását vetíti előre, amelyek a szolgáltatás nyújtásában az elvárt üzemeltetési paraméterek teljesítésében kockázatot hordoz.*

Jelen követelmény előrevetíti, hogy a tesztelés folyamán teljesítménytesztelésre is szükség lesz. Általában a végleges teljesítménytesztekhez szükség van egy – a későbbi valós környezetnek teljesen megfelelő – környezet kialakítására, illetve a teljesítménytesztéléshez általában más – a tesztelés végrehajtásához szükséges – hardverelemek és szerverek beépítésére is szükség lehet.

Hasonlóan szükséges lehet hasonló elemek beépítésére más teszttípusok esetén is (pl. biztonsági vagy automatikus tesztelés esetén).

Teszt stratégia és módszertan

A tesztkörnyezetek elsődleges célja, hogy megfelelő módon támogassa a tesztelést, mint folyamatot, ezért annak folyamata és módszerei fontos követelményeket adnak a tesztkörnyezetek felépítéséhez.

A tesztelési fázisok, tevékenységek, a párhuzamos fejlesztések száma meghatározza, hogy milyen és mennyi környezetre van szükségünk, illetve fordítva is igaz, a rendelkezésre álló tesztelési környezetek hatnak a megvalósítható tesztelési módszerekre, illetve korlátként szolgálhat a párhuzamos fejlesztések számára. Figyelni kell a párhuzamosan futó tesztelési tevékenységekre, hogy azoknak időben és esetlegesen fizikailag is elkülönülő tesztkörnyezetet kellhet biztosítani. Ilyen igény lehet például egy teljesítménykritikus rendszerben egy folyamatos teljesítményteszt ellenőrzés a fejlesztés korai szakaszától kezdve, ahol ennek a tevékenységnek folyamatosan szükséges egy környezet biztosítása, ahol nincs funkcionális tesztelés, illetve fejlesztői munka. (Általában ebben az esetben nem a teljesítménytesztet zavarja meg az egyéb tevékenység, hanem annak nagy a valószínűsége, hogy a teljesítményteszt tartós szolgáltatás-kiesést okoz, és ezen keresztül a többi tevékenységet hátráltatja).

Nagyobb szervezeteknél, ahol sok, egymással párhuzamosan futó projektet és szolgáltatást kell támogatni, előbb utóbb szükségessé válik egy egységes teszt stratégia és módszertan kidolgozása, amelynek összhangban kell lennie más – a szoftverfejlesztéshez kapcsolódó – folyamatokkal. Ezen folyamatokban szokás szabályozni a tesztkörnyezetekre vonatkozó szabályozásokat is, amelyekkel biztosítható a megfelelő elérhetőség, illetve a hatékony erőforrás-kiosztás.

A tesztkörnyezetek felhasználása és menedzsmentje általában az egyik központi elem a teszt stratégia megalkotásakor. A teszt stratégia természetesen nem független a projekt többi elemétől, és harmonizálni kell a többi folyamattal. A legfontosabb külső folyamatok, amelyek hatással vannak mind a teszt stratégia, mind a tesztkörnyezetek kialakítására vonatkozó részével: a projektmenedzsment, a konfigurációmenedzsment (verziókövetés), a kiadás (release) menedzsment területe illetve a meglévő infrastruktúra támogatása.

A teszt stratégia felölel(het) más, az adott fejlesztéshez kapcsolódó kockázatokat, módszertani igényeket, amelyek kivetülhetnek tesztkörnyezetbeli igénnyé. Ilyen lehet egy olyan kockázatos elem a fejlesztés folyamán, ami korai tesztelést, és ezért egy speciális környezetet igényel, vagy olyan komponensek, külső szolgáltatások, ahol tesztkörnyezet nem, vagy csak limitáltan áll rendelkezésre, vagy a már említett teljesítménytesztelés is.

Architektúra és technikai megvalósítás

Aligha képzelhető el tesztkörnyezet, ami nem reflektálna a végleges használatra, nem szabad azonban elfelejteni, hogy a tesztkörnyezet csak egy modell a végleges termékhez, és attól függően, hogy mit akarunk tesztelni, nem biztos, hogy a teljes infrastruktúrára szükségünk van. Egyes megoldások, különösen azok, amelyek nagyon sok felhasználót szolgálnak ki, nagyon költséges hardver platformon futhatnak, így nem lehetséges őket több példányban létrehozni, mint tesztkörnyezetet. A legtöbb esetben nincs is rá szükség, hiszen a tesztelés folyamán a végleges felhasználók számának csak töredéke a terhelés, így ahhoz, hogy funkcionálisan a terméket ellenőrizzük, nincs szükségünk az összes olyan elemre, amelyek csak arra szolgálnak, hogy a sok konkurens felhasználó által generált teljesítményt biztosítsák. Ugyanakkor szükséges lehet egy nagy komplexitású környezet is, ahol a teljesítményt, illetve a terhelés felvételére szolgáló elemek funkcionalitását is tudjuk ellenőrizni, de a nagyteljesítményű környezetek száma általában korlátozott.

A tesztkörnyezeteknél nagy többletköltséget jelenthetnek a szoftver licenckőltségek. Egyes szoftvereknél csak a végfelhasználók után kell megfizetni a licencdíjat, így annak nincs költsége a tesztkörnyezetek esetében, viszont más gyártóknál minden futtatott példány, így a teszteszközökön lévő is licencdíj-köteles. Ez egyes esetekben jelentős kiadást jelenthet, ezért itt érdemes lehet megvizsgálni, hogy ezek a költségek hogyan csökkenthetőek.

A tesztkörnyezetek definíciója általában együtt halad az architektúra tervezésével, és minden lényeges változtatás az architektúra elemein leképződik a tesztkörnyezetekben is.

Projektmenedzsment módszertan[4][5][6][7][8][9][10]

Bár léteznek más módszerek is, alapvetően a két legelterjedtebb modell, amit jelenleg az iparág használ, az a vízesés vagy az agilis szoftverfejlesztési modellek egyike. Vannak még lassabb ciklusú iteratív modellek is használatban (pl. a spirál modell), de ezek tesztelési környezetek szempontjából valamelyikhez közelítenek, a ciklus hosszától függően, általában negyedéves ciklushossz felett inkább vízeséshez közeli a megvalósítás.

A két módszertanról elmondható, hogy vannak közös elemek, de a környezetek létrehozásának menetrendje jelentősen eltér egymástól, illetve más kihívások elé állítja a projekt résztvevőit.

A vízesés modellnél nagyon pontos tervezés szükséges (csakúgy, mint más területeken is), viszont a megvalósításnak nem feltétlenül célja a flexibilis alakíthatóság, illetve jelentősen több idő áll rendelkezésre a legtöbb környezet felépítésére, mert a fő tesztelési fázisok időben később kezdődnek.

Agilis környezetben kiemelt fontosságú, hogy flexibilis környezetet tudjunk biztosítani, mert az architektúra folyamatosan változik a módszertanból adódóan, illetve szükség lehet speciális tesztelési igények gyors kielégítésére.

A gyors változások kockázatot jelenthetnek a végső felhasználói környezet definíciójában, amennyiben annak kialakítására olyan technológiákat használunk, amelyek nem ennyire flexibilisek.

Az elfogadási és végső rendszerintegrációs teszteket általában a végső felhasználói környezettel megegyező, vagy nagyon hasonló környezetben futtatjuk. Egy késői olyan változtatás, amelynek hatása

van a végfelhasználói környezetre, olyan változtatások végrehajtását igényelheti, amire már nincs idő az eredetileg tervezett menetrend szerint (mert például a beszerzési határidő hosszú, vagy a hálózati konfiguráció módosításának felülvizsgálata túl hosszú időt igényel). Ezen kockázatok kivédésére érdemes az agilis módszertanoknál olyan prioritásrendszert alkalmazni, ahol az architekturálisan szignifikáns elemeket tartalmazó követelmények a fejlesztés elején bekerülnek a fejlesztendő elemek közé.

Külső folyamatok

Manapság már ritkán beszélünk egyedi fejlesztésekről, egyedülálló rendszerekről, még viszonylag kisméretű szervezeteknél is jellemző, hogy a szolgáltatások jelentős része integrált más szolgáltatásokkal. A tesztkörnyezeteknek kettős célt kell szolgálniuk, egyrészt az adott szolgáltatás fejlesztését, üzemeltetését kell szolgálniuk, másrészt más szolgáltatások számára, azok tesztjeihez szolgálnak integrációs végpontként. Hiszen nem várható el, és gazdaságilag nem megvalósítható illetve hatékony, hogy minden egyes projekt a teljes infrastruktúrát felépítse magának.

A tesztkörnyezetek definíciójakor az előbb vázolt okokból, meg kell felelni az adott szervezet által használt folyamatoknak. Ezek egyes elemei lehetnek az organizáció szintjén definiáltak, mások csak projekt/szolgáltatás szinten, illetve lehetséges, hogy a szolgáltatás kiegészíti a teljes organizáció által definiált folyamatot. Alapvetően, ha egy folyamat az organizáció szintjén definiált, azt szolgáltatás szintjén csak kiegészíteni lehet, és csak úgy, hogy az organizációs szinten definiált elvárásoknak teljesen megfeleljen. Szervezeti szintű folyamatok áthágása más szolgáltatások üzemeltetését és fejlesztését számukra nem átlátható módon akadályozhatja. A továbbiakban a néhány legfontosabb területet érintjük

Kiadás (release) menedzsment[11]

Ez a terület felelős azért, hogy definiálja, hogy milyen folyamaton keresztül lehetséges valamilyen változtatás a felhasználói környezetben. Általában legalább két típusú módosítást szoktak definiálni.

Az egyik, amikor előre tervezetten kerülnek új funkciók a rendszerbe. Ebben az esetben általában definiáltak bizonyos minőségi célok, illetve bizonyos ellenőrzési pontok, jóváhagyások, aminek meg kell felelni. A teszt stratégiának és a tesztkörnyezeteknek, ennek a kritériumrendszernek meg kell felelnie. Általános tapasztalat, hogy az itt definiált elvárások esszenciálisak, azaz organizációs szinten a minimumkövetelményeket rögzítik, amit szolgáltatás szinten tovább finomítanak.

A másik típusú módosítás, amikor nem várt esemény miatt kell beavatkozni. Ezek általában nem várt, nagy súlyosságú hibák javításakor fordulnak elő, amit bár igyekszünk elkerülni előzetes teszteléssel, még igen fejlett, nagy hatékonyságú minőségorientált szervezeteknél is fel kell készülni az adott eshetőségre. A legtöbb szervezet az ilyen módosítást kivételként kezeli, és csak súlyos problémákat kezelnek ilyen módon. A kisebb fajsúlyú problémákat összegyűjtve, az előre tervezett egyéb módosításokkal együtt javítják, csökkentve evvel a nem kívánt esetleges mellékhatások kockázatát.

Szintén szokás definiálni magasabb szinten elvárást, hogy milyen tesztkörnyezeteket kell minimálisan biztosítani, és ott milyen verziókat kell, illetve lehet installálni, illetve milyen rendelkezésre állást kell biztosítani mások számára. Ennek kialakítása nagyban függ az adott szervezet módszereitől. Egyes szervezetek ebből a szempontból monolitikusak, azaz előre meghatározott időpontokban (pl.

negyedévente egyszer) lehet módosítani, mindenkinek egyszerre, míg más szervezeteknél teljesen rugalmasak a módosítási időpontok, azokat a szolgáltatások maguk definiálják.

Mindkét módszernek vannak előnyei és hátrányai. A monolitikus módszer módszertanilag egyszerű, és alacsonyabb adminisztrációt követel, viszont az ütemtervet módosítani nehéz, ahhoz mindenkinek alkalmazkodnia kell, ami egy nagyobb szervezetben előbb-utóbb nem megvalósítható. A rugalmas módszerek komplexitása viszont lényegesen magasabb, szabályozott környezetet követel, és olyan szervezetet, amely ezeket a szabályokat betartja.

Ugyanígy definiálni kell az adatokat, amire számítani lehet a tesztkörnyezetekben, illetve annak frissítését, és annak ütemét. Lényeges, hogy az adatok konzisztenciáját is biztosítani kell, különben egy rendszerintegrációs teszt megbukhat azon, hogy a két adathalmaz különbözik a rendszereken.

Konfigurációmenedzsment[12][13][14]

A konfigurációmenedzsment a konzisztenciát biztosítja a fejlesztésben, azaz azt biztosítja, hogy a különböző fejlesztési elemek (forráskód, dokumentumok, bináris állományok stb.) összepárosíthatóak legyenek. Bár többen azonosítják a verziókövetéssel (számozással), ez csak egy – igaz, nagyon fontos – eleme a területnek. A konfigurációmenedzsment szorosan integrált a kiadásmenedzsmenttel és a változásmenedzsmenttel.

A konfigurációk segítséget nyújthatnak abban, hogy segítségével definiálhassunk, hogy milyen elvárásoknak kell megfelelni egy adott környezetben, hogy a célja szerint felhasználható legyen (például az egyik helyen csak az aktuális kiadott konfiguráció lehet, a másikon csak a következő kiadás jelölt stb.).

Biztonság

A különböző szervezetek egyre nagyobb figyelmet fordítanak a felhasználói környezetek biztonságára, különösen az adatok és a működőképesség védelmére. Jogosan felmerül a kérdés, hogy miért is befolyásol ez minket, hiszen mi „csak” tesztkörnyezetekről beszélünk.

A tesztkörnyezetek általában tartalmaznak teljes vagy részleges formában éles adatokat, amelyeket ugyanúgy védeni kell, mintha az a végfelhasználói rendszerben lenne. A tesztrendszereket biztonsági szempontból sokan elhanyagolják, ami könnyű támadási célponttá teheti azokat. A tesztrendszerek, mivel általában a szervezetek külső védvonalán belül helyezkednek el, ugródeszkát jelenthetnek más rendszerek megtámadásához is, amit kívülről nem lehetne végrehajtani.

A továbbfertőzés elkerülése végett érdemes a tesztrendszereket a végfelhasználói rendszerektől fizikailag is elválasztani, és azok között a kapcsolatokat a minimálisra korlátozni, azokat pl. tűzfalakkal védeni. A teljes szeparáció – nagyon erős biztonsági érdekek kivételével – természetesen nem megoldható, hiszen a tesztelőnek is el kell érnie az alkalmazást, de például a direkt kapcsolatot a végfelhasználó környezet szervereihez lehet és kell blokkolni.

Mindezeken kívül a biztonsági rendszerek a végfelhasználói környezetekben potenciális hibaforrások. Nem megfelelő beállításokkal az e nélkül amúgy jól működő szolgáltatások is meghibásodhatnak. Ha ezek a védelmi rendszerek a tesztkörnyezetekből hiányoznak, az abból fakadó hibákat nem lehet detektálni. A biztonsági teszteléshez szükséges egy olyan tesztkörnyezet, amely tartalmazza az összes

biztonsági eszközt a megfelelő beállításokkal (természetesen apró módosításokkal, pl. az IP címeket az adott környezet paramétereihez kell alakítani).

Adatkezelés

A teszteléshez szükségünk van adatokra, és a tesztelés fázisa és célja határozza meg, hogy milyen és mennyi adatra van szükségünk. Néha csak néhány speciális adatra van szükségünk, amit kézzel, vagy valami direkt manipulálással létrehozunk a rendszerben, de különösen a magasabb szintű teszteléseknél már végfelhasználói mennyiségű és minőségű kell. Ezek a legtöbb esetben reálisan nem generálhatóak, és leginkább a valós adatok többé-kevésbé rendszeres másolásával, frissítésével szoktuk előállítani. Mindez persze felvet néhány problémát.

A végfelhasználói rendszerek tartalmaznak olyan adatokat, amelyek titkosak (pl. jelszavak), vagy nagyon szűk azoknak a köre, akik megismerhetik az adatokat. Ilyen korlátozott körben ismerhető adat például a fizetési adatok, amelyet csak az adott személy, a felettesei és általában a humán erőforrás osztály munkatársai láthatnak, de a terület tesztelője és fejlesztője nem (pedig nekik is tesztelniük kell valahogy). A végfelhasználók által elért rendszerben ez megoldott, mert a hozzáférések általában erősen ellenőrzöttek, de a tesztrendszerben másoknak is hozzáférést kell biztosítani. Ilyen esetekben egy jó megoldás lehet egy utólagos lépés a tesztkörnyezet frissítésére, amikor beépítünk ezekre az adatokra egy manipulációs lépést, ami véletlenszerűen megváltoztatja azokat.

Rendezni kell a tesztkörnyezetekhez való hozzáférések rendszerét is. A probléma lényege, hogy általában nem azok tesztelnek, akik használni fogják a rendszert, azaz hozzáférést kell biztosítani a tesztelőnek a megfelelő helyeken. Az alap megoldás az lehet, hogy a jelszavakat minden felhasználónál átállítjuk egy bizonyos jelszóra. Ez a megoldás bár egyszerű – különösen, ha a jelszó egyszerű, és nem változik túl gyakran – jelentős biztonsági kockázatokkal rendelkezik. A biztonságon lehet javítani a rendszeres jelszócserével, a jelszó bonyolultságával, illetve a jelszó, mint információ megosztásának korlátozásával, de mindenképpen marad egy bizonyos szintű kockázat. Vannak rendszerek, ahol ez a kockázat vállalható, ezt mindig egy biztonsági kockázatelemzés alapján lehet eldönteni.

Mit tegyünk, ha nem akarjuk ezt a kockázatot vállalni? Ebben az esetben több tervezésre van szükség. Az egyik megoldás, hogy a tesztelőknek további jogokat biztosítunk, illetve generálunk azonosítókat, vagy megváltoztatjuk a jelenlegi azonosítók jelszavát, és azt egyedileg kiosztjuk a megfelelő tesztelő(k)nek. Bár ez jóval nagyobb biztonságot produkál, külön folyamatokat kell felállítani, hogy hogyan gyűjtsük az igényeket, illetve hogyan kezeljük infrastrukturális oldalról az adott módosításokat. Megoldás lehet még a végfelhasználók bevonása a tesztelésbe, ha az ilyen igényeket támasztó tesztesetek száma viszonylag alacsony. Hosszú tesztelés esetén ez valószínűleg túl nagy erőforrás kivonásával járna az üzleti oldalon.

A rendszeres adatfrissítés felveti az adatkonzisztencia problémáját. Különböző rendszerek különböző idejű adatfrissítése felveti a lehetőséget, hogy a rendszerek integrációja ennek következtében sérül. Gondoljunk bele abba a szituációba, amikor a két rendszerből származó adatok összekapcsolódnak, majd az egyik rendszerből eltűnik (vagy éppen megjelenik) a kapcsolt adat egy adatfrissítés miatt, míg a másik oldal erről nem tud, és az integrációs tesztelés egy olyan adatot érint, ahol az adatkapcsolat hibás. Nagy valószínűséggel a teszt megbukik, de a teszt nem valós, hiszen a végfelhasználói rendszereket nem szoktuk frissíteni más adatokkal. Több megoldás is lehetséges a problémára:

- A kritikus rendszerek egyszerre történő frissítése
- Szinkronizációs mechanizmus, ami a megfelelő hivatkozásokat korrigálja
- Olyan adathalmaz meghatározása, amit az adott szinkronizációs probléma nem érint. Például az előző lezárt negyedéves, illetve az azt megelőző adatok biztosan jók ebből a szempontból

A megfelelő megoldás lehet ezek közül csak az egyik, vagy ezek kombinációja. A megfelelő megoldást az adott technológia figyelembevételével kell megtervezni.

Adatinkonzisztencia lehet még abban az esetben, amikor egy tesztkörnyezet nem rendelkezik avval a kapacitással, amivel a végfelhasználói környezet. Ezért az adatoknak csak egy részét másoljuk át. Könnyen látható, hogy amennyiben az integrált rendszer nem a teljes adathalmazt tartalmazza, akkor fennáll az adatok közötti kapcsolatok hibájának lehetősége, amennyiben a két rendszeren nem azonos módon történik az adatok szűkítése. A fenti megoldások mellett ilyen esetben fennáll a lehetősége annak, hogy a frissítéseket az integrált rendszerekben azonos szabályok szerint végezzük (pl. csak az elmúlt három év adatai).

Üzemeltetés

Jogosan merülhet fel a kérdés, hogy az üzemeltetés folyamataival miért is kell foglalkozni, hiszen az a projekt lezárása után van? Ez valóban így lenne egy hagyományos szolgáltatásnál, de a való életben egy fejlesztés a legritkább esetben végleges az első verzióval, így egyszerre üzemeltetjük és fejlesztjük a szoftvert. Ráadásul, még ha csak egy verzió is van az adott fejlesztésből, hibajavítások, apróbb módosítások még mindig szükségesek. Ezekhez ugyanúgy tartozik tesztkörnyezet, illetve olyan környezet, ahol a hibát analizálják (reprodukálják, megtalálják az okát), és ezekre rendszerint az eredeti tesztkörnyezeteket használjuk. Érdemes teát azt úgy definiálni, hogy számítunk a későbbi felhasználásra, és figyelembe vesszük azokat az igényeket is, elkerülve evvel a felesleges beszerzési költségeket. Mindemellett, az üzemeltetésnek nem csak a felhasználók által használt környezetet kell támogatni, hanem tesztrendszert is kell biztosítani az integrált rendszerek számára, hogy azok fejlesztését és tesztelését biztosítsák.

Amikor a fejlesztés és egy korábbi verzió üzemeltetése egyszerre folyik, általában nem tartunk fenn külön környezetet mind a fejlesztésnek, mind az üzemeltetésnek, hanem lehetőség szerint azonos infrastruktúrát szeretünk használni, hogy azt jobban kihasználva költséget takarítsunk meg.

A közös infrastruktúrához viszont megfelelő folyamatok, hatáskörök és prioritások lefektetése szükséges, hogy mindkét igény megfelelő kompromisszumokkal teljesítésre kerüljön. Megfelelő kompromisszumot jó kockázatelemzéssel és kezeléssel a legegyszerűbb kialakítani. Hiszen amennyiben számítunk a problémára, és arra megfelelő válaszokat tudunk adni, jelentősen lerövidül a reakcióidő, és az előre egyeztetett megoldások miatt csökken a konfliktusok kialakulásának kockázata.

A későbbiekben a szerver-kliens architektúra részeként ezt a területet jobban kiértékeljük, mert itt a leggyakoribb a közös és drága infrastruktúra elemek használata.

Projekt büdzsé

Nehéz megkerülni a kérdést, a projektek költségvetése alapvetően meghatározza meddig ér a takaró, azaz mit lehet, és mit nem lehet megvalósítani. Egy-egy szolgáltatás hardverkörülményeinek megteremtése meglehetősen nagy anyagi ráfordítást is igényelhet. A projekt tervezőit, menedzsereit – különösen, ha nem gyakorlott személyek – meglepheti, és készületlenül találhatja, hogy a tesztkörnyezetek létrehozása ezt az összeget akár meg is többszörözheti. Megfelelő folyamatokkal csökkenthető a tesztkörnyezetekre fordított összeg, például egyes környezetek teljesítményének csökkentésével.

Fontos megjegyezni ugyanakkor, hogy az alacsony, alultervezett büdzsé nem indokolja, hogy a minőségbiztosítási/tesztelési igényeket nem vegyük figyelembe. Jelentősen alultervezett környezetekkel nagy a kockázat, hogy emiatt súlyos rejtett hibák maradjanak a fejlesztett szolgáltatásban. Ezek összességében jelentősen nagyobb költséget jelenthetnek, mint a környezeteken „megtakarított" összeg. A súlyos rejtett hibák indoka lehet, hogy:

- Nem történt tesztelés a végfelhasználói környezettel megegyező, vagy ahhoz nagyon hasonló környezetben, ami funkcionalitási problémákat okoz (például egy teljesítményelosztott rendszer nem teljesítményelosztott rendszerben való tesztelése nem fedi fel az ilyen jellegű hiányosságokat).
- Teljesítményproblémák, mert a teljesítményt nem lehetett rendesen tesztelni a teljesen más felépítésű tesztkörnyezet miatt.

A megfelelő kompromisszum megtalálása ilyen esetekben igen nehéz, de több irányban is lehet keresni a megoldást a projekten belül.

- A legfontosabb, hogy a tesztkörnyezetek tervezése már akkor megkezdődjön, amikor a büdzsé tervezése folyik, ilyenkor lehet a legegyszerűbben elkerülni a meglepetést mindkét oldal számára.
- Alacsony prioritású követelmények elhagyásával csökkenthető a fejlesztési költség, amit megfelelően tudunk használni ezen a területen. A tapasztalatok szerint, egy átlagos projektben, az emberi erőforrás-költséghez képest a tesztkörnyezetek költsége nagyságrendileg alacsonyabb. Így viszonylag kicsi erőforrás-vágással is jelentősen növelhető arányaiban az erre a területre fordítható összeg.
- A büdzsé (bár fontos tényező egy projektben) végső esetben, amikor nyilvánvalóan hibás és nem tartható fenn, változtatható.
- Ha a környezetek tervezésénél megfelelő kompromisszumkészséget mutatunk, és bemutatjuk, hogy hol vágtunk már a költségekből, könnyebb máshol is áldozatot hozni.
- A túlzásba vitt költségcsökkentés egy komponensen, jelesül a tesztkörnyezeteken, azt is eredményezheti, hogy hosszú távon jelentősen többet kell költeni üzemeltetésre és karbantartásra, ami jelentősen meghaladhatja azt az összeget, mint az alacsony prioritású

követelmények fejlesztésének hozadéka. Az is előfordulhat, hogy már a fejlesztés hatékonyságát is annyira rontja, hogy összességében már ott magasabb emberi erőforrás többlettel kell(ene) számolni, amely már magasabb, mint a környezetek beszerzési költségének többlete (például mert az adatokból vágni kell, és a konzisztencia biztosítására egy több hetes fejlesztés szükséges). Mindig a teljes költségre koncentráljunk, mert kis összegek befektetésével is jelentősen tudjuk csökkenteni az összköltséget.

Tesztkörnyezet megvalósításának menetrendje

Egy tesztkörnyezet megtervezése és felépítése időigényes feladat. Mindemellett, hacsak nem nagyon alapszintű hardverelemekről van szó, a beszállítóktól is megfelelő szállítási határidőkkel kell számolni. Ezért fontos, hogy a környezetek felépítésének folyamata proaktív, jól tervezett és jól szervezett legyen.

Ebben a fejezetben egy-egy ajánlott ütemtervet mutatunk be a vízesés és az agilis modellhez. Ez egy alapterv, amit mindig az adott körülményekhez kell igazítani.

Vízesés modell[15][16][19][20]

A vízesés modell egy erősen tervalapú modell, amely alapvető célokkal rendelkező külön fázisokra bontható. A tervezési hibák, hiányosságok a későbbiekben súlyosan megbosszulhatják magukat. Az alábbiakban fázisonként megvizsgáljuk, hogy melyik fázisban mi a célunk, milyen tevékenységeket kell elvégeznünk, és milyen buktatók lehetnek.

Előkészítési fázis: Ebben a fázisban általában a megoldandó probléma elemzése folyik, és annak eldöntése, hogy a fejlesztés megéri és megvalósítható. A fázis végén születik egy döntés a projekt elindításáról vagy elvetéséről. Ehhez általában egy alapköltségvetést és alapmenetrendet állítanak össze egy nagyon felszínes megvalósítási modellel. A fázis hossza attól függ mekkora elemzést kell ahhoz végrehajtani, hogy ezeket az alapkérdéseket meg tudjuk válaszolni, ami néha meglehetősen hosszú idő (akár évek is lehetnek egyes esetekben), de az esetek többségében a pár hét a jellemző idő. Tesztelés szempontjából ilyenkor becsülni kell a tesztelési költségeket, ami egyrészről a tesztelés munkaerőigénye, másrészről annak eszközigénye határozza meg. Az eszközigény meghatározásához a következő szempontokat szoktuk általában figyelembe venni:

- **Technológiai részletek:** nagyon fontos, hogy milyen technológiával valósítjuk meg a fejlesztést. Ami alapvetően befolyásolja, hogy milyen eszközöket kell beszerezni. Általában eddigre meghatározásra kerül egy technológiai alap, amihez tartozik egy költségbecslés is. A technológiai tervek jó alapot biztosítanak a tesztelési eszközök becsléséhez. Ne feledjük el, hogy a tesztelési eszközök nem feltétlenül csak olyan eszközöket jelent, amit tesztelünk, hanem olyanokat is, amivel tesztelünk, vagy amivel menedzseljük a tesztelést

- **Magas szintű teszt stratégia:** A módszertan, ahogy tesztelünk, meghatározhatja a környezetek darabszámát, illetve a velük szemben támasztott teljesítményelvárásokat. Az egyéb tesztelési eszközök beszerzésére is csak egy vázlatos teszt stratégia alapján lehet javaslatot tenni.

- **Erőforrás becslés:** Az erőforrások száma befolyásol(hat)ja a megfelelő eszközök számát, amit használnunk kell a tesztelés folyamán

Összefoglalva:

- **Cél:** Alaprészletek, költségek becslése, megvalósíthatóság.
- **Tevékenységek:** Szükséges eszközök felmérése, becslése, költségelemzésük.
- **Kockázatok:** Korlátozott ismeretek a megvalósításról, a valós költség az ebben a fázisban becsült költség 50% és 200%-a közé tehető. Sok projekt elköveti azt a hibát, hogy ilyenkor lezárja a költségvetést, és nem tájékoztatja a megrendelőt a becslés bizonytalanságáról.

Követelmények meghatározása: Ebben a fázisban konkretizálódik a megoldandó feladat, az előzetes tervek is sokat finomodnak. Ahogy a feladat konkretizálódik, a környezetek terve is kiegészül, és beépülnek az új információk, valamint a teljes menetrend is sokat finomodik. Itt már el kell kezdeni a környezetek beszerzésének és beállításának menetrendjét is. Lehetségesek olyan követelmények, amelyek tesztelés szempontjából költségesek lehetnek (esetlegesen pont amiatt, hogy a tesztelés eszközigénye magas), ezen követelmények elemzésekor ezt jelezni kell. Amennyiben olyan követelmények maradnak benn a megvalósítandó elemek között, amelyeknek további eszközigényük van a tesztelés folyamán, ezekkel a környezetre vonatkozó terveket módosítani szükséges, illetve ezek költségigényét egyeztetni kell a projekt vezetésével.

Bár a követelmények meghatározása után még általában van egy időszak, amikor a tervezés folyik, figyelembe véve a környezetekre a beszállítási határidőket, szükség lehet egyes, a fejlesztést és a korai egységtesztelést segítő környezetek beszerzésére.

A teszt stratégia általában a fázis végére lezárásra kerül, amiben a tesztkörnyezetek számára és használatára vonatkozó irányelvek véglegesednek. A környezetek részletes tervének meghatározására ebben az időben már elkezdődnek, egy alapverzió rögzítésre kerül, de a tervezési fázis még jelentősen alakíthat rajta, amennyiben szükséges.

Összefoglalva:

- **Cél:** A tervek becslések finomítása, a követelmények elemzése, tesztkörnyezetek számosságára és használatának módjára vonatkozó irányelvek lefektetése.
- **Tevékenységek:** Teszt stratégia és annak környezetekre vonatkozó részének megtervezése, a követelmények elemzése környezetek és tesztelés szempontjából, a kockázatos (drága) elemek jelzése, előzetes architektúra tervek a tesztkörnyezetekre, előzetes menetrend a tesztelési és a környezetek beszerzésének és felépítésének tevékenységeire, amennyiben szükséges, a fejlesztés és egységtesztek környezetei beszerzésének elindítása.
- **Kockázatok:** Architektúra még változhat, ami módosíthatja a tesztkörnyezet igényeket, bár a becslések általában a fázis végén már javulnak, a végleges költség a becslés 67% és 150%-a között mozog. Szükséges lehet megrendelni egyes környezeteket, amelyeken később módosítani kell az esetleges tervezéskor felmerülő architekturális változások miatt.

Tervezési fázis: Az elsődleges cél a technikai megvalósítás és az architektúra részletes meghatározása. Ezek pontosítják a tesztkörnyezetekkel szembeni elvárásokat is, amelyek megjelennek a konkrét tervekben. Ilyenkor határozzuk meg a teljes technikai specifikációt a teszt stratégiában definiált környezetekhez.

A projekt menetrendje és költségei eddigre konkretizálódnak, bár a jelenlegi becslések pontossága még mindig nem teljes, a végeleges költség a jelenlegi becslés 80%-a és 125%-a közé várható.

A tervek véglegesítésekor el kell indítani a beszerzési folyamatokat, illetve a beszállítási határidők figyelembevételével estelegesen prioritásokat alkalmazva a korai tesztkörnyezeteket viszonylag hamar specifikálni kell, és el kell kezdeni a megrendelést, hogy a megfelelő időben elérhető legyen. A fejlesztés a fázis lezárásakor megkezdésre kerül, elengedhetetlen, hogy addigra a fejlesztés megkezdéséhez szükséges környezetek rendelkezésre álljanak.

Ugyanekkor a konkrét teendők mellé meg kell határozni a konkrét felelősöket, és biztosítani kell a megfelelő elérhetőséget. A konkrét feladatokért felelős lehet a fejlesztő, infrastruktúra fejlesztését és beszerzését irányító személy, a tesztelők, vagy egyéb feladatot ellátó, a projekthez közvetlenül nem tartozó személyek, mint például a pénzügy vagy a beszerzés munkatársai. A tesztkörnyezetek felépítése ennek a sokszínű csoportnak a koordinált együttműködésével jön létre, és nem szabad elfeledkezni a megfelelő bevonásról, hogy a megvalósítás sikeres legyen.

Ilyenkor már el kell kezdeni a teszteléshez kapcsolódó egyéb rendszerek konfigurálását és/vagy beszerzését is (például teszt menedzsment eszközök, tesztautomatizáló eszközök stb.)

Összefoglalva:

- **Cél:** A konkrét részletek meghatározása, az ütemtervek, tevékenységek és erőforrások véglegesítése, a beszerzések elindítása, a fejlesztés megkezdésekor szükséges környezetek felállítása.
- **Tevékenységek:** Tervek finomítása, a konkrét architektúra meghatározása, a megrendelések összeállítása, és az aktuális megrendelések elindítása, menetrend véglegesítése, erőforrás allokáció, a fejlesztéshez szükséges környezetek megépítése és előkészítése, teszteléshez kapcsolódó rendszerek beszerzése/felállítása.
- **Kockázatok:** A fejlesztői környezetek megrendelése általában korán – a tervezési fázis egy olyan fázisában, ahol a tervek még nem teljesek – történik, ami okozhat megfelelőségi problémákat, a több funkcionális csoportot felölelő tevékenységek koordinálása nehézkes lehet, érdemes bevonni a különböző csoportokat a fejlesztésbe, a szállítói határidők meglehetősen hosszúak lehetnek, a beszerzést érdemes lehet korán bevonni a tervezésbe.

Fejlesztési fázis: A fejlesztés folyamán a tesztelők a konkrét teszteseteket állítják elő, amelyeknek vannak tesztkörnyezetekre vonatkozó hatásai. Jó esetben az architektúra rendben van, mivel azt ilyenkor már nehézkesebb változtatni, de a szükséges adatokra vonatkozó követelményeket általában ilyenkor lehet összegyűjteni. Érdemes a tesztadatokra, azonosítókra, jogosultságokra és más egyéb konfigurációs

paraméterekre vonatkozó követelményeket és terveket egy helyen összegyűjteni, ami segíti a környezetek előkészítését.

Nem szabad elfelejteni, hogy egyes teszt folyamatok már a fejlesztés alatt folynak (például egységtesztek vagy egység integrációs tesztek), amelyhez tesztkörnyezet is szükségessé válhat.

Mindemellett szükséges a tesztkörnyezetek ütemterv szerinti összeállítása és előkészítése (tesztadatok, hozzáférések stb.). illetve a tesztelési eszközök teljes felállítása.

Összefoglalva:

- **Cél:** A fejlesztéssel párhuzamosan végrehajtott tesztelés támogatása, a tesztelési fázisra való előkészület, a tesztelési fázishoz szükséges eszközök teljes felkészítése.
- **Tevékenységek:** Tesztelési környezetek összeállítása, a konfigurációs igények felmérése és biztosítása, tesztelési eszközök teljes rendelkezésre bocsájtása.
- **Kockázatok:** Az előző fázisokból örökölt tervezési hibák egy bonyolult rendszernél súlyos problémákat okozhatnak, egyes adatelemeket bonyolult lehet biztosítani a teszteléshez, adatkezelési és biztonsági szabályokkal való konfliktusok, lassú reakcióidő egyes nem a projekt hatáskörében lévő csoportokban (például a hálózati csoportnak szigorú szabályai és bürokratikus folyamatai lehetnek egyes hálózati paraméterek megváltoztatásában, mint például tűzfal paraméterek megváltoztatása).

Tesztelési fázis: Itt már a cél a tesztelési folyamat fenntartása és támogatása, jó esetben építési feladatok nem szerepelnek a feladatok között. Szükséges lehet viszont a tesztrendszerek adatállományának, illetve azok egyéb tartalmának (konfiguráció, kód stb.) rendszeres frissítésére, az előzetes terveknek megfelelően.

Összefoglalva:

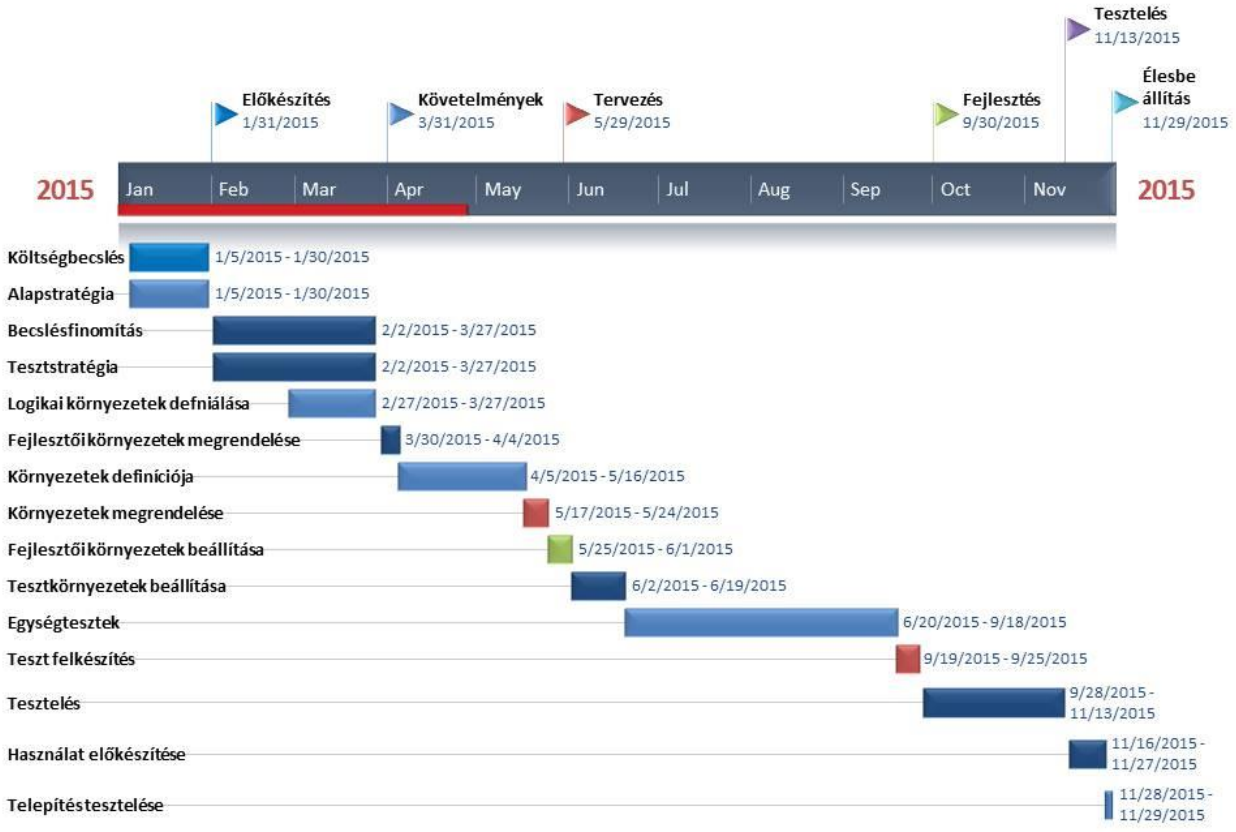
- **Cél:** A tesztelés céljainak és folyamatosságának biztosítása.
- **Tevékenységek:** A tesztkörnyezetek karbantartása, igény szerinti frissítése, módosítása.
- **Kockázatok:** Váratlan meghibásodás, alacsony teljesítmény (ez ebben a fázisban szinte biztosan újratervezéssel jár, ha ennek hatása van a végfelhasználói környezetre is), nem megfelelő előkészítés.

Üzemeltetési fázis: Az üzemeltetési fázisban általában a jelenlegi környezetek fenntartása és karbantartása a cél, ahol megfelelő frissítésekkel kell támogatni az üzemeltetési illetve a továbbfejlesztési igényeket. Természetesen, ha a szolgáltatást párhuzamosan fejlesztjük is, akkor szükséges lehet a környezetek módosítása, de mivel ekkorra már jelentős tapasztalattal rendelkezünk, ezek a módosítások általában egyszerűbbek, és kevesebb kockázatot, kihívást tartalmaznak.

Az üzemeltetés folyamán bármikor adódhatnak megoldandó problémák, amelyek megoldása megkövetel(het) tesztkörnyezeteket, amit megfelelő karbantartással tudunk elérni.

Összefoglalva:

- **Cél:** A meglévő környezetek karbantartása, megfelelő szinten tartása a problémák elhárítására, illetve a futó fejlesztések – mind a szolgáltatásé, mind az integrált rendszereké – biztosítása.
- **Tevékenységek:** A tesztkörnyezetek rendszeres frissítése, karbantartása, az esetleges továbbfejlesztések szerinti aktualizálása.
- **Kockázatok:** Alapvetően alacsony kockázatú fázis, a továbbfejlesztések változásai okozhatnak konfliktusokat, illetve egy karbantartási folyamat közben fellépő súlyos probléma kivizsgálása esetleg késedelmet szenvedhet.



Ábra 1 - Tesztkörnyezethez kapcsolódó tevékenységek egy példa vízesés projektben

Agilis modell[16][17][18][19]

Az agilis modell egy gyorsabb, agresszívabb indítású modell, amelyik jobban eltűri a változásokat. Mindezen tulajdonságok olyanok, amelyek a tesztkörnyezetek felépítésétben és karbantartásában kihívást jelentenek, mert ott alkalmazkodni kell a környezet nem feltétlenül ugyanolyan gyors szállítási

határideihez. Lényeges, hogy amit felépítünk, annak flexibilisnek kell lennie, és fel kell készülni a meglehetősen sűrű változtatásokra. Mivel az agilis projektek startja általában igen gyors, és a fejlesztés megkezdéséhez szükséges, hogy legyenek megfelelő környezetek, azokat nagyon rövid időn belül kell összeállítani. Hasonló a szituáció az egyéb tesztelési eszközökkel, azokra szinte a projekt kezdeteitől szükség van.

Előkészítési fázis: Ez a fázis bár igen gyors, általában komplexebb, mint a vízesés modellnél. Mindazon dolgok mellett, amit a vízesés módszernél végrehajtunk, itt fel kell állítani a fejlesztő csapatot, meg kell lennie legalább alapszinten, hogy hogyan működik a projekt, és már itt is rendelkezni kell egy alapszintű tesztelési stratégiával. Ugyan a folyamatok egy agilis fejlesztésnél egyszerűbbek, azok már az elején is szükségesek (tudván azt, hogy azok is alakulni fognak, és nem feltétlenül annyira dokumentáltak, mint a vízesés modellnél).

Ahhoz, hogy a fejlesztést el tudjuk kezdeni, szinte azonnal szükséges, hogy a fejlesztéshez, és a teszteléshez használt környezetek rendelkezésre álljanak, valamint a tesztelési eszközöknek is készen kell állnia a felhasználásra. Az alapkörnyezeti modellnek lehetőleg flexibilisnek kell lennie. Érdemes lehet egy felhőszolgáltatótól bérelni kezdetben az infrastruktúrát, hogy akkor fektessünk be a környezetekbe, amikor már megfelelő információnk van az igényekről, illetve az ilyen környezetek elindítása megfelelően gyors lehet a hagyományos beszerzéshez képest, illetve nagyon gyorsan változtathatóak a paraméterek, amennyiben szükséges.

Összefoglalva:

- **Cél:** Alaprészletek, teljes tesztelési ciklus terve, azok tesztkörnyezeti igényei és költségbecslése, menetrendi becslés a teljes projektre, fejlesztés elindításának biztosítása, az alapkörnyezetek létrehozása.
- **Tevékenységek:** Teszt stratégia meghatározása, alapbecslés a környezetekre, az azonnal szükséges környezetek definíciója, beszerzése és üzembe helyezése, a későbbi tesztelési igények becslése, az ott használt környezetekkel támasztott követelmények összegyűjtése, a megvalósítási alapmenetrend felállítása.
- **Kockázatok:** Nagyon kevés információval kell a kezdeti környezeteket felépíteni, a későbbiekben ez nem biztos, hogy megfelelő, vagy megfelelően rugalmas az igények szerinti módosítására. Érdemes ideiglenes megoldásként tekinteni a kezdetekre.

Fejlesztési fázis: A fejlesztési fázisban egyre több információ keletkezik a szükséges infrastruktúráról illetve architektúráról. Ezeket folyamatosan be kell építeni a szükséges tesztelési infrastruktúrába, és a tesztelési környezetek folyamatosan épülnek. Ez a folyamatos építkezés néha problémákat eredményezhet, és megkövetelhet akár szignifikáns újrastrukturálást is. Érdemes az architektúra meghatározó elemeit a korai fázisokban bevonni a fejlesztésbe, mivel ilyenkor még könnyebb alkalmazkodni és alakítani az infrastruktúrán, valamint felkészülni a komplexebb tesztfeladatokra.

Az agilis modellekben a tesztelés folyamatos, ezért folyamatosan fenn kell tartani a tesztkörnyezeteket, aminek lépést kell tartani a fejlesztéssel.

Az agilis modellekben általában benne foglalt a fejlesztési folyamatok folyamatos fejlesztése is, amelyek módosításokat igényelhetnek a tesztelési infrastruktúrában (környezetek, eszközök).

Összefoglalva:

- **Cél:** A meglévő tesztelési infrastruktúra elérhető szinten tartása, az alakuló architektúra és fejlesztési folyamatok változásának követése, a telepítés fázisának előkészítése, az ahhoz szükséges infrastruktúra felépítése.
- **Tevékenységek:** Eszközök beszerzése, beépítése, átstrukturálása, a tesztelési módszerek és eszközök fejlesztése, az alapmenetrend és tervek karbantartása.
- **Kockázatok:** A folyamatos fejlesztéssel nem mindig lehet lépést tartani, a szállítási határidők és egyéb külső tényezők megnehezíthetik az időbeli változtatásokat, felesleges beszerzések történhetnek a későbbi változtatások miatt, késői fázisokban történő architektúra változásoknak jelentős menetrendbeli változtatásokat okozhat.

Telepítési fázis: Ez nem minden agilis projektnek sajátja, de ha egy bonyolultabb infrastruktúrába kell a kész alkalmazást integrálni, néha elengedhetetlen egy végső tesztelési fázis beiktatása. Tipikusan ilyen egy rendszerintegrációs tesztelési, illetve egy felhasználói átvételi fázis, illetve dobozos termékek esetén egy alfa és béta teszt. Ezek a tesztek általában a végfelhasználó környezettel lényegében megegyező környezetben zajlik.

Mivel a környezet paraméterei inkrementálisan állnak elő ilyen esetben, amennyiben lényeges elemei az architektúrának későn véglegesednek, nagy a valószínűsége, hogy a külső, nem megváltoztatható tényezők miatt a megfelelő környezet nem áll időben rendelkezésre.

Összefoglalva:

- **Cél:** A tesztelés céljainak és folyamatosságának biztosítása, a biztonságos telepítés előkészítése, meggyőződni, hogy a végfelhasználói infrastruktúrába ágyazva a szolgáltatás megfelelően működik.
- **Tevékenységek:** A tesztkörnyezetek karbantartása, igény szerinti frissítése, módosítása.
- **Kockázatok:** Váratlan meghibásodás, alacsony teljesítmény (ez ebben a fázisban szinte biztosan újratervezéssel jár, ha ennek hatása van a végfelhasználói környezetre is), nem megfelelő előkészítés, lényeges architektúra elemek változása miatti csúszás az előkészítésben.

Üzemeltetési fázis: Lényegesen nem különbözik üzemeltetés szempontjából az agilis és a vízesés modellel előállított szolgáltatások üzemeltetése. Az egyetlen lényeges különbség lehet, hogy amennyiben az üzemeltetés mellett fejlesztés is folyik, és az továbbra is agilis modellel történik, akkor a tesztkörnyezeteknek hosszabb (állandó) rendelkezésre állást kell biztosítani.

Tesztkörnyezetek különböző architektúrákra

Az architektúra alapvetően befolyásolja a tesztelési módszereket illetve ezen keresztül a tesztkörnyezetek kezelését. A következő fejezetekben áttekintünk néhány gyakran használt infrastruktúrát, és az ott alkalmazott környezeteket, illetve a tipikus problémákat, megoldásokat.

Szerver-kliens architektúra

Ez talán a leggyakoribb olyan architektúra elem, ami előfordul a mai alkalmazásokban. Szinte nincs olyan alkalmazás manapság, az internet elterjedése után, ahol ez valamilyen kontextusban ne lenne része egy mai alkalmazásnak. A korábban tipikusan egyedülálló szoftverekben például játékok, fejlesztői eszközök, irodai programok stb. is megjelennek olyan funkciók, amelyek használnak ilyen kapcsolatokat (több résztvevős játékok, regisztráció, programfrissítések, szolgáltatás kiegészítések stb.).

Bár technológiai szempontból jelentős különbségek vannak a megvalósításokban, az alapelvek, ahogyan egy tesztkörnyezetet felépítünk, viszonylag kevés paramétertől függenek. Általánosságban elmondható, hogy azon esetekben, amikor már egy végfelhasználásra tervezett szolgáltatást nyújtunk (tehát nem egy általános szerveralkalmazást gyártunk), a kiszolgáló oldal specifikációja meglehetősen kötött, míg a kliensek általában nagyon sokszínűek lehetnek (természetesen vannak esetek, amikor ez is kötött, de ez inkább kivételnek tekinthető).

Más a helyzet, amikor egy dobozos szerverterméket fejlesztünk, amit később testre szabnak az adott végfelhasználási célnak megfelelően. Amikor ilyen terméket tesztelünk, akkor a várható specifikációknak és felhasználói igényeknek megfelelő tesztkörnyezetek építése szükséges.

Technológiai fogalmak

Ebben a fejezetben néhány fogalmat tisztázunk, ami segíti a megértést. A fogalmak után zárójelben szerepeltetjük az angol nyelvű megfelelőjét.

Szerver/Kiszolgáló (Server): Szervernek olyan, általában nagy teljesítményű számítógépet és/vagy szoftvert nevezünk, ami más számítógépek számára nyújt hozzáférést bizonyos szolgáltatásokhoz (például adatok, számítási kapacitás, nyomtató stb.) A szerverek általában egyszerre tartalmaznak hardver és szoftver komponenseket, bár egyes szerverek lehetnek annyira specializáltak (például egyes hálózati kiszolgálók, mint egy proxy szerver), ahol ez az úgynevezett firmware tartalmazza magát a szoftverkomponenst, és egyéb szoftver telepítésére nincs szükség illetve lehetőség.

Kliens (Client): A kliens egy olyan számítógép vagy szoftver, amely hozzáfér egy szerver által nyújtott szolgáltatáshoz.

Vékony kliens (Thin/Slim/Lean/Zero Client): A vékony és ellentéte a vastag kliens fogalmakat szokták definiálni mind hardveres, mind szoftveres értelemben. A hardveres értelmezés olyan számítógépet jelent, aminek szoftverei, adatai, és fő számítási kapacitása egy szerveren történik, és a kliens csak a megjelenítésért felel. Általában ezek a kliensek nem tartalmaznak nagy mennyiségű adat tárolására alkalmas eszközöket, illetve teljesítményük is behatárolt, és erőteljesen támaszkodnak a szerver által biztosított erőforrásokra, csak a szerverrel csatlakoztatva működőképesek.

Szoftveresen olyan architektúrát értünk alatta, amelyben a kliens csak a felhasználóval való interakcióért felel, a tényleges számítások illetve az adattárolás a szerveren valósul meg. Rendszeresen használják szinonimaként a webes fejlesztések architektúrájára. Ez az analógia jellegéből adódóan fennáll, de kezd eltolódni az irány egy hibrid architektúra irányába, ahol a kliens egyre komplexebb feladatokat vesz át a szerver oldaltól, illetve léteznek más vékony kliens megoldások is (bár nem elterjedtek).

Vastag/Gazdag/Kövér kliens (Thick/rich/Fat Client): A vastag kliens olyan számítógép, amely gazdag funkcionalitással rendelkezik, amelyik független a központi szervertől. A vékony kliens ellentéteként szokás meghatározni, általában saját háttértárat és programokat tartalmaz, nagyobb a számítási kapacitása és nem feltétlenül követel meg állandó szerverkapcsolatot.

Értelmezhető szoftveres környezetben is, ilyenkor általában olyan programról, atrchitektúráról beszélünk, amelyik a fő számítási teljesítménye a kliens oldalról származik, és a szerver valamilyen kiszolgáló funkciót lát csak el (pl. adattárolás).

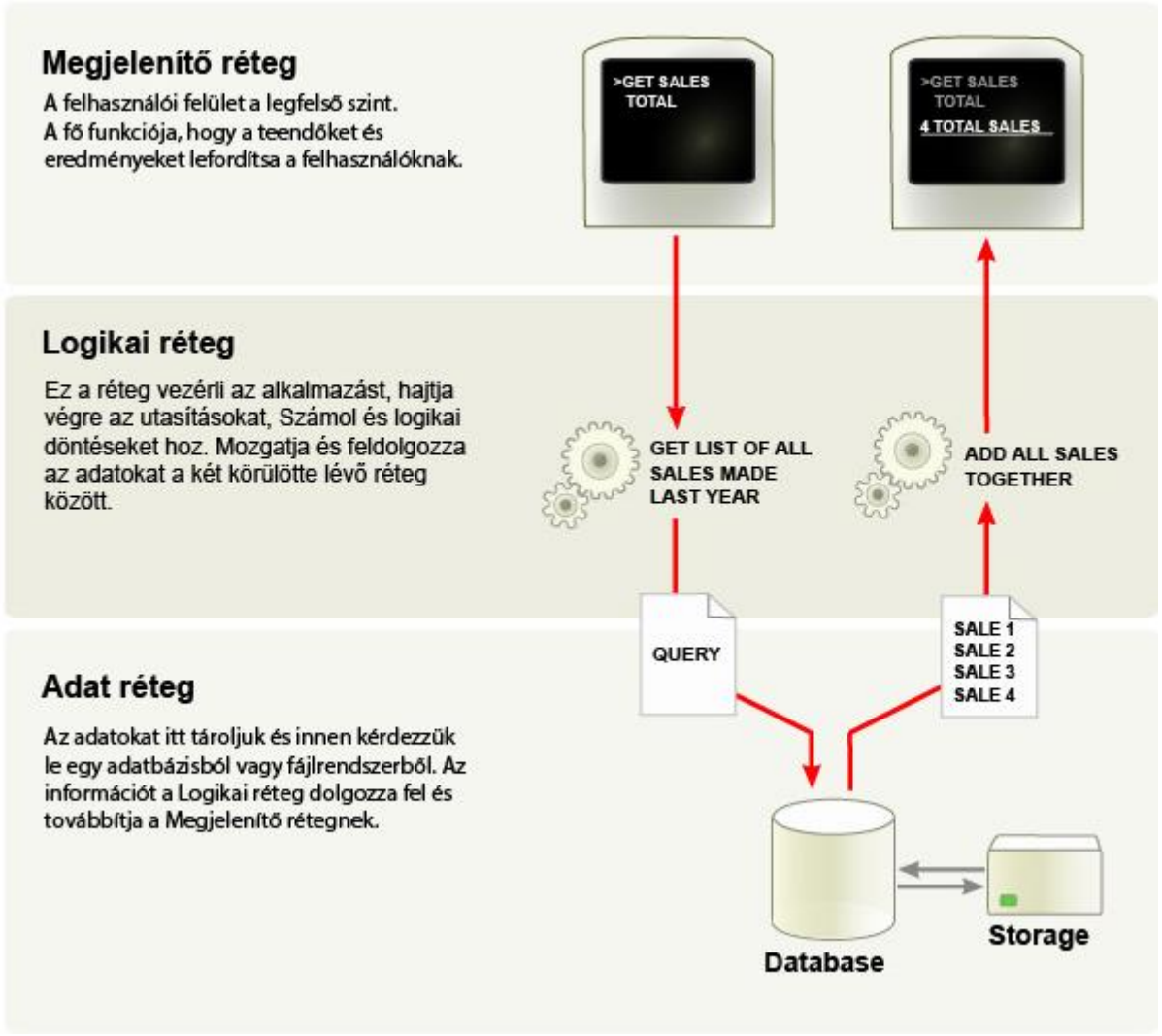
Mind a vékony, mind a vastag kliensnek vannak előnyei és hátrányai, amit a tervezésnek figyelembe kell vennie. A vastag kliens a szerver oldalon alacsonyabb teljesítmény igényt és költséget jelent, gazdagabb megjelenítést tesz lehetővé (pl. multimédiás tartalmak), viszont kliens oldalon magasabb a beruházási igény, és általában a kliens oldali architektúra változatos. Vékony kliens esetén a kliens oldal jól menedzselhető, és egységes, viszont csak a központi szerverrel együtt működik, a szerver oldalon nagyon magas számítási kapacitás szükséges, bizonyos funkciók csak nehezen biztosíthatóak.

Habár egy friss beruházásnál valószínűleg a vékony és a vastag klienses megoldás nagyságrendileg hasonló nagyságrendű beruházást igényel, egy meglévő infrastruktúrában már alkalmazkodni kell a meglévő környezethez. Új beruházásnál az előnyök és hátrányok gondos mérlegelésével kell eldönteni, hogy mire van szükség.

Hibrid kliens (Hybrid client): A vékony és a vastag kliens közötti kompromisszum, amit általában a való életben alkalmazunk. Ebben az esetben általában a kliens oldal is elég erős, lokálisan is dolgozik, de felhasznál valamennyit a vékony kliens koncepcióból, ami segíti a menedzselhetőséget és a rugalmasságot. Bár a webes alkalmazásokat általában vékony kliens architektúrának állítjuk be, manapság egyre több számítás kerül le kliens oldalra, valamint egyre több multimédiás képességet várunk el tőlük, ezért egyre inkább hibrid alkalmazásnak tekinthetőek.

Többrétegű alkalmazásfejlesztés (Multitier/n-tier Architecture): Egy olyan szerver-kliens architektúrára alkalmazott alkalmazásfejlesztési koncepció, amikor a különböző rétegekben jól szeparáltan más és más feladatokat látunk el. Ezek a rétegek megjelennek a hardver architektúrában. A leggyakoribb megvalósítások két illetve három réteget alkalmaznak, de előfordulhat több réteg is.

Háromrétegű alkalmazások (3-Tier Architecture): A többrétegű alkalmazások egy speciális formája, amit a webes alkalmazások általánosan használnak. A három réteg a megjelenítés, az üzleti logika és az adat réteg.



Ábra 2 - Háromrétegű alkalmazásmodell[21]

Kétrétegű alkalmazások (2-Tier Architecture): A háromrétegű alkalmazásokkal szemben definiált architektúra, amikor a megjelenítő és az üzleti logika rétegeket egyben valósítjuk meg.

Virtuális gép/szerver (Virtual Machine): A virtuális gép egy számítógépes program által szimulált számítógépet jelent. A szimuláció lehet teljes rendszerszintű, amikor komplett számítógépet emulálunk, illetve folyamatszintű, amikor egyetlen folyamat futtatására szolgál egy gazdaszámítógépen, aminek célja, hogy platform független környezetet teremtsen. Ez utóbbira a legismertebb példa a Java virtuális gép. Szerver-kliens alkalmazások esetében a teljes szimuláció az érdekesebb változat (habár a második is része lehet a megoldásnak, de az inkább rendszeradminisztrációs és architekturális kérdés).

Virtuális gépekkel tudunk szeparálni környezeteket, miközben nincs szükség több hardver hozzáadására a rendszerhez, illetve tudunk létrehozni különböző kliens oldali konfigurációkat anélkül, hogy jelentős hardverberuházást kellene tennünk.

Terheléselosztás (Load Balancing): Terheléselosztási technikákkal el tudjuk osztani a terhelést több különálló erőforrásra (ez lehet teljes számítógép, vagy csak elemek, mint feldolgozó egységek, vagy adattároló egységek). Az elsődleges cél, hogy olyan terhelést is ki lehessen szolgálni, amit az egyedi elemek önmagunkban nem lenének képesek. Az ilyen környezetek felépítése általában jelentős befektetést igényel, és szükség van olyan tesztkörnyezet(ek)re, ahol ezek a technológiák aktívak, mert potenciális hibaforrást jelenthetnek. A magas befektetés miatt ezen környezetek száma általában korlátozott, és a tesztelési feladatokat érdemes elosztani olyan módon, hogy azok nagy része olcsóbb környezeteken is kivitelezhető legyen.

Fürt (Cluster): Egy fürt több számítógépet tartalmaz gyenge vagy szoros kapcsolatban, amelyek úgy dolgoznak együtt, hogy az kívülről sok tekintetben egy egyszerű rendszerként viselkedik. A fürt minden eleme azonos feladatot lát el, és egy szoftver felügyeli a terhelés elosztását. Általában a teljesítmény és a megbízhatóság növelése céljából alkalmazzák (miközben általában olcsóbb megoldás, mint egyetlen számítógéppel helyettesíteni, ami hasonló teljesítményt és rendelkezésre állást produkál).

Csonk (Stub): egy szoftverkomponens speciális célú vagy részleges megvalósítása. A csonkot arra használjuk, hogy támogassuk a komponens(ek) fejlesztését vagy tesztelését. Helyettesíti a meghívott komponenst. Bár egyszerűbb teljes integrált környezetekkel dolgozni, erre nem mindig van lehetőség (mert például még nincs kész a komponens, vagy olyan kommunikációt indítana, amit kerülnie kell stb.)

Meghajtó (Driver): egy szoftverkomponens, vagy teszt eszköz, amely kiváltja azt a komponenst, amely egy másik komponens, vagy a rendszer vezérlését, és/vagy felhívását végzi. Hasonló szerepet tölt be, mint a csonk, csak nem az általunk végzett műveletekre ad meghatározott választ, hanem vezérli a kommunikációt. Használatára hasonló indokok vannak, mint a csonkok használatára.

DTAP modell [22][23][24][25][26]

Bár architektúra szempontjából a szerver-kliens alkalmazások sokszínűek, tesztelési folyamat szempontjából nem feltétlenül különböznek jelentősen, és ahogyan a tesztkörnyezeteket felépítjük és használjuk, sem különbözik alapjaiban. A DTAP modell a szerver oldali tesztkörnyezet felépítésére szolgál, olyan esetekben, amikor végleges szolgáltatást készítünk. Ez az eset fedi le az esetek nagy részét, kivételt képeznek azonban a szervertermékek, mint úgynevezett dobozos termékek fejlesztése és tesztelése, ami jobban hasonlít más, nem szerver, de dobozos termék teszteléséhez. A DTAP modell elnevezés angol alapú, a környezetek felhasználásának kezdőbetűiből áll:

Development (Fejlesztői)
Test (Tesztelési)
Acceptance (Átvételi)
Production (Végfelhasználói)

Szokásos elnevezések még a tesztkörnyezetre az integrációs teszt (Integration Test, IT) elnevezés is, illetve az átvételi környezetet szokás még minőségbiztosítási (Quality Assurance/QA/Stage) környezetként is használni. Ez a négyszintű környezetrendszer, bevett iparági gyakorlatként, alapként szokott szolgálni a további testre szabáshoz a tesztelés folyamata és a minőségi követelmények függvényében. Az alaprendszer flexibilis, és jól variálható, de vannak olyan esetek, amikor ez nem

elégséges, mert a minőségi követelményrendszer olyan magas, hogy további környezetek definiálása is szükséges lehet. Általános felhasználási feltételek mellett a négyszintű rendszer egy megfelelő kompromisszum szokott lenni a kockázatok, költségek és fejlesztési hatékonyság szempontjából.

Előfordulhatnak olyan esetek, amikor olyan fejlesztési feladat adódik, aminek speciális tesztelési igényei lehetnek, és olyan környezeti igényei vannak, ami nem fér bele az adott sémába. Ilyenkor szokás úgynevezett Sandbox (Homokozó) környezeteket létrehozni. Ezek a környezetek általában ideiglenesek, és rövid életűek, de előfordulnak esetek, amikor ezeket hosszabb távon be kell építeni a tesztelési rendszerbe.

Az egész rendszert egy alapesetből, mint modellből építjük fel, és egymás után adjuk hozzá azokat a tényezőket, amik az egész rendszert képessé teszik egyre bonyolultabb kihívások kezelésére.

Az alapeset

Vegyünk egy olyan fejlesztést, ahol a fejlesztés egyszeri, a rendszer továbbfejlesztését az átadás után nem tervezzük, és nincs jelentős külső kapcsolata, integrációja.

Ez a legegyszerűbb eset, habár meglehetősen ritka, mind a továbbfejlesztés jogos igény szokott lenni, és az sem szokásos, hogy ne lennének külső kapcsolatok. Ilyenkor még elég három környezet is, sőt, bizonyos kockázatok felvállalása esetén akár kettő is.

A tesztkörnyezetek életciklusa szempontjából kétfelé tudjuk bontani az igényeket.

1. **Szakasz – Fejlesztés**
2. **Szakasz – Karbantartás**

Fejlesztési szakasz

A fejlesztési ciklusban általában szükség van egy olyan környezetre, ahol a fejlesztők tudnak dolgozni, tudják integrálni a kódot, és ahol a kódbázis nagyon gyorsan tud változni, és nem kell foglalkozni más igényekkel, stabilitással, illetve mindenképpen szükséges egy szeparált tesztelési környezet, amelyben el tudunk érni stabilitást, és a tesztelési igények szerint történnek benne a változások.

Környezet	Felhasználás	Követelmények
Fejlesztői (Development)	Fejlesztési feladatok, fordítás, hibakeresés, nagyfokú hozzáférhetőség a fejlesztőknek, olyan egységtesztek, amelyek nagyfokú hozzáférést követelnek, alapvető egységtesztek	- Nagyfokú hozzáférés - A végfelhasználó környezetet megközelítő szoftver és hardverkörnyezet - Megengedett lehet az alacsonyabb teljesítmény és tárhely (megfelelő tervvel a kockázatok kezelésére)
Teszt (Test)	Integrációs, rendszer, rendszerintegrációs, átvételi tesztek, teljesítménytesztelés,	- Stabilitás - Korlátozott, végfelhasználói jellegű hozzáférések (kivételkezeléssel,

	felhasználói tréning	amennyiben bizonyos tesztek megkövetelik) - A végfelhasználó környezettel megegyező szoftver és hardverkörnyezet illetve adatmennyiség és minőség

Táblázat 2 - DTAP alapeset, fejlesztés közbeni környezethasználat

Karbantartási fázis

Amint a projekt elkészült teljesen más igényeknek kell megfelelni, megjelennek a felhasználók, és a végfelhasználói környezet, amelynek nagy rendelkezésre állási időt kell biztosítani, nagyon kötött szabályokkal. Mivel nincs további fejlesztés, akár gondolhatnánk, hogy több környezetre ilyenkor már nincs is szükség, de minden terméknél számolni kell váratlan helyzetekkel, nem tervezett meghibásodásokkal, esetleg olyan apró változtatási igényekkel, amelyek további apró fejlesztést és tesztelést követel. Ezen tevékenységek elvégzése a végfelhasználói környezetben nagyon kockázatosak, ezért szokás párhuzamos környezeteket fenntartani, hogy az ilyen váratlan helyzetek kezelésére legyenek olyan környezet(ek), amelyek lehetővé teszik az adott hibajelenség vizsgálatát, esetleges javítását, és annak tesztelését. Itt is érvényes, hogy szerencsés a tesztelési és fejlesztési munkákat szétválasztani, bár amennyiben tényleg csak hibajavítás, és sürgősségi beavatkozás szükséges, elégséges lehet egyetlen környezet fenntartása.

Környezet	Felhasználás	Követelmények
Fejlesztői (Development)	Fejlesztési feladatok, fordítás, hibakeresés, nagyfokú hozzáférhetőség a fejlesztőknek, olyan egységtesztek, amelyek nagyfokú hozzáférést követelnek, alapvető egységtesztek	- Nagyfokú hozzáférés - A végfelhasználó környezetet megközelítő szoftver és hardverkörnyezet - Megengedett lehet az alacsonyabb teljesítmény és tárhely (megfelelő tervvel a kockázatok kezelésére)
Teszt (Test)	Hiba reprodukálása, javítások, apróbb fejlesztések ellenőrzése és tesztelése a végfelhasználói környezetben való alkalmazás előtt, tréningek, konfigurációs változtatások próbája, tesztelése	- Stabilitás - Korlátozott, végfelhasználói jellegű hozzáférések (kivételkezeléssel, amennyiben bizonyos tesztek megkövetelik) - A végfelhasználó környezettel megegyező szoftver és hardverkörnyezet illetve

		adatmennyiség és minőség
Végfelhasználói (Production)	Csak végfelhasználói használat **Nem engedélyezett tevékenységek:** - Tesztelés (kivéve egy rövid tesztet módosítás után) - Hibakeresés - Kód és konfigurációmódosítások előzetes tesztelés nélkül **Kerülendő tevékenységek:** - Tréningek - Adatmódosítással járó tesztek	- 24/7 működés - Stabilitás - Korlátozott jogosultságok

Táblázat 3 - DTAP alapeset, karbantartás közbeni környezethasználat

Amennyiben úgy döntünk, hogy nincs szeparált fejlesztői illetve tesztkörnyezetünk, akkor a tesztkörnyezet erősebb követelményeit kell figyelembe venni, de valószínűleg mellette szükségünk lesz a meggyengített hozzáférés-kontrollra.

Beszerzéskor érdemes mind a fejlesztői, a teszt és a végfelhasználói környezetet létrehozni, ahol fejlesztés alatt a végfelhasználói környezet szolgálhat még egy extra tesztkörnyezetként, ami egy rövid projekt menetrendnél nagy segítséget tud biztosítani (ilyen lehet például, hogy a teljesítményteszteket a végfelhasználói környezetben futtatjuk, minimalizálva evvel az esetleges nem várt negatív mellékhatásokat, következményeket). Amennyiben nem lesz külön fejlesztői és tesztelői környezet a támogatáshoz, akkor a végfelhasználói környezet a korábbi tesztkörnyezet és a fejlesztői környezetből lesz a kombinált fejlesztői és tesztkörnyezet (lehet fordítva is, de tapasztalatok alapján ez az egyszerűbb).

Néhány esetben a végfelhasználói környezet hardverigénye nagyon magas (mert nagy felhasználószámot kell kiszolgálnia), ezáltal nagyon drága lenne egy ugyanolyan tesztkörnyezetet hosszú távon fenntartani. Jogos igény ilyenkor, hogy egy alacsonyabb fenntartási költségű alternatíva álljon rendelkezésre.

Ilyen esetekben általában a komponenseket arányos mértékben csökkentjük úgy, hogy a lényeges elemek még mindig meglegyenek (például egy 8 szerveres klasztert 4 vagy 2 szerveres klaszterrel helyettesítünk, de nem helyettesíthetjük egyetlen elemmel, mert akkor már jelentősen módosítjuk az architektúrát, vagy a szerverekben mondjuk 32 mag helyett csak 8-at, vagy 4-et alkalmazunk). Az ilyen teljesítményt jelentősen módosító változtatások nagyban módosítják egy teljesítményteszt eredményét. Ilyen esetekben a teljesítményteszteket a végfelhasználói környezetben hajtjuk végre először (mielőtt használatba vennénk), és a teljesítményteszt célja nem csak annak megállapítása, hogy a végfelhasználói rendszer hogyan viselkedik, hanem egy olyan modell megalkotása is, ami megmutatja, hogy a teszt és végfelhasználó környezet teljesítménye között milyen korreláció van, és a későbbiekben hogyan következtethetünk a tesztkörnyezetben mért adatokból a végfelhasználói környezetben várható viselkedésre.

Továbbfejlesztéssel együtt

Ez az eset hasonló az előzőhöz, azaz a szolgáltatás nem tartalmaz jelentős külső integrációt, viszont számítunk arra, hogy az első kiadás után lesznek még jelentős további fejlesztések a rendszeren, de egy alkalommal csak egy jövőbeli kiadás készül.

Az első kiadás fejlesztése ugyanaz az eset, mint az előző, ebben nincs lényeges különbség, viszont amint az első kiadás használatba kerül, két tesztkörnyezetre lesz szükségünk. Egyre, ami a jelenlegi aktív kiadást, és egyre, ami a következő kiadás fejlesztését támogatja.

Amennyiben megoldható, hogy mindkét tesztkörnyezet azonos teljesítményű legyen, mint a végfelhasználói környezet, akkor több stratégia is rendelkezésre áll a kiadáskezelés számára. A folyamat szempontjából legegyszerűbb, hogy a tesztkörnyezetben folyik a fejlesztés és tesztelés végig, és amikor a végfelhasználói környezetet frissítik, akkor frissítik a végfelhasználói környezet tesztkörnyezetét is.

Drága és bonyolult környezetek esetén viszont két drága tesztkörnyezetet nem feltétlenül érdemes fenntartani. Tehát, ha van egy gyengébb, illetve egy erősebb tesztkörnyezetünk, jogos a kérdés, hogy melyik legyen a fejlesztéshez, illetve melyik legyen a karbantartáshoz?

Ha megvizsgáljuk, hogy miért van szükség teljesen végfelhasználói szintű környezetre, akkor azt lehet látni, hogy azért, mert lehetnek olyan hibák, amelyek az eltérő teljesítmény illetve architektúra miatt nem jönnek elő. Illetve lehetnek olyan tesztesetek, amelyek annyi, vagy olyan minőségű adatot követelnek, amelyhez csak a nagyteljesítményű környezetben van megfelelő kapacitás. Vagy lehet, hogy a teljesítménytesztek követelnek meg olyan környezetet, amelyik megegyezik a végső környezettel. Látható, hogy ezek speciális esetek, és a tesztelés nagy része elvégezhető egy kisebb teljesítményű környezeten is.

Ezek után még mindig jogos a kérdés, hogy melyik legyen akkor a nagyteljesítményű. A javasolt, hogy a karbantartásra használt változat legyen az. Amikor probléma adódik a végfelhasználói környezetben, akkor azonnali és sürgős beavatkozás lehet szükséges. Ráadásul szerepet játszhat a hiba okában egy olyan tényező (például klaszterkonfigurációs hiba), ami a gyengébb tesztkörnyezetben nem reprodukálható.

Felmerül a kérdés, hogy akkor hogyan ellenőrizzük azokat a speciális eseteket, amit az előbb megemlítettünk. A megoldás, hogy szakaszoljuk a tesztelést. A legtöbb időt a gyengébb tesztkörnyezetben töltjük el, és amikor elértük azt a minőséget, amit később szeretnénk látni, leteszteljük a speciális eseteket immár a végfelhasználói tesztkörnyezetben.

De ha használjuk a végfelhasználói környezet tesztkörnyezetét is, akkor mit csinálunk, ha hiba van abban? A kérdés jogos, hiszen ebben az időszakban elveszítjük a lehetőségét az azonnali hibaelhárításnak, ami egy kockázat ebben az esetben. Vannak esetek, amikor ez a kockázat egyáltalán nem engedhető meg, de a legtöbb esetben nem ez a helyzet.

Ha megvizsgáljuk a kockázatot, a tapasztalatok azt mutatják, hogy a Pareto-elv erre is érvényes, azaz a problémák 80%-a a kiadás használatának első 20%-ban fordul elő, tehát a kiadás használati idejének a legvégén már nagyon ritka, hogy váratlan hiba lépjen fel. Az viszont fontos, hogy az itt töltött tesztelési

időt minimalizáljuk, tehát itt már tényleg csak a végső tesztek futnak (általában a rendszerintegrációs vagy teljes folyamat (End to End) tesztelés és az átvételi (Acceptance) tesztek), innen származik az átvételi környezet elnevezés. Gyakori hiba, amikor a határidő közeleg, hogy már akkor átlépnek ebbe a környezetbe, amikor még jelentős nyitott hibák vannak az adott kiadásban. Sokszor próbálkoznak így behozni a lemaradást, de legtöbb esetben ez nem sikerül. Sőt a tapasztalat inkább az, hogy szükségtelenül növeli a projekt komplexitását, és inkább késlelteti azt, mintsem felgyorsítaná. Természetesen, mint mindig, lehet helye a kivételnek, ha csak egy vagy két nagyobb probléma van, de nagy mennyiségű probléma esetén, vagy befejezetlen tesztelésnél nem javasolt ez a fajta időspórolás. Illetve mivel nagy a valószínűsége, hogy az adott kiadás késni fog, és beragad abba a környezetbe, ami a végfelhasználói környezet esetleges hibajavításához kell, feleslegesen kockázatként egy jelentősebb szolgáltatás kiesést okozhat.

Mi tehetünk, ha mégis baj történik a végfelhasználói környezetben? Először is, szükséges egy megfelelő válságtervvel készülni az ilyen szituációra, hogy ne abban a pillanatban kelljen rögtönözni. Több lehetőség is van az adott probléma kezelésére. Hogy melyiket alkalmazzuk, azt mindig a megfelelő kockázatelemzéssel kell eldönteni.

1. **Lehetőség – Az átvételi környezet visszaállítása:** Ez bármikor alkalmazható, az adott tesztelést meg kell szakítani, az adott állapotot, adatokat a hiba sürgőssége alapján lehet menteni (ha van rá lehetőség és idő, hogy ne kelljen esetleg függőben maradt teszteseteket újrakezdeni), visszaállítani a tesztelés megkezdése előtti állapotot, a hibát elhárítani, majd a tesztelési rendszert visszaállítani és folytatni a tesztelést. A tesztelés újraindítása előtt analizálni kell a tesztelést befolyásoló történéseket, és szükség lehet bizonyos tesztesetek újbóli futtatására.

2. **Lehetőség – Az új kiadás idő előtti használatba vétele:** Ez a lehetőség függ a végső tesztek állapotától. Ha ezek megfelelően előrehaladott állapotban vannak, és semmi váratlan probléma nem lépett fel, érdemes lehet előrehozni a kiadás megjelenését. Az ilyen döntések természetüknél fogva erős kockázatot jelentenek. A döntés meghozatala előtt gyors, de alapos kockázatelemzés szükséges, és megfelelő tervekkel kell rendelkezni a kiadás visszavonására, amennyiben probléma adódik vele. Itt is ellenőrizni kell az adott végfelhasználói környezetben tapasztalt hibajelenséget. Amennyiben reprodukálható, az adott hibát előbb még ki kell javítani a következő kiadásban is (nem feltétlenül ez a rosszabbik eset, hiszen így szinte biztosan megtaláltuk és javítottuk a hibát, és nem fedte el esetlegesen valami más módosítás megváltoztatva annak viselkedését).

Ez az az eset, amikor először beszélhetünk DTAP modellről. Az alábbi táblázatban foglaljuk össze a DTAP modell környezeteinek tulajdonságát.

Környezet	**Felhasználás**	**Követelmények**
Fejlesztői (Development)	Fejlesztési feladatok, fordítás, hibakeresés, nagyfokú hozzáférhetőség a fejlesztőknek, olyan egységtesztek, amelyek	- Nagyfokú hozzáférés - A végfelhasználó környezetet megközelítő szoftver és hardverkörnyezet - Megengedett lehet az alacsonyabb

	nagyfokú hozzáférést követelnek, alapvető egységtesztek	teljesítmény és tárhely (megfelelő tervvel a kockázatok kezelésére)
Teszt (Test)	Integrációs, rendszer, rendszerintegrációs tesztek, előzetes teljesítménytesztek	- Viszonylagos stabilitás - A tesztelési igények prioritása - Korlátozott, végfelhasználói jellegű hozzáférések (kivételkezeléssel, amennyiben bizonyos tesztek megkövetelik) - A végfelhasználó környezetet megközelítő szoftver és hardverkörnyezet - Megengedett lehet az alacsonyabb teljesítmény és tárhely (megfelelő tervvel a kockázatok kezelésére)
Átvételi (Acceptance)	Hiba reprodukálása, javítások, apróbb fejlesztések ellenőrzése és tesztelése a végfelhasználói környezetben való alkalmazás előtt, tréningek, konfigurációs változtatások próbája, tesztelése Végső tesztelés egy végfelhasználóinak megfelelő környezetben, átvételi tesztek, végső teljesítménytesztek	- Stabilitás - Korlátozott, végfelhasználói jellegű hozzáférések (kivételkezeléssel, amennyiben bizonyos tesztek megkövetelik) - A végfelhasználó környezettel megegyező szoftver és hardverkörnyezet, illetve adatmennyiség és minőség
Végfelhasználói (Production)	Csak végfelhasználói használat **Nem engedélyezett tevékenységek:** - Tesztelés (kivéve egy rövid tesztet módosítás után) - Hibakeresés - Kód és konfigurációmódosítások előzetes tesztelés nélkül **Kerülendő tevékenységek:** - Tréningek - Adatmódosítással járó tesztek	- 24/7 működés - Stabilitás - Korlátozott végfelhasználói jogosultságok

Táblázat 4 - DTAP teljes, környezethasználat

Mit tehetünk, ha csak három környezetre van finanszírozás? Feladatok összevonásával valamennyire kezelhető a probléma, de ezzel beépítünk egy többlet kockázatot a rendszerbe.

Összevonhatjuk a fejlesztési és tesztkörnyezeteket, ilyenkor a gyorsan változó környezet megnehezítheti a tesztelést és a hibák reprodukálását. Vagy összevonhatjuk a tesztelési és átvételi környezeteket, ekkor viszont a végfelhasználói környezetben váratlanul fellépő hibák vizsgálata és javítása lesz nehézkes. Mindig az adott kockázatok szerint érdemes mérlegelni, hogy melyik a jobb megoldás ilyen esetekben. Illetve meg kell vizsgálni a lehetőségét, hogy egy alacsonyabb teljesítményű és költségű fejlesztési és tesztkörnyezettel esetlegesen megoldható-e a szeparált környezet. Esetleg szerver virtualizálással lehet kétfelé osztani a tesztelési és fejlesztői környezeteket, mivel az alacsonyabb teljesítmény ezekben a környezetekben általában jól kezelhető.

Ha csak két környezetre van finanszírozás, egy fejlesztési/teszt és egy végfelhasználói környezetre, a kompromisszumok és kockázatok még erősebben jelentkeznek. Általános tapasztalat, hogy ilyenkor már akkora a veszteség a humánerőforrás oldalon ezeknek a kockázatoknak a kezelésén, hogy jelentős megtakarítást hoz, ha a fizikai infrastruktúrába fektetünk. Talán csak nagyon kicsi projekteknél és szolgáltatásoknál lehet ennek létjogosultsága.

Bár csak elméleti lehetőségként említem meg, játszunk el az egyetlen környezetes gondolattal. A kezdeti projekt még csak-csak megoldható, bár ott sem lesz ideális, viszont a továbbiakban egy futó szolgáltatás melletti fejlesztés vállalhatatlan kockázatot és kompromisszumokat jelent, még úgy is, hogy a két oldal (a végfelhasználói és a tesztelési) szoftveresen elkülönül, könnyen előidézhető olyan váratlan szituáció, ami jelentősen befolyásolja a szolgáltatás színvonalát.

Több párhuzamos fejlesztés

Ebben az esetben nem csak egy kiadás fejlesztése folyik egy adott időben, hanem több kiadáson folyik fejlesztés egyszerre. Ilyen lehet például, amikor van egy havi kiadás az apró fejlesztéseknek, illetve egy negyedéves a mélyre hatóbb változásoknak.

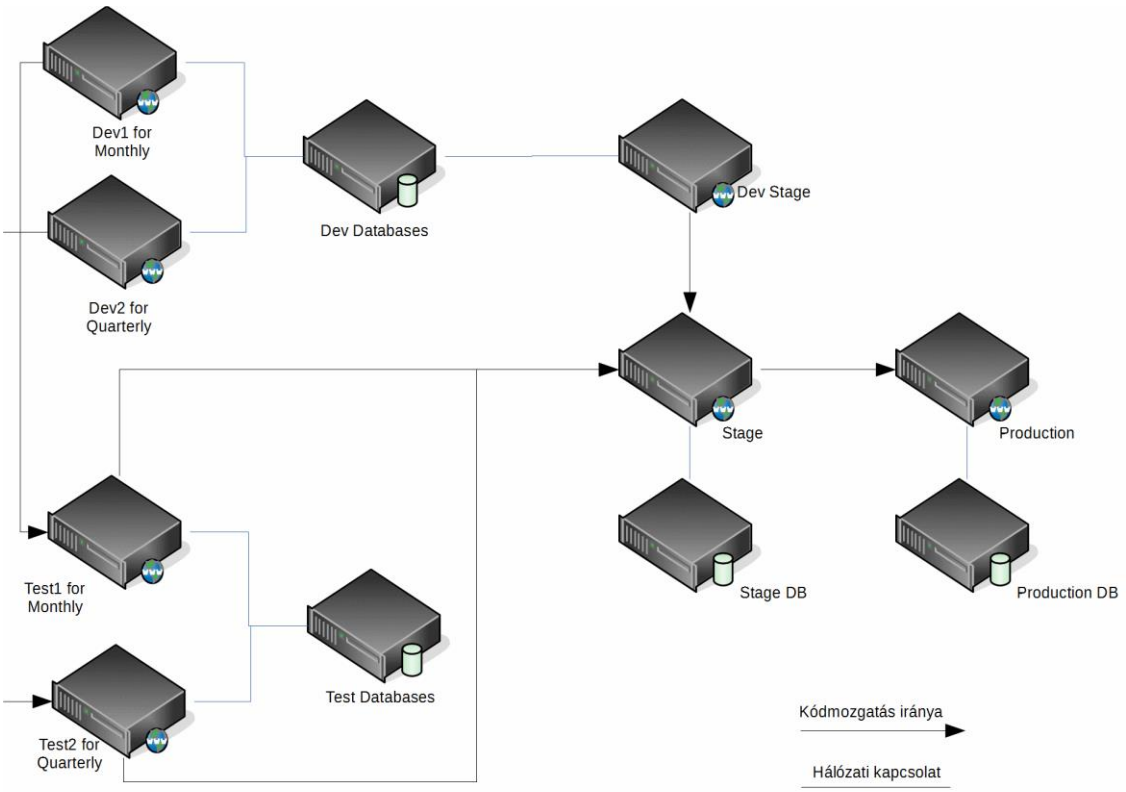
Tulajdonképpen a probléma már az előző változatban is adott, csak nem feltétlenül túl bonyolult, hiszen amikor hibát javítunk a végfelhasználói kiadásban, akkor párhuzamosan két kiadáson dolgozunk, még ha a hibajavítás csak amolyan mini kiadás is.

Hogy ezeket a problémákat megoldjuk, megfelelő konfigurációmenedzsmentet kell fenntartanunk. Már az előző példában is, amikor csak sürgős hibákat javítunk, biztosítani kell, hogy a megfelelő javítás belekerüljön a következő kiadásba is, különben az új kiadással a hiba ismét megjelenhet. Több kiadás egyszerre fejlesztésénél ez a probléma még bonyolultabb.

Az egyik megoldás az adott problémára, hogy annyi fejlesztői és tesztkörnyezetet tartunk fenn, ahány kiadáson egyszerre dolgozunk, és nem keverednek a kódbázisok. A konfigurációmenedzsment alapja az, hogy az alacsonyabb verziószámú (hamarabb kiadott) konfigurációból a változások az összes magasabb verziószámú konfigurációba migrálódnak. A modell kétségtelen egyszerűsége és hatékonysága mellett azonban meg kell említeni, hogy a modell eszközigénye és így fenntartási költsége is magasabb, mintha csak egy fejlesztői és tesztkörnyezetünk lenne. A költségtöbblet a környezetek természetes költségei (beszerzési ár, avulás, karbantartás, meghibásodások javítása) mellett humánerőforrásban is jelentkeznek, hiszen jelentősen nőnek az adminisztrációs költségek és terhek is. Előre meg kell jegyezni,

hogy bonyolultabb infrastruktúra és jelentős integrációk mellett ez a modell nagyon nehezen fenntartható, amit bővebben megvizsgálunk egy kicsit később.

Egy ábrán szemléltetve az alább látható egy példa egy ilyen konfigurációra.



Ábra 3 - Dedikált fejlesztési és tesztelési környezet minden tervezett kiadásnak

A modell korlátozza a környezetek számát, és csak egy fejlesztői és egy tesztelői környezetet engedélyez. A kulcs itt is a konfigurációmenedzsment, és az adott konfigurációk engedélyezése a különböző környezetekben, azaz a különböző környezetekben különböző fejlesztési fázisok szerepelnek.

Környezet	**Engedélyezett kiadás**
Fejlesztői (Development)	- Bármely már engedélyezett megkezdett fejlesztés
Teszt (Test)	- Csak és kizárólag az adott végfelhasználói kiadás, illetve a következő még nem kiadott kiadás, vagy más néven kiadás jelölt (Release Candidate)

Átvételi (Acceptance)	- Általánosan az adott végfelhasználói kiadás - Az adott végfelhasználói kiadáson végzett sürgősségi javítás - Kiadás jelölt: o amennyiben az eddig a pontig előírt tesztelés (általában minimum a rendszerteszt) lezárult o az összes megkövetelt fejlesztés lezárult o elérte a végfelhasználói kiadással szemben támasztott minőségi követelményeket o megkapta a megfelelő engedélyeket a tesztelés utolsó fázisára
Végfelhasználói (Production)	- Adott, elfogadott végfelhasználói kiadás

Táblázat 5 - DTAP, környezetek konfigurációmenedzsment előírásai

Ez a rendszer viszonylag bonyolult konfigurációmenedzsmentet követel meg, ezért lehet egy minimális többletterhelés a fejlesztőkön, hogy minden módosítást megfelelően címkézzenek. Illetve elképzelhető, hogy lesznek olyan hibák, amelyet a fejlesztői környezetben a jövőbeli fejlesztések elnyomnak, vagy éppen előhoznak, ami megnehezíti a hibakeresést és javítást. Viszont vannak előnyei is ugyanennek a megoldásnak. Egyrészt tudunk tesztelni későbbi fejlesztéseket, és a korábbi fejlesztések olyan hibái, ami egy későbbi fejlesztésre vannak hatással, hamarabb előjönnek, illetve jelentősen csökkenthető a környezetekre fordított teljes költség (Total Cost of Ownerhsip ,TCO).

A fenti felosztás hatással van általában a tesztelési folyamatra is, és meghatározott tesztszakaszokat a megfelelő tesztkörnyezetben kell végrehajtani. Egy ajánlott felosztást tartalmas az alábbi táblázat, ami segíti a korai tesztelést, és minimalizálja az esetleges menetrendi ütközéseket:

Környezet	**Engedélyezett kiadás**
Fejlesztői (Development)	- Egységtesztek, egységintegrációs tesztek
Teszt (Test)	- Egységintegrációs tesztek, rendszertesztek
Átvételi (Acceptance)	- Váratlan végfelhasználói hibák javítása és tesztje - Végtesztelés (rendszerintegrációs teszt vagy E2E teszt) - Átvételi tesztek - Teljesítménytesztek
Végfelhasználói (Production)	- Nincs tesztelés

Táblázat 6 - DTAP tesztkörnyezetek és tesztfázisok

Hogy melyiket válasszuk a két modell közül? Mindkettőnek vannak előnyei és hátrányai. Ha nagy és integrált rendszereink vannak, amiben a különböző rendszerek menetrendje nem összehangolt, az első megoldás szinte megvalósíthatatlan, és kontrollálhatatlan. Azonban ha egy szeparált szolgáltatásról van szó, akkor mindkét opció működőképes alternatíva. A legjobb megoldás megvizsgálni ilyenkor a költségeket. Ha az első megoldás minimális költséggel megoldható, egy jó alternatíva lehet a konfigurációk keveredésének minimalizálása. Viszont a sokkal kisebb menetrendbeli megkötések miatt, még jelentős beruházások mellett is inkább a második lehetőség tűnik vállalható alternatívának.

Integrált környezet

A jelenkor nagyvállalati infrastruktúrája rengeteg különböző szolgáltatást tartalmaz(hat), melyeket általában integrálni kell egymással. Amint integrációról beszélünk, a saját tesztkörnyezeteink már nem csak a saját céljainkat szolgálják, de egyben szolgáltatásként biztosítjuk azt a velünk integrált szolgáltatást nyújtó másik egységnek is. Mivel a szolgáltatások, mint egy lánc felfűzve szinte mindannyian kapcsolódnak egymáshoz, egyértelműen szükséges egy közös szabályzás bevezetése. Elkerülhető ezzel az, hogy a szolgáltatások eltérő folyamatai és rugalmatlansága miatt teljesen más szolgáltatásokra legyenek negatív hatással.

Kisebb rendszereknél, ami néhány szolgáltatást tartalmaz csak, alkalmazhatunk egy olyan megoldást, amelyik meghatározott napokon állítja élesbe a kiadásokat, és más alkalommal ezekre nincs lehetőség. Evvel a technikával egy olyan rendszert tudunk elérni, ami nagyban hasonlít egy nem integrált környezetre, mert az egész integrált rendszert egységesen kezeli. Bár ez így egyszerűbben kezelhető folyamatot biztosít, a projektek menetrendjére nagyon erős megkötéseket alkalmaz. Ezek a megkötések megnehezíthetik nagyobb projektek kivitelezését, illetve sokszor kell kompromisszumot kötni mind a minőségi elvárások, mind a projekt által megvalósítható funkciók területén.

Ahogy nő egy adott rendszer, és nő az abban nyújtott szolgáltatások száma, illetve a megvalósítandó feladatok komplexitása, ezek a megkötések egyre inkább elfogadhatatlanok egy szervezet számára, és az egyes szolgáltatások életciklusában egyre nagyobb szabadságra van igény a menetrendek és a minőség meghatározására.

Amikor mindenki flexibilis menetrenddel dolgozik, akkor nehéz garantálni egy olyan rendszer létrejöttét, ami pontosan azt a konfigurációt szimulálja, ami a jövendőbeli állapot lesz a rendszernél, amikor a konfigurációváltás megtörténik.

Mennyire nagy ez a probléma? Az esetek nagy részében, amikor a szolgáltatások közötti felületek jól definiáltak, általában egy-egy kiadás között korlátozott a valószínűsége, hogy probléma keletkezik, viszont erről meg is kell győződni. Ha jól megfigyeljük, a probléma nagyon hasonló ahhoz, mint amikor a tesztkörnyezetek teljesítményénél az egyik tesztkörnyezet kisebb teljesítményű, és a megoldás is hasonló. Mivel a normál tesztkörnyezetekben csak az aktuális, vagy a rákövetkező kiadás szerepelhet, ezért a funkciók ott is nagy biztonsággal ellenőrizhetőek, azok integrációja is kezelhető. Amikor a fejlesztés, tesztelés befejeződött és a minőségi követelményeket teljesítettük, kell, hogy legyen egy feltehetően rövid tesztidőszak, ahol már csak egy végső tesztelés folyik az átvételi környezetben. Azért az átvételi környezetben, mert az integrált szolgáltatások itt pontosan azon a szinten (konfiguráción)

vannak, mint ahogy az élesbe állításkor lesznek (ha egyszerre van a két szolgáltatás élesbe állítása, ami néha szükséges, akkor az új, ha külön-külön, akkor még a jelenlegi konfigurációban).

A teljes rendszer (amikor már nagyfokú integráltsággal rendelkező rendszerekről beszélünk) megkövetel az IT szervezettől egy bizonyos fokú érettséget és minőségbiztosítást. Bár a rendszer nagyon hatékony és flexibilis, amennyiben az alapszabályok betartatásával probléma adódik, és ez egy gyengébben kontrollált fejlesztésnél – amikor a határidő közeleg – könnyen előfordul, az jelentős zavarokat tud okozni más területeken. Ráadásul kiterjesztjük saját kockázatainkat az integrációkon keresztül a többi szolgáltatást fejlesztő partnerre is.

Egy javasolt minőségbiztosítási módszer, hogy belépési és kilépési kritériumokat (Entry and Exit Criteria) definiálunk bizonyos környezetek használatba vételére. Ezek lehetnek tesztfázisok határán is, mivel azonban a tesztfolyamat változatos lehet, a következő példában inkább az adott környezet használata szerint nézzünk egy példát.

Környezet	**Belépési kritérium**	**Kilépési kritérium**
Fejlesztői (Development)	- A módosításhoz tartozó követelmény engedélyezett - A követelmény egy már engedélyezett kiadáshoz tartozik (nem olyan kiadáshoz, amihez a fejlesztés még nem engedélyezett)	- Minden egységteszt lefutott - Az eredményeket jelentettük - A hibák, amelyeket javítani kellett kijavításra kerültek - A javított hibák újratesztelése megtörtént
Teszt (Test)	- A módosításhoz tartozó követelmény engedélyezett, és a következő tervezett kiadáshoz tartozik - A módosítás egységtesztje sikeresen lefutott	- A kiadás minden követelményének fejlesztése befejeződött - Az egységtesztek befejeződtek, az eredmények elérhetőek - Az egységintegrációs tesztek befejeződtek, az eredmények elérhetőek - A rendszertesztek befejeződtek, az eredmények elérhetőek - A kijelölt hibák javítása megtörtént - A javított hibák újratesztelése megtörtént és a tesztek pozitívan zárultak - A jelenlegi hibaszint megfelel a minőségbiztosítási irányelveknek
Átvételi (Acceptance)	- A módosítás egy a jelenlegi kiadást érintő súlyos és sürgős hibát érint vagy, - A következő kiadásnak a tesztkörnyezet kilépési kritériuma teljesült és - A következő kiadás megkapta az	- A jelenlegi kiadást érintő hiba javításának újratesztelése sikeresen megtörtént, vagy - A végső tesztelés lezárult - Az eredmények elérhetőek - A talált kritikus hibákat javították - A javítások újratesztelése

	engedélyt a végső tesztek végrehajtására	megtörtént, és a javítás sikeres - Engedély az új kiadás használatba vételére
Végfelhasználói (Production)	- Az új kiadás vagy hibajavítás engedélyezett - A telepítés után az ellenőrzés sikeres	- Újabb engedélyezett kiadás vagy - Szolgáltatás leállítása

Táblázat 7 - DTAP, Környezetek Belépési és Kilépési kritériuma

Kérdés lehet még, hogy hogyan építsük fel az integrációkat az adott környezetekben? A javasolt metódus, hogy minden szolgáltatás tartsa fenn a négy környezeti szintet, és azok szintenként a végfelhasználói szintnek megfelelő integrációval rendelkezzenek. Ennek a megoldásnak számtalan előnye van:

- Nincsenek adatintegritási problémák (amikor az egyik környezetben létrehozunk egy adatot, ami azután szinkronizálódik több más rendszerrel, ha egy rendszer nem végfelhasználói környezetszerűen egy az egyhez csatlakozik, hanem több az egyhez vagy több a többhöz, nehéz lesz egy hibáról eldönteni, hogy az valóban hiba, vagy adatszinkronizációs probléma)
- Egyszerűbb karbantartás
- Állandó kapcsolatok, kevesebb konfigurációs probléma a tesztelésnél
- Hamarabb látható logikai hibák a környezetek összeállításánál
- Magasabb rendelkezésre állás

A másik felmerülő kérdés, amit általában tisztázni kell, hogy mit csináljunk azon szolgáltatásoknál, ahol nem folyik aktív fejlesztés. Ott is kell a négy környezet? A válasz az, hogy általában igen, mert azok a környezetek nem csak az adott szolgáltatás fejlesztéséhez kellenek, hanem a környező, integrált szolgáltatások fejlesztését és tesztelését is segítik. Ez így persze úgy hangzik, hogy felesleges kiadásra kötelezünk egyes szolgáltatásokat, de ha az összköltséget nem szolgáltatás, hanem teljes szervezeti szinten vizsgáljuk, ez a hatékonyabb megoldás.

DTAP kivételkezelés integrált környezetben

Maga a rendszer akkor működik optimálisan, ha minden szolgáltatás biztosítani tudja a négyszintű környezetet, és azok egymáshoz megfelelően integráltak. Azonban nem élünk ideális világban, és mindig vannak kivételek.

Kezdjük azokat a rendszereket, ahol ezeket kerülni javasolt:

- Erősen integrált rendszerek (annyi helyen kell kivételt kezelni, ha egyáltalán lehet, hogy az összköltsége a kapcsolódó szolgáltatásoknál valószínűleg jelentősen meghaladja a megtakarítást)

- Kétirányú integrációk megléte (nagyon könnyen okozhat adat egység problémákat)
- Kritikus rendszerek (megengedhetetlen az üzemeltetési kockázatok növelése)

Ahol kivétel egyáltalán szóba tud kerülni, azok olyan szolgáltatások általában, amelyek gyenge integrációval kapcsolódnak a többi rendszerhez, inkább egyedülálló szolgáltatásként tekintünk rájuk. Itt is meg kell vizsgálni az integrációkat, és azoknak egyirányúnak kell lenniük (azaz csak az egyik oldal tud változtatni az adatokon), és ez úgy is teljesül, hogy minden rendszert figyelembe veszünk (azaz nincs olyan útvonal, ahonnan az adat visszajut, esetleg több szolgáltatáson keresztül). Ilyen esetekben a kockázatot minimálisan lehet tartani, és lehetséges más környezeti struktúrát kialakítani, viszont minden egyes ilyen kivételt megfelelő kockázatelemzéssel kell megvizsgálni.

Egy-egy ilyen kivétel kezelése további kivételkérelmekkel is járhat (például egy ilyen kivétel igényelhet további biztonsági rendszabályok alóli felmentési igényt, hogy össze lehessen kötni különböző szinten lévő környezeteket).

Természetesen egy nagyvállalati rendszeren belüli teljes mértékben független szolgáltatás is felmentést kaphat az adott szabályok betartása alól, mivel nem hordoz kockázatot más szolgáltatások számára.

DTAP és az agilis szoftverfejlesztés kapcsolata

A DTAP modell egyik pozitívuma, hogy nem egy projektmenedzsment metodológiára optimalizált, hanem az adott projekt érettsége alapján szabályozza, hogy milyen tevékenységeket szabad végezni az adott környezetben. Mivel viszonylag kevés megkötést tartalmaz, remek módon alkalmazható mind vízesés mind agilis környezetben, illetve vegyes környezetben is, amikor egyszerre futnak vízesés jellegű és agilis fejlesztések is.

Agilis módszertanok jelenleg is többtízes nagyságrendben állnak rendelkezésre. Ezek elsősorban abban különböznek egymástól, hogy mennyire előíróak (például RUP) vagy adaptívak (például SCRUM vagy Kanban[28]), illetve, különösen az adaptív oldalon, hogy melyik szervezet hogyan implementálja őket. A sokszínűség miatt leginkább az Agilis kiáltvány (Agile Manifesto)[5] az, amire ebben a fejezetben támaszkodunk.

Az agilis módszertannak biztosítani kell az állandó tesztkörnyezetet, ami jelen modellben a fejlesztői és tesztkörnyezet, amit folyamatosan használni lehet fejlesztésre és tesztelésre, mert minden alkalommal működő végfelhasználásra kész terméket kell produkálni, amihez folyamatos tesztelés szükséges.

Az állandó integráció és rendelkezésre álló tesztkörnyezet jelentős segítséget nyújt bármely agilis projektnek.

Bár egy agilis projekt elméletileg minden iteráció végén egy kiadható terméket állít elő, nem feltétlenül adjuk azt ki, dönthetünk úgy is, hogy bár megtehetnénk, inkább fejlesztünk még rajta.

Komplex integrációkkal tűzdelt környezetben viszont 100%-osan biztosítani azt, hogy az adott időpillanatban minden kompatibilis a meglévő konfigurációval, szinte lehetetlen. Emiatt szükséges, hogy az átvételi környezetben zajló lezáró folyamat, vagy egy külön iteráció keretében, vagy a többi fejlesztéssel párhuzamosan, de mindenképpen megtörténjen. Mivel az agilis iterációk végeredménye

általában tökéletesen megfelel az átvételi környezet belépési kritériumának, ez az igény könnyedén integrálható és implementálható, illetve ez az a tesztelési fázis, amikor az agilis és vízesés modellek fejlesztési ciklusa összekapcsolódik.

Egy nem integrált környezetben akár magának az iterációnak is része lehet egy átvételi teszt illetve egy végteszt az átvételi környezetben, ott nincsenek külső környezeti hatások.

DTAP – az integrált környezetben megvalósított követelmények

Ez egy visszatekintés arra, hogy milyen követelményeket tudunk teljesíteni az előbb leírt rendszerrel, illetve melyek azok a kockázatok és megkötések, amelyeket elfogadunk.

Követelmények:

- A fejlesztések ütemterve egymástól nagy függetlenséget élvez
- Lehetséges több különböző kiadás egyszerre történő fejlesztése
- Lehetséges egy, a következő kiadásnak megfelelő konfiguráció tesztelése
- A projektek menetrendje szabadon választható az igényeknek, a minőségi követelményeknek és az adott státusznak megfelelően, maga a környezet elérhetősége nem helyez nyomást a megfelelő kompromisszumok megkötésében
- Biztosított a környezet ahhoz, hogy minden tesztelésnek megfelelő minőségű környezet álljon rendelkezésre, és az ne akadályozza a tesztelés végrehajtását
- Biztosított a környezet a végfelhasználói környezetben esetlegesen felmerülő súlyos problémák hibakeresésére és a megoldásának tesztelésére (lásd a Kockázatok és megkötések részt is)
- Biztosított, hogy az integrált szolgáltatásoknak az integrációs teszteléséhez van megfelelő környezet
- Alacsony humán erőforrás igény a tesztkörnyezetek fenntartására, az állandó kapcsolatok fenntarthatósága miatt
- Magas rendelkezésre állás
- A különböző szerepkörök hatékony támogatása a környezetek jó szeparációjával és megfelelő szerepköri prioritásokkal
- Kevés megkötés a tesztfolyamatban (lásd Kockázatok és megkötések részt is)
- Egyszerre támogatja a vízesés és agilis metódusokat is

Kockázatok és megkötések:

- Egyes esetekben, az adott kiadás életciklusának a végén, amikor a sürgős hibák amúgy nem jellemzőek, már nincs pontos mása a jelenlegi végfelhasználói környezetnek

- A tesztfolyamat egyes lépései meghatározottak, azokat nem lehet átlépni
- Szigorú minőségbiztosítás, és erős kritérium szükséges a végső tesztfázisokhoz, az átvételi környezetbe való belépéshez, amit akkor is be kell tarta(t)ni, amikor a projekt menetrendje nyomás alatt van
- Az áthágott minőségbiztosítási modellnek következménye lehet más szolgáltatásra is
- A rendszer üzemeltetése fegyelmezettséget, és az alaprendszer alapos ismeretét várja el a teljes fejlesztési szervezettől

Speciális igények

Bár a DTAP modell egy flexibilis modell, mégis lehetségesek olyan szituációk vagy követelmények, amikor ki kell egészíteni, vagy mást kell alkalmazni.

A modellből adódóan van egy kis kockázat a végfelhasználói környezet támogatásában, amikor a következő kiadás már az átvételi környezetben van. Ez egyes kritikus szolgáltatások esetén elfogadhatatlan lehet, mert olyan rendelkezésre állási időket kell produkálni, ami nem tudja biztosítani a kivételkezelésre szükséges időt. Ilyenkor szoktak egy ötödik környezetet is beállítani, amin csak a végfelhasználói környezet támogatása folyik, és csak akkor frissül, amikor a végfelhasználói környezet, ami persze felvet néhány kérdést.

Miért kell ötödik környezet? Miért nem tesztelünk végig a tesztkörnyezeten? Ilyen kritikus rendszer általában csak olyan szervezeteknél fordul elő, és olyan szolgáltatásokat tartalmaz, amelyek általában erősen integráltak. Egy ilyen integrált környezetben az átvételi környezet szerepe nem csak az más esetben sem, hogy biztosítsa a karbantartás számára a környezetet, hanem az is, hogy a különböző menetrendek miatt eltérő időben élesedő szolgáltatásoknak biztosítson egy olyan integrációt, ami az adott konfiguráció élesítésekor várható. Ez normál esetben két jól kombinálható funkció, jelen esetben azonban nem lehetséges ennek biztosítása. Természetesen nem integrált esetben egy végfelhasználói szintű tesztkörnyezettel nincs szükség ötödik környezetre (csak a kritikus egyedülálló rendszerek nagyon ritkák).

Jogosan adódik a kérdés, hogy ha itt ötödik környezetre van szükség, akkor hogyan ellenőrzünk ebben integrációt? Ha nulla kockázatra törekszünk, akkor szükséges lehet ennek az ötödik szintnek a létrehozására minden szolgáltatásnál, viszont ennek nagyon magas költségvonzata lehet. A köztes megoldás az, hogy csak a legkritikusabb rendszerekből építjük fel ezt az ötödik szintet, és az olyan integrációkat, amelyek nem okozhatnak olyan problémát, amire 24-48 órán belül mindenképp megoldást kell találni, meghagyjuk a DTAP rendszerben, mert azokat az eseteket már tudjuk avval is kezelni.

Ezen kívül előfordulnak olyan esetek, speciális tesztelési igények, amikor ezt az adott rendszerrel nem tudjuk lefedni, és a speciális igényekhez külön környezeteket kell építeni. Ezeket a környezeteket angolul Sandboxnak hívjuk (szó szerinti fordításban homokozó, de talán szerencsésebb magyarul valami más elnevezést, talán demó környezetként hivatkozni rá). Ezek a környezetek egyedi igényt szolgálnak ki, és nagy részük csak egy rövidebb időre léteznek. Néhány tipikus eset, amikor ilyeneket építünk:

- Koncepció kipróbálása (Proof of Concept) egy projekt elején
- Fejlesztőgépek (ahol nincsenek megkötések)
- Nagyon korán megkezdett későbbi fejlesztés, aminek nincs pontos céldátuma/cél kiadása, ezért jelenleg nem akarjuk betenni a hivatalos kiadásokba
- Speciális tesztelési igények, például hibatűrési tesztek, amelyek jelentősen megzavarnák a munkát a normál környezetekben

Egy-egy nagyobb szervezetnél érdemes ilyen célokra egy kisebb készletet tartani bármikor bevethető környezetekből, amit dinamikusan lehet használni különböző célokra, hogy elkerülhető legyen a körülményes beszerzési procedúra. Kisebb szervezeteknél azonban ez nem feltétlenül jó opció, tekintettel arra, hogy az ilyen eszközök kihasználtsága valószínűleg alacsony lesz. Kis szervezeteknél az infrastruktúra bérlése, vagy innovatív, többcélú technológiák (mint például virtuális gépek, felhőszolgáltatások) nyújthatnak megoldást.

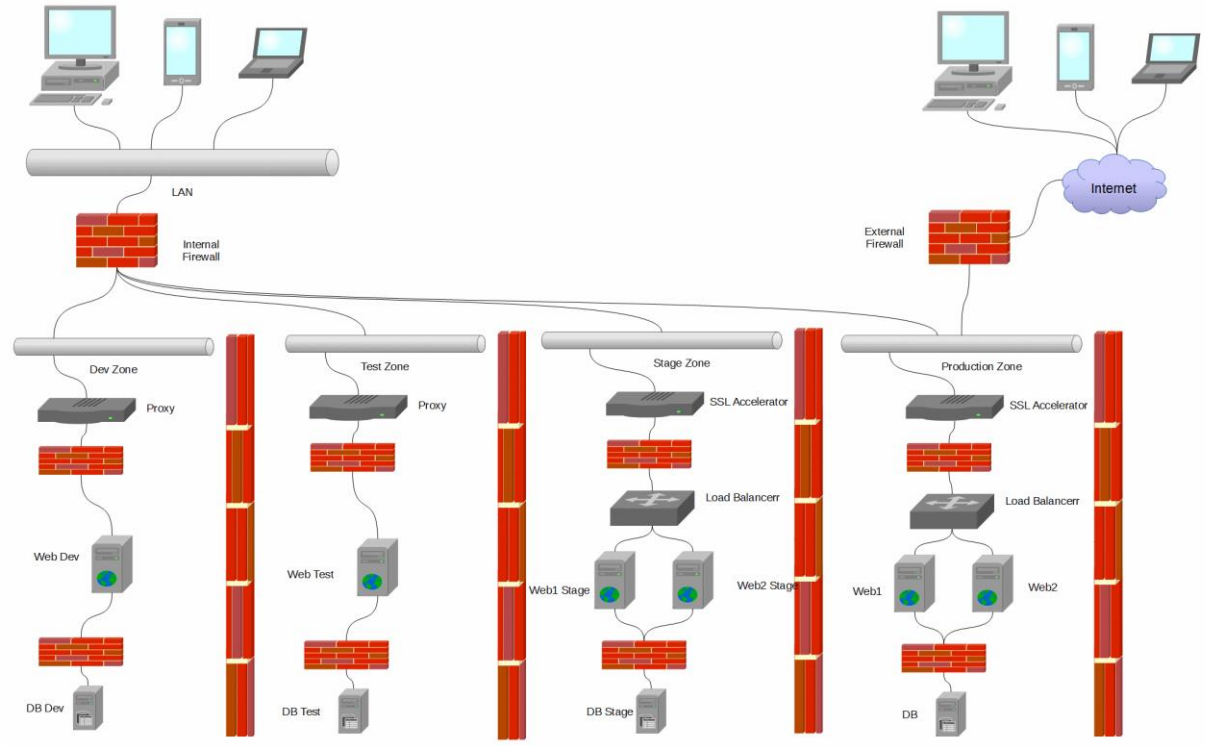
Biztonság

A biztonság kérdése szerver-kliens alkalmazásoknál nem kerülhető meg, a szolgáltatás színvonala, funkciói és minősége mellett az egyik legfontosabb paraméter. A biztonság alatt itt most elsősorban adatbiztonságot értünk, ami a mai üzleti világban az egyik, ha nem a legfontosabb vagyonelem, amire a végfelhasználói környezetekben általában nagyon részletes szabályozást készítenek a cégek. Ugyanakkor nem gondolnak a tesztkörnyezetekre, mint potenciális veszélyforrásra. Mi is okozza a veszélyt? A tesztkörnyezetekbe kényelmi illetve megvalósíthatósági szempontból általában éles adatok másolódnak rendszeres időközönként. Amennyiben a tesztkörnyezeteket kevésbé védjük, könnyű célpontot jelentenek, és kerülő úton a féltve őrzött adatvagyon megszerezhető.

Az első védelmi vonal, hogy lehetőleg a tesztkörnyezeteket tegyük elérhetetlenné a publikus hálózatok (például az internet) számára. Amennyiben szükséges külső erőforrásoknak hozzáférést biztosítani azt lehetőleg egy közbülső belépéssel, például VPN (Virtual Private Network, Virtuális Magánhálózat) segítségével valósítsuk meg, amivel megfelelően szabályozni tudjuk a hozzáféréseket.

Korlátozzuk a direkt adathozzáféréseket. Bár a tesztelést gyorsabbá teheti, a direkt adathozzáférés megkönnyíti és felgyorsítja az adatok teljes mértékű exportját a rendszerből. Használjunk hasonló szabályozást, mint a végfelhasználói környezetekben. Amennyiben adatbányászatra van szükség, vagy használtassunk beépített riportokat, vagy kérjük el a specifikált igénnyel az adatgazdáktól.

Az adatokat tároló szerverek elérésére hasonló biztonsági előírásokat kell alkalmazni, mint a végfelhasználói környezetre (azonos tűzfalrendszer, stb.), illetve alapértelmezett szabályzásként ne engedjünk olyan hozzáféréseket, amik kivezetnek az adott szintről a DTAP rendszerben (például nem lehet elérni a végfelhasználói környezetet tesztrendszerből, vagy az átvételiből a tesztet).



Ábra 4 - DTAP modell, biztonságilag szeparált környezet

A másik biztonsági kérdés, amit az adatmásolás felvet, hogy bár az adatok nagy többsége általában valóban olyan, hogy az a céges dolgozók, így a tesztelők számára nyílt, vannak kivételek, olyan adatok, amelyek csak nagyon kevesek számára hozzáférhetőek (például fizetésadatok, jelszavak), melyek nem kerülhetnek a tesztelőkhöz, viszont számukra is szükséges megfelelő tesztadatok biztosítására. Ezeket az adatokat általában irányított módon megváltoztatjuk az adatok betöltése folyamán. Fontos, hogy a változtatás olyan legyen, amiből az eredeti értékek nem következtethetőek vissza, valamint a historikus adatok módosítása is szükséges.

Karbantartás

A tapasztalatok azt mutatják, hogy általában a tesztkörnyezetek karbantartása legalább akkora, ha nem nagyobb munkát jelent, mint a végfelhasználói környezetek karbantartása. Ez először talán meglepőnek tűnik, de ha belegondolunk, a tesztkörnyezetekben sokkal gyakrabban cserélünk kódot, és sokkal kevésbé stabil kódot, ami bármikor okozhat váratlan szituációt. A szoftvereket normál használatra és nem tesztelésre tervezzük, és az ottani felhasználói csoportokra optimalizált a használata. Tesztelés közben több különböző csoport verseng néha a környezetek használatáért, amely tevékenységek kizárhatják egymást, és lehetséges, hogy más követelményeik vannak, amely miatt másképp kell a környezeteket beállítani hozzájuk. (Csak egy példa, egy teljesítménytesztnek lehetnek olyan alkalmai, amikor az a funkcionális tesztelést lehetetlenné teszi, vagy éppen egy megkezdett manuális tesztet elront.)

A viszonylag szerteágazó felhasználás miatt nagyon erős kommunikáció, valamint a megfelelő menedzsment szükséges, hogy minden érdekelt igényét megfelelően ki lehessen szolgálni, illetve az információk mindenkihez eljussanak. Javasolt, hogy minden egyes környezetnek meghatározott tulajdonosa legyen, aki dönteni tud a különböző igények megfelelő kiszolgálásáról, és ezt megfelelően és tervezetten kommunikálja. A DTAP modell szerint egy javasolt példa a tulajdonosokra az alábbi felosztás:

Környezet	Legfontosabb felhasználói csoport(ok)	Javasolt tulajdonos
Fejlesztői (Development)	- Fejlesztők - Egységtesztelők	- Technikai projek menedzser
Teszt (Test)	- Rendszer és integrációs tesztelők	- Tesztmenedzser
Átvételi (Acceptance)	- Üzemeltetés - Rendszertesztelők - Átvételi tesztelők	- Szolgáltatás vezető
Végfelhasználói (Production)	- Felhasználók	- Szolgáltatás vezető

Táblázat 8 - DTAP, környezet tulajdonosok

Kliensoldali tesztelési igények

Amíg a szerveroldalon a technológia bonyolultsága és ára, valamint a sok különböző igény kielégítése hatékony módon jelent kihívásokat, a kliensoldalt a technológiai sokszínűség jellemzi, aminek szintén meg kell felelni.

Az első és legfontosabb, amit fel kell mérnünk, hogy mi a célplatform, amit támogatni szeretnénk, és melyek azok a paraméterek, amelyek a lényeges különbségeket okozzák.

A szűkítés alapja, hogy miben különbözhetnek lényegesen a támogatott platformok. A következő pár fejezetben a különböző klienstípusokat nézzük át, illetve az itt lehetséges jelentős módosító tényezőket.

Ha a teljes kombinációs halmazt kellene vizsgálni egy általános esetben, az akkora lenne, hogy azt reálisan és elvárható időben nem lehet tesztelni, ezért az első teendő, hogy reálissá szűkítsük a kört. Ehhez használhatunk ekvivalencia osztályokat, szélsőértékekre nagyobb hangsúlyt fektethetünk, illetve használhatjuk a páronkénti tesztelés technikáját is. A tesztelői tapasztalat nagy segítséget nyújt a megfelelő kiválasztásban.

Csak egy gyors példa erre: például felmerül az igény, hogy minden Windows változaton tesztelni kell. Reálisan megnézve jelenleg (2015-ben) az alábbi Windows változatoknak van még támogatása: Windows Vista, Windows 7, Windows 8 és Windows 8.1, és minden változatnak létezik 32 és 64 bites változata, ami egyből 8 platformot jelent, viszont már így is rengeteg irreleváns platformot elhagytunk. A Windows 7 és Windows 8 között létezik egy erős technológiai váltás. Ha az nem lenne, valószínűleg érdemes lenne

csak Windows Vistára és Windows 8.1-re szűkíteni, mint a két szélsőértékre (ahogy a Windows 8 előtti időben valószínűleg a Windows XP és a Windows 7 lett volna a két platform), de így a váltás miatt a Windows 7 és a Windows 8 is határértékké válik. Persze mondhatjuk, hogy nekünk ez a 8 változat megfelel, de ha nincs elég idő tesztelni, és tovább kell szűkíteni, akkor elő tudjuk venni azokat az információkat is, amelyekkel rendelkezünk az adott platformokról.

- A 64 és 32 bites architektúrában lehetséges, hogy különbözőképpen működnek szoftverek
- A Windows Vista az iparági tapasztalatok alapján bizonytalanabb működésű, mint a Windows 7
- A Windows Vista sokkal kevésbé használt, mint a Windows 7
- A Windows 8 és 8.1 között nincs lényeges különbség a működésükben (vannak olyan termékek, ahol ez nem feltétlenül igaz, de most maradjunk egy ilyen példánál), viszont a 8.1-es változat kedveltebb, és szabadon frissíthető

A szűkítésre több stratégia is létezhet ilyen esetben:

A) Mivel a Windows Vista bizonytalanabb és régebbi, azon tesztelünk, ha hiba van, akkor megnézzük, hogy az Windows 7-en is reprodukálható-e

B) Mivel a Windows 7 és a Vista között nincs nagy különbség, és a Vista stabilitása is sokat javult, a gyakrabban használt Windows 7-t használjuk tesztelésre

C) Úgyis frissíthető 8.1-re a Windows 8 és kedveltebb, ráadásul az az utolsó változat, inkább azt használjuk tesztelésre, mint a korábbi változatot

D) A Windows 8 volt a korábbi változat, valószínűleg több a gyermekbetegsége, azt használjuk, és lesz egy referenciakörnyezet Windows 8.1-gyel, hogy azon is reprodukálható-e a hiba

A preferenciák szerint általában A és D vagy B és C választások a hasonlóak. Amennyiben az A és D válasz szerint tesztelünk, akkor:

- Az elsődleges tesztkörnyezeteink: Windows Vista 32-bit, Windows Vista 64-bit, a Windows 8 32-bit és a Windows 8 64-bit lesz,
- Illetve referenciakörnyezetként használunk: Windows 7-t illetve Windows 8.1-t, választás szerint mind 64- és 32-bites változatban, vagy csak ez egyik választott módon.

Ha a B és C opció fontosabb számunkra, akkor:

- Az elsődleges tesztkörnyezeteink: Windows 7 32-bit, Windows 7 64-bit, a Windows 8.1 32-bit és a Windows 8.1 64-bit lesz.
- Nincs referenciakörnyezetünk Windows Vistára és Windows 8-ra.

Hogy akkor melyik is a jobb megoldás? Egyik sem vagy mindkettő. Ez egy kompromisszum, és a megfelelő kompromisszumot az adott fejlesztés kockázatai alapján kell megválasztani (akár az is lehet a megoldás, hogy egyáltalán nem szűkítünk a listán).

Vékony kliensek (hardver)

A vékony kliensek használata általában jelentősen egyszerűsíti a tesztelést a központi menedzselhetőségnek és a varianciák számának alacsonyan tartásának köszönhetően. Az egyik, és talán legnagyobb rizikófaktor a szerver oldali teljesítményigény, illetve ezek tesztelhetősége (a teljesítménytesztelő eszközök nagy többsége hálózati forgalom alapján szimulál, és egyes protokollok, amennyiben ritkák, hiányozhatnak az eszköztárból).

Természetesen számíthatunk rá, hogy az évek során több generációban szerzünk be vékony klienseket, ezért ezekből érdemes beszerezni néhány példányt tesztelési célokra is. Egy átfuttató tesztet, ahol a legfontosabb funkcionalitást használjuk, érdemes átfuttatni mindegyik használatban lévő eszközön. Nagyobb volumenű tesztelést viszont csak a jelentősen különböző hardverű eszközökkel érdemes csinálni. Tapasztalat alapján a legrégebbi (az alacsony teljesítmény miatt), illetve a legújabb (kevés tapasztalat) a két legnagyobb kockázatot hordozó komponens.

Webes alkalmazások

A webes alkalmazások talán a leggyakoribb szerver-kliens alkalmazások. A fő nehézség a nagy variancia, amit a böngészők nagy száma, az azokat futtató platformok szintén nagy varianciája és az esetlegesen beépülő modulok okoznak.

Az első és legfontosabb, hogy meg kell határozni azokat a kombinációkat, amelyeket tesztelni szeretnénk, és ez alapján felépíteni a tesztelési stratégiát, és evvel együtt a környezetek stratégiáját, előrebocsátva, hogy nem tesztelhető minden kombináció, azok számossága miatt. Amennyiben a működés stabilitása kiemelt fontosságú, inkább írjunk elő ajánlott konfigurációkat. Ellenkező esetben határozzuk meg, hogy melyek a fontos tényezők, amelyek jelentősen befolyásolhatják a működést. Az alábbi tényezők mind befolyásolhatnak, de nem feltétlenül jelentős mindegyik, azt az adott fejlesztés kockázatai alapján kell eldönteni:

- Böngésző fajtája
- Böngésző verziószáma (melyik verzió érdekes?)
- Operációs rendszer (tényleg fontos?, melyik kritikus?)
- Beépülő modulok (plugin) és azok verziószáma (a verziószám is fontos?)
- Hardver teljesítmény (hardverigényes oldal, van cél minimum konfiguráció?)
- Regionális beállítások
- Nyelvi beállítások

Nagy valószínűséggel egy olyan halmazt kapunk, amit egyetlen számítógéppel nem tudunk kielégíteni (mert különböző operációs rendszereket, böngészők különböző verzióit, beépülő modulok különböző verzióit kellene egyszerre installálni, ami nem lehetséges).

Az adott problémára több megoldás is születhet.

A) Minden célplatformból beszerzünk akkora mennyiséget, amennyi egyszerre használatban van, és ezeket leosztjuk a tesztelők között, akik használat függvényében hozzáférnek. Hogy ez a módszer működjön, erős menedzsment szükséges, hogy ne legyen egy adott elem kihasználatlanul, miközben valakinek szüksége van rá, csak mert az adott eszközt nem találjuk

B) Virtuális gépeket definiálunk, mint rendszerképet (image), és azokat mindenki a saját számítógépén egy megfelelő szoftverrel futtatja

C) Beállítunk egy központi terminálszervert az adott rendszerképekkel, és azt terminálelérésen keresztül használjuk

Az utóbbi két megoldás talán kevesebb adminisztrációval, viszont bonyolultabb karbantartással jár. Nagyvállalati környezetben fontos kérdés még ilyenkor tisztázni a különböző licencszerződéseket, amelyek eltérőek lehetnek. Van, amikor az A) és a B) vagy C) megoldásokat kombinálni kell, mert például bizonyos eszközök (például mobiltelefonok) nem érhetőek el mint rendszerkép.

Egyéb kliens szoftverek

A web egy olyan megoldás, ahol a megjelenítést egy adott szabvány szerint, az adott szabványkommunikációval végezzük, és több megjelenítő program is létezik. Az ilyen jellegű megvalósításban a web szinte egyeduralkodóvá vált, de amennyiben más hasonló megoldást tesztelünk, arra szinte teljes mértékben analóg megoldást kapunk.

A másik megközelítés, amikor a megjelenítést is programozott módon fejlesztjük, ekkor viszont előtérbe kerül már a hardver illetve az operációs rendszer környezet. Ez akkor is fontos lehet, ha a kliens elvileg platformfüggetlen alkalmazás (például Java alapú), mert bizonyos dolgokat az fogadó operációs rendszer másképp kezelhet (például fájlrendszer). Ebben az esetben a figyelembe veendő paraméterek változnak kissé, és itt is igaz, hogy a megfelelő lényeges különbségeket okozó pontokat kell megtalálni:

- Operációs rendszer
- Operációs rendszer nyelve és verziószáma
- Telepített egyéb komponensek és azok elvárt verziószáma
- Hardver variációk
- Regionális beállítások

Maguknak a kliensszoftvereknek a felépítése nagyon hasonló, mint a webes alkalmazásoknál, amikor már megvannak a kívánt kombinációk. Amire még ebben az esetben figyelni kell, hogy a kliensprogram is folyamatosan változik ilyen esetekben, és annak a telepítéséről is gondoskodni szükséges.

Teszteszközök

Jogos kérdés lehet, hogy foglalkozunk a teszteszközökkel? Használjuk őket tesztelésre, de milyen relációban állnak magával a tesztkörnyezetekkel? Ezeknek az eszközöknek is el kell érniük a megfelelő szolgáltatásokat és szervereket, ráadásul egyeseknek ezt úgy, hogy másképp és máshonnan érik el a rendszereket, mint a rendes felhasználók.

Nézzük végig a teszteszközöket és a problémákat, amit meg kell oldani.

Teljesítménytesztelő eszközök

A teljesítménytesztelő eszközök általában nagymértékű terhelést generálnak az eszközökön. Mindezt általában úgy szeretnénk megtenni, hogy az infrastruktúra többi elemére lehetőleg minél kisebb terhelést jelentsen, ezért az eszközöket a lehető legközelebb szeretnénk elhelyezni a célszerverekhez, amelyek általában egy lezárt szerverszobában találhatóak. Mindez lehetséges, hogy megfelelő konfiguráció elvégzését is igényli a tűzfalakon. Ha több környezetben is tesztelünk, akkor vagy több ilyen terhelést generáló eszközre van szükségünk, vagy több környezetre is át kell engedni a tűzfalon. Mindezen kívül ezekkel a terhelést generáló számítógépekkel is kommunikálni kell (például tesztmenedzsment eszköz), amihez lehetséges, hogy végfelhasználói környezetben lévő komponenseket használunk. Olyan helyeken, ahol olyan biztonsági szabályzatok vannak, amelyek esetleg nem engedik a más környezeti szinttel való kommunikációt, csak mint kivételt (tehát például a végfelhasználói eszközt nem engedik kommunikálni a teszt rétegben lévő eszközzel, és csak kivétel-kezelési kérelemmel lehet az esetet kezelni). Az ilyen kivételek, illetve tűzfal konfigurációs lépések általában hosszadalmas bürokrácia útján engedélyezhetőek (jogosan, mert az ilyen kivételek súlyos biztonsági kockázatot is jelenthetnek). Mindezek miatt fontos, hogy a teljesítménytesztéléshez szükséges eszközöket foglaljuk bele a környezetek tervébe, hogy azok megvalósítása ne csússzon ki a határidőkből.

Tesztautomatizáló eszközök

Kétféle tesztautomatizáló eszközt lehet telepíteni. Az egyik kód illetve protokoll bázisú és nagyrészt egységteszteknél használjuk. Lehetnek olyan tesztek, amikor a teljesítménytesztelő eszközökhöz hasonlóan ezt közel kell tenni a szerverekhez, mert a tesztek másképp nem végrehajthatóak. Ebben az esetben hasonló problémákat kell megoldani, mint a teljesítménytesztelő eszközöknél, és a megoldás is hasonló.

A másikfajta tesztautomatizálási eszköz általában felhasználói felület alapú. Itt a kliensoldalon kell a megfelelő eszközt telepíteni, amihez a megfelelő környezeteket létre kell hozni.

Mindkét esetben meg kell teremteni az egyéb teszteszközökkel a megfelelő kapcsolatot, amennyiben szükséges.

Teszt- és hibamenedzsment eszközök

Mint már említettük, egyes tesztmenedzsment eszközök lehetővé teszik, hogy automatizált tesztjeinket (legyenek azok az funkcionális vagy teljesítménytesztek) menedzselten el lehet indítani belőlük, és az eredményeket visszakapjuk a tesztmenedzsment eszközbe. Akár az is előfordulhat, hogy automatikusan a hibajegyeket is felvesszük.

A másik fontos teendő, hogy miután a tesztkörnyezetek architektúrája kialakult, a megfelelő értékeket (amik arra szolgálnak, hogy a teszteket, illetve a hibákat a megfelelő módon lehessen hozzárendelni) létrehozzuk, vagy a megfelelőre módosítsuk. Hasonlóképpen kell eljárni, hogy a megfelelő konfigurációt (verziót) megfelelően azonosítani lehessen.

Felhőalkalmazások[29][30][31]

A felhő, és a felhőalapú megoldások az utóbbi pár évben terjedtek el. Ez egy feltörekvő ágazata a számítástechnika iparnak, amelynek használatát még mindig tanulja az iparág. A felhő alatt egy olyan szolgáltatást értünk, amikor a szolgáltatás nem egy dedikált hardveren, hanem a szolgáltató eszközein elosztva, a szolgáltatás részleteit a felhasználó elől elrejtve valósul meg. Egy felhőalkalmazás egy virtuális gép/szerver formájában látható a felhasználó számára. Az igazi újdonság, hogy nem látható a hozzá tartozó gazda számítógép.

A szolgáltatás helye szerint megkülönböztetünk **privát** és **publikus** felhőt. A publikus felhőt a felhasználók nyílt hálózaton (interneten) keresztül érik el, a privát felhőt pedig értelemszerűen helyi, nem publikus hálózaton. Amikor mind privát mind publikus elemeket kombinálunk, **hibrid** felhőről beszélünk. Fontos különbség a publikus és a privát felhő között, hogy míg a publikus felhőben egy szolgáltató gondoskodik a megfelelő infrastruktúra felépítéséről és karbantartásáról, a privát felhőben ezt nekünk kell megtenni (vagy gondoskodni egy olyan szolgáltatóról, amely létrehozza és karbantartja ezt számunkra a saját létesítményeinken belül).

A felhőalkalmazásokat felhasználási módjuk szerint három csoportba tudjuk osztani:

- **Infrastruktúra szolgáltatás (Infrastructure as a Service, IaaS):** a legegyszerűbb szolgáltatói modell, ahol a szolgáltató egy – általában virtuális – számítógépet a meghatározott paraméterekkel (tárhely, számítási kapacitás, hálózati elérhetőséget) biztosít, és annak menedzsmentje az ügyfél feladata (beleértve akár annak operációs rendszerrel való ellátását is). Több szolgáltató más erőforrásokkal is támogathat, mint például alaprendszer képekkel, tűzfalakkal, terheléselosztó alkalmazásokkal vagy egyéb más szoftvercsomagokkal az igényeknek megfelelően. A felhasználó felel a karbantartásért, a számlázás az allokált és felhasznált erőforrások függvényében történik.

- **Platform szolgáltatás (Platform as a Service, PaaS):** egy magasabb szintű szolgáltatás, ahol a szolgáltató a teljes platformot biztosítja, amiben benne van az operációs rendszer, az adatbázis, a webszerver, az esetleges programvégrehajtáshoz szükséges végrehajtó környezet (például JVM). Mindezen felül a szolgáltató gondoskodik az erőforrások skálázásáról (azaz amennyiben több tárhely vagy számítási kapacitás szükséges, az adott felhasználáshoz igazítja a megfelelő erőforrásokat). Platform szolgáltatás esetén a szolgáltató biztosítja a karbantartást, mind a hardver, mind a szolgáltatásban foglalt szoftverek esetén.

- **Szoftver szolgáltatás (Software as a Service, SaaS):** esetén magát a szoftvert, illetve annak használatát engedélyezzük a felhasználóknak. Ebben az esetben nem kell a felhasználónak foglalkoznia a szerver és a szoftver karbantartásával, hanem erről a szolgáltató gondoskodik. A

számlázási modell az ilyen szolgáltatásoknál általában felhasználó alapú és havi vagy éves ciklusú, ami a felhasználói igények szerint skálázódik.

A fenti szolgáltatói és hozzáférhetőségi modelleken kívül léteznek még más modellek is, de ezek a gyakorlatban ritkábban jelentkeznek. A mindennapi életben talán leggyakoribb ilyen a tárhely szolgáltatás (Storage as a Service).

Infrastruktúra és platform szolgáltatások a tesztelésben

Vannak esetek, amikor a megoldás tartalmaz olyan elemeket, amelyek ilyen szolgáltatást tartalmaznak. Bár ilyenkor a megoldás egy része, vagy teljes egésze a felhőben van, ha logikailag és funkcionálisan nézzük, akkor nem sok különbség van egy fizikai hardveren megvalósított megoldáshoz képest. Az egyetlen különbség, hogy egyes hardverelemek nem teljesen specifikáltak, esetleg valahol az interneten vannak. Ezeket az elemeket a tesztkörnyezetekben is klónozhatjuk, ami lehet szintúgy felhőbe telepített, vagy akár fizikailag létező elem is.

Ugyanakkor jelentős a különbség, ha a nem funkcionális dolgokat vizsgáljuk. A hardver ebben az esetben nincs a mi kezünkben, sőt a teljesítménye talán nem is állandó. A teljesítmény függhet a szolgáltató adatközpontjának, illetve az oda vezető hálózat terheltségétől, illetve platformszolgáltatás esetén egy teljesítménytesztelés beindítja a folyamatot, hogy az igények alapján felfelé skálázza a környezetet.

A hálózat terheltsége (nem feltétlenül a szolgáltató miatt, hanem a köztünk és a szolgáltató közötti sávszélességben bárhol megbúvó szűk keresztmetszett miatt) a terhelés véges lehet. Vannak szolgáltatók, akik elosztott módon tudnak terhelést nyújtani, ami segít ennek a problémának a leküzdésében. A tesztelés eredménye ilyen esetekben a válaszidők lesznek, egyéb terheltségi adatok a virtualizálás miatt nem biztos, hogy relevánsak.

Amikor platformszolgáltatást használunk, figyelnünk kell az automatikus skálázásra. Lehet, hogy az eredmények megfelelőnek tűnnek, de valójában csak azért, mert a rendszer erőforrásait nagymértékben kiterjesztettük. Ez azért probléma, mert a szolgáltatás árazása a felhasznált erőforrásoktól függ, ami magas fenntartási és jelen esetben esetleg tesztelési költséget is jelenthet. Egy jól optimalizált szoftverrel jelentős költségmegtakarítás is elérhető.

Amennyiben ki tudjuk nyerni az erőforrás-használat mennyiségét az adott tesztidőszakban a szolgáltatótól, az segíthet annak eldöntésében, hogy az eredmények így is megfelelőek-e.

Egyes esetekben érdemes lehet egy ellenőrzött hardveren végrehajtani a teljesítményteszteket, és ott optimalizálni a szolgáltatást, mert ott sokkal kevesebb megkötéssel kell számolnunk. Ez különösen igaz azokra a tesztekre, amelyek a lehetséges szűk keresztmetszetekre, illetve az optimalizálásra koncentrálnak, amit egy skálázott szolgáltatásnál esetlegesen nagyon nehéz elérni.

A másik lehetséges használat az ilyen szolgáltatásoknak, amikor magához a teszteléshez használunk felhő alapú eszközöket. Ennek oka lehet, hogy teszt hardverhez jussunk, vagy használhatunk felhő alapú szolgáltatást, mint tesztszolgáltatást.

A megoldás előnyei:

- **Gyorsan elérhető.** Egy normál beszerzési ciklus több hét vagy hónap is lehet, így akár órákon belül elérhető a szolgáltatás.
- **Alacsony induló költségek.** Nincs azonnali induló beszerzési költség, csak a használat után kell fizetni.
- **A teljesítmény és más paraméterek könnyedén növelhetőek, csökkenthetőek.** Különösen egy projekt elején nehéz meghatározni, hogy milyen erősségű hardverre lesz igény. Ami jelen esetben flexibilisen átskálázható.
- **Nincsenek használaton kívüli maradványeszközök.** Amikor egy projekt lezárul, vagy menet közben változnak a tervek, általában vannak olyan eszközök, amelyekre a továbbiakban nincs szükség, és ezeknek nem mindig lehet megfelelő hasznosítást találni az adott szervezeten belül.

Mindezen karakterisztikák különösen alkalmassá teszik áthidaló megoldásnak, még akkor is, amikor később esetleg hagyományos megoldásokkal váltjuk le. Gyakran előálló szituációk, amikor jó megoldás lehet felhő alapú szolgáltatásokhoz nyúlni:

- **Koncepciókészítés (Proof of Concept)**
- **Gyorsan létrehozandó környezet,** amíg a végleges környezet nem áll rendelkezésre
- **Ideiglenes környezetek,** amikor csak rövid időre, speciálisan kell egy környezet
- **Nagyszámú, de alacsony hardverigényű környezetek,** mint például a különböző nagyszámú klienskonfiguráció (böngésző, nyelv, operációs rendszer stb.)
- **Agilis szoftverfejlesztés,** gyorsan alkalmazkodik a változásokhoz

Egyes szervezetek, különösen friss kezdő cégek a teljes tesztelési infrastruktúrát egy felhőbe költöztetik a gyorsan változó igények miatt. Ez a kezdetben általában nyilvános felhő, és ahogy nő a cég esetlegesen átmozgatják egy privát felhőbe.

Szoftverszolgáltatások

Amikor szoftverszolgáltatást vásárolunk és integrálunk a saját rendszerünkbe, akkor általában egy szerver-kliens alkalmazás szerver oldalát integráljuk a saját hagyományos és/vagy felhő infrastruktúránkkal. Tesztelés szempontjából tehát egy szerver-kliens architektúrájú fejlesztésről és tesztelésről beszélünk, annak a technikáját, technológiáját használjuk. Van azonban néhány lényeges különbség, amik közül a legfontosabb, hogy két (vagy esetleg több, ha több hasonló szolgáltatást is igénybe veszünk) fejlesztési folyamatait és kultúráját kell összehangolni. Az egyszerű esetben, amikor csak használunk egy szolgáltatást, és nem tartalmaz számunkra egyedi funkciót és beállítást, illetve nincs integrálva más szolgáltatással, akkor persze nincs sok dolgunk, egyszerűen használjuk a szolgáltatást, és vagy van egy átvételi tesztünk a szolgáltató által biztosított környezetben és időpontban, vagy csak egyszerűen elkezdjük használni a szolgáltatást a megfelelő frissítésekkel. Amikor azonban testreszabott és/vagy integrált szolgáltatásokat használunk, az adott folyamat akár nagyon komplex tud lenni.

A következő alfejezetekben megvizsgáljuk mindezt az ügyfél, illetve a szolgáltató oldaláról is. Itt már csak arról lesz szó, amikor integráció és/vagy egyedi igények kialakítása szükséges.

Szoftverszolgáltatás az ügyfél szempontjából[32][34][35]

Amikor egy új, ilyen jellegű szolgáltatást integrálunk, a két szervezet kiadáskezelését és teszt módszertanát össze kell hangolni. Az összehangolásnál sokat segít, ha mindkét oldalon olyan az adott folyamat, ami megfelelő rugalmasságot biztosít.

Az ügyfél, mivel a szolgáltatás infrastruktúráját nem kontrollálja, fel kell adjon pár dolgot az eddigi szabadságából, cserébe azért, hogy a rendszeradminisztráció terhét leveszik a válláról. A leggyakoribb kompromisszumok:

- Az új kiadás használatbavételi dátuma teljesen időintervallumhoz kötött. Különösen igaz ez olyan szolgáltatások és szolgáltatók esetén, ahol a szolgáltató infrastruktúrája több ügyfelet szolgál ki párhuzamosan a háttérben ugyanazon az infrastruktúrán
- A tesztkörnyezetek struktúrája adott, attól eltérni nem, vagy csak további díjfizetés esetén lehet
- A nem a szolgáltató által végzett egyedi fejlesztésért a szolgáltató nem vállal felelősséget, arról az ügyfélnek kell gondoskodni, hogy megfelelően működjön az új kiadásban

Ahhoz, hogy egy kiadást az adott időintervallumban át tudjuk venni több lehetőség is van, attól függően, hogy az ügyfél hogyan alakította ki a saját rendszerét.

Amennyiben egy olyan rendszert használ, amikor a teljes infrastruktúrán előre meghatározott időpontban van a karbantartás, akkor a karbantartási ciklusnak igazodnia kell a szolgáltató ciklusához. Abban az esetben, amikor több ilyen jellegű szolgáltatást használunk, ez nagyon nehézkessé, vagy lehetetlenné válhat, mivel az reálisan nem várható el, hogy több ilyen szolgáltató igazodjon egymáshoz.

Amikor azt a modellt használjuk, hogy minden szolgáltatás esetleg kisebb megszorításokkal ugyan, de szabadon határozhatja meg a saját kiadási menetrendjét, sokkal egyszerűbbé válik ezeknek a külső függőségeknek a kezelése, hiszen csak az adott szolgáltatás menetrendjével kell egyeztetni a szolgáltató kiadásait.

Az adott szolgáltatás ára nem minden esetben tartalmaz tesztkörnyezeteket, vagy a benne foglalt környezetekre bizonyos megszorítások érvényesek. További környezetek létrehozása többletköltséget hoz magával, ami egyes esetekben jelentős terhet is jelenthet.

A megszorítások lehetnek:

- Csökkentet felhasználószám
- Csökkentet adatmennyiség
- Funkciók tiltása

Az ilyen korlátozott környezetek általában olcsóbbak, mint egy teljes képességű környezet, viszont az integrálásukat meg kell oldani, illetve megfelelő tesztelési és adatkonzisztencia stratégiával kell rendelkezni.

Kell azonban olyan környezet, ami végfelhasználói környezet jellegű. A végső tesztekhez, illetve a végfelhasználói támogatáshoz is ilyen környezet szükséges. A szerver-kliens architektúra fejezetében ismertetett módon az esetek többségében egy teljes tesztkörnyezettel ez a két funkció általában jól kezelhető. Mivel nincs kezünkben a környezet, a visszaállítás opció, amennyiben mégis váratlanul sürgősen megoldandó végfelhasználói incidens történik, általában nem reális opció, mert a szolgáltatási szerződésben ezeknek a teljes környezeteknek a frissítési paraméterei nem teszik mindezt lehetővé.

Ugyanígy tisztázandó kérdés a kiadások menedzsmentjének finomhangolásakor, hogy mikor elérhető a következő kiadás, melyik környezetben, és hogyan kell frissítenünk a tesztkörnyezeteinket, hogy minden tevékenységnél a megfelelő környezetet használjuk. Általában a szolgáltatók egy meghatározott periódussal korábban elérhetővé teszik a tesztkörnyezetekben a teszt kiadást, és az vagy szabályozható, amikor az adott környezetet frissítjük, vagy egy adott időpont után már automatikusan az új változat települ minden frissítésnél. Az adott környezetet akkor érdemes frissíteni az új kiadásra, amikor már az ott végzett tevékenységek egy olyan kiadásra készülnek, ami már az új alapkiadást használja.

Minden általunk végzett fejlesztés egy potenciális konfliktuspont az új kiadással, ezért általában a javasolt metódus egy szoftverszolgáltatásnál, hogy lehetőség szerint kerüljük a hozzáadott komponenseket, és használjunk inkább konfigurációs változtatásokat, és a már beépített funkciókat, amelyeket a szolgáltató tesztel, így nagy valószínűséggel nem okoz problémát egy verzióváltás. Ugyanakkor vannak olyan igények, amikor nem tudjuk elkerülni az egyedi fejlesztést. Hosszú távon ilyen esetben érdemes automatizált egységteszteket fejleszteni, sőt egyes platformok a fejlesztési fázis folyamán ezeket futtatják is, és az esetlegesen felmerülő hibákat javítják, vagy jelzik az ügyfélnek, hogy az új verziónál az adott teszt megbukik, és az új verzióval konfliktus várható. Mindez sokat segít a jövendőbeli fejlesztéseknél, illetve jelentősen rövidítheti a tesztelési időt, amire nagy szükség lehet, tekintettel a rendelkezésre álló tesztelési időre az új kiadással.

Szoftverszolgáltatás a szolgáltató szempontjából

Amikor szolgáltatóként fejlesztünk ilyen alkalmazásokat, a saját minőségbiztosítási rendszerünk mellett figyelni kell arra is, hogy az ügyfél is végez fejlesztéseket, illetve integrálja a rendszert a saját rendszerébe, ami akár nagyon komplex is lehet, és megfelelő tesztelési idő és átfutás szükséges egy új kiadás használatba vételére.

Természetesen ez nem jelenti azt, hogy az ügyfél minden igényét ki kell elégíteni, általában egy nagy ügyfélkörrel rendelkező szolgáltatásnál nem is lehet, mert az igények olykor konfliktusba kerülnek. Viszont egy jó, rugalmas rendszer segít abban, hogy a saját folyamatait sikeresen illessze az általunk nyújtott infrastruktúrával.

Az egyik gyakori ügyfélkérés, hogy a frissítések felvétele opcionális legyen. Ez csak akkor működik, ha az egyes ügyfelek dedikált infrastruktúrát kapnak. Amennyiben azonos infrastruktúrán szolgálunk ki több ügyfelet, ott csak akkor tudnánk ezt megtenni, ha minden ügyfél ezt kérné, ami nem reális alternatíva.

Ilyen esetekben az ügyfélnek kell alkalmazkodnia. Amennyiben megtehető, akkor is érdemes korlátozni a támogatott kiadásokat, mert hosszabb távon több kiadás támogatása jelentősen növeli a támogatás ráfordításait. Azon ügyfeleket, akik több rugalmasságot szeretnének a kiadások kontrolljában, ajánlott egy, a saját infrastruktúrában létrehozott hagyományos telepítés felé terelni, amennyiben van ilyen licenc módozat is.

Vannak azonban olyan általános ügyféligények, amit mindenképpen tudnunk kell biztosítani, vagy a szolgáltatási csomag részeként, vagy külön díj ellenében. A legfontosabbak:

- Kell olyan tesztkörnyezetet biztosítani, amely funkcionalitásban, adatmennyiségben és teljesítményben megfelel a végfelhasználói környezetnek
- Legalább egy teljes környezet kell, de vannak organizációk, amelyek ennél többet igényelnek
- Részleges, olcsó környezetekre szükség lehet a fejlesztéshez
- Biztosítani kell, hogy az új kiadás tesztelhető legyen megfelelő hosszúságú ideig (minimum egy hónap periódus, de 2-3 ajánlott), mielőtt a frissítés a végfelhasználói környezetben megtörténik
- Biztosítani kell, hogy a tesztkörnyezetben szabályozható legyen, hogy az aktuális, vagy a jövőbeli kiadással dolgozunk
- Amikor az új kiadás megjelenik a tesztkörnyezeten, annak stabilnak és teszteltnek kell lennie, nem az ügyfél dolga, hogy tesztelje a szolgáltatásunkat, ők a saját fejlesztéseikre koncentrálnak
- Az ügyfélnek képesnek kell lennie a saját környezetei menedzselésére (létrehozás, frissítés, adatmásolás stb.)
- A tesztkörnyezetek infrastruktúrája szeparált a végfelhasználói környezettől, hogy semmilyen módon ne lehessen hatása a tesztelésnek a felhasználókra
- Megfelelő kommunikációval kell rendelkeznünk a saját kiadásainkról, a frissítések időpontjáról és azok tesztkörnyezetbeli elérhetőségéről
- Megfelelő módon ismertetni kell az ügyfelekkel a saját kiadási folyamatunkat, illetve annak ajánlott integrációját az ügyfél kiadáskezelésébe

Gyakran felmerülő kérdések a felhőkkel kapcsolatban

Bár kétségtelenül jól felismerhető előnyei vannak a felhő technológia használatának, vannak olyan kérdések, amelyek rendszeresen felmerülnek a technológiával kapcsolatban. Nézzünk meg egy párat ebben a fejezetben.

Publikus felhőnél az egyik kérdés az adatbiztonság kérdése. Ebben az esetben ugyanis átadjuk az adataink kezelését egy harmadik félnek, amely ezt felhasználhatja, módosíthatja, esetleg törölheti azokat. Az utóbbi esetek inkább üzemeltetési kérdések, amelyet a komoly szolgáltatók megfelelő folyamatokkal és technológiával kezelni tudnak, de a felhasználás már jogi problémákat is felvet (például személyes adatok, amelyet velünk osztottak meg, de nem engedélyezett annak kiadása harmadik fél

számára, az országonként eltérő adatkezelési szabályok betarthatósága). Az adatok tulajdonjogáról nem minden esetben rendelkeznek pontosan a szolgáltatási feltételekben.

Határon túli felhőszolgáltatónál kérdéses, hogy melyik hatóság, illetve bíróság illetékes, hogy az adatokba való betekintést kérje, illetve engedélyezze, a felhasználó és a szolgáltató országának szabályozása jelentősen eltérhet.

Létezhetnek iparági biztonsági előírások, aminek a felhőszolgáltató nem felel meg (például hitelkártya tranzakciók kezelése esetén).

Mikor egy nyilvános felhőben lévő szolgáltatást integrálunk a saját belső infrastruktúránkkal, akkor a saját szolgáltatásainkhoz hozzáférést kell biztosítani egy külső entitásnak. Az ilyen kivételek a tűzfalakon nagy odafigyelést követelnek, hogy csak a még megfelelő, de a lehető legminimálisabb kivételt hozzuk létre a tűzfalakon. Az adatkapcsolatok titkosított csatornát használjanak a külső szolgáltatóval. A csak IP cím alapú azonosítás nem megfelelő, más technikákat is be kell vetni a pontos azonosításra, megakadályozva evvel a közbeékelődő támadásokat.

A szolgáltató rendszeradminisztrátora olyan adatokhoz férhet hozzá, amelyek titkosak (például fizetések, vagy egyéb ipari titkok, stb.).

Bár a felhőszolgáltatók általában jelentős méretű, nagy tapasztalattal rendelkező vállalatok, amelyek nagy biztonságtechnikai szaktudással rendelkeznek és ezáltal sokkal jobban védettek, mint egy átlagos szolgáltatás, itt is előfordulhatnak hibák. Mivel ezek a szolgáltatók sok adatot kezelnek, potenciális célpontok a rosszindulatú betörések szempontjából.

A megbízhatóság kérdése is fel szokott merülni, hiszen az üzemeltetést kiadjuk a kezünkből, így a felelősséget is. Ha az alapmutatókat nézzük, a felhőszolgáltatók általában nagyon magas rendelkezésre állási mutatókkal rendelkeznek, ami egyrészt adódik a technológiai fejlettségből, és az abból származó hibatűrésből. Illetve a nagy üzemeltetési gyakorlatból, viszont ugyanúgy készülni kell az esetleges szolgáltatás-kimaradásra, mint bármely hagyományos szerviz esetén, hiszen a szolgáltatónál is bekövetkezhetnek előre nem látható események (például természeti katasztrófa).

A privát felhők használata válasz lehet a legtöbb biztonsági és felelősségi problémára.

Fel szokott merülni még, hogy akkor most publikus vagy privát felhő megoldást válasszunk, esetleg maradjunk a hagyományos megoldásoknál?

Általában nagyobb szervezetek számára érdemesebb a privát felhő használata, illetve a hagyományos megoldások választása nagyobb szolgáltatások esetén. Azonban kezdő illetve kisvállalkozások esetén a publikus felhő használata jelentős megtakarítást hozhat a rugalmassággával, illetve amiatt, hogy nem kell rendszeradminisztrátorokat foglalkoztatni. Hasonló a helyzet, amikor egy szoftvernek létezik szoftverszolgáltatás alapú, illetve hagyományos formája is. Kisebb szervezeteknél, illetve alacsony felhasználószám esetén jobb lehet a szoftverszolgáltatás, és gyorsabban használatba lehet venni, viszont sok felhasználó esetén már olcsóbb és rugalmasabb lehet a saját rendszeren karbantartott változat.

Mindezek persze csak ajánlások, mindig az adott költségstruktúra, az igények, illetve az adatbiztonsági kérdések szerint mérlegelve kell a döntést meghozni.

Egyedül álló szoftverek tesztkörnyezetei

Olyan szoftverek tartoznak ide, amelyek hagyományosan önállóan működnek, és nem támaszkodnak hálózati tartalmakra. Egyes funkciók persze lehetnek olyanok, amelyek szerver-kliens alapúan működnek, és azon funkciók tesztelése – a szerver környezettel együtt – annak a tesztmetódusnak megfelelően tesztelődik, de a funkciók nagy része nem ilyen, és ez más kihívásokat jelent.

Egyrészről a függőségek kisebbek egy ilyen fejlesztésnél, ami könnyebbé teszi a tesztelést. Ugyanakkor jelentős kihívást jelent a futtatókörnyezetek sokszínűsége, amit ugyan már érintettünk a szerver-kliens architektúra kliens oldalán, de a probléma egy ilyen szoftver esetén általában hatványozottabban jelentkezik.

Néhány példa, ami jelentősen befolyásolhatja a szoftver működését:

- Alap architektúra (például PC, Mac, Tablet stb.)
- Operációs rendszer
- Processzor
- Hangeszközök
- Videó eszközök
- Memória mennyiség (általában, mint minimum)
- Háttértár (mennyiség, sebesség és eszköztípus)
- Egyéb szoftverkomponensek (driverek, futtató környezetek (JVM, DirectX stb. és ezek verziói)

Az első probléma, amivel találkozunk, hogy a lehetséges kombinációk száma nagyon magas, még a megközelítő tesztelése sem megoldható saját eszközökkel.

A másik probléma, hogy a több különböző platform megkövetelhet több fejlesztői környezetet (egyszerűen szükséges lehet a fordítás elvégzésére is), amit szintén fel kell építeni. A különbség a fejlesztői és tesztelői környezetek között, hogy a variancia sokkal kisebb, itt arra törekszünk, hogy az összes szükséges platformon végre lehessen hajtani a megfelelő feladatokat.

A harmadik speciális probléma ebben az esetben, hogy a tesztkörnyezeteket folyamatosan frissíteni kell a legújabb buildeknek megfelelően. Ezeket megfelelő módon elérhetővé kell tenni egy olyan helyen, ahonnan a frissítés végrehajtható a tesztelőknek, valamint a hibák reprodukálásához rendelkezésre kell állni az előző buildeknek is.

Sokszínűség kezelése

Az első és legfontosabb teendő, hogy reális mennyiségű kombinációra szűkítsük a tesztelést. A szűkítéshez az első lépés, hogy a funkcionalitás, a technikai felépítés és a támogatott platformok alapján meghatározzuk azokat az elemeket, amelyek lényegesek tesztelési szempontból.

Ez az adott feladat szempontjából teljesen más lehet, amit néhány példán keresztül talán könnyebb megvilágítani.

1. Példa, Játékfejlesztés: A játék- és multimédia-fejlesztésekben általában az alábbi paraméterek döntőek fejlesztési és tesztelési szempontból:

- Platform és operációs rendszer, amin meg akarunk jelenni (például PC Windows, PC Linux, Mac, Playstation, Xbox, Wii, Android, iOS stb.)
- Multimédia-futtató környezetek (például DirectX, OpenGL, egyes esetekben a platform definiálja)
- Videokártya típusok és teljesítményszintek
- Hangkártya típusok
- Memóriamennyiség
- CPU osztály
- Játékeszközök (például joystick, kormány stb.)

2. Példa, Java alapú üzleti szoftver esetén például a variancia sokkal kisebb, és kevésbé érzékeny az alapplatformra. Itt általában az alábbi paraméterek döntőek:

- JVM verzió
- Operációs rendszer, amin meg akarunk jelenni (de a Java maga ezt is nagyrészt eltakarja)

3. Példa, Natív üzleti szoftver esetén az előzőhöz képest ismét nagyobb varianciával találkozunk,de a multimédiás képességek marginálisak. Itt általában az alábbi paraméterek döntőek:

- Platform és operációs rendszer, amin meg akarunk jelenni (például PC Windows, PC Linux, Mac, mobil platformok)
- Memóriamennyiség
- CPU

4. Példa, Webes tartalomkezelő szoftver, mivel itt magát az alapszoftvert fejlesztjük, ezért itt inkább az egyedülálló szoftver tesztelési technikáját alkalmazzuk. Itt általában az alábbi paraméterek döntőek:

- Platform és operációs rendszer, amin meg akarunk jelenni (például PC Windows, PC Linux, egyéb Unix platformok)

- Támogatott webszerverek
- A fejlesztés szerveroldali technológiája (J2EE, PHP, CGI stb.)
- Támogatott böngészők
- Memóriamennyiség
- CPU osztály

Amint megvannak azok a lényeges paraméterek, amelyek lényegesen befolyásolják a működést, meg kell határozni, hogy mi az elvárt követelmény, mit szeretnénk támogatni. Valószínűleg ennél a pontnál a teljes kombinációs halmaz elemeinek száma még mindig túl magas ahhoz, hogy érdemben tesztelni lehessen az alkalmazást.

A következő lépés egy minimális konfiguráció definiálása, aminél kisebb, gyengébb hardverrel a terméket nem vizsgáljuk. Amikor ez egy szervezet specifikus igényeire fejlesztett szoftver, akkor ez valamivel egyszerűbb, mert az adott organizációban felmérhető, hogy mi a leggyengébb hardver, amivel működnie kell, azonban amikor a célközönség nem definiált, ezt nekünk kell meghatározni.

Nem definiált célközönség esetén meg kell találnunk az arany középutat. Amennyiben túl megszorítóak vagyunk, és túl magasra pozícionáljuk a minimumkörnyezetet, potenciális felhasználókat veszíthetünk, amennyiben túl alacsonyra, esetlegesen szükségtelen nehézségeket okozunk önmagunknak, és nagyon megnöveljük a tesztelendő konfigurációk számát. Ökölszabályként elmondható, hogy egy 3-4 éves, akkor középkategóriásnak minősíthető konfiguráció támogatásával az ügyfelek megfelelő részét meg lehet szólítani, ezek azok a konfigurációk, amelyeket a szoftvereket rendszeresen vásárló ügyfelek általában elkezdenek újabbra cserélni. Mint minden ökölszabály, itt sem alkalmazható mindez minden esetre. Például jóval erősebb és/vagy újabb hardver megkövetelése egy realisztikus grafikájú csúcsjáték, vagy egy multimédiaszerkesztő program esetén elfogadott az ügyfelek számára, ugyanakkor alapszoftvereknél, például biztonsági programcsomagnál elvárható, hogy egy öt-hat éves számítógépen még elfogadható teljesítményt nyújtson.

A minimumkonfiguráció az, amelyen mindenképpen tesztelni kell az alkalmazást. A másik érdekes konfiguráció a technológiailag legkésőbbi komponensek használata. A teszteléshez kockázatelemzéssel kell meghatározni azokat a konfigurációs pontokat, amelyeket tesztelni kell, és ezeknek megfelelő tesztkörnyezetet kell építeni. Egyes konfigurációs elemek tesztelése megoldható virtuális gépekkel, a különböző operációs rendszerek, illetve szoftverkörnyezetek erre remek lehetőséget biztosítanak, viszont a hardveres különbségekre nem nyújtanak reális alternatívát, azokat fizikai valójukban be kell szereznünk, hogy megfelelő módon le tudjuk tesztelni.

Amikor „dobozos” terméket fejlesztünk, valószínűleg rengeteg olyan elemmel találkozik majd a jövendőbeli szoftver, amit mi nem tudunk leképezni és tesztelni (ebben lehetnek hardver és szoftverelemek is). A probléma áthidalható a leendő ügyfelek bevonásával alfa illetve béta tesztelőként. Itt nem az általunk biztosított kontrollált és steril környezettel találkozik a szoftver, hanem már valós végfelhasználók környezetével, ahol előjöhetnek a leggyakoribb problémák és a végleges megjelenés

előtt korrigálni tudjuk a hibák nagy részét. A megfelelő időzítés azonban ezeknél a teszteknél nagyon fontos, nem helyettesíti az általunk elvégzett szerteágazó teszteket. Ezekben a tesztekben, különösen béta fázisban, már annyira stabil szoftver az elvárás, hogy az ügyfelek nagy többségének kevés és nem zavaró hibát szabad csak megtapasztalnia. Egy rossz minőségű béta kiadása, és az abban talált sok probléma lerombolhatja a leendő ügyfelek bizalmát.

Konfigurációkezelés és környezetmenedzsment

Az ilyen jellegű projektek általában nagyszámú teszteszközzel rendelkeznek, amelynek menedzsmentje (melyik eszköz kinél van, kinek, mikor, melyikre van szüksége, a tesztkörnyezet specifikus hibák vizsgálata miatt mikor van szükség esetlegesen fejlesztői példányra, melyik konfigurációból hány darabra van szükség stb.) nagyon fontos az effektív tesztvégrehajtáshoz, illetve a megfelelő környezetek felépítéséhez, beszerzéséhez is rendszerezés és előkésztés szükséges.

Mindezen teendőket és felelősséget ajánlott egy központi személyre például a tesztelés menedzserére, vagy annak asszisztensére bízni, hogy egyetlen központi helyre kelljen fordulni, ami jelentősen megkönnyíti és felgyorsítja a kommunikációt. Nagyobb tesztprojektek esetén, nagyon nagyszámú teszt eszköz esetén érdemes lehet szoftveres megoldást, tesztkörnyezet menedzselő szoftvereket használni.

Ugyanígy fontos, hogy megfelelően azonosítsuk a tesztelendő szoftvert, amit a teszteszközökre vagy központilag telepítünk, és ekkor nyilván kell tartani a teszteszközön az adott telepített tesztszoftver verzióazonosítóját, vagy elérhetővé tesszük a tesztelők számára a nekik megfelelő build telepítését az eszközökön. Minden esetben meg kell tartanunk több tesztelt buildet is elérhető formában, és képesnek kell lennünk azonosítani az adott buildhez tartozó forráskódot, mert a hiba reprodukálásához és megtalálásához szükségünk lehet rá. Megfelelő verziókezelő szoftver nagy segítséget nyújthat ebben számunkra.

Mobilfejlesztések

Az okostelefonok és tabletek rohamos elterjedésével a mobilalkalmazások megkerülhetetlenné váltak- Mobilfejlesztésen több különböző dolgot is értünk, érthetünk:

- Weboldal mobil változatának elkészítése
- Applikáció fejlesztése
- Mobiltelefon vagy tablet fejlesztése
- Mobil operációs rendszer fejlesztése

Mindegyik fejlesztési kategória más kihívásokat és követelményeket támaszt, ami megjelenik a tesztkörnyezetek kezelésében is.

Weboldal mobilváltozata

A weboldalak használatában a mobil környezet nagyon hasonló problémákat hordoz, mint normál számítógépes környezetben. Ami a legnagyobb különbséget jelenti, az a képernyő mérete, illetve az, hogy a nagyon gyors fejlődés miatt nagyon sokféle variáció van a piacon. Bár ahogy stabilizálódik az iparág, ez a változatosság jelenleg csökkenőben van (sokkal egységesebb kínálat jelenik meg).

A tapasztalatok szerint a legfontosabb tényező az ilyen alkalmazásoknál a mobilböngésző típusa, az alap operációs rendszer (kevésbé jellemző) és a képernyő mérete illetve felbontása (amiből a méret inkább ergonómiai kérdés).

Alapvetően az ilyen környezetben a tesztelésre két lehetőség van, vagy beszerzünk megfelelő mennyiségű eszközt, és azon hajtjuk végre a tesztelést, vagy emulátorokat használunk, amiből több is létezik a piacon, mind ingyenes, mind kereskedelmi változatban. Az aktuális eszközökön való tesztelés mindenképpen fontos, hogy legyen aktuális tapasztalatunk, de a tesztelés nagy részére az emulátor jól használható költséghatékony alternatíva.

Az eszköznyilvántartás és menedzsment itt is fontos része a tesztmenedzsment funkcióknak.

Mobil applikáció fejlesztése [36][37]

Jelenleg több mobil applikációs platform is elterjedt. Az alkalmazásfejlesztés az adott platformonokon alapvetően különbözik, így tulajdonképpen az alkalmazás másik platformra történő átvitele az alkalmazás újraírását is jelenti a legtöbb esetben.

Bár a platformok jelentősen különböznek, vannak közös pontok, amelyek minden platformra jellemzőek:

- Az applikációkat az operációs rendszerhez köthető alkalmazásbolton keresztül lehet terjeszteni
- Applikációk direkt installációjára az eszközök általában védettek, egyes esetekben elég egy beállítást megváltoztatni, más esetekben viszont illegális változtatásokat kell végezni az eszközön hozzá
- A fejlesztői eszközök lehetőséget biztosítanak az alkalmazások feltöltésére tesztelés céljából, viszont ehhez megfelelő regisztrációs lépések szükségesek lehetnek
- Az eszközök operációs rendszere magasabb verzióra könnyen cserélhető, a visszatérés alacsonyabb verziószámra azonban nem lehetséges

A platformok szerint megkülönböztetünk:

- **Zárt forráskódú és zárt rendszerű (iOS, Blackberry OS),** ahol az adott operációs rendszer forráskódja nem publikus, és nem módosítható, illetve magát az operációs rendszert csak a gyártó által engedélyezett és/vagy gyártott eszközök futtatják
- **Zárt forráskódú és nyílt rendszerű (Windows Phone, Windows RT),** ahol a forráskódja az operációs rendszernek még mindig nem publikus, viszont azt megfelelő licencszerződés betartása mellett, az adott rendszer minimumkövetelményeinek megfelelő rendszerekre telepíthető, esetleges licencdíj megfizetése után
- **Nyílt forráskódú és nyílt rendszerű (Android),** ahol az operációs rendszer forráskódja nyílt és szabadon felhasználható eszközök készítésére annak változatlan formában, vagy módosításokkal való felhasználásával

Az elsőtől a harmadikig haladva elmondható, hogy az adott platform egyre több féle és változatos eszközöket jelent, ami nehezíti a tesztelést.

Az eszközök, mint bármely számítástechnikai terméknél, általában gyorsan avulnak, általában 3-4 éves eszközök azok, amelyekre még érdemes támogatást biztosítani (sőt néha csak 2-3 éves periódusról beszélnek az iparágban). Ugyanez elmondható az operációs rendszerekre, mivel a frissítések jól megoldottak, az alkalmazások általában csak az előző operációs rendszer főverziót támogatják (vagy csak az aktuális verziót).

Zárt platform esetén csak néhány eszköz jelenik meg évente a piacon, amely jelentősen szűkíti a céleszközök számát, jelentősen egyszerűsítve a tesztelés ezen szeletét. A viszonylag gyors avulás, és az évenkénti kevés eszköz általában 20 alá csökkenti azon eszközök számát, amit teszteléskor figyelembe kell venni, és egyes lényeges paraméterekben (mint például a kijelzők felbontása), még kevesebb változatot lehet megfigyelni, ugyanakkor az operációs rendszerek verziói itt is növelik a lehetséges varianciák számát, ha sok régebbi verziót is támogatni szeretnénk.

Zárt forráskódú, de nyílt rendszer esetén valamivel szélesebb körű a választék, de még mindig viszonylag könnyen be tudunk szerezni néhány eszközt, amivel jól feddhető az adott környezet.

A harmadik eset viszont, amikor a forráskód nyílt (és ingyenes, valamint módosítható), egyrészt kedveltté tette a platformot a gyártók között, viszont nagyon nagy változatosságot hozott mind a hardver teljesítményében és minőségében, mind a szoftveres módosításokban, ami nehézzé teszi az applikációk fejlesztését és tesztelését.

Az első két esetben egy optimális összeállítás lehet:

- A legkisebb, legrégebbi eszköz, a legrégebbi még támogatott operációs rendszer változattal
- Ugyanez az eszköz egy friss operációs rendszerrel
- Egy friss eszköz a legfrissebb operációs rendszerrel
- Képernyő méretenként egy eszköz
- A második esetben, amennyiben teljesítmény és grafika intenzív az alkalmazás, a piacon lévő CPU teljesítményszintek és grafikai gyorsítók szerinti készülékcsoportok

A harmadik esetben annyi variációs lehetőség létezik, hogy minden esetre általában nem lehet eszközöket beszerezni, illetve azokon tesztelni. Általában az ilyen operációs rendszerekhez is van olyan eszköz, amit referenciaeszköznek lehet tekinteni (azaz azt mondjuk, hogy ha az applikáció ezen rendesen fut, akkor azt feltételezzük, hogy a hiba az eszköz hibája). A nyílt rendszereken az ajánlott eszközök, amin érdemes tesztelni:

- Referenciakészülékek
- Valamely gyengébb, de még elfogadható készüléke egy régebbi, de még általunk támogatott verzióval

- Egy ugyanilyen, vagy hasonló minimumkészülék a legfrissebb operációs rendszerrel
- Nagy gyártók 1-1 készüléke
- Amennyiben teljesítmény és grafika intenzív az alkalmazás, a piacon lévő CPU teljesítményszintek és grafikai gyorsítók szerinti készülékcsoportok

A fenti ajánlás egy közepes, alacsony közepes minőségbiztosításnak felel meg. Egy kisforgalmú applikációhoz ez sok, és valószínűleg 1-2 eszközzel is elégséges lefedettséget tudunk nyújtani, míg egy többszázezres letöltést generáló alkalmazásnál valószínűleg ennél sokkal több eszközt fogunk használni. A nagyobb iparági szereplők az egyes termékeket általában 30-60 eszközzel tesztelik, de egyes szereplők akár több száz eszközt is bevetnek. Tanulni, esetlegesen csak próbaképp, minimális befektetéssel elindított vállalkozásnál, amennyiben egy-két készüléket tudunk vagy akarunk csak megfinanszírozni, akkor használjunk egy közepes árfekvésű, sokak által használt készüléket a zárt forráskódú rendszereknél, illetve referenciakészüléket a nyílt forráskódú rendszereknél. Ezekkel egy viszonylag jó lefedettséget tudunk garantálni, viszont számítsunk arra, hogy lesznek eszközfüggő hibáink.

Amikor már van applikációnk egy-egy alkalmazásboltban, akkor már kapunk statisztikákat a felhasználókról, amiben benne vannak az eszközök, és azok megoszlása is. Ennek segítségével javítani tudunk a tesztelés minőségén úgy, hogy a felhasználóknak megfelelő eszközöket kezdünk használni teszteléshez.

A tesztelés menete

A mobil applikációkat először általában emulátorokban teszteljük, ahol be tudunk állítani különböző felbontásokat és más egyéb hardverparamétereket. Általában már a fejlesztői környezet is tartalmaz emulátort, de gondoskodni kell annak telepítéséről a tesztelők számítógépére.

Az emulátor egy jó segédeszköz, sokkal könnyebb és gyorsabb hibát keresni és javítani a segítségével, de nem mindig pontosan olyan eredményeket ad, mint az aktuális eszköz. Amikor az emulátorban az alkalmazás kezd stabilan működni, egyre inkább vonjuk be a különböző eszközöket a tesztelésbe.

A tesztelésünk végén a saját döntésünk szerint vagy elérhetővé tesszük az alkalmazást az alkalmazásboltban, vagy még beiktatunk egy közönségtesztelési (béta teszt) fázist.

A tesztkód mozgatása az eszközökre

A tesztkód átmozgatása az eszközökre általában nem egyszerű, különböző feltételeknek kell megfelelni biztonsági okokból. Ebben a fejezetben áttekintjük a három legnagyobb platform folyamatát, ahogy az a tananyag írásának időpontjában, 2015. elején éppen aktuálisan működik.

Android [50][51]

Az Android a legegyszerűbb az összes közül. Egyrészt létezik olyan biztonsági beállítás, amivel engedélyezhető telepítés az alkalmazás áruházon kívülről, valamint a fejlesztői környezetből USB kábelen keresztül az alkalmazás feltölthető.

Ahhoz, hogy a debuggolás is menjen, néhány beállítást el kell végezni. Az első, hogy a *manifest* vagy *build.gradle* fájlban be kell állítani, hogy az alkalmazás debuggolható:

```
android {
    buildTypes {
        debug {
            debuggable true
        }
```

A második, hogy az eszközön engedélyezni kell az USB-n keresztüli debuggolást.

Android 3.2 és előtte esetén **Settings > Applications > Development** helyen.

Android 4.0 és újabb verzióktól **Settings > Developer options** helyen. Android 4.2 és utána lévő változatoknál ez az opció rejtett, eléréséhez a **Settings > About phone** szekcióban a **Build number**-re kell hétszer egymás után rábökni, hogy megjelenjen.

A harmadik lépésben az USB kapcsolatot kell beállítani. Ez Windowson egy meghajtó telepítését igényli, OS X-en alapból működik, Linuxon `udev` rule-t kell létrehozni minden telefongyártóhoz, amin tesztelni szeretnénk.

Alfa- és bétatesztelés Androidon [52]

Ahhoz, hogy egy alkalmazás megjelenjen a Play Store alkalmazás áruházban, egy **Google Play Developer Console** [53] előfizetésre van szükség. Az összes beállítást ezen keresztül lehet megtenni. Amikor egy alkalmazást feltöltünk, beállítható, hogy az végleges, alfa vagy béta változat, és az alfa és béta változatnál megadható, hogy mely felhasználók férhetnek hozzá.

Ugyanitt lehet megadni, ha az In App Billing tranzakciókat egyes felhasználóknak ne számlázza ki a rendszer, ami előnyös tesztelési beállítás (bár egy béta tesztnél ezek már lehetnek valós tranzakciók is).

Windows Phone

A Windows Phone fejlesztői környezet is tartalmaz emulátorokat, amelyekkel a tesztelést el tudjuk kezdeni, de mindenképpen szükséges tesztelni az eszközökön is. A tesztelés engedélyezéséhez a telefont regisztrálni kell a megfelelő programmal, hogy elfogadjon fejlesztői változatokat.[55][57][59][60]

Egy Microsoft azonosító egy telefon megnyitását engedélyezi fejlesztési célokra. A fejlesztői azonosítóval alapesetben három telefont lehet engedélyeztetni, de a Microsoft támogatásán keresztül ez a korlát megfelelő indokkal (például cég mérete) emelhető. A fejlesztői azonosító mindenképpen szükséges a Windows áruház használatához így a terjesztéshez. A fejlesztői azonosítónak éves díja van (tanulóként van lehetőség ingyenes regisztrációra), és két szintje van, egyéni, illetve céges. [56][61][62][63]

A tesztelés ajánlott menete [54][59][58]:

1. **Tesztelés emulátorban** – a funkcionális tesztelés nagy része, illetve a hibakeresés, hibajavítás effektíven és gyorsan végezhető emulátorban. Érdemes az első teszteket, amíg az adott stabilitást nem értük el, emulátorban végezni

2. **Tesztelés fejlesztésre nyitott telefonon** – egyes teszteket nem lehet emulátorban elvégezni. Hogy az alkalmazás milyen teljesítményt nyújt egy telefonon, illetve milyen erőforrásokat használ, azt csak az aktuális eszközön tudjuk tesztelni. Teljesítményanalizálási eszközökkel jobb betekintést nyerhetünk, hogy mely részek milyen terhelést jelentenek az adott eszközön, hol kell esetleg javítani a teljesítményen

3. **Béta App publikálása** – Egy béta applikáció az alkalmazásbolton keresztül elérhető, de csak azon azonosítók számára, amelyeket regisztráltunk hozzá. Az így közölt tartalmakhoz már nem szükséges regisztrált és megnyitott mobil, valamint eltűnnek azok az apró különbségek, amelyek a nyitott mobil és a letöltött applikáció között van. Amennyiben népes tesztcsapatunk van, érdemes azokat evvel a módszerrel ellátni az alkalmazásokkal az után, hogy az alkalmazást stabilizáltuk, és már egy stabilabb szakaszába léptünk a tesztelésnek. Az engedélyezett felhasználók táborának kiterjesztésével bevonhatunk külső tesztelőket, potenciális felhasználókat. A kiterjesztés különösen javasolt egy új alkalmazás esetén. A különbség egy béta app illetve egy végleges között, hogy csak a meghatározott Windows ID-vel rendelkezők érik el, illetve a különböző vásárlások nem generálnak valós tranzakciókat.

4. **Rejtett App publikálása** – ez egy opcionális lépés, ha népesebb bétatesztelést szeretnénk végrehajtani, de már úgy, hogy a tranzakciók valósak, akkor ezzel a módszerrel úgy publikáljuk az alkalmazást. A rejtett app nem listázódik, illetve nem jelenik meg a keresési eredmények között, csak direkt link ismeretével férhetünk hozzá. Ez nem teljes körű védelem, a külső keresőmotorok könnyen ráakadhatnak, vagy a felhasználók is küldözgethetik a linket egymás között, viszont az ilyen alkalmazásokkal szemben még mindig kisebb az elvárás, mint a teljesen nyilvánosakkal.

5. **App nyilvános publikálása** – Amikor úgy gondoljuk, hogy már minden rendben van, akkor rendes módon nyilvánosan publikáljuk az alkalmazást

A két legfontosabb különbség, amikor emulátorban illetve fejlesztői telefonon tesztelünk az áruházból letöltötthöz képest[55]:

1. **„Networking capability”** – mint erőforrás automatikusan hozzáadódik, amikor emulátorban vagy fejlesztői telefonon tesztelünk, viszont ha erre szüksége van az appnak, és nincsen benne a buildben, akkor az áruházból letöltött alkalmazás hibásan fog működni

2. **Az app képes írni a saját könyvtárába** – ahová telepítjük, amikor emulátorban vagy fejlesztői telefonon futtatjuk, ami hasznos lehet bizonyos funkciók ellenőrzésekor, de az áruházból történő letöltésnél ez már nem lehetséges. Éppen ezért bizonyos teszteket esetleg nem tudunk elvégezni a béta app segítségével.

iOS [38][40][41][49]

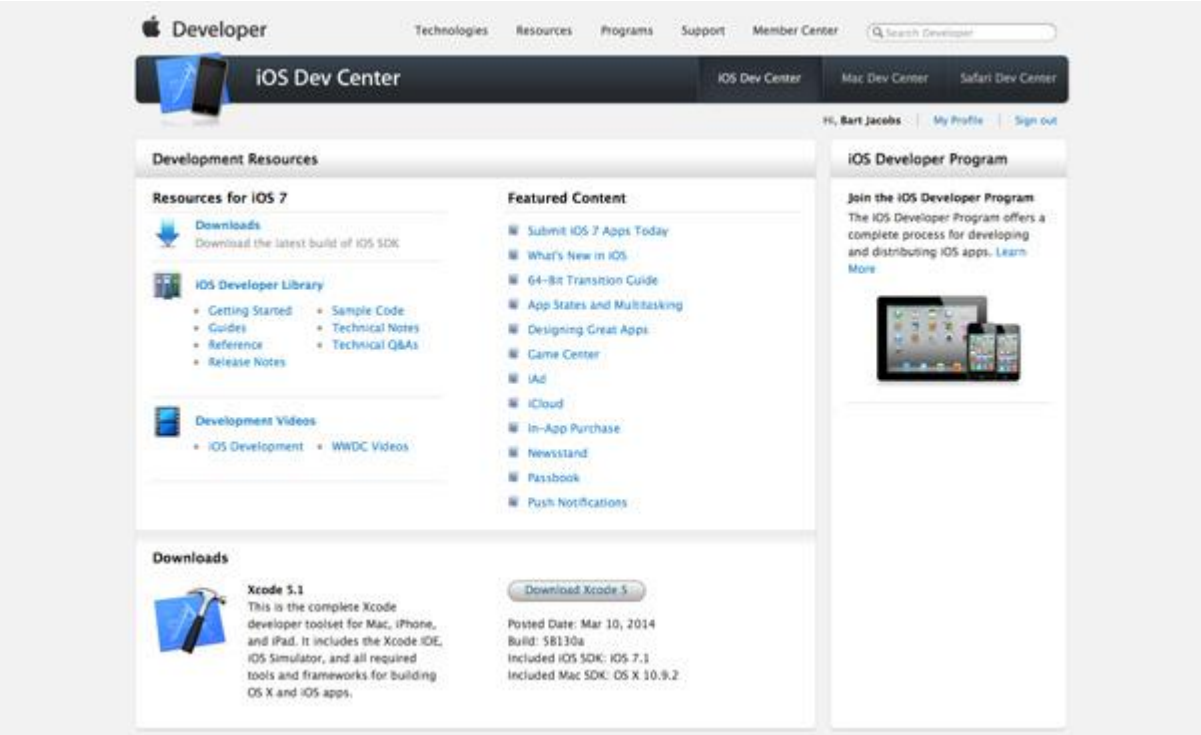
Az iOS eszközökön „Provisioning”-nek hívják azt a folyamatot, amin keresztül engedélyezzük egy applikáció teszt eszközökre való telepítését. A folyamat meglehetősen bonyolult első ránézésre, viszont része a biztonsági rendszernek, aminek segítségével a visszaéléseket kivédik, illetve a felelősöket azonosítani tudják.

iOS fejlesztési környezet csak és kizárólag OS X (Apple) környezetben lehetséges, amennyiben ebben a környezetben szeretnénk fejleszteni, akkor mind a fejlesztőknek, mind a tesztelőknek (ez nem feltétlenül szükséges, de ajánlott) ilyen eszközöket kell biztosítani. Az Apple licencszerződések nem teszik lehetővé a saját fejlesztőeszközeinek használatát más környezetekben, egyedül az iTunes szoftver elérhető más platformokon, de az csak korlátozottan, néhány funkcióra használható.

Az iOS fejlesztési környezetek felállítása meglehetősen bonyolult, viszont a megfelelő felépítés elengedhetetlen ahhoz, hogy a terméket sikeresen ki tudjuk fejleszteni, illetve le tudjuk tesztelni. Az alábbiakban egy egyéni fejlesztő beállításait mutatjuk be, ami ahhoz szükséges, hogy tudja tesztelni a kódját már eszközökön is. Ez az alapeset, amin keresztül az alapfogalmak megérthetőek, amikor azonban csapatban egy cégnél fejlesztünk, az még komplexebb előkészületeket követel, viszont az alapfogalmak tisztázása után az már nem akkora ugrás. A fejezet végén összefoglaljuk a különbségeket, és megadjuk azokat a hivatalos leírásokat, ahol a teljes rendszernek utána lehet nézni, illetve kitérünk még néhány tesztelési opcióra.

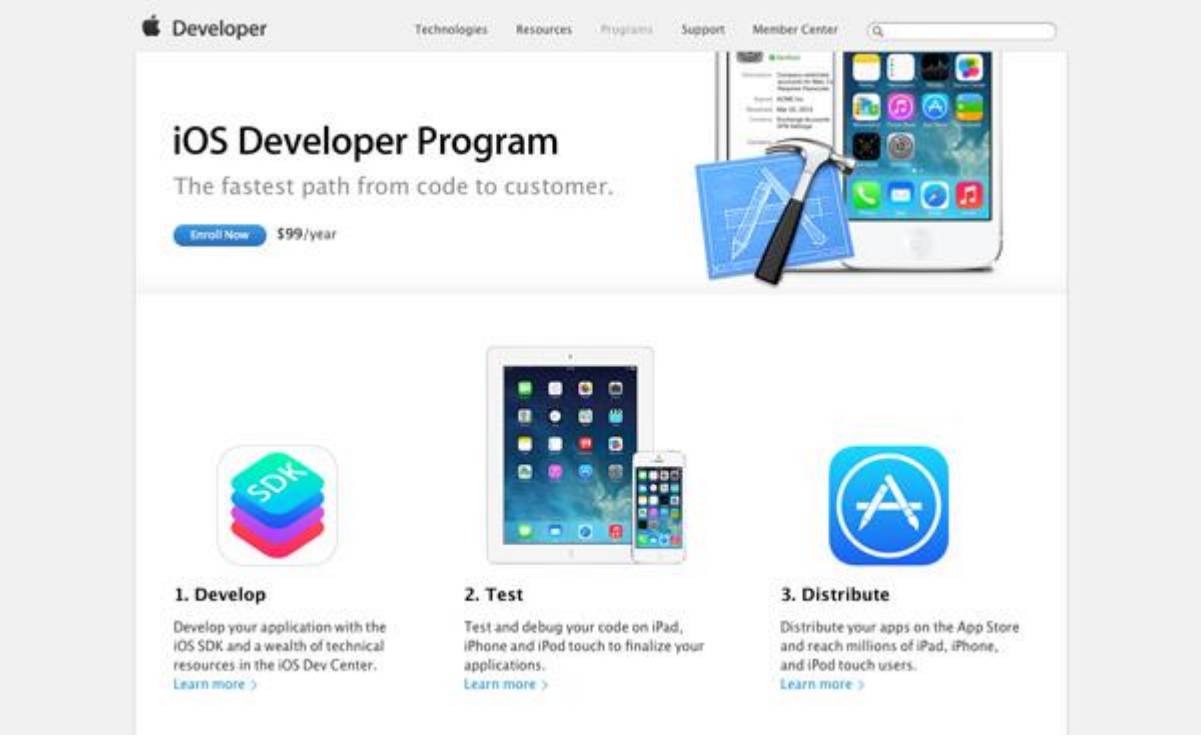
iOS Fejlesztői Program

Az első lépés, hogy be kell lépni az iOS fejlesztői programba (iOS Developer Program), ehhez először regisztrálni kell egy azonosítót az iOS Dev Center-ben (https://developer.apple.com/devcenter/ios/index.action), majd a jobb oldali **iOS Developer Program** dobozban a jobb oldalon a **Join the iOS Developer Program** részben a **Learn More** linkre kell kattintani.



Ábra 5 - Képernyőkép iOS Dev Center [38]

A következő oldalon az **Enroll Now** gombra kattintva lehet a folyamatot elindítani.



Ábra 6 - Képernyőkép, csatlakozás az iOS Developer Programhoz [38]

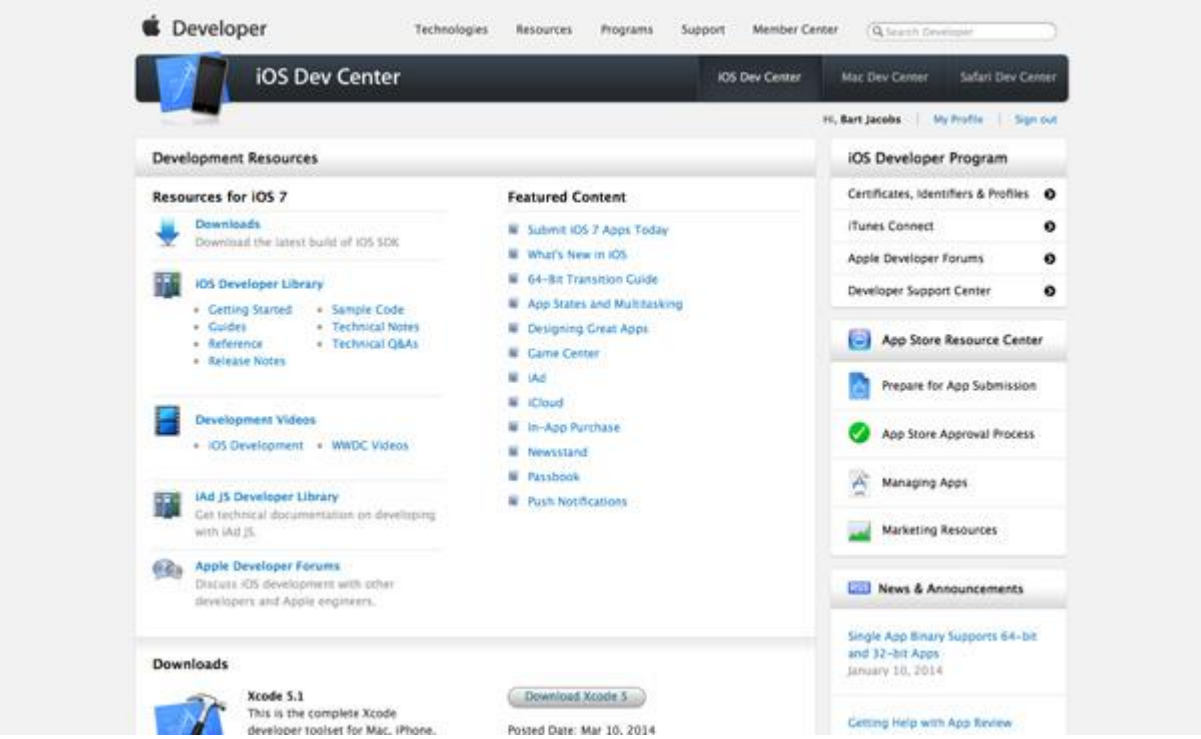
A programban való részvétel nem ingyenes, $99 USD a díja évente, kivéve tanulóknak, akik az alábbi linkről (https://developer.apple.com/programs/ios/university/)[39] indulva ingyenesen csatlakozhatnak.

A harmadik fejlesztői program az iOS Developer Entreprise Program (https://developer.apple.com/programs/ios/enterprise/)[39], amely nagyobb cégek belső saját használatú applikációinak fejlesztését teszi lehetővé.

Amennyiben céges felhasználásra regisztrálunk, akkor szükség van a D-U-N-S számra, ami a céget azonosítja, amit Magyarországi központtal a http://www.dbhun.hu/adatbazis/duns-szam/duns-szam-igenyles címen lehet megtenni.

A döntés nem automatikus, néhány napig eltarthat a folyamat, amíg az Apple elfogadja a jelentkezést.

A jelentkezés elfogadása után az első lépés a digitális aláírások elkészítése, amit az ekkor már megváltozott iOS Dev Center felületén lehet megcsinálni. A változás az **iOS Developer Program** dobozt érinti, ahol a jelentkezés helyett más funkciókat találunk.



Ábra 7 - Képernyőkép, iOS Dev Center csatlakozás után [38]

Az aláírásokhoz szükséges tanúsítványokat a **Certificates, Identifiers & Profiles** részben kell létrehozni.

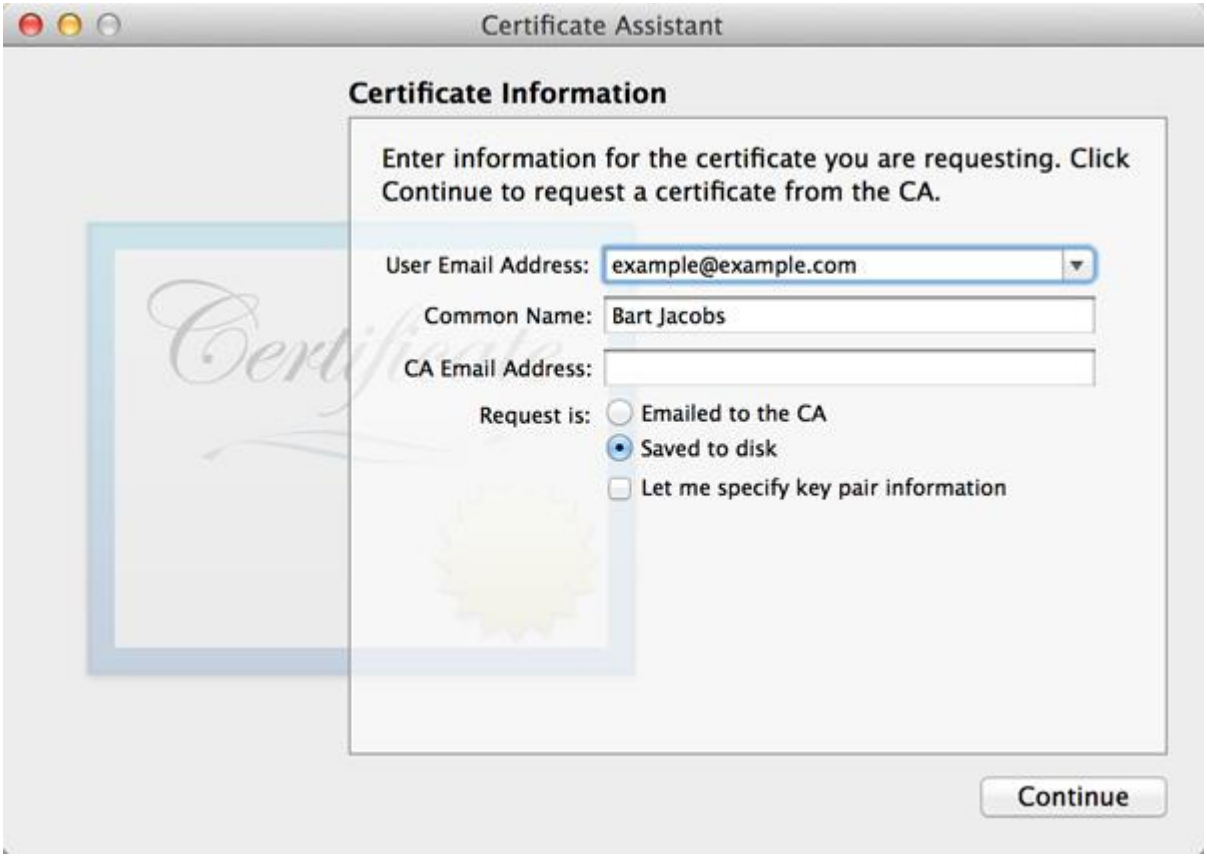
iOS Development Certificate Létrehozása

Először egy **iOS Development Certificate**-t kell létrehozni, ami digitális azonosításra szolgál, és tartalmazza az egyéb azonosításhoz szükséges adatokat, mint név, e-mail cím, céges információk, valamint egy nyilvános és egy privát kulcsot, hasonló módon egy weboldal SSL tanúsítványához.

Az XCode (fejlesztői környezet) a privát kulcs használatával digitálisan aláírja az elkészült alkalmazás bináris állományát.

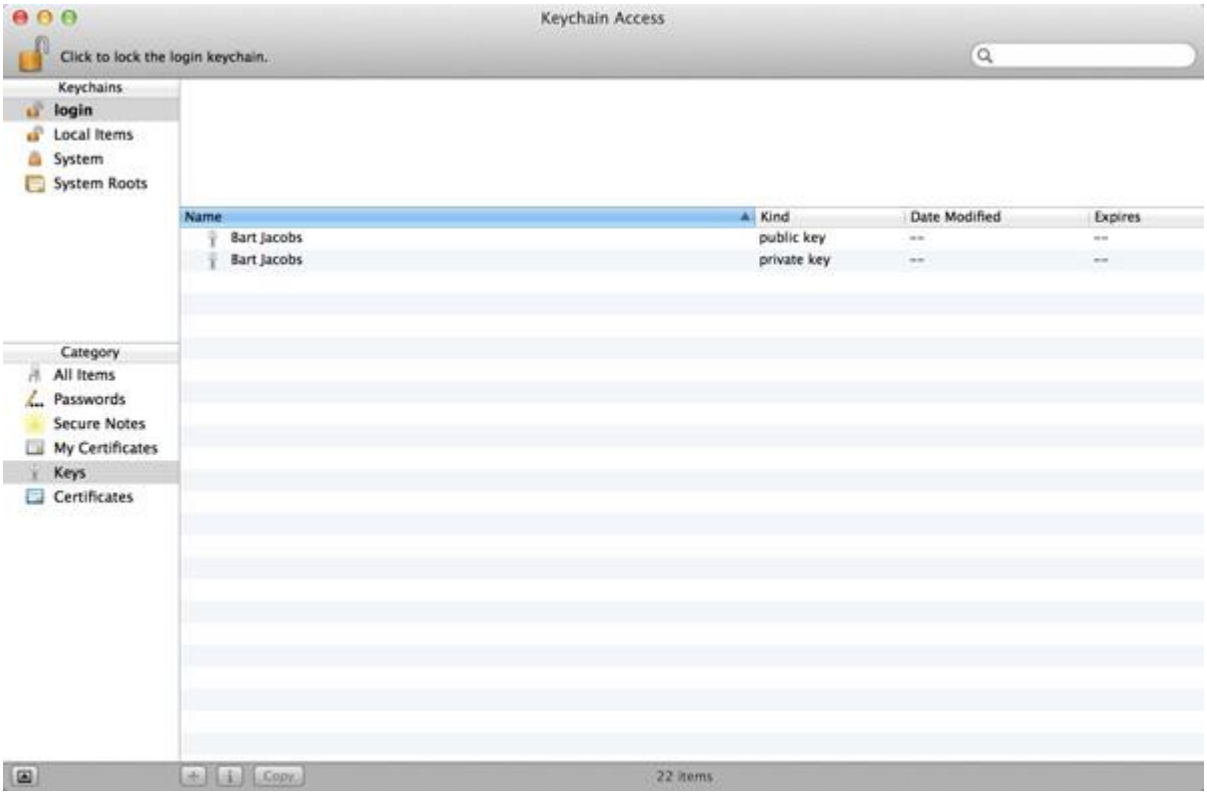
Az első lépés egy tanúsítvány aláírási kérés (certificate signing request, CSR) elindítása. Ezt a **Keychain Access** alkalmazással lehet létrehozni OS X alatt, ami az **Applications** könyvtár **Utilities** alkönyvtárában található. Itt a **Keychain Access** menüben a **Certificate Assistant** pontot kell választani és utána a **Request a Certificate From a Certificate Authority** opciót.

A megnyíló űrlapot ki kell tölteni, avval a névvel és e-mailcímmel, amit regisztrációkor megadtunk, a **CA Email Address** mezőt hagyjuk üresen, válasszuk a **Saved to disk** opciót és hagyjuk bejelöletlenül a **Let me specify key pair information** részt. Kattintsunk a **Continue** gombra, válasszuk ki a mentési helyet, majd kattintsunk a **Save** gombra, amivel létre is hozzuk a CSR-t.



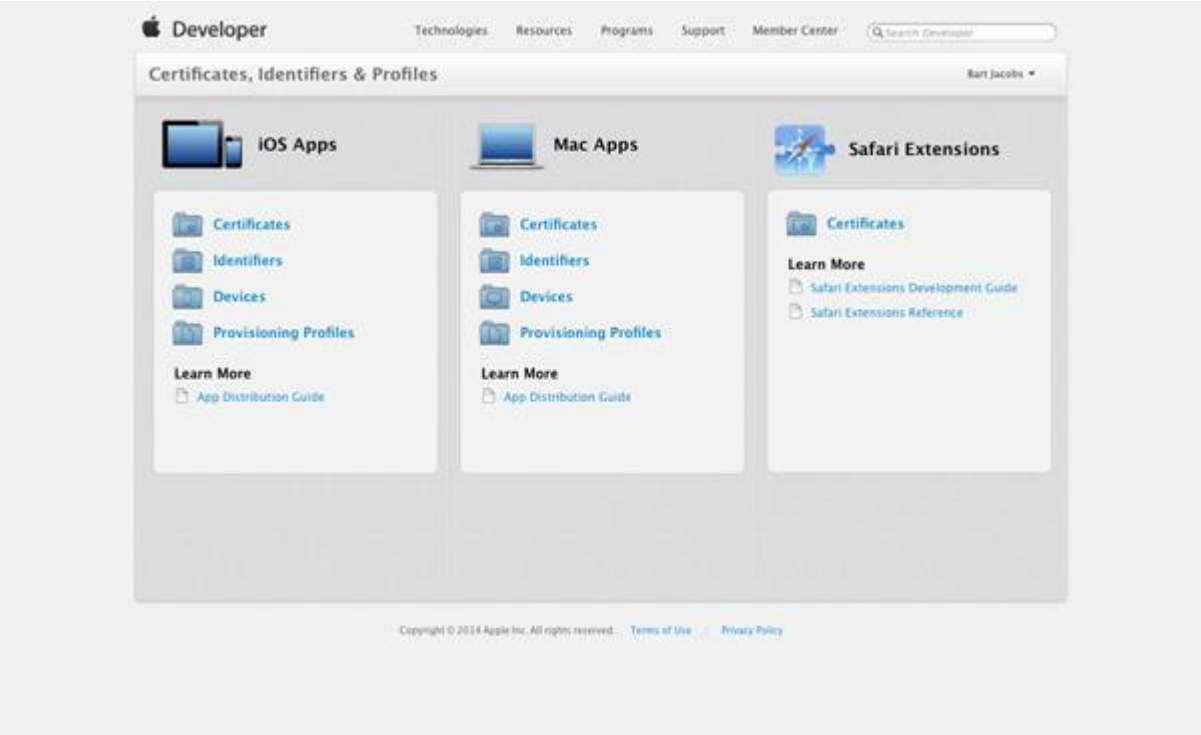
Ábra 8 - Képernyőkép - OS X Certificate Assistant [38]

Ellenőrizzük, hogy valóban létrejött a CSR a mentési helyen. A **Keychain Access**-ben a **Keys** kategória alatt a login keychain-hez hozzáadódott a nyilvános és a privát kulcs.



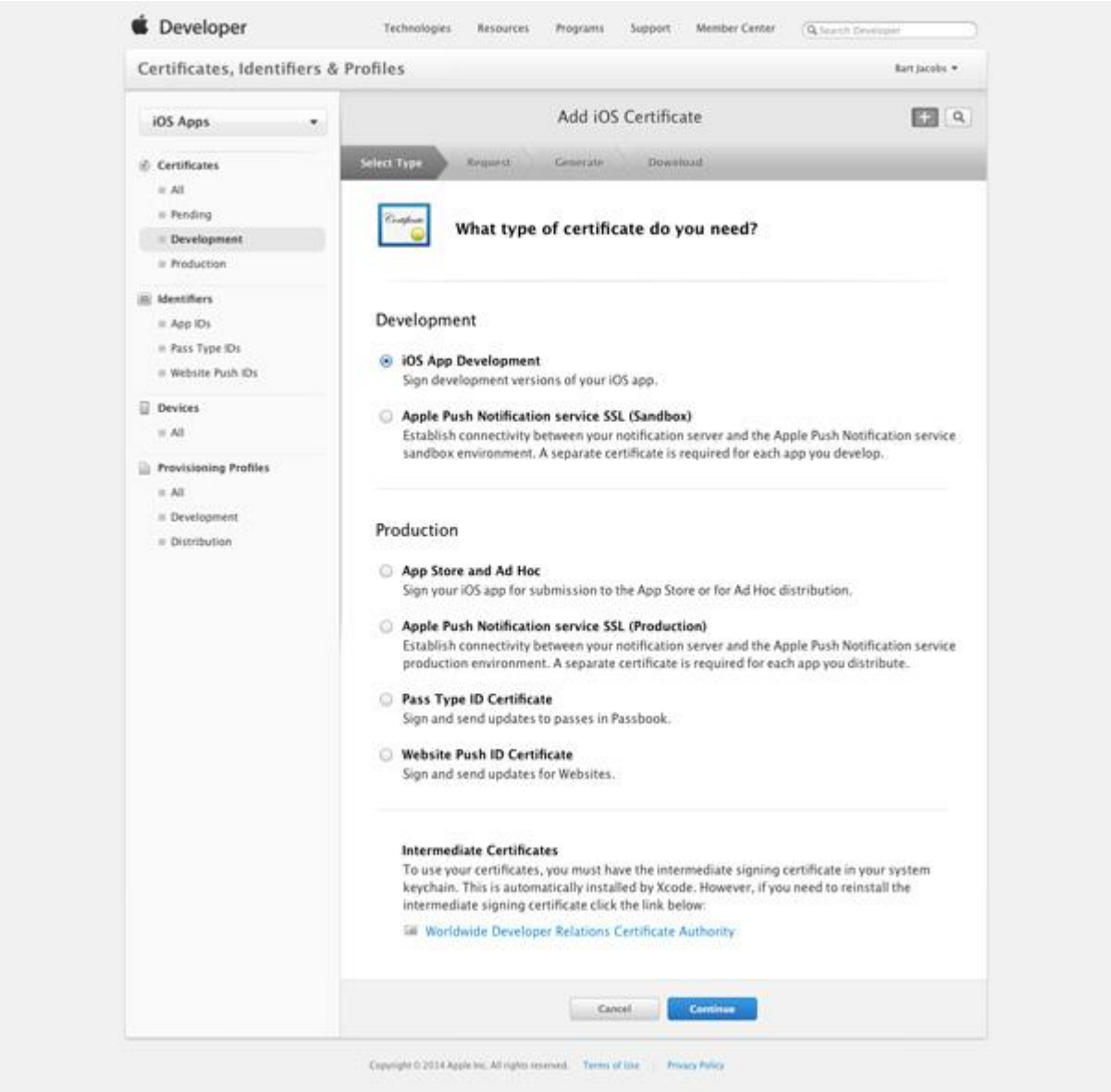
Ábra 9 - Képernyőkép, OS X Keychain [38]

A következő lépés, hogy létrehozzunk egy fejlesztői tanúsítványt. Ehhez térjünk vissza a **Certificates, Identifiers & Profiles** részhez az **iOS Dev Center**-ben. Válasszuk a **Certificates** menüpontot az **iOS Apps** alatt.



Ábra 10 - Képernyőkép, iOS Dev Center Certificates, Identifiers & Profiles [38]

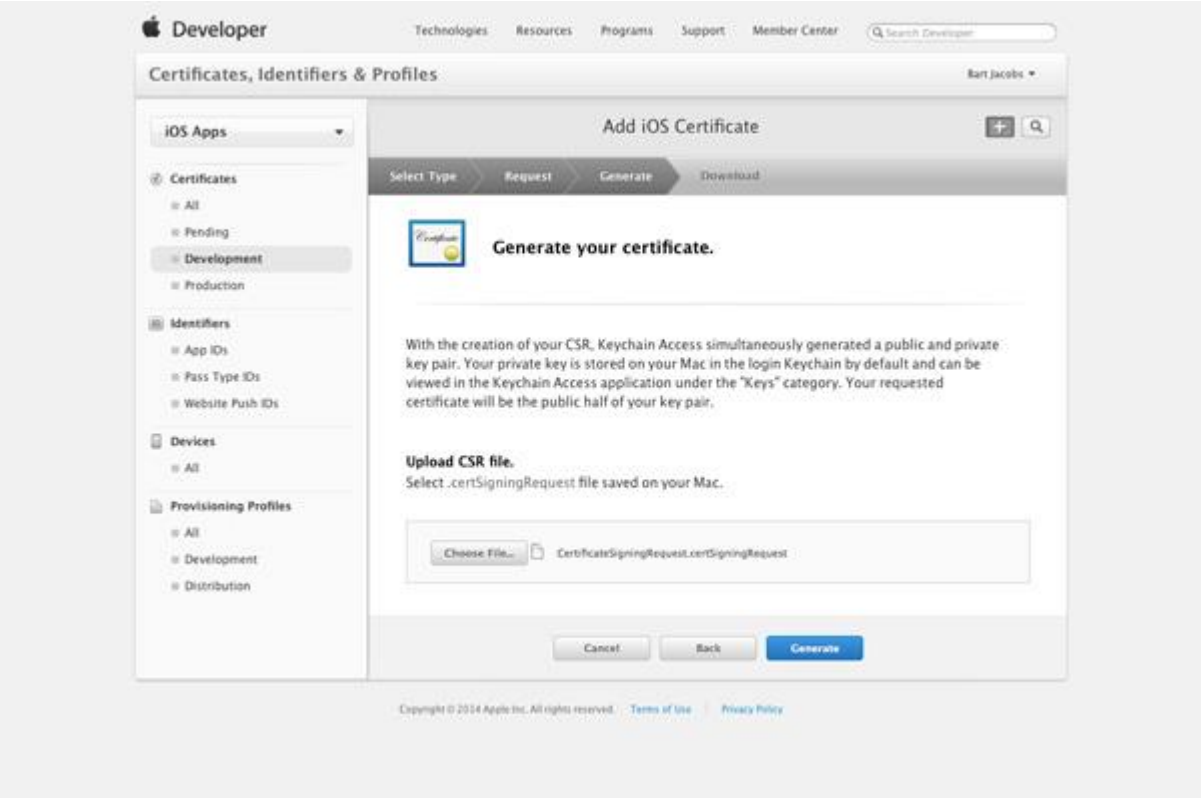
A jobb felső sarokban a pluszjelre kattintva lehet elkezdeni a folyamatot. A **Development** részben a tetején az **iOS App Developmentet** kell választani, majd a **Continue** gombbal tovább kell menni.



Ábra 11 - Képernyőkép, iOS Dev Center, Add iOS Certificate [38]

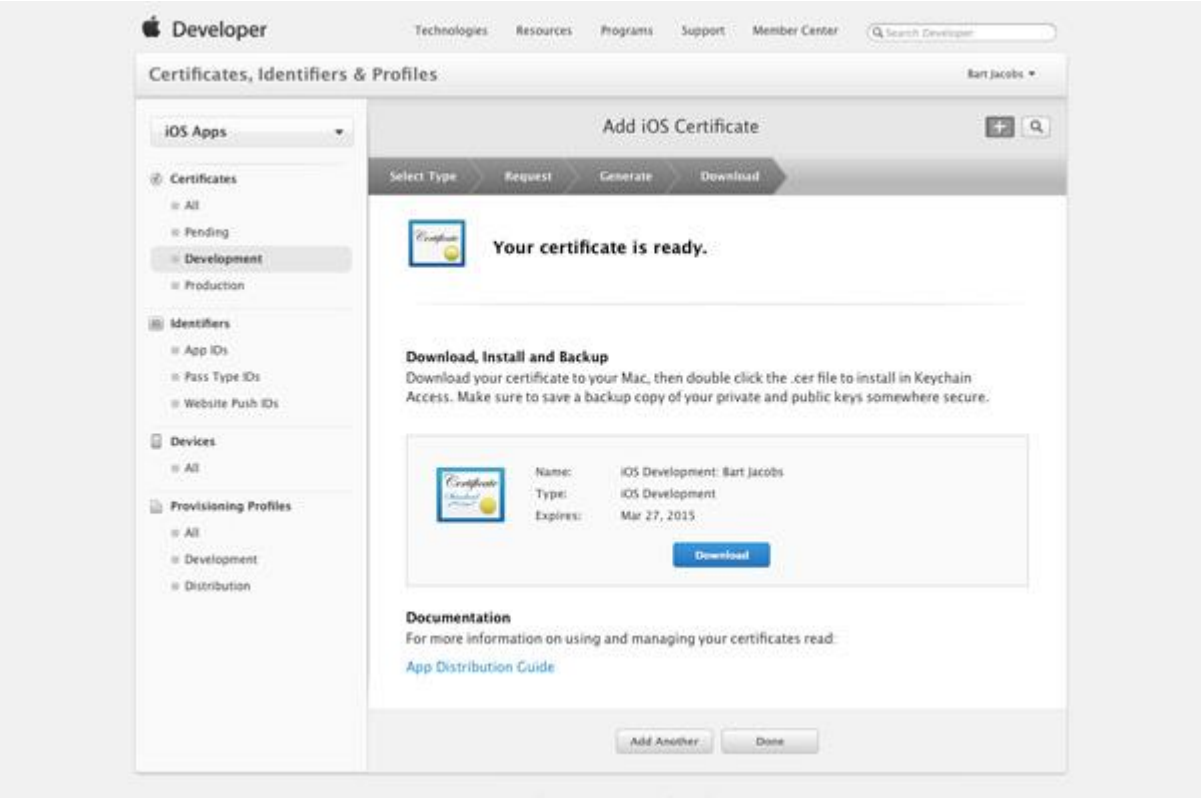
Ezután egy leírás következik, hogy hogyan kell CSR-t készíteni, amit az előző lépésben megtettünk, a **Continue** gombbal nyugodtan továbbléphetünk.

Most fel kell tölteni a CSR fájlt, amit az előzőekben létrehoztunk. A fájlt a **Choose File** gombbal tudjuk kiválasztani, majd a **Generate** gombbal elindítani a folyamatot, ami általában néhány másodpercig eltart.



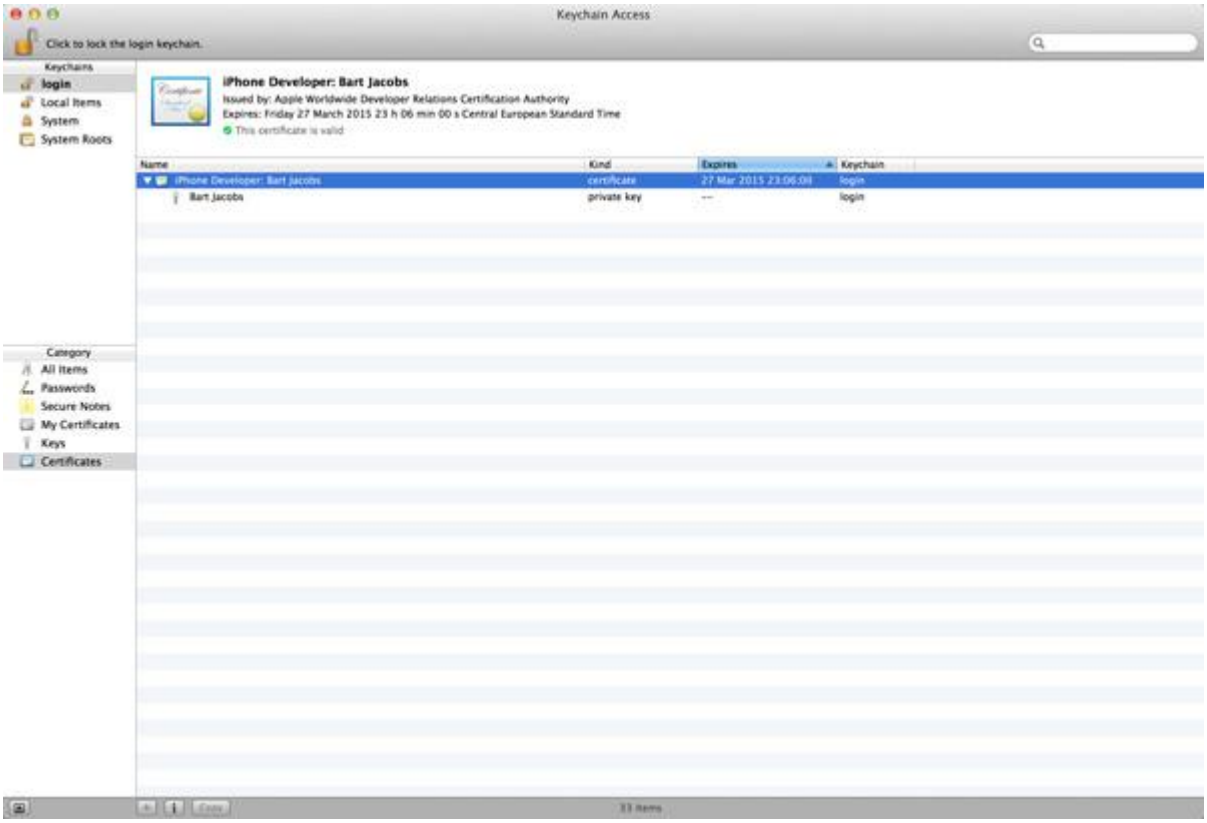
Ábra 12 - Képernyőkép, iOS Dev Center, Tanúsítvány generálása [38]

A **Download** gombbal a tanúsítvány letölthető, ami egy évig érvényes.



Ábra 13 - Képernyőkép, iOS Dev Center, tanúsítvány letöltése [38]

A letöltött fájlra duplán klikkelve a tanúsítványt hozzá kell adni a keychain-hez.

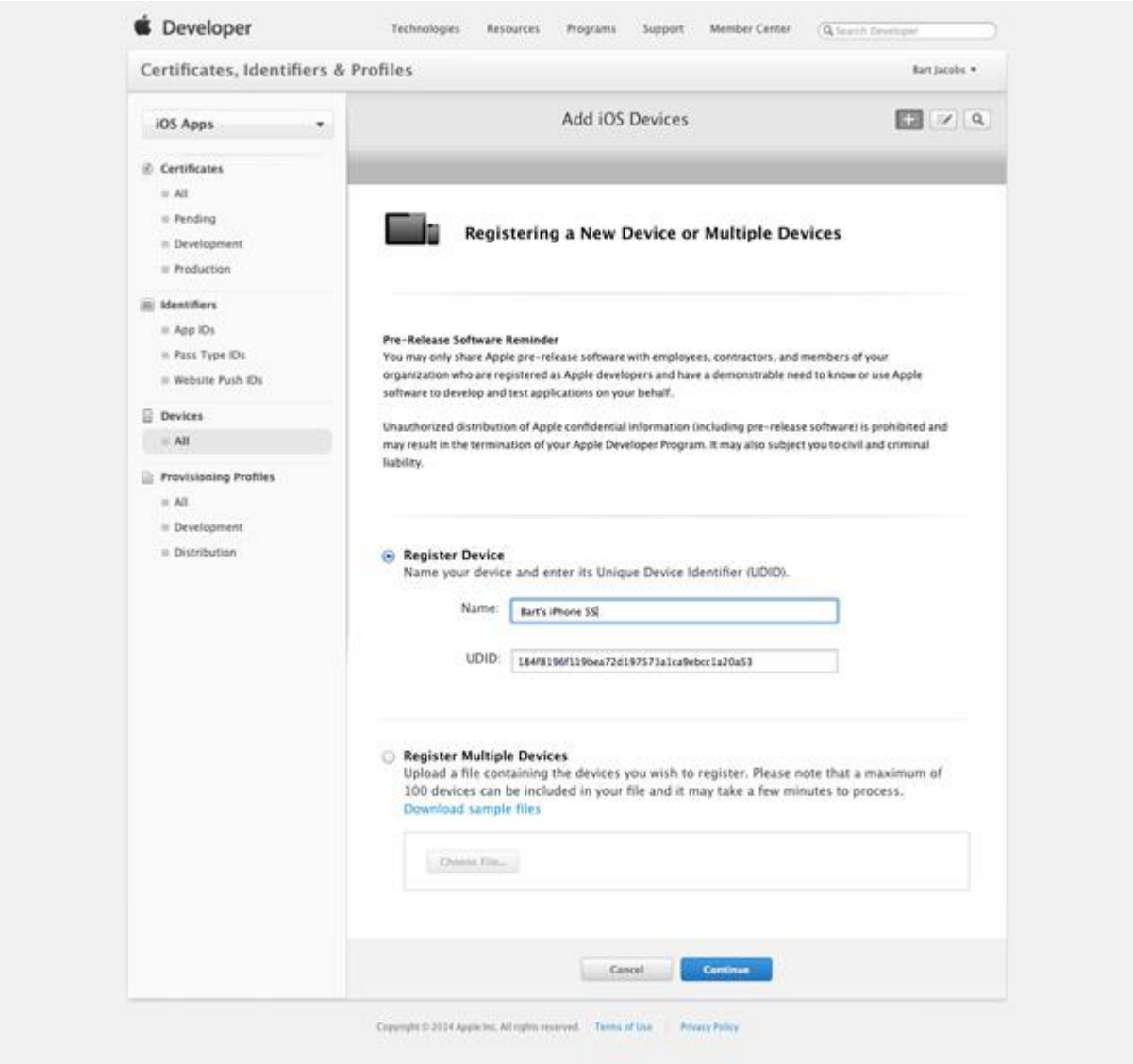


Ábra 14 - Képernyőkép, OS X Keychain, fejlesztői tanúsítvány [38]

Eszközök hozzáadása

A következő lépésben hozzá kell adni az eszközöket, amit használni akarunk a teszteléshez, fejlesztéshez. Ez azért szükséges, mert véletlenszerűen kiválasztott eszközökön nem futtatható egy iOS alkalmazás, csak a hozzáadott eszközökön lesz lehetséges használni a teszteléshez.

Az **iOS Dev Center**nek a **Certificates, Identifiers & Profiles** részén a **Devices** linkre kell kattintani az **iOS Apps** szekcióban. A pluszjelre kattintva a jobb felső részen megnyílik az a szekció, ahol hozzá lehet adni egy eszközt. A regisztrációhoz meg kell adni az eszköz UDID-ját, illetve a nevét. A UDID egy egyedi azonosítója az eszköznek, ami nem egyezik meg a szériaszámmal.



Ábra 15 - Képernyőkép, iOS Dev Center, eszköz hozzáadása [38]

A UDID-t az XCode fejlesztőeszközzel lehet kiolvasni az **Oragnizer** segítségével, ami a **Window** menü alól nyílik. Itt a **Devices** rész alatt kiválasztva az eszközt találjuk a 40 karakteres UDID-t az **Identifier** címkénél.

Ábra 16 - Képernyőkép, XCode Organizer, eszköz részletei [38]

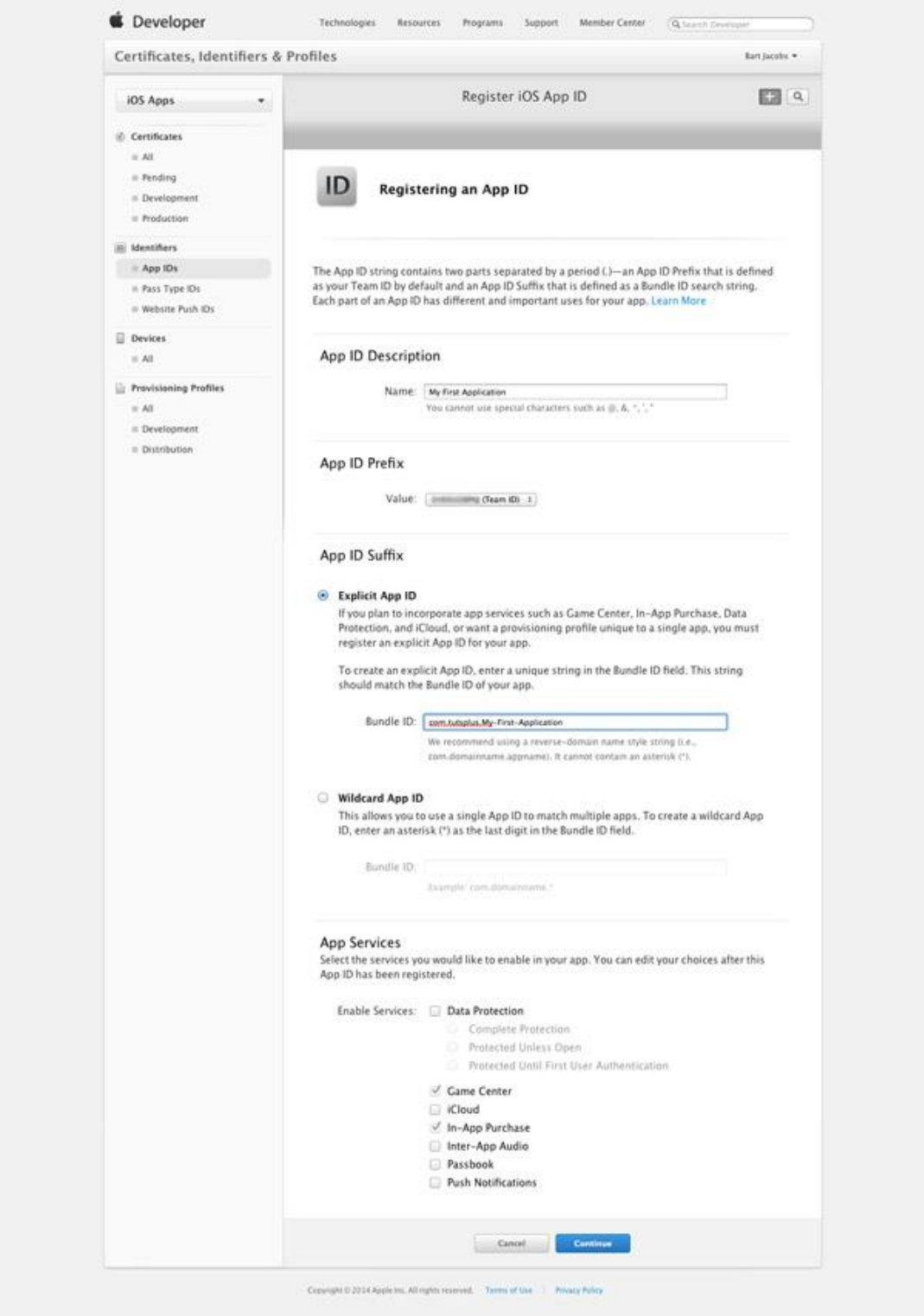
Az eszközök regisztrálásánál látható, hogy van opció több eszköz regisztrációjára egyszerre. Ez különösen egy cég esetén lehet hasznos, ahol sok eszközzel dolgoznak, érdemes lehet egy közös fájlba minden eszközt bepakolni és egyszerre regisztrálni.

A másik lehetőség, ahogy a fenti képernyőképen is látszik, hogy a **Use for Development** gomb segítségével is lehet regisztrálni az eszközt, ekkor vagy a beállítások alapján (**Preferences > Accounts**), vagy ha nincs beállítva akkor az azonosító/jelszó párra való rákérdezéssel kapcsolódik az **iOS Dev Centerrel** az XCode és automatikusan elvégzi a regisztrációt.

App ID Létrehozása

A következő lépés egy **App ID** létrehozása. Az App ID egy alkalmazás egyedi azonosítója, ami a **bundle seed ID**-ból, ami egy 10 karakteres generált egyedi azonosítóból és a **bundle identifier**-ből áll. Az ajánlás az úgynevezett fordított domain név konvenció szerinti elnevezés. A képernyőképen is egy ilyen elnevezés szerepel, ahol a fejlesztő a **tutsplus.com** domain nevet használó cégben a **my-first-application** nevű alkalmazáshoz a **com.tutsplus. my-first-application**-t használja mint bundle identifier. Az App ID pedig **xxxxxxxxxx. com.tutsplus. my-first-application**.

Az App ID-t a szokásos módon regisztráljuk az iOS Dev Centerben, a **Certificates, Identifiers & Profiles** szekcióban az **iOS Apps** alatt az **Identifiers**-t választva meg kell nyomni a pluszjelet a jobb felső sarokban, majd ki kell tölteni az alábbi űrlapot.



Ábra 17 - Képernyőkép, iOS Dev Center, App ID regisztrálása [38]

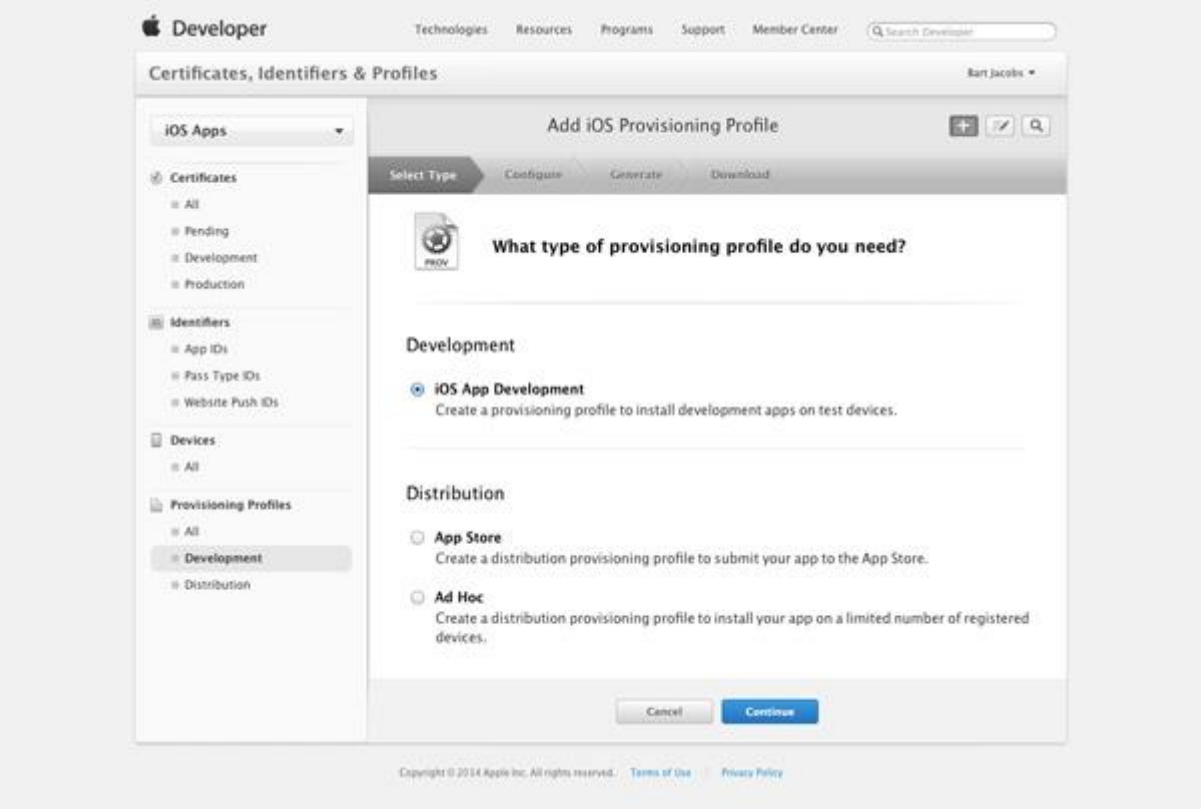
Ne változtassuk meg az **App ID Prefix**-et, és töltsük ki a Bundle ID-t. Kétr lehetőségünk van, az **Explicit App ID** egy App ID-ját tartalmazza, de van lehetőség egy egész csoportot is létrehozni a **Wildcard App ID** opcióval, ha van egy olyan alkalmazáscsoportunk, amit ugyanúgy akarunk kezelni. Ugyanitt meg kell adni azokat az egyéb szolgáltatásokat, amit használ az alkalmazás. Egyes ilyen szolgáltatások kizárják a **Wildcard App ID** használatát.

Provisioning Profil létrehozása

A következő lépés a **Provisioning Profil** létrehozása. Először tisztázzuk a **Provisioning Profil** fogalmát, amimek megértése általában gondot okoz kezdő iOS fejlesztőknek. „A provisioning profil rendelkezésre álló elemek gyűjteménye, ami egyedülálló módon összeköti a fejlesztőket és eszközöket egy feljogosított iOS fejlesztő csapattá, és engedélyezi az eszközöket a tesztelés céljaira”.[1]

Ismét a szokásos módon az iOS Dev Centerben, a **Certificates, Identifiers & Profiles** szekcióban az **iOS Apps** alatt az **Provisioning Profiles**-t választva meg kell nyomni a pluszjelet a jobb felső sarokban.

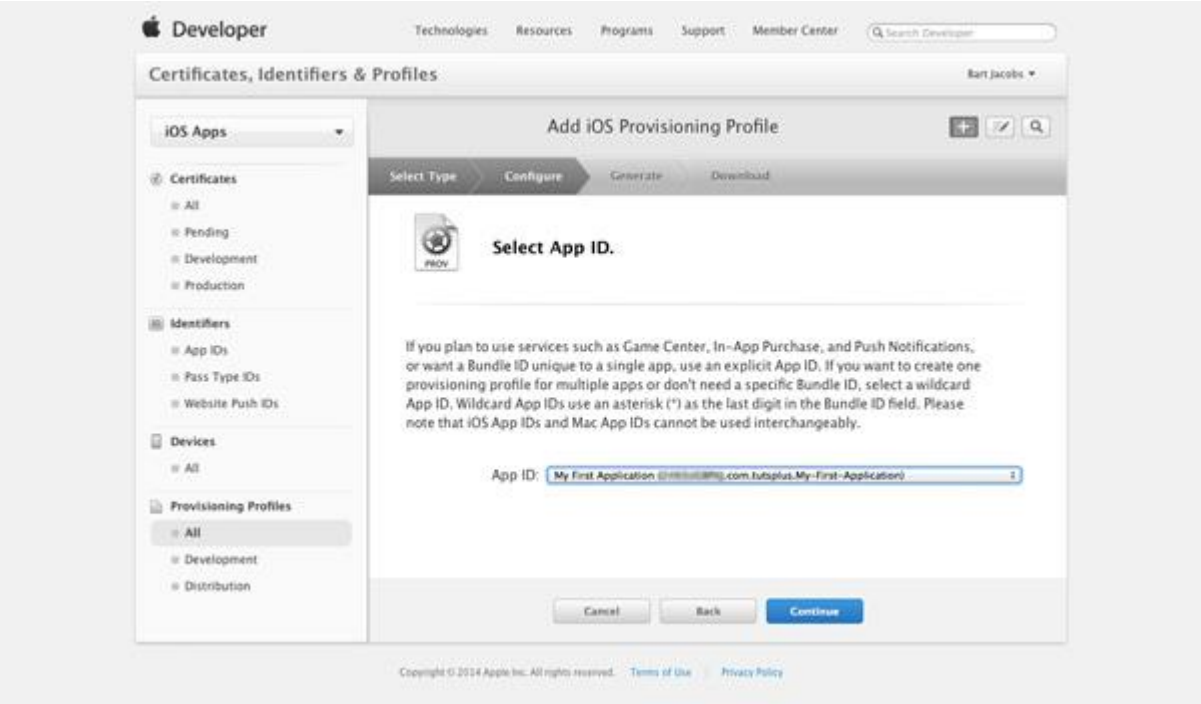
Az **iOS App Development**-et választva meg kell nyomni a **Continue** gombot.



Ábra 18 - Képernyőkép, iOS Dev Center, fejlesztői Provisioning profil létrehozása [38]

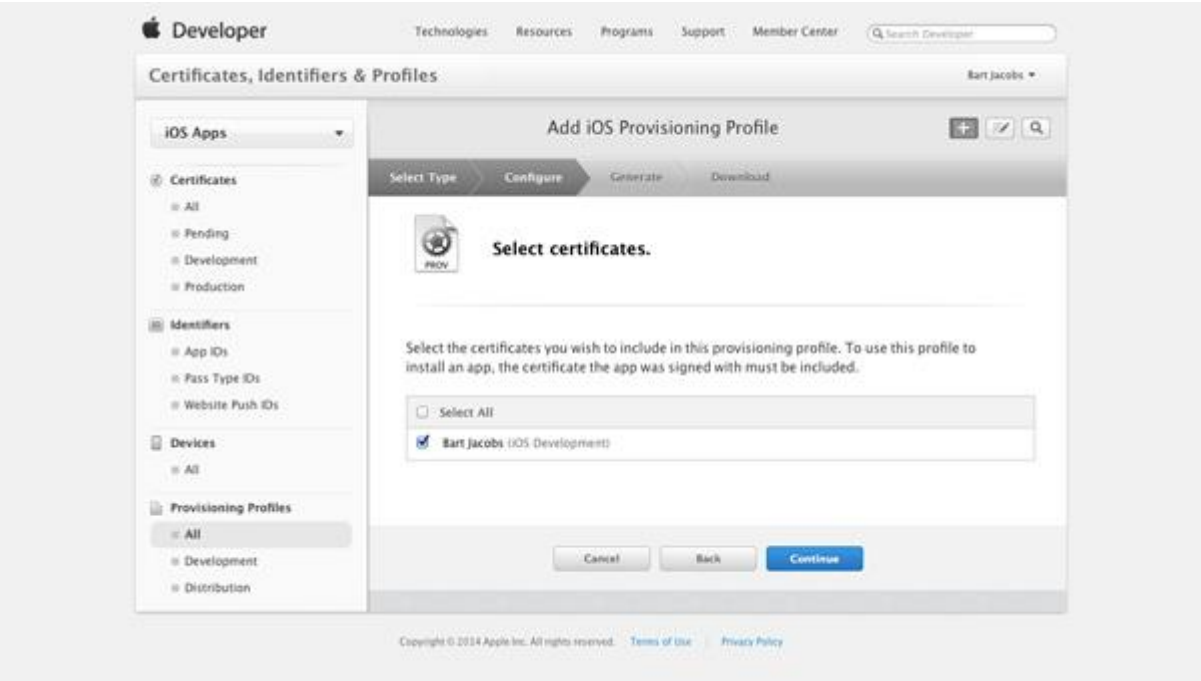
A következő lépésben ki kell választani az **App ID**-t, és a **Continue** gombbal folytatni.

[1] A provisioning profile is a collection of assets that uniquely ties developers and devices to an authorized iOS Development Team and enables a device to be used for testing.



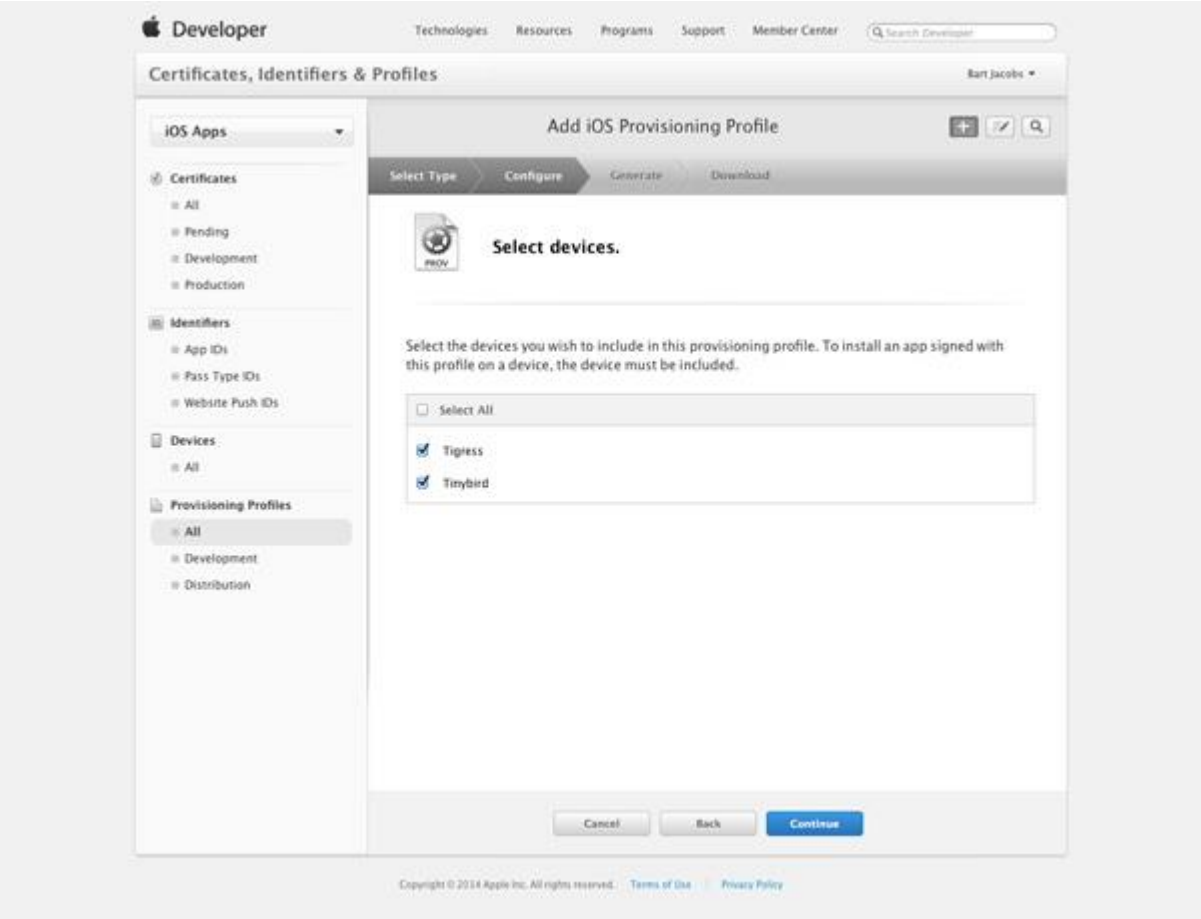
Ábra 19 - Képernyőkép, iOS Dev Center, App ID hozzáadása a Provisioning profilhoz [38]

A következő lépésben a **Development Certificate**-t kell hozzáadni.



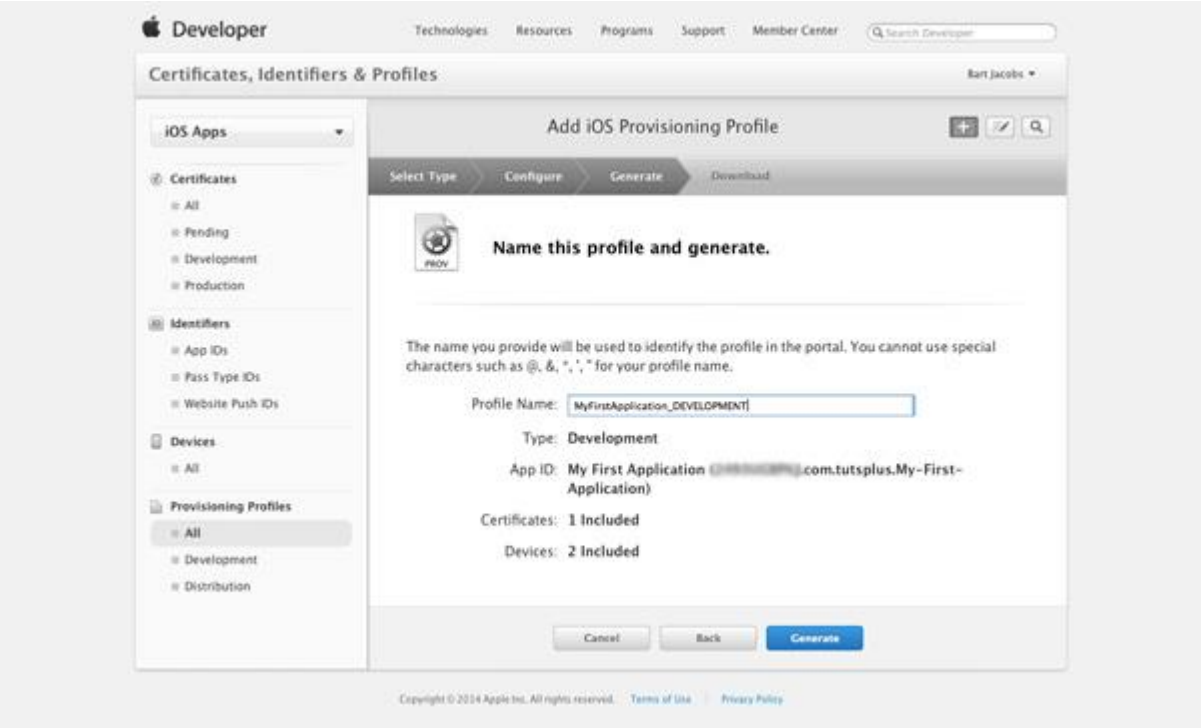
Ábra 20 - Képernyőkép, iOS Dev Center, fejlesztői tanúsítvány hozzáadása a Provisioning profilhoz [38]

Majd hozzá kell adni az eszközöket is. Csak ezek az eszközök lesznek képesek arra, hogy fejlesztés alatt futtassák az adott App-ot.



Ábra 21 - Képernyőkép, iOS Dev Center, eszközök hozzáadása a Provisioning profilhoz [38]

A következő lépésben már csak egy nevet kell adni és a **Generate** gombbal letölthetjük az elkészült profilt, amit dupla kattintással importálhatunk az Xcode-ba.

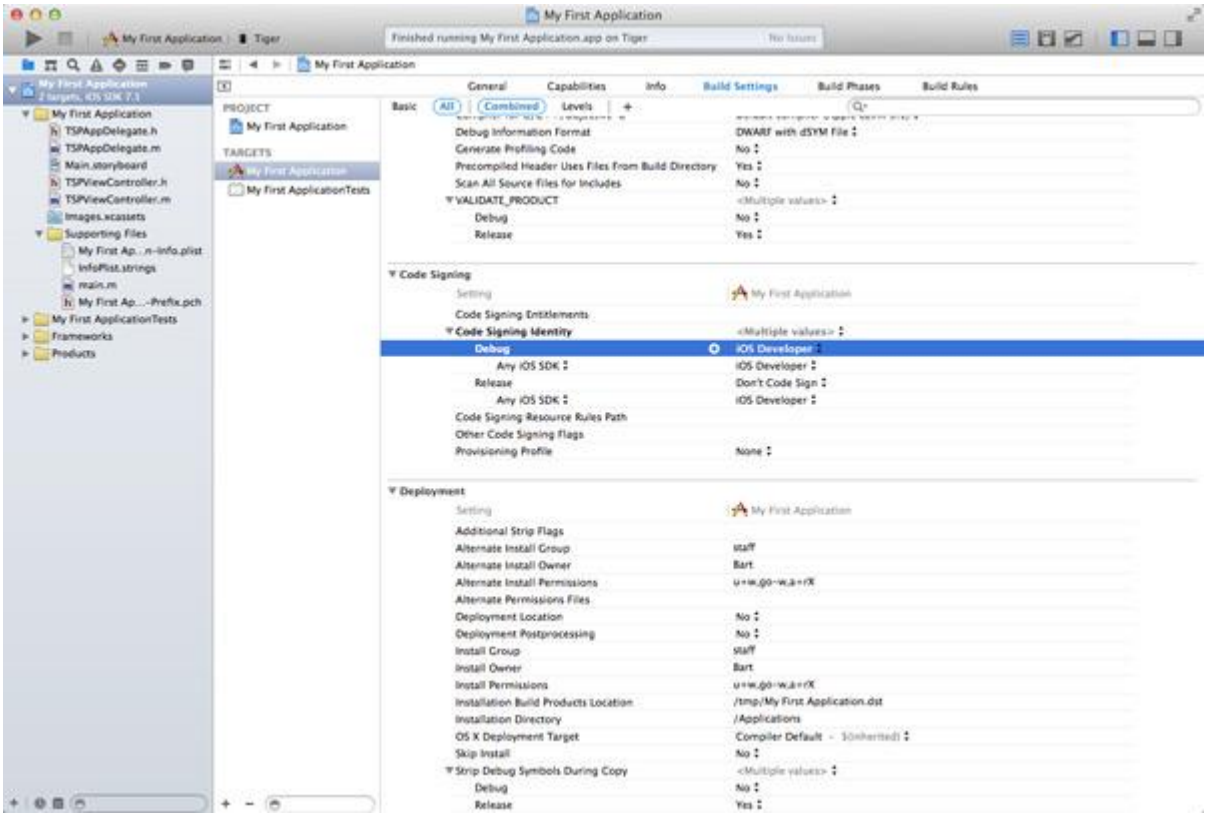


Ábra 22 - Képernyőkép, iOS Dev Center, a Provisioning profil generálása [38]

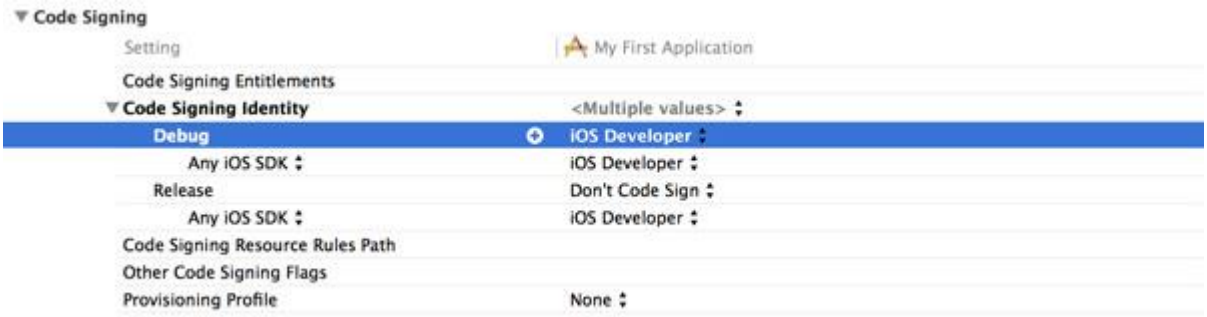
Amennyiben utólag hozzá szeretnénk adni még eszközöket, a profil szerkesztésével és újraimportálásával megtehetjük.

A Projekt konfigurálása

Ahhoz, hogy a projekt debuggolható legyen meg kell változtatni egy beállítást, ami az Xcode-on belül a **Project Navigator > Build Settings** alatt található, és ott a **Code Signing > Code Signing Identity > Debug** opciót be kell állítani a megfelelő fejlesztőre



Ábra 23 - Képernyőkép, XCode projektbeállítások [38]



Ábra 24 - Képernyőkép, XCode projektbeállítások, Debug beállítások [38]

Ezután már nincs más hátra, mint fordítani és futtatni.

Csapatmunka

Céges azonosító esetén a **Member Center** [43][44] segítségével tudunk több személyt is összekapcsolni. A tagoknak különböző jogosultságaik vannak:

Szerepkör	Leírás
Team Agent	A Team Agent a jogilag felelős személy a csapatért, és az elsődleges kapcsolattartó az Apple felé. A Team Agent tud meghívni új embereket a csapatba, és beállítani a jogosultságokat. Csak egyetlen Team Agent lehet.
Team Admin	A Team Admin a Team Agent kivételével mindenkinek be tudja állítani a jogosultsági szintjét, meg tud hívni új tagot. Menedzseli az aláírási jogosultságokat mind fejlesztés alatt, mind a külső kiadásoknál. Csak a Team Admin képes olyan kiadást létrehozni, ami nem a fejlesztői eszközökön fut. Engedélyezi a Team Member-ek aláírási tanúsítványait.
Team member	A Team Member fejlesztői változatot tud aláírni, de csak azután, miután egy Team Admin ezt engedélyezte

Táblázat 9- iOS fejlesztés szerepkörök

A **Member Center** működéséről itt található egy részletes leírás: https://developer.apple.com/library/mac/documentation/IDEs/Conceptual/AppDistributionGuide/ManagingYourTeam/ManagingYourTeam.html#//apple_ref/doc/uid/TP40012582-CH16-SW1

Tesztelés nem fejlesztői eszközön [41]

Az eddig ismertetett módszerekkel a tesztelőnek is szüksége van fejlesztői hozzáférésre. Egy másik lehetőség arra, hogy a tesztelő fel tudja telepíteni a csomagot egy **Ad Hoc Provisioning Profil** létrehozása. Ehhez már nem elég az **iOS Developer** tanúsítvány, hanem **iOS Distribution** kell hozzá, ami céges fejlesztésnél csak az adminoknak lehet. Ebben a Provisioninig Profile-ban meg lehet adni azokat a UDID-ket, amelyekkel tesztelni fogunk, és evvel készítjük el az appot, majd ezt iTunes segítségével lehet telepíteni azokra az eszközökre, amit megadtunk.

Béta tesztelés [41][42][46][49]

Béta tesztelés lebonyolítására több lehetőség is van. Az egyik, hogy igénybe vesszük az Apple által kiépített infrastruktúrát és iTunes Connect vagy a TestFlight app segítségével hívunk meg tesztelőket.

A másik lehetőség az előbb említett Ad Hoc Provisioning profil segítségével tesszük elérhetővé az alkalmazást a tesztelők számára.

App Services tesztelése [45][46][47][48]

Az Apple biztosít tesztszervereket, amin a különböző App Service-eket tesztelni lehet (mint például Game Center vagy App in Purchase). Alapvetően, amíg a disztribúció nem végleges, az eszköz alapvetően a tesztrendszerhez, és nem a végfelhasználói környezethez kapcsolódik. Az iConnecten belül lehet létrehozni teszt azonosítókat (megfelelő jogosultsággal).

Amikor ilyen jellegű szolgáltatást kell tesztelni, mindenképpen érdemes elolvasni a megfelelő leírásokat, mielőtt belekezdünk.

A céleszközök sokszínűségének kezelésére alkalmazott módszerek

Egy nagyobb fejlesztőnek sok termékkel, alkalmazottal és már fenntartható bevételekkel nem probléma egy akár több száz eszközös tesztflotta (sok különféle eszköz, és azon különböző operációs rendszer flotta) fenntartása, hiszen azt a több projekt és nagyszámú alkalmazott miatt jó kihasználtsággal tudja alkalmazni, de ez egy kisebb fejlesztőnek nem reális opció, a kihasználtság alacsony lenne, és akkora beruházást igényelne, ami nem kitermelhető a bevételekből. A megoldásra emeljünk ki két lehetséges alternatívát:

1. **Online szolgáltatások** – léteznek olyan online szolgáltatások, amelyekkel interneten keresztül felhőalkalmazás formájában elérhetővé tesznek rengeteg eszközt manuális és/vagy automatikus tesztelés céljából. Ezek az eszközök általában havidíjasok, és meghatározott mennyiségű használatot tartalmaznak.

2. **Crowd testing szolgáltatók** – olyan szolgáltatók, akik tesztelési szolgáltatás adnak bérbe. Ennek lényege, hogy a tesztelést nem főállású tesztelők végzik, hanem saját környezetben dolgozó külsős tesztelők, akiket egy online disztribúciós rendszer köt össze a megbízóval. A rendszer előnye az, hogy a tesztelők sokféle eszközzel rendelkeznek, amiből összeállítható a megfelelő kollekció, valamint a tesztelést is megcsinálják, a hátránya, hogy mivel nem feltétlenül profi tesztelőkkel dolgozunk, megfelelően kell specifikálni a feladatot és az elvárásokat, valamint a rendszerjutalékot is meg kell fizetni.

Hardverfejlesztés

Egy hardverfejlesztés nem csak az elektronika fejlesztését jelenti, hanem hozzá kapcsolódóan több szoftveres feladat is van, ami együtt jár annak tesztelésével. Ezek a projektek nagyon sokfélék lehetnek, és most nem is fogunk kitérni minden részletre, inkább a közös jellemzők azok, amire koncentrálni fogunk, a specifikumokkal valószínűleg több könyvet is megtölthetnénk.

Nézzünk néhány példát, milyen projektekről is beszélhetünk jelen pillanatban:

- Számítógépes alkatrészek – ahol egyrészt az alkatrészekhez meghajtókat kell fejleszteni, illetve a legtöbb ilyen alkatrész belső szoftverkomponensekkel úgynevezett firmware-rel is rendelkezik

- Beágyazott rendszerek – célfeladatot ellátó célszámítógépek, amelyek bizonyos feladatra specializáltak. Kevesen gondolnak bele, de manapság egy autóban akár húsz-harminc ilyen beágyazott rendszer, komplett számítógép is dolgozik, és egy nagyon bonyolult hálózatot alkotnak, de olyan hétköznapinak számító tárgyak, mint egy set top box, vagy egy laposképernyős televízió is egy ilyen rendszer, csak nem feltétlenül így tekintünk rájuk

- Komplett számítógépek (laptop, tablet, okostelefon), amelyeket, mint komplett rendszert kell integrálni.

Az ilyen termékek követelményei egyrészt adódnak a funkcionális követelményekből, amit az adott elemnek szánunk, de emellett rengeteg egyéb előírásnak kell megfelelni, ami a további követelményeket biztosítja. Néhány ilyen forrás példaként:

- Egy autóipari rendszernél törvényileg szabályozott biztonsági követelmények, túlbiztosítások
- Egy autó utasterében elhelyezett elemeken (például akár egy utólagos telefon-kihangosító kijelzőjén) előírhatnak fejre ütközési tesztet (head impact test)
- A GCF[64] teszt egy olyan teszt, ami biztosítja, hogy a mobiltelefonok megfelelően működnek a hálózaton. Ezt a független tesztet általában megkövetelik a mobiloperátorok az adott telefon forgalmazásához
- Érintésvédelmi szabványoknak meg kell felelni a terméknek
- Különböző klimatikus viszonyokat át kell vészelnie, illetve az adott viszonyok között működnie kell az eszköznek. Ezek a viszonyok az elvárt üzemeltetési körülményektől függenek, egy autó utasterében például a hőmérséklet nagyon tág határok között mozoghat
- A tartósságot is ellenőrizni kell, hogy a megkövetelt élettartamot kiszolgálja

Látható, hogy a követelmények egy része hardveres, de sok követelménynek szoftveres teszteket követel, sőt olyan is van, ami egyszerre igényel hardveres és szoftveres tesztelést. Ismét csak egy példa, egy adott készüléknek működnie kell egy bizonyos hőmérséklettartományban. Itt funkcionálisan kell ellenőrizni az eszközt bizonyos határértékeken, amit általában klímakamrában tudunk elvégezni.

Teszteszközök

Egy ilyen fejlesztésnél a teszteléshez nem csak a szokásos számítógépes környezetet kell biztosítani, de nagyon sok egyéb eszközt is be kell vetnünk. Ilyen eszközök lehetnek (a teljesség igénye nélkül):

- Mérőműszerek, mérőeszközök
- Klimatikus tesztkamrák
- Különböző eszközök, amelyek a hosszú távú használatot szimulálják
- Speciális célú teszteszközök (telefonfejlesztéshez például elengedhetetlen olyan eszközök használata, amivel a hálózati eseményeket szimulálni tudjuk)
- Speciális érzékelők
- Csonkok és meghajtó eszközök, amivel az esetlegesen nem létező környezetet szimuláljuk
- Prototípusok

Az eszközök egy részére egy időszakban állandóan szükségünk van, ezeket be kell szerezni, és/vagy hozzá kell rendelni állandó jelleggel az adott projekthez, de vannak eszközök, amelyek meglehetősen drágák, és csak az idő egy töredékében kell.

Azon drága eszközöknél, amelyek csak időnként kellenek, általában több megoldás közül választhatnak a cégek. Amennyiben több projektjük is fut, aminek ilyen eszközre szüksége van, és azt így jó kihasználtsággal tudják hasznosítani, akkor általában beszerzik a megfelelő eszközt, és egy jó

időmenedzsmenttel megosztják a projektek között. Annak költségét használatarányosan visszaosztják a projektre. Egy jó eszközmenedzsment-rendszer jelentős megtakarítást tud hozni összességében az eszközök minél jobb kihasználásával.

Vannak azonban olyan eszközök, amelyekre csak alig van szükség, ilyenkor általában bérelik a berendezést külső beszállítótól (általában egy specializált tesztközponttól), ami így jelentős megtakarítást hozhat. Egyes esetekben nem csak az eszközt lehet bérbe venni, hanem a teljes tesztelést is ki szokták ilyenkor adni, mert a tesztközpontban, ismerve a saját berendezéseiket, hatékonyabban tudják megtervezni az adott tesztet.

A projektek tervezésekor az eszközigényeket és allokációkat itt sokkal jobban kell tervezni, mint egy sima szoftveres projekt esetén, általában sokkal több a megosztott erőforrás egy hardvert is fejlesztő projektben. A tervezés nem elég, hanem folyamatosan kontrollálni kell a fejlesztés menetét. Amennyiben valamilyen változás van, ami hatással van egy olyan tesztelés menetrendjére, ami osztott erőforrást használ, akkor azokat időben át kell tervezni. Ha annak az áttervezése nem megoldható, akkor a megfelelő kockázatot kezelve biztosítani kell, hogy a késést okozó adott tevékenységek visszakerüljenek az eredeti menetrendnek megfelelő időpontra.

Egy-egy termék, amit fejlesztünk, maga is egy másik nagyobb termék (általában új termék) része, ami nem áll rendelkezésre a teszteléshez. Ilyen esetekben meghajtók és csonkok segítségével egészítjük ki a saját rendszerünket, és a tesztelés céljaitól függően akár több ilyenre is szükség lehet. Ez csak egy számítógépes interfész és egy program, de akár mechanikus szerkezetek is tartozhatnak hozzá, ami a termék fizikai valóságát szimulálja. Egyes esetekben, amikor egyedi feladatra kell, és készen nem kapható hasonló a piacon, előfordul, hogy a teszt eszközt is meg kell építeni és le is kell tesztelni.

Amikor a mi termékünk beépül egy másik nagyobb termékbe (és speciálisan ahhoz készül), a tesztelés folyamán eljutunk egy olyan pontra, amikor már a másik termékbe integrálva teszteljük az eszközöket. Ilyenkor a két projektcsapatot – a beépülő termék és a teljes termék csapatait – integrálni szükséges, hogy pontosan tervezni lehessen az adott tesztidőpontokkal, a tesztelési felelőségi körökkel, és azok eszközigényével.

Prototípusok

A prototípusok fontos részei a tesztjeinknek, mint tesztalanyok. Egyes darabok a tesztek során meghibásodnak, megsemmisülnek. Egy részük tervezett módon a teszt részeként (például egy ütközésteszt nagy valószínűséggel tönkreteszi az egységet), mások pedig olyan hibák következtében, amit nem terveztünk előre.

Mivel a hardvert is fejlesztjük és teszteljük, általában a prototípusokból több generációt is legyártanak. Ezeket általában kis szériában manufakturális módszerekkel állítják elő, a végtermék gyártási költségének többszöröséért. A prototípusok egyre közelítik kidolgozottságban a végterméket. Míg az első generáció lehet, hogy csak egy mérnöki tábla (engineering board), ahol az alkatrészeket egy megfelelő táblán összerakjuk, és az első prototípus nem rendelkezik minden mechanikai elemmel (például burkolattal), a későbbi generációk már jobban hasonlítanak a végtermékre, az utolsó 2-3 generáción már jó esetben alig vannak változtatások.

Mivel a prototípusokat meghatározott időnként gyártják le, azok darabszámát tervezni kell, hogy mind a fejlesztési folyamathoz, mind a teszteléshez megfelelő számú prototípus álljon rendelkezésre, amibe bele kell számolni a prototípusok esetleges meghibásodását, megsemmisülését is. Az alultervezés a projektben lassítja a munkavégzést, a jelentős túltervezés viszont a drága gyártási folyamat miatt jelentős extra költséget jelent. A tapasztalatok alapján az használható módszer, hogy megtervezzük, hogy mennyi szükséges a projekttervek, tevékenységek és a résztvevők száma alapján, és arra 10-20%-nyi többletet teszünk a váratlan események kezelésére.

A tapasztalat azt is mutatja, hogy a szükséges prototípusok száma a projekt előrehaladtával növekszik, különösen a befejező fázisban, amikor a tesztelés már a fő tevékenység.

A prototípusok életciklusához hozzá tartozik azok megsemmisítése is. Ezek az eszközök, bár a fejlesztés fontos részei, nincsenek megfelelő hatósági engedélyeik a használathoz, illetve, mivel nem végleges a termék, nem egyezik a végső kinézettel és specifikációval, amit a nagyobb gyártók a márka védelme érdekében nem engednek végfelhasználásra. A prototípusokat, megelőzendő annak elvesztését, illetéktelen kezekbe kerülését, illetve azt, hogy ne legyen a szükséges helyeken elérhető prototípus, nagyon szigorúan nyilvántartják, és annak elvesztéséért jelentős anyagi felelősségvállalással tartoznak az azokat használók. Az anyagi felelősségvállalás nem vonatkozik a meghibásodásra és fizikai sérülésre, hiszen nem ezen események megakadályozása a célunk.

Titoktartás és biztonság

A hardverfejlesztés általában termékfejlesztést is jelent, aminek különböző szempontok szerint határozzák meg azt az időpontját, amikor azt a nagyközönség tudomására hozzák, illetve hogy hogyan „csepegtetik” az információt. Egy szoftvertermék esetén nem feltétlenül beszélnénk erről a dologról, hiszen nem maga a tesztkörnyezet az, ami nem nyilvános, hanem a termék. Viszont itt maga a prototípus, amin tesztelünk látható, és nagyon sok olyan információt hordozhat, amit nem kívánunk megosztani egy bizonyos időpontig.

Vannak olyan esetek, amikor a prototípusnak el kell hagynia a fejlesztőközpont területét. Ezek lehetnek olyan tesztek, amit külső tesztközpontban végzünk, vagy a fejlesztésnek több központja van, és ezek között kell szállítani, esetleg az új prototípus generációt kell szállítani a gyártási hely és a fejlesztőközpontok között. Ilyen esetben gondoskodni kell a prototípusok megfelelő csomagolásáról és szállítási módjáról, úgy, hogy illetéktelenek ne férjenek hozzá. Ez általában nem csak annyiban merül ki, hogy veszünk egy dobozt és lezárjuk. Még az sem feltétlenül elég, hogy egy jól zárható megerősített tárolóba csomagoljuk, hanem a szállítás minden részletét meg kell tervezni.

Néhány eset példaként, amire fel kell készülni. Légi szállításnál biztonsági és vámvizsgálatra is készülni kell. Feladott poggyásznál hasznos lehet egyeztetni a légitársasággal előre, hogy milyen csomagot szállítunk, az miért kényes, továbbá kidolgozni egy olyan eljárást, ami mindkét félnek megfelelő. A kézipoggyászos szállítás esetén, az adott prototípus át fog menni a biztonsági átvilágításon, és mivel elektronikát tartalmaz, elképzelhetően gyanússá válik (azaz megkérnek, hogy vegyük ki a prototípust, mindezt nagy nyilvánosság előtt, amit épp el szeretnénk kerülni). Ilyen esetekre érdemes megfelelő módon készülni, és megfelelő módon reagálni. Lehet kérni ilyenkor, hogy a csomagot egy zárt helyen

nyissuk fel. Hogy elkerüljük a szóbeli kommunikációt, ilyen esetre jó, ha egy nyomatott oldalon elmagyarázzuk, hogy mit szeretnénk, és miért kérjük mindezt.

Csomagküldés esetén figyelni kell a nyomon követhetőségre. A legtöbb egyszerű postai szolgáltatás emiatt kiesik azon kockázat miatt, hogy a csomag menet közben elveszik. Az ilyen esetekben is a csomagolásnak nem csak a terméket kell megvédeni a fizikai behatásoktól, hanem gondoskodni kell a véletlen vagy szándékos csomagkinyitások ellen. A vámeljárás egy másik akadály lehet egy ilyen esetben.

Szárazföldi szállítás esetén is figyelni kell arra, hogy illetéktelenek véletlenül se láthassák meg az adott eszközöket még olyan váratlan szituációkban sem, mint egy esetleges baleset, vagy a tároló rekesz nyitása.

A másik eset, amikor kikerül a termék nyilvános helyre, amikor az valós környezetben teszteljük (field test). Általában az a megszokott eljárás, hogy a termék ilyen jellegű tesztjét nem kezdjük el, amíg maga a termék nincs bejelentve. Ekkor általában erős álcázással kerülnek ki az első prototípusok, amelyek kamuflázsa segíti elrejteni a termék valódi kinézetét és kulcstulajdonságait. Ahogy egyre több információ jelenik meg publikusan a termékről, azt egyre kevésbé kell álcázni, amikorra pedig minden adat nyilvánossá válik, általában az álcázás teljesen elmarad.

Vannak esetek, amikor egyes tulajdonságokat már akkor az utcán kell tesztelni, amikor azt a fenti szabályok keretei között még nem lehetne megtenni. Ilyen esetekben általában egy már rég a piacon lévő terméknek álcázzák az újat, hogy ne legyen feltűnő, így a legfontosabb tulajdonságok korai tesztelése megoldható.

Komponensek és meghajtók fejlesztése

Bár általában komplett rendszereket fejlesztünk, előfordul, hogy a végtermék egy másik eszközbe szabványosan beépülő komponens, a legegyszerűbb példa erre egy videokártya, ami egy szabványos PCI Express csatolót kap, de megemlíthetünk egy PXI ipari szabványú kártyát is.

Amikor egy ilyen eszközt fejlesztünk, akkor a termék egyrészt magából az eszközből, illetve egy meghajtó program(ok)ból (driver) áll. A tesztelés nehézsége ilyenkor abból áll, hogy nagyon sokféle rendszerhez illeszthető egy ilyen eszköz, amihez megfelelő tesztállomásokat kell létrehozni. Nyílt szabványok esetén minden gyártó elviekben minden kompatibilis eszközén és szoftveres környezetén tesztelni szinte lehetetlen. Egy-egy komponens gyártó általában nem csak egyféle komponenst gyárt, hanem a termékskálából szinte tetszőleges komplett rendszer összeállítható. Az egyik prioritás, hogy a saját termékekkel való teljes kompatibilitást ellenőrizzük, mert az itteni összeférhetetlenség egyértelműen a legnagyobb presztízsveszteséget okozza, de ez nem elégséges, külső gyártók termékeivel is tesztelni kell.

Egy jó teszthalmaz összerakása az egyik fontos kérdés a tesztelésben. Előbb a lényeges jellemzők meghatározása szükséges (amiből egy ilyen esetben elég sok van). A videokártyánál maradva nézzünk egy példát, hogyan állítható össze a megfelelő tesztkörnyezetek. Mivel ez most egy példa, egy kicsit elnagyolt, egy valós eset sokkal több paramétert vizsgál, inkább csak bemutatásra szolgál, hogy milyen problémák léphetnek fel menet közben. A termék (videokártya) egy komplett számítógépbe kerül bele, amiben van:

- Alaplap
- Memória
- CPU
- Háttértárak
- Hálózati kapcsolat
- Operációs rendszer
- ….
- Egyéb perifériák

Ezekből először is eltávolítjuk azokat, amelyek kevés befolyással rendelkeznek a tesztelésre, hogy csökkentsük a lehetséges kombinációk számát, ezért elhagyjuk a

- memóriát (feltételezzük, hogy elégséges a rendszer futtatásához, és nem szokott inkompatibilitási problémákat
- háttértárakat (nem befolyásolják jelentősen a grafikai teljesítményt és működést)
- hálózati kapcsolatot (csekély a befolyása a grafikai rendszerre)
- hangkártyát (bár ugyanaz a multimédiás motor hajtja meg, nagyon különböző csatornákon és módon működnek, az egymásra hatás esélye minimális, a kockázatot úgy csökkentjük, hogy véletlenszerűen különböző hangrendszereket telepítünk)
- egyéb perifériákat (a grafikai motorral való kapcsolatuk minimális)

Megmaradt nekünk az alaplap, a CPU és az operációs rendszer hármasa, mint tesztelendő terület, amit további jellemzőkre tudunk bontani.

- Alaplap
    - gyártó
    - chipset
    - formátum (ATX, Mini-ATX stb.)
    - SLI/Crossfire támogatás (a kártyától függ melyik, vagy egyáltalán szükséges-e)
    - PCI Express támogatott verziószáma
- Processzor
    - Intel

    - AMD
- Operációs rendszer
    - Windows
        - nyelv
        - verzió
        - 32/64 bit
    - Linux
    - OS X

Mondhatnánk, hogy akkor szedjük össze az összes támogatott opciót, és keressünk egy ideális konfigurációt, és meg is lennénk. De ez nem feltétlenül ilyen egyszerű, ugyanis az opciók nem függetlenek egymástól. Ilyen például a chipset és a CPU kérdése, ezeknek az eszközöknek kompatibilisnek kell lennie.

A rendszer chipsetje alapvetően meghatározó tényező, meghatározza a PCI Express verziót, illetve a processzorok körét is behatárolja. A PCI Express verziószámát ezért el is hagyhatjuk. Az egyszerűsítés miatt tételezzük fel, hogy a kártyánk nem támogatja az SLI vagy Crossfire szabványt sem, így ez a szempont sem szükséges.

A chipset alapvetően meghatározza, hogy AMD vagy Intel CPU-t tudunk használni, ezért itt vegyük kétfelé a konfigurációkat, és eszerint két készletet generálunk, az egyiket az AMD kompatibilis a másikat az Intel kompatibilis chipsetekből. Így az alaplapnál marad a chipset, illetve gyártó, mint jellemző paraméter a rendszerünkre. Maradjunk az Inteles oldalon, hogy befejezzük a példát. Az Intel processzorválasztéka nagyszámú elemet tartalmaz, viszont nem mindegyik kompatibilis mindegyik chipsettel, ami viszont jellemző, hogy generációnként igen, illetve a CPU-k különböző osztályokba sorolhatóak, és azok általában csak sebességben térnek el, tehát minden generációban az alábbi osztályok vannak (ez egy példa, lehetne tovább menni, ha nagyon pontosak akarunk lenni):

- Celeron
- Pentium
- Core i3
- Core i5
- Core i7
- Xeon

Hogy megszüntessük az egzakt függőséget a chipset és a CPU között, az osztályokat használjuk helyette. Ezután már csak az operációs rendszer marad. A Windows különböző változatai a célközönségünk, ezért erre sok variációval készülünk tesztelni. A Linuxos közösség sokkal kisebb, viszont nagyon sok disztribúció adott a felhasználótábor méretéhez képest, ezért abból kiválasztjuk a három leggyakrabban használt disztribúciót néhány alapkonfigurációval, és nem a páronkénti tesztelést alkalmazzuk.

Amennyiben az adott eszköz belekerül egy vagy több Apple termékbe, akkor az OS X is célplatform lehet. De ez csak nagyon kevés komponensre igaz, és azok is viszonylag jól meghatározott komponensekkel épülnek egybe, így az konkrét rendszertesztet igényel, ezért most azt is kihagyjuk.

Amikor mindezzel megvagyunk, az alábbi paramétereket kapjuk:

- Alaplap gyártó
- Chipset
- Alaplapformátum
- Processzorgeneráció
- Windows verzió
- Windows nyelv
- 32/64 bites Windows változat (lehet része a Windows verziónak is)

Ezekre a paraméterekre meghatározzuk a lehetséges opciókat, és meghatározunk egy páronkénti tesztelés táblát, amivel egy elég jó közelítéssel optimális tesztkörnyezeti együttest kapunk Windows-ra Intel környezetben.

Az ilyen tesztkörnyezetek létrehozása ugyan nagy befektetés, viszont használható több termék fejlesztésére is egyszerre egy megfelelő időmenedzsment bevezetésével.

Érdemes megbízható partnereket bevonni az alfa és a béta tesztelésbe, a megfelelő titoktartási szerződések megkötésével. A videokártyás példánál maradva, a játékgyártók természetes partnereink lehetnek, amivel le tudják mérni az adott kártya és meghajtó program hatékonyságát, és esetleges hibákat találhatnak, amely a saját termékeik használatával jönnek elő. Ki tudják próbálni az új funkciókat, és véleményt tudnak róla mondani, amelyek alapján még javítható a teljesítmény és funkcionalitás a megjelenés előtt.

Operációs rendszerek fejlesztése

Az operációs rendszer fejlesztések az egyik legösszetettebb fejlesztési feladatnak számítanak, és több különböző fejlesztési feladatra bonthatóak, ahol minden feladatnak más specialitásai vannak. Ha jobban felosztjuk, az operációs rendszer fő részei:

- **Rendszermag (Kernel),** az a központi mag, ami rendszerhardver erőforrásainak kezeléséért felelős, és kiosztja azokat a megfelelő szoftvereknek. Általában egységes felületet biztosít a

hardverek kezelésére, hogy lehetővé tegyen egy az aktuális hardver részleteitől független programozást

- **Meghajtók (Driver),** amik felelősek azért, hogy az egységes felületet biztosítsák a rendszermag számára az adott hardver erőforrás optimális használatára

- **Rendszerkönyvtárak,** amely segítségével a felhasználói interfészt és egyéb más szolgáltatásokat el lehet érni (néhány példa: MFC, COM, .NET, DirectX, X11, Unix Shell, stb.)

- **Felhasználói felület,** ami a felhasználók által is értelmezhető kommunikációt biztosítják magával az operációs rendszerrel

- **Alkalmazások,** bár nem része az operációs rendszernek, ha szigorúan definíció szerint nézzük, egy operációs rendszer kiadásának jelentős részét teszik ki a hozzá csomagolt különböző alkalmazások és eszközök. Ezek egy része olyan, ami nélkül az operációs rendszer szinte használhatatlan lenne (például a Control Panel felület nélkül nehéz lenne a rendszer adminisztrációja és beállítása), valamint olyanok is, amik egyértelműen olyan alkalmazások, amelyek hozzácsomagolt egyéb szolgáltatások (média lejátszók, web böngészők, szövegszerkesztők stb.).

Alapvetően mindegyik réteget másképpen teszteljük, más követelményeknek kell megfelelni a tesztkörnyezetnek, legalábbis az egységtesztek időszakában. Egy idő után, amikor a rendszer integrálása folyik, és immár teljes rendszert tesztelünk, akkor is más stratégiára van szükség.

Az operációs rendszereket különböző üzleti stratégia mentén fejlesztik.

- **Zárt rendszer, csak meghatározott eszközökre szánt operációs rendszer** – fejlesztés és tesztelés szempontjából a legegyszerűbb eset, a tesztelési rendszerek a célplatformokból állnak. Ebbe a csoportba az tartoznak például az Apple operációs rendszerei, az OS X vagy az iOS.

- **Nyílt rendszer** – Amikor a célhardverrel szemben csak követelmények vannak lefektetve, de azokhoz a biztosított és egységesített csatolófelületen keresztül a meghajtó programok megírásával szabadon fejleszthető hardver. Ilyen például a Windows vagy az Android (bár az Androidot nem telepíti a végfelhasználó, hanem a gyártótól kapja a frissítést).

Az operációs rendszer gyártója szállítja a rendszermagot. A többi komponens esetén vagy maga a gyártó fejleszti őket, vagy kooperációban más gyártókkal. Jellemző módon a meghajtó programok esetén a legnagyobb a kooperáció a külső gyártókkal, akik részt vesznek a saját termékeikhez tartozó meghajtók fejlesztésében, hogy azok már a hivatalos kiadásban is benne legyenek, és ne utólag kelljen telepíteni a felhasználóknak. Az operációs rendszer gyártója a rendszerintegrátor, de a tesztelésből a partnerek is jelentős részt vállalnak, akik lehetnek hardverkomponens gyártók, vagy márkás számítógép gyártók, akik a későbbiekben telepítik az operációs rendszert.

Egy általános operációs rendszer támogatott konfigurációinak száma olyan magas, hogy a teljes tesztelés nem lehetséges. A partnereket bevonva, illetve megfelelően nagy variációt biztosítva elérhető egy viszonylag jó minőség, de a jó alfa- és béta tesztelői háttér elengedhetetlen ebben az esetben.

Tesztrendszerekkel szemben támasztott követelmények

Végignéztünk már több technikai, technológiai környezetet, és azt láttuk, hogy a tesztkörnyezetek nagyon sokfélék lehetnek, illetve az is látszik, hogy nem lehet lefedni minden eshetőséget tesztkörnyezetekkel, hanem megfelelő kompromisszumokat kell kötni. A kompromisszumok pedig a követelmények között lehetségesek. Egyeseket mindenképpen meg kell valósítanunk, a másikon gyengítünk, a harmadikat esetleg elvetjük.

A követelményeknek több forrása van. Vannak egyrészt funkcionális és nem funkcionális követelmények, amelyek a projekt végtermékét írják le, illetve követelményeket támaszt maga a projekt, annak folyamatai illetve a minőségi célok, amelyek megint lehetnek a termék nem funkcionális követelményei között, de támaszthat ilyet a környezet is külső auditok, törvényi szabályozások esetleg belső előírások alapján.

Minden követelményre nem lehet kitérni, hiszen az a fejlesztési célok alapján nagyon változni tud, de van néhány olyan általános és implicit (nem megfogalmazott, természetesnek vett) követelmény, amiről mindenképpen érdemes beszélni. Az implicit követelmények veszélye, hogy nem mindenki ugyanazokat a követelményeket veszi természetesnek, ezért könnyen félreértések keletkezhetnek. Ami az egyik embernek megfelelő, az a másiknak esetleg nem, mert bizonyos követelmények, amit természetesnek gondolt, nem jelennek meg az adott tesztkörnyezetekben.

Tesztkörnyezetekkel szemben támasztott általános követelmények

Azonosság

Amikor tesztelünk, egy bizonyos modell szerint ellenőrizzük a tesztalanyát, hogy az megfelel az elvárt követelményeknek. A tesztkörnyezetnek, ezen modell szempontjából, azonosnak kell lennie avval a termékkel, ami a tesztalanya.

A fenti mondat szerint, tehát nem a teljes azonosságot követeljük meg, hanem a modell szerinti azonosságot.

Ez az azonosság néhány esetben valóban szinte teljes, mert a teljes rendszert vizsgáljuk, és ahhoz olyan körülmények szükségesek, mint ahogy a szerver-kliens architektúránál is láttuk, hogy szükséges egy, a végfelhasználóival szinte teljesen megegyező környezet, más esetekben viszont a tesztkörnyezet akár jelentősen el is térhet az adott végterméktől. Minél alacsonyabb a teszt szintje, és minél inkább a komponenseket vizsgáljuk, az azonossági kritériumok annál gyengébbek.

Biztonság

A tesztkörnyezetekkel szemben többféle biztonsági követelményt is támasztunk.

Személyes biztonság, amire talán nem is mindig gondolunk, és egy tisztán szoftveres projekt esetén az esetek többségében fel sem merül, mert annyi láthatatlan biztonsági követelménnyel vesszük már magunkat körbe, hogy azok az eszközök, amelyek kereskedelmi forgalomban kaphatóak, rendeltetésszerű használat mellett nagyon kis valószínűséggel okoznak balesetet. Ez a nagyfokú biztonság azonban nagyon hosszú, minőségbiztosítási és biztonsági fejlesztés és szemlélet eredménye.

Amikor azonban nem csak szoftvert fejlesztünk, illetve tesztelünk, kapcsolatba kerülhetünk olyan eszközökkel, amelyek még nem teljesen teszteltek a biztonsági szabványok szerint. Az ilyen eszközök használatánál a kockázatokat elemezni kell, és a kockázatok alapján megfelelő munkavédelmi tréningekre szükség lehet (például érintésvédelem, vagy mozgó alkatrészek okozta sérülésveszély esetén).

Anyagi biztonság. Jogos elvárás, hogy a tesztelés ne okozzon járulékos anyagi veszteséget. Ezt mind a tesztmódszerekben, mind a környezetekben lecsapódik.

Ebből a jogos elvárásból következik sok olyan követelmény, amit szinte természetesnek vesznek a tesztelési szakemberek, például, hogy a végfelhasználói környezetben nem tesztelünk, mert nem tudjuk garantálni, hogy nem lesznek járulékos, előre nem látott költségei az ilyen tesztelésnek. Amennyiben ez az alapelv nem tudatos egy szervezeten belül, ott előfordul olyan, ahol a végfelhasználói környezetbe tesztelés nélkül kerül át új funkcionalitás, mondván „alacsony a kockázat”, amikor pedig megtörténik a baj, a legritkább esetben gondolnak arra, hogy a nem megfelelő kockázatkezelés és folyamatok a mélyben húzódó kiváltó ok.

Tesztelés közben máshol is előfordulnak anyagi veszteség jellegű kockázatok. Az összes olyan teszteset, kapcsolat, amely külső pénzügyi transzakciót generálhat, potenciális veszélyforrás. Nagyon fontos meggyőződni arról, hogy nincs olyan kapcsolat a tesztkörnyezetünkben, amely éles pénzügyi tranzakciót generálhat.

Az olyan béta tesztek, amikor valós pénzügyi tranzakció már lehetséges, különös figyelmet igényelnek, és csak nagyon erősen tesztelt funkcionalitásnál jelent elfogadható kockázati szintet. Fel kell készülni a reklamációk kezelésére, ha valami mégsem az elvártak szerint működik.

Az anyagi biztonság nem jelenti azt, hogy nem teszünk tönkre semmit (végül is tesztelünk, lehet az a cél, hogy tönkretegyük, hibát találjunk). A tervezett felhasználással tönkretett teszteszköz még mindig kisebb veszteség, mint a végfelhasználás során bekövetkező hasonló káresemény. Ugyanakkor figyelni kell arra, hogy amikor tesztelünk, a módszereknek figyelembe kell vennie a kockázati szinteket. Egy extrém példánál maradva, egy atomerőmű vezérlőrendszerét nem tesztelhetjük az elejétől fogva úgy, hogy rákötjük a reaktort, mert egy kezdeti fázisnál a magas hibavalószínűségek mellett ez túl kockázatos lenne. Természetesen szükséges olyan teszt is, amikor azt rákötjük, de megfelelően tesztelt állapotban, és megfelelő óvintézkedésekkel.

Egyes tesztesetek esetén előre tudhatjuk, hogy a tesztelésnek az lesz az eredménye, hogy valami végérvényesen tönkremegy (a töréstesztnem azért van, hogy az eszköz túlélje), de az ilyen eseteknek konkrét célja van, és nincs más mód, hogy felmérjük az eredményt, vagy egyszerűen a más módszer magasabb költségű. Ugyanakkor a járulékos költség minimalizálása lehet és kell is, hogy cél legyen.

Adatbiztonság, ami egy speciális formája az anyagi biztonságnak, ha az adatra, mint értékre tekintünk. A tesztrendszerekben lévő adatvagyonra ugyanolyan módon kell vigyázni, mint az úgymond „rendes” adatokra, hiszen azok nagyrészt akár meg is egyeznek azokkal.

Magára a fejlesztésre is tekinthetünk úgy, mint értékes adatok összességére, amit nem feltétlenül szeretnénk idő előtt megosztani. A tesztkörnyezetekből jól kikövetkeztethető a tervezett végtermék több fontos adata, ezért azt néha nagyobb védelem övezi, mint a már múltat jelentő valós adatokat. Egy híres példa erre az első iPhone fejlesztése, ami az Apple cég nagyon gyors felemelkedésének kulcsa lett. Annak a fejlesztésnek a titkait, a teszt- és fejlesztői környezetét az egyik legsikeresebb módon sikerült megvédeni, amiben része volt maguknak fejlesztőknek, és az ő titoktartásuknak, illetve a nagyon szigorú biztonsági intézkedéseknek. Ha nem sikerül ennyire jól megtartani ezt a titkot, nehéz lett volna ennyire átütő piaci sikert elérni.

Elérhetőség

A legjobb tesztkörnyezet sem ér semmit, ha nem vagy nem akkor elérhető, amikor szükség van rá.

Ahhoz, hogy megfelelő elérhetőséget biztosítsunk, több folyamatnak is megfelelően kell működnie.

Amennyiben egy új szolgáltatásról beszélünk, megfelelő tervezés szükséges, hogy az igényeknek megfelelő környezetet lehessen építeni a megfelelő időre. Egy új környezet felépítése egyrészt technikai kérdés és kihívás. Meg kell tervezni, hogy milyen eszközök kellenek, azokat milyen szoftverkörnyezettel kell ellátni, hogyan kell beállítani, illetve integrálni a meglévő környezetbe. A másik része a megoldásnak egy pénzügyi és logisztikai probléma, hogy a megfelelő beszerzések időben és gördülékenyen történjenek. A harmadik rész a technikai felkészítés a tesztelésre, amelyek lehetnek egyedi fejlesztések, adatmozgatás stb.

Egy környezet előkészítése lehet nagyon egyszerű, illetve nagyon bonyolult és hosszadalmas. Ki kell dolgozni a megfelelő eljárásokat és folyamatokat, hogy ezek gördülékenyek legyenek, különösen egy komplex rendszernél, azt ugyanis általában rendszeres időközönként végre kell hajtani. A jól definiált (és dokumentált) folyamatok segítenek abban, hogy a tervezett időpontra az adott környezet valóban bevethető legyen. Egy bonyolult rendszer esetén számítani kell arra, hogy az első pár alkalommal ez nem sikerül tökéletesen, és lesznek kisebb nagyobb hibák, csúszások. A jó folyamat egy tanulási folyamat eredménye. Az a szerencsés, ha a hibákra válaszként a folyamatainkat megfelelően javítjuk, hogy a legközelebbi alkalommal ugyanazt a hibát már ne kövessük el. A tapasztalatok folyamatos beépítésével elérhető, hogy a folyamat egyre gördülékenyebb és megbízhatóbb legyen.

Nagyon fontos, hogy a folyamatok kövessék a szervezet igényeit. Sokszor előfordul, hogy egy szervezet a növekedéssel egyre növeli a tesztkörnyezetek számát. Azt tapasztalják, hogy a tesztkörnyezeteket ugyan csak az idő 30-40%-ban használják tesztelésre, nagyon nagy az erőforrás és a karbantartás igénye, mégis az alacsony elérhetőségre panaszkodnak a tesztelők. A növekedést és a nagyobb komplexitást, az addig hatékonyan működő folyamatok nem mindig tudják jól kezelni, mert több az integráció, a komplexebb projekteket már nem lehet letesztelni ugyanannyi idő alatt, több projekt fut egyszerre, amelyek konfliktusba kerülnek egymással. Ha a folyamataink nem követik ezeket a változásokat, előbb-utóbb több helyen is problémákat tapasztalhatunk, amelynek az egyik jele lehet, hogy a környezetek elérhetősége nem optimális a feladatokhoz. Egyes esetekben elég egy picit módosítani, vagy javítani a folyamatokon, más esetekben viszont gyökeres változtatásra is szükség lehet. Nagyobb változtatások általában további változtatásokat indukálnak más folyamatokban is, vagy a máshol végzett módosítások

igényelhetnek változtatást a környezetek karbantartási folyamatán. A fejlesztési folyamatok erősen integrált rendszert alkotnak és hatnak egymásra.

Folyamattámogatás

Már az előző fejezetben is érintettük, hogy a környezetek akkor vannak jól felépítve, ha harmonizálnak a fejlesztési folyamatokkal (amelyek jó esetben szintén harmonizálnak a fejlesztési igényekkel). A tesztkörnyezetek és a fejlesztési folyamatok folyamatos kölcsönhatásban vannak, és az a szerencsés, ha valóban hathatnak egymásra, és nincs kiemelt szerepkör. Egy a fejlesztési folyamatra optimalizált tesztkörnyezet, bár magát a folyamatot lehet, hogy optimálisan kiszolgálja, és a fejlesztés nagyon hatékony, ha emiatt egy nagyon drága környezeti struktúrát és sok erőforrást lekötő szolgáltató csapatot kell fenntartanunk, az nem szerencsés. Persze az sem jó, ha a jelenlegi környezeti képességeinkhez és folyamatainkhoz kell igazítani minden tevékenységet, ha az valójában az esetek többségében nem hatékony.

A kezdetekben, egy kisebb szervezetnél az a szerencsés, ha a projekt illetve a szolgáltatás definiálja a megfelelő folyamatokat, ahol figyelembe veszik a fejlesztési igényeket, és optimalizálják a technológiai realitásokhoz, hogy mindkét oldalon ideálisnak mondható kompromisszum jöjjön létre. Amikor egy szervezet elkezd növekedni, egy adott idő után szükségessé válhat bizonyos folyamatok szabályozása. Ekkorra általában összegyűlik a megfelelő tapasztalat, hogy melyek azok a módszerek, amelyek egy adott szervezetnél jól működnek, és melyek azok a gyakorlatok, amelyeket jobb elkerülni a negatív tapasztalatok miatt.

Az első lépésként általában csak irányelveket határoznak meg, és ahogy nő a szervezet érettsége, egyre jobban definiálják a folyamataikat, amiből a projektek a saját folyamataikat származtatják. Egy jól definiált érett szervezet összességében többször hatékonyabb lehet[66].

Függetlenség

Egy projektnek, szervezetnek, hogy céljait elérje, szüksége van bizonyos fokú függetlenségre (olyan paraméterekre, tevékenységekre, amelyeket szabadon megválaszthat). Mint minden szabadságnak, ennek is ára van. Ahhoz, hogy valamiben nagyfokú szabadságot, és függetlenséget lehessen elérni, más helyeken szükségszerű, hogy kötöttségeket vállaljunk annak érdekében, hogy a saját és mások hatékonyságát fenn tudjuk tartani.

Meg kell tudni határozni, hogy mi az, amiben a legnagyobb szabadságot akarjuk élvezni, melyek azok a döntések, ahol a legkevésbé tudjuk elfogadni a kompromisszumokat, és melyek azok a paraméterek, ahol könnyedén alkalmazkodni tudunk. Így lehet feloldani a *függetlenség* és a folyamatok szabályozottságával biztosított *hatékonyság* közötti ellentétet.

A folyamatainknak, és azon belül a tesztkörnyezeteink felépítésének ott kell szabadságot biztosítani, ahol az számunkra a legfontosabb, és ott lehet megszorításokat alkalmazni, ahol nekünk ez kevésbé jelent problémát. Mindezt leírni sokkal egyszerűbb, mint megvalósítani, hiszen egy projekten belül rengeteg érdek és nézőpont találkozik, és ami az egyik embernek fontos, az a másiknak esetleg kevésbé az.

Paraméterek, ahol fontos lehet a szabadság egy projekten belül (nem egyszerre, mert ezek egymást kizáró tényezők lehetnek):

- Technológia
- Határidők, menetrend
- Minőségi kritériumok
- Projektmenedzsment módszertan
- Teszt stratégia
- Projekt hatóköre
- Tesztkörnyezetek elérhetősége
- Tesztkörnyezetek technikai paraméterei
- Projekt költségvetése
- Többi projekt állapota és menetrendje

Hatékonyság

Az erőforrások megfelelő felhasználása egy szervezetnél szükségszerű a jobb működés és a jobb eredmények elérése szempontjából. A jó szemléletmód az, ha a teljes szervezet hatékonysága felől közelítjük meg a kérdést. A tesztkörnyezetek fenntartása egy befektetés, amely más területeken a kockázatok, költségek csökkenésével vagy az eredményesség növelésével járnak.

Egy szerver-kliens architektúrában egy réteg megszüntetése, ha csak a környezetek költségét nézzük, jelentős megtakarítást eredményezhet. Viszont jelentősen ronthatja a fejlesztések és a tesztelés hatékonyságát, ami azt eredményezheti, hogy ugyanannyi ember kevesebbet tud átadni, megtermelni ugyanannyi idő alatt, vagy az átadott termék minősége csökken, aminek a járulékos költségei még kevésbé láthatóak az adott szervezeten belül. A minőségcsökkenés egy ilyen esetben ráadásul nem feltétlenül látványos, mert a korábbi fejlesztések még rendelkeznek egy beépült minőséggel, és a leépülés hosszú távon lassan történik meg, amikor már azt esetleg senki nem köti össze a környezeti struktúra megváltoztatásával.

Ugyanakkor az sem feltétlenül igaz, hogy a tesztkörnyezetekbe mindig jó befektetni. Bár néha ez tűnik a legegyszerűbbnek, néha egyszerű folyamatváltoztatással megoldható problémát (például egy hatékonyabb konfigurációmenedzsment igényt) kezelünk többlet környezetek használatával.

A megfelelő hatékonyságot globális szemléletmóddal, megfelelő folyamatokkal és folyamatmenedzsmenttel, illetve a folyamatok hatékonyságának mérésével lehet elérni.

Megbízhatóság

A fejlesztési folyamat erősen támaszkodik a tesztkörnyezetekre, azok megbízhatósága elsőszámú hatékonysági kérdés egy projekt számára.

A megbízhatóság nem csak technikai paramétereket jelent, hogy például az adott környezet elérhető volt a megfelelően vállalt százalékban, hanem rengeteg más egyéb paramétert is, amelyek egy része akár

egy kicsit szubjektív is lehet. Ha megbízható környezeteket szeretnénk kialakítani, az alábbi estekre való odafigyelés jelentős segítséget nyújthat:

Felügyelet. A tesztkörnyezetek felügyelete, és a hibák gyors azonosítása nagyon fontos a hatékony munkavégzéshez. Amennyiben a problémákra nem reagálunk megfelelő időn belül, az jelentékeny többletmunkához vagy akár a tesztelés felfüggesztéséhez is vezethet. Természetesen a végfelhasználóké az első számú prioritás, viszont ha a fejlesztésre túl kevés erőforrás marad, az jelentős hatékonyságproblémákat okoz.

Arra kell számítanunk és készülnünk, hogy egy tesztkörnyezetben a problémák általában sokkal gyakrabban fordulnak elő, mint amit a végfelhasználóknál tapasztalunk.

Karbantartás. A tesztkörnyezetek folyamatos karbantartást igényelnek, sőt általában több törődést igényelnek, mint egy normál végfelhasználói környezet.

A tesztkörnyezetekben alapvetően olyan termékeket használunk, amelyek nagy valószínűséggel tartalmaznak hibákat. Egy ilyen termék használata nem segíti a megbízhatóság fenntartását. A konfiguráció a legtöbb ilyen eszközön nem stabil, folyamatosan változnak a telepített komponensek, így az ilyen telepítések során keletkezett apróbb hibák könnyen összeadódhatnak. Egyes esetekben a rendszerben tárolt adatoknak is jelentősége lehet, amiket a tesztelés, a nem feltétlenül üzleti jellegű felhasználás miatt, folyamatosan erodál.

A fenti okok miatt, a környezeteinket általában rendszeresen frissítjük. Ez a frissítési folyamat jó esetben hatékony, megbízható és az igényeket kielégítő módon megy végbe.

A kommunikáció az egyik legfontosabb szubjektív szempont. Az érintetteknek tudniuk kell arról, amikor a tesztkörnyezet nem elérhető, hogy a tevékenységeiket megfelelően tudják tervezni. A nem várt problémák nagyon rontják a megbízhatóság szubjektív megítélését.

A megfelelő kommunikációt egy jó kommunikációs terv készítésével és betartásával, illetve a tapasztalatok alapján történő folyamatos fejlesztéssel lehet a leghatékonyabban elérni.

Egy jó kommunikációs terv figyelembe veszi, hogy milyen eshetőségek vannak, amikor kommunikáció szükséges (például tervezett karbantartás, váratlan esemény stb.), kik az adott esemény kommunikációs célpontjai, melyik a leghatékonyabb kommunikációs csatorna, melyik a megfelelő időpont a kommunikációra, illetve egy jó kommunikációs tervben sablonok is szerepelnek az adott szituációkra.

Minőségbiztosítás. A tesztelés, bár szerepe megkérdőjelezhetetlen, nem az egyetlen, és messze nem a leghatékonyabb módja a minőségbiztosításnak. Egyes környezeteknek adott minőségi elvárásoknak kell megfelelniük, amit, ha a fejlesztés menetében nincsenek meg a megfelelő ellenőrzési pontok, esetlegesen nem tudunk teljesíteni, mert nem a megfelelő érettségű alkalmazást használjuk, és annak problémáit a környezet problémáiként azonosítjuk.

Tesztelési technológiák követelményei

Követelmények adódhatnak abból is, hogy milyen tesztelést hajtunk végre, ahhoz milyen technológiákat használunk. A két leggyakoribb technológia, ami a tesztkörnyezetekkel szemben külön követelményeket támaszt az a tesztautomatizálás és a teljesítményteszt.

Tesztautomatizálás követelményei

Általában két tesztautomatizálási technika terjedt el.

Az egyiket egységtesztekre használjuk, és általában kód alapú, és a forráskód vagy a technikai felületeken keresztül teszteli a részegységeket és/vagy a terméket. A másik technológia a felhasználói felületet használja, és a felhasználók által megszokott módon, manuális teszt szerűen használja az alkalmazást. A két technikához tartozó követelmények jelentősen különböznek.

A kód alapú tesztelő eszközökhöz az alábbi követelményeket kell kielégíteni a tesztelő eszközöknek:

- **Fejlesztői környezet,** ahol magukat a teszteket össze lehet állítani, amennyiben szükséges fordítani és utána a megfelelő helyen tárolni.
- **Tesztelési környezet,** az elkészült teszteket magukat is tesztelni kell. Ezt egyes fejlesztések külön környezettel oldják meg, máshol a fejlesztői környezet szolgál erre a célra is.
- **Futtató környezet,** megint nem feltétlenül szeparált környezet, bár a fejlesztés, tesztelés és futtatás egy környezetben néha konfliktusokhoz vezethet, a futtató környezetet érdemes szeparálni.
- **Elérhetőség,** a felhasznált eszköztől függően akár már fejlesztésnél szükséges lehet a tesztalanyhoz, jelen esetben a forráskódhoz és a tesztelendő felülethez való hozzáférés. A teszteket ugyanúgy tesztelni kell, ez az a fázis, ahol már egyértelműen a tesztalanyra is szükség van. A futtató környezetnek pedig teljes elérést kell biztosítani. Egyes esetekben olyan felületet is tesztelhetünk ilyen módszerekkel, amit a normál hálózati helyekről nem lehet elérni, tűzfal mögött van, ilyenkor a futtató állomást is be kell helyezni a tűzfal mögé, vagy számára megfelelő átjárást biztosítani.
- **Integráció,** az eszköz több egyéb rendszerrel integrált lehet, és annak futtató környezetét integrálni kell ezekkel az eszközökkel. Az egyik ilyen fontos integrációs lehetőség a tesztmenedzsment eszközökkel való integráció, ahonnan elindíthatjuk, ütemezhetjük a teszteket és visszakapjuk az eredményt. A másik fontos integrációs pont a szoftver konfigurációmenedzsment eszközzel való integráció. Egyes ilyen verziókezelő eszközök integrálva vannak olyan teszttámogató szoftverkörnyezettel (test harness), amelyek futtatják ezeket a teszteket. Egyes esetekben ez csak jelenti a hibát, amit tapasztal, szigorúbb rendszerek akár visszautasíthatják a változtatást, ha ezek a tesztek megbuknak.

A felhasználói felület alapú rendszerek másképp épülnek fel, ezért más jellegű követelményeket támasztanak:

- **Fejlesztői/tesztelői/futtató környezetek,** ugyanúgy szükségesek, mint az előző alkalommal.

- Az **elérhetőség** sokkal fontosabb az ilyen tesztelő rendszerek számára. A legtöbb ilyen eszköz tartalmaz felvétel és lejátszás (Record & playback) opciót, ami nagyon lerövidíti az ilyen tesztek fejlesztését, viszont ennek használatához hozzáférés szükséges a működő szoftverhez, ami nem mindig biztosítható, mert vagy a szoftver még fejlesztés alatt van, vagy egyes szervezetek tesztszerver elérhetősége korlátozott. Mivel a futtatáshoz felhasználói felületet használ, speciális elhelyezés nem szükséges, mivel a szabványos felületen éri el az alkalmazást.

- **Integráció** szempontjából a tesztmenedzsment eszközzel való integráció a lényeges, ritkán szánnak az eszköznek az előbb említett egységteszt feladatokat, inkább rendszerteszt a feladat.

- **Stabilitás,** mivel az eszköz a felhasználói felületen kommunikál a tesztelendő szoftverrel, szükséges, hogy az a tesztelő eszköz szempontjából stabil legyen, a sűrű változás nehezen alkalmazhatóvá teszi az eszközt. Bár az automatizált teszteket érdemes robosztusra készíteni, és felkészíteni a hibákra, váratlan eseményekre, egy instabil környezetben a tesztautomatizálás ezen módjának a felhasználhatósága meglehetősen kérdéses.

Teljesítménytesztelés követelményei

A teljesítménytesztelő eszközök olyan specializált tesztautomatizáló eszközök, amelyek elsődleges célja a nagymennyiségű transzakció generálása. Két filozófia szerint találunk eszközöket, az egyik a használatot a hálózati protokoll szerint szimulálja, a másik az alkalmazás aktuális futtatásával. Mindkét filozófiának vannak előnyei és hátrányai, egyaránt használt eszközök. Amit környezet szempontjából mindenképpen meg kell jegyeznünk, hogy a felhasználói felület alapú teljesítménytesztelés futtatása jelentősen több erőforrást igényel a futató környezetben. A teljesítményteszt által generált igények:

- **Környezetek,** hasonlóan a funkcionális tesztautomatizáláshoz itt is szükséges biztosítani a tesztek fejlesztését, tesztelését és futtatását.

- Az **elérhetőség**-re nagy szükség van. Az összes jelentős eszköz felvétel és lejátszás rendszerű, a fejlesztéshez szükség van az alkalmazásra. Ezeket a teszteket általában minél közelebbről szeretjük futtatni, hogy az infrastruktúra egyéb elemeit lehetőleg ne terhelje, viszont a futtató állomásokat kívülről kontrolláljuk, ezért lehet, hogy a tűzfalakat meg kell nyitni számára. A tesztelt objektumok teljesítményparamétereinek rögzítésére is szükség lehet a teszt alatt, amihez szintúgy tűzfal mögötti eszközökhöz kell hozzáférést biztosítani.

- **Stabilitás,** a protokoll alapú tesztek különösen érzékenyek lehetnek a változtatásra. A teszt biztonságos végrehajtása stabil funkcionalitást követel. Ha a funkciók nem megfelelően működnek, az eredmények nem valósak, a terhelés nem realisztikus.

- **Azonosság.** A teljesítménytesztek környezeteinek a mérési eredmények pontossága érdekében meg kell egyezni a célplatform konfigurációjával teljesítményjellemzőivel. Bár léteznek extrapolációs eljárások, amiből következtetnek egy másik környezeten mért eredményekből, a tapasztalat az, hogy csak speciális körülmények között működnek megfelelően.

A tesztkörnyezetek felépítésében gyakran alkalmazott technikák, technológiák

Ebben a fejezetben olyan technológiákról lesz szó, amelyek nagyban segítik a tesztkörnyezetek kiépítését. Ezek közül néhányat már korábban is érintettünk, itt inkább egy összefoglalóként fogunk róla beszélni.

Virtualizáció

A virtualizáció segítségével szimulált számítógépes környezeteket hozhatunk létre egy másik számítógépen. Egy ilyen virtuális számítógép legfőbb tulajdonságaiban, funkcionalitásában tökéletesen megfelel egy hagyományos számítógépnek. Hasznos és optimális kiegészítője a tesztelési környezetünknek

- Amikor gyorsan kell felépíteni környezeteket.
- Amikor nagyszámú szoftverkonfigurációt kell ellenőriznünk. Ilyen lehet például, amikor különböző operációs rendszer és böngészőpárosításokat tesztelünk.
- Egyes speciális esetekben rövidítheti a tesztelési időt. Egy ilyen példa: tegyük fel, hogy a telepített szoftver regisztrációját teszteljük. Amint a regisztrációt megtettük, akkor azt többször már nem tudjuk megtenni, csak akkor, ha a szoftvert eltávolítjuk és újratelepítjük. Ez meglehetősen időigényes folyamat, jobb lenne ezt mindig onnan kezdeni, hogy csak a regisztrációs eljárást kell lefolytatni. A megoldás, hogy csinálunk egy olyan virtuális számítógép képet, amin még nincs meg a regisztráció, majd ebből klónozunk egy új példányt minden alkalommal, amikor a regisztrációt teszteljük, így nem „romlik el” a tesztkörnyezetünk. Léteznek olyan virtuális környezetet biztosító eszközök is, amelyekkel pillanatfelvételt (snapshot) készíthetünk, akár többet is egy adott virtuális géphez, és bármelyik adott állapotot kiválaszthatjuk, hogy onnan folytassuk a munkát.

A virtuális számítógépet általában egy fájlban tároljuk, ami tartalmazza az adott számítógép konfigurációját, illetve a virtuális merevlemezek állapotát. Egy új virtuális gép létrehozható egy ilyen képből klónozással, illetve alapállapotból a virtuális gépen telepítéssel. A klónozás folyamán lehetőség van teljes klón létrehozására, amikor a két kép teljesen szétválik, illetve kapcsolt klón létrehozására, amikor csak a különbség mentődik a két kép között, és a klónozott használja az eredeti képet is. A kapcsolt klón hatékonnyá tudja tenni az adminisztrációt, mert nem kell minden egyes képre egyesével feltelepíteni az operációs rendszer frissítéseit. De figyelni kell rá ilyen esetben, hogy a fő képen végzett módosítások minden származtatott rendszeren megjelennek, aminek lehetnek nem kívánatos mellékhatásai.

Több szoftverplatform támogatja a virtuális számítógépek létrehozását, valamint az interneten is lehet találni már megépített virtuális gépeket, amelyeket lehet használni alapként, hogy ne kelljen a teljes telepítési folyamatot végigcsinálni. A nyilvánosan fellelhető képfájlokkal kapcsolatban viszont két dologra mindenképpen oda kell figyelni:

- Ez is egy számítógép, ami tartalmazhat vírust, rosszindulatú kódot, amit ráadásul be is vihetünk a védett, belső zónába. Biztonsági szempontból nagyon fontos, mint bármely külső forrásból származó futtatható állomány esetén, hogy fokozott elővigyázatossággal kezeljük őket

- Az interneten elérhető képfájl még nem jogosít a használatra. Figyeljünk az operációs rendszerek és egyéb esetlegesen telepített szoftverek licencszerződéseire, ahhoz, hogy az adott képfájlt futtassuk, rendelkezni kell az adott szoftverek felhasználói licencével az adott célra.

A virtuális gépek képeire létezik egy közös szabvány az Open Virtualization Format (OVF, nyitott virtualizációs formátum), amely segíti a különböző szoftverplatformok közötti átjárhatóságot. Amennyiben több virtuális platform között ezen a felületen keresztül biztosítjuk az átjárhatóságot, ellenőrizni kell, hogy melyek azok a kivételek, amelyeket egyik, vagy másik platform nem támogat, hogy elkerülhetőek legyenek a konfliktusok.

A licencfeltételeket ellenőrizni kell azon esetekben is, amikor saját képfájlokat hozunk létre. A feltételek több tényezőtől is függenek, érdemes megkeresni azt a személyt, aki felelős az adott szervezetnél a licencszerződésekért és a szoftverbeszerzésekért, amikor virtuális környezeteket kezdünk el létrehozni.

Alapvetően két módon szokás a virtuális környezeteket beépíteni a tesztelésbe:

- A virtuális gépek a tesztelő számítógépén futnak
    - Kialakíthatóak evvel különböző klienskonfigurációk, vagy akár egy komplett szerver-kliens architektúra is a tesztelő számítógépén, de általában kliensek illetve egyedülálló szoftverek tesztelésére használják ezt a konfigurációt.
    - Erős hardver szükséges a virtuális gépek futtatásához, különösen abban az esetben, ha több számítógépet futtatunk egyszerre.
    - Ha több tesztelőnek esetleg fejlesztőnek szüksége van különböző konfigurációkra, akkor érdemes egy képfájl könyvtárat létrehozni, ahonnan mindenki másolni tudja magának azt, amire éppen szüksége van.
- Központi szerveren futnak a virtuális gépek és azt távoli kapcsolattal lehet elérni
    - A tesztszervereket általában így hozzuk létre, és nem a tesztelő számítógépén.
    - Erős szervereken szolgálhat ez, mint kliensszolgáltató, és a tesztelők gépe mint vékonykliens funkcionál.
        - Ez jóval egyszerűbbé teszi a központi adminisztrációt
    - A szerverek virtualizációja segíthet szeparálni a környezeteket különböző célokra anélkül, hogy új hardvert kell beszerezni. Ez különösen ajánlott lehet az alapvetően alacsony teljesítményigényű szerverkörnyezetekben.
    - Segíthet ad hoc környezetek gyors létrehozására.

- Segíthet abban, hogy az adott igények szerint osszuk ki a rendelkezésre álló teljesítményt.
- Működhet egy erős szerveren futtatva, vagy több szervert egy privát felhőben egybegyúrva is.

Emulátorok

Az emulátor lehetővé teszi, hogy egy számítógép (gazda) egy másik számítógépként (vendég) viselkedjen, és annak szoftvereit és perifériáit futtassa, illetve használja.

A végfelhasználók általában akkor találkoznak emulátorokkal, amikor valamilyen másik (általában régi) platformra írt szoftvert futtatnak.

A tesztelők és fejlesztők általában különböző eszközökre (telefonok, konzolok stb.) írt programok tesztelésére és hibakeresésére használják. Az emulátorok előnye:

- Gyorsabban fejleszthető vele az adott program, mert a fejlesztőeszközön (számítógépen) is futtatható.
- Több platformon is kipróbálható (különböző operációs rendszer verziók, felbontások stb.).
- Olyan hardverre is lehet előzetesen fejleszteni, ami maga is fejlesztés alatt áll, és nem elérhető még.

A gazda és vendég számítógép felépítése akár jelentősen is eltérhet. Az eltérésből adódóan az emulátor megírása egy meglehetősen nehéz programozói feladat.

Az operációs rendszert és a programokat az emulátor tolmácsolja a gazdaszámítógépnek megfelelően. Ez nem csak az utasításkészlet fordítását jelenti, hanem bizonyos egyéb hardverelemeket is megfelelő módon kell tudni fordítani (például egyes memóriaterületek felelhetnek annak, hogy mi jelenik meg a képernyőn). Különösen a régebbi rendszereknél jellemző, hogy a fejlesztők speciális programozási technikákat használtak, amelyek nem dokumentált esetei voltak az adott hardvernek. Ezeket nagyon nehéz megfelelő módon emulálni.

A mai modern rendszerek általában szabványos rendszerhívásokon keresztül kommunikálnak a hardverrel, így elég azoknak a fordítását elvégezni.

Bár az emulátorok igen jó minőségűek, és nagyon felgyorsítják a munkát, nem tudják teljes mértékben helyettesíteni a valós eszközöket és az azon való fejlesztést, mert az emuláció sohasem teljesen pontos. Az emulátor is tartalmazhat programozási hibát, ami elfedheti az általunk elkövetett hibát, illetve a hardverteljesítmény sem feltétlenül azonos, ami más időzítéseket eredményezhet, és emiatt más lehet a működés is.

Meghajtók és csonkok

A tesztelés folyamán vannak esetek, amikor úgy kell tesztelnünk valamilyen funkcionalitást, hogy a kapcsolódó részek, amik a végtermékben szerepelni fognak, nincsenek jelen. Ennek több oka is lehet, ezek közül néhány:

- Az adott komponens még fejlesztés alatt/előtt áll, és nem elérhető a teszteléshez.
- Az adott környezetben nem lehet létrehozni valamilyen megkötés miatt.
- Olyan funkciót lát el az adott komponens, amit nem szeretnénk engedélyezni a tesztelés folyamán (például éles pénzügyi tranzakciók, e-mail küldése az adott szervezeten kívülre stb.).

Ilyen esetekben az adott komponenst egy általunk fejlesztett komponensre kell cserélnünk, ami pótolja az integrált funkcionalitást olyan módon, hogy az a tesztelést lehetővé tegye. Ezt megtehetjük úgy, hogy az integrált komponensen végzünk némi módosítást, ha ki akarjuk venni a nem kívánt funkcionalitást. Illetve, ha a komponens nem áll rendelkezésre, akkor általában egy olyan minimális komponenst fejlesztünk, ami nyújtja a megfelelő csatlakozási felületet, de a válaszok nagyon egyszerű módon programozottak, és nem követi az üzleti logikát. Vannak esetek, amikor az adott komponens a mi részünk vezérlését látja el (például a felhasználói felületet kell pótolnunk), akkor egy olyan komponenst kell fejlesztenünk, ami a megfelelő hívásokat a megfelelő paraméterezéssel elvégzi.

Szerepük szerint kétféle ilyen komponens létezhet:

- **Csonk (stub):** „egy szoftverkomponens speciális célú vagy részleges megvalósítása. A csonkot arra használjuk, hogy támogassuk a komponens(ek) fejlesztését vagy tesztelését. Helyettesíti a meghívott komponenst. [IEEE 610]”[1]
- **Meghajtó (driver):** „egy szoftverkomponens, vagy teszteszköz, amely kiváltja azt a komponenst, amely egy másik komponens, vagy a rendszer vezérlését, és/vagy felhívását végzi” [1]

Tesztkörnyezetek menedzsmentje

Egy kis szervezetben, viszonylag kis projekteknél, kevés külső függőség esetén maga a tesztkörnyezetek menedzsmentje általában nem szokott problémát okozni, ilyen esetekben egy reaktív szemlélet megfelelő agilitással, illetve a beszerzések tervezésével elégséges lehet egy ilyen környezetben.

Ahogy a szervezetek, projektek növekedni kezdenek, annál több konfliktust kell kezelni, és annál inkább szükség van a proaktív tervezésre. Általában a szervezetek úgy kezdik az életciklusukat, hogy kicsikék, és egyszerűbb projekteket, változásokat kell menedzselniük, majd a sikeres szervezetek elkezdenek növekedni, és evvel előkerül a komplexitás kérdése.

A következő növekedési szakasz, amikor irányelveket és megkötéseket vezetünk be, hogy a függőségeket kezeljük. Ezek a módszerek ismét jól működnek egy ideig, majd amennyiben a méret tovább növekedik, előbb-utóbb ezek a megszorítások elkezdik nehezíteni más stratégiai célok elérését, és szükség lesz a szervezeten belül arra, hogy nagyobb flexibilitást biztosítsunk bizonyos területeken, és javítsuk a

működés hatékonyságán. Akkor érdemes ilyen megkötéseken dolgozni, amikor a különböző projektek elkezdenek konfliktusba kerülni egymással.

A nagyobb szervezetek általában akkor működnek megfelelően, ha a működésük és folyamataik megfelelően szabályozottak, dokumentáltak, illetve nagyon fontos, hogy a szervezeten belül ismertek és betartottak legyenek. Ahhoz, hogy ezeket a célokat elérjük, olyan rendszerek kialakítása szükséges, amelyek bár összetettek, az egyes részek, amelyeket bizonyos szerepkörökben végre kell hajtani, egyszerűek, könnyen értelmezhetőek és a szerepkörben dolgozó emberek megfelelő képzést kapnak a használt eljárásokról. Egy ilyen eljárásrendszer kidolgozása jellemzően nagy munka és évekig eltart. Az ilyen összetett folyamatot érdemes szakaszolni, és egy megfelelő keretrendszert (process framework) alkalmazni (például CMMI[67] vagy ITIL[68]). Ebbe a szakaszba általában akkor szoktunk belépni, amikor a megszorításokat, amiket kitűzünk, nem tudjuk betartani, illetve azok olyan körülményeket biztosítanak, amelyek egyre több projekt számára nem ideálisak, rontják azok hatékonyságát. A tapasztalatok szerint, mindez együtt jár az átadott minőség fokozatos romlásával is, ami növeli a szolgáltatások működtetésének költségét. Egyes proaktív kultúrájú szervezetek átlépik a közbülső szakaszt, és egyből belépnek ebbe a szakaszba.

A megfelelő folyamatfejlesztésnek nem csak a tesztkörnyezetekre illetve a tesztelésre, minőségre van pozitív hozadéka, hanem ez az egyik legjobban kiaknázható hatékonyságnövelő potenciál a legtöbb szoftverfejlesztéssel foglalkozó szervezet számára[66]. A tesztkörnyezetek menedzsmentje általában nem egyetlen terület, hanem elosztott több különböző terület között, és különböző szerepkörök felelősek egyes elemekért.

A méret lényeges ezekben a folyamatokban, ami egy nagy szervezetben a hatékonyságot növeli, az egy kis szervezet számára felesleges bürokráciát jelent. A következő fejezetekben méret szerint haladva közelítjük meg, hogy mely folyamatok lényegesek az adott méretnél.

Egyes fogalmak több szabvány szerint értelmezhetőek, és bár általában az azonos nevű területek központi része hasonló funkciót lát el, de vannak eltérések a méretükben, és abban, hogy a kapcsolatokból melyik rész tartozik az adott területhez, és mik a kapcsolódó területek. Ebben a fejezetben a *CMMI*[67] szerinti tagolást fogjuk használni.

Kis méretű szervezetek

A kis méretű szervezeteknél jellemző módon kevés a függőség, általában projekten belül folyik az összes igény menedzsmentje. Bár jellemzően lefedik bizonyos részeit egy-egy CMMI területnek, a megvalósítás tagolása még nehezen értelmezhető a csapat számára. A jellemző fő területek, amit le kell fedni:

- A tesztkörnyezetek tervezése
    - Technika, technológia
    - Tesztelési stratégia
    - Projekt-menetrend, saját menetrend
    - Karbantartási menetrend

  - Esetleges külső függőségek, szolgáltatások
- Tesztkörnyezetek nyilvántartása
  - Ki mikor használja, foglalja
  - Milyen igényeknek kell adott teszteléshez megfelelni
  - Tesztkörnyezetek állapota
- Tesztkörnyezetek karbantartása
  - Adott tesztelésre való felkészítés
  - Adatfrissítés
  - Új build telepítése

Tesztkörnyezetek tervezése és nyilvántartása

A tesztkörnyezetek tervezése a projekt tervezésével együtt, azt támogatva történik. Ilyenkor általában egyszerre tervezzük az architektúrát, a projekt menetrendjét, költségeit, erőforrásokat, folyamatokat, a tesztelést és annak stratégiáját, amelyek egyszerre nyújtanak információt ahhoz, hogy mikor és milyen formában lesz szükségünk a különböző környezetekre. A folyamatok egymásra hatva alakítják egymást.

A környezeteknek általában van néhány jellemzője, amit érdemes a tervezés közben rögzíteni:

- **Név,** amire hivatkozni tudunk, ezáltal elősegíti a kommunikációt és az információ rendszerezését. Bár triviálisnak tűnik, a jó elnevezések megtalálása nagyon fontos. Egyes esetekben az elnevezések nem konzisztensek a különböző csoportokban, ami rendszeres forrása lehet a kommunikációs bakiknak. Amikor ilyet tapasztalunk, lehetőleg korrigáljuk, és állapodjunk meg egy közös névhasználatban, ami mindenki számára elfogadható. Ne feledjük el a megfelelő dokumentációt javítani a megállapodás után.
- **Cél,** amire az adott környezetet elsősorban használni szeretnénk. A későbbiekben, amikor ellenőrizzük a terveket, hogy mindenhol rendben vannak és konzisztensek-e, illetve egyes esetekben a későbbi konfliktusok kezelésében jelentős segítséget nyújthat.
- **Technikai paraméterek,** azaz hogy pontosan mit tartalmaz az adott környezet.
- **Beszerzési feladatok és azok menetrendje,** hogy időben elkészüljön, és mindenki tudja a feladatát.
- **Felhasználás menetrendje,** amiben rögzítjük, hogy kik, mikor, mire tervezik használni az adott környezetet, annak milyen feltételeknek kell megfelelni, mi az adott tevékenység státusza.
- **Karbantartási feladatok és azok menetrendje,** hogy a megfelelő állapotot készítsük elő a tevékenységekhez.

A **név, cél** és **technikai paraméterek** általában valamelyik tervben (például a mester teszttervben) definiáltak, és általában szöveges formájúak. A különböző menetrendeket pedig általában külön dokumentum(ok)ban helyezzük el. Ez megjelenhet egy projekttervezésre használt eszközben, vagy csak egyszerű táblázatként is. Lehetnek azonban olyan rendszeres feladatok, amelyeket nem definiálunk minden alkalommal (például a kódfrissítések), hanem azokat, mint rendszeres tevékenységeket a projekttervben, vagy a mester teszttervben szerepeltetjük. Az alábbi sablon egy példa arra, hogyan dokumentáljuk egy kis projekt tervét egy tesztkörnyezetre, és abban hogyan tartsuk nyilván az aktuális állapotot.

Környezet neve: Teszt:

Kezdés	Befejezés	Tervezett tevékenység	Résztvevők	Feltételek	Státusz
2015.04.01	2015.04.03	Megrendelés elindítása	Projektmenedzser, beszerzés	Architektúra definíció kész	100%
2015.04.06	2015.04.10	A hálózati infrastruktúra megtervezése	Rendszeradminisztrátorok	Architektúra definíció kész	100%
2015.04.13	2015.04.30	A hálózati infrastruktúra előkészítése	Rendszeradminisztrátorok	Architektúra definíció kész, hálózati eszközök rendelkezésre állnak	
2015.05.04	2015.05.04	Az eszközök átvétele	Rendszeradminisztrátorok	A rendelés beérkezett	100%
2015.05.05	2015.05.08	Az eszközök beépítése, az operációs rendszerek és szoftverkörnyezet telepítése	Rendszeradminisztrátorok	A rendelést átvették, minden eszköz elérhető	100%
2015.05.11	2015.05.11	Infrastruktúra tesztje	Rendszeradminisztrátorok	A környezet felépítése befejeződött	100%
2015.05.04	2015.05.22	Alapadathalmaz előállítása	Tesztelők, fejlesztők, rendszeradminisztrátorok		100%
2015.05.25	2015.05.29	Környezet előkészítése az integrációs tesztelésre, részletes terv: <link>	Tesztelők, fejlesztők, rendszeradminisztrátorok	A környezet felépítése befejeződött, az alapadathalmaz definiált, létezik telepíthető buiild	100%
2015.06.01	2015.07.17	Integrációs teszt	Tesztelők, fejlesztők	Az integrációs tesztkörnyezet működik	45%
2015.06.15	2015.06.19	Teljesítményteszt igények tervezése	Teljesítménytesztelő	Teljesítményteszt szcenáriók készen vannak	100%
2015.06.29	2015.07.17	Teljesítménytesztek fejlesztése és tesztelése	Teljesítménytesztelő	Az integrációs tesztkörnyezet működik	15%
2015.07.20	2015.07.21	Rendszerkarbantartás - teljesítményteszt előkészítése	Teljesítménytesztelő, rendszeradminisztrátorok	Az integrációs teszt befejeződött, a funkciók fejlesztése lezárult	0%
2015.07.22	2015.07.24	Teljesítménytesztelés első kör	Teljesítménytesztelő	Teljesítményteszt előkészítése	0%
2015.07.20	2015.07.24	Rendszerteszt adatigényeinek előkészítése	Tesztelők, fejlesztők	Az integrációs teszt befejeződött, a funkciók fejlesztése lezárult, rendszerteszt terv elkészült	0%
2015.07.27	2015.07.31	Rendszerkarbantartás - rendszerteszt előkészítése, részletes terv: <link>	Tesztelők, fejlesztők, rendszeradminisztrátorok	A rendszertesztelés belépési kritériuma teljesül	0%
2015.08.03	2015.08.14	Rendszerteszt	Tesztelők	A rendszer előkészítése sikeres	0%

2015.08.03	2015.08.14	UAT előkészítése	Tesztelők, fejlesztők	UAT teszterv elkészült	0%
2015.08.17	2015.08.18	Rendszerkarbantartás, - teljesítményteszt előkészítése	Teljesítménytesztelő, rendszeradminisztrátorok	A rendszerteszt sikeresen lezárult	0%
2015.08.19	2015.08.21	Teljesítményteszt 2. kör	Teljesítménytesztelő	A rendszer előkészített a teljesítménytesztelésre	0%
2015.08.24	2015.08.28	Rendszerkarbantartás - A UAT előkészítése. Részletes terv: <link>	Tesztelők, fejlesztők, rendszeradminisztrátorok	UAT előkészítése befejeződött, rendszerteszt sikeres	0%
2015.08.31	2015.09.04	UAT	UAT tesztelők	A rendszer előkészítése a UAT-re sikeres	0%

Táblázat 10 - Sablon egy környezetmenetrendhez kis projektnél

Vannak olyan tesztelési projektek, amikor különböző eszközöket kell biztosítanunk a teszteléshez. Ilyenek lehetnek a prototípusok, vagy egy mobilfejlesztés. Ilyen esetekben szükség van egy olyan eszköznyilvántartásra, ahol a tesztelőknek kiadott eszközöknek a jelenlegi használóját és a foglalásokat rögzíteni tudjuk, illetve a teljes leltárt nyilvántartjuk.

Közepes méretű szervezetek

Amikor egy szervezet elkezd növekedni, megjelennek a függőségek és az integrációk, amelyek bizonyos fokú szinkronizáltságot megkövetelnek. Hogy mekkora szervezet tartozik a közepes méretbe, azt néha nehéz megmondani. Néha egészen kicsike szervezetek is beleesnek, mert sok, esetlegesen külső, függőségük van, és néha egészen nagy szervezetek is sokáig megmaradnak ezen a szinten.

Az első természetes reakció az ilyen jellegű kihívásra, hogy közvetlenül a kihívásra reagálva megtiltunk néhány problémát okozó dolgot, és bevezetünk némi egységesítést és minőségbiztosítást. Ezek az egységesítések befolyásolják, hogy miképpen lehet, illetve kell definiálnunk a tesztkörnyezetet.

A leggyakoribb egységesítési technikák:

- **A különböző projektek menetrendjeinek szinkronizálása,** ami általában azt jelenti, hogy bizonyos tesztelési fázistól a projektek azonos menetrenddel rendelkeznek, és egyszerre van az új kiadásra való átállása mindenkinek előre meghirdetett időpontban. Ez biztosítja, hogy az integrációk okozta függőségeket könnyen teljesíteni lehessen.

- **Egységes tesztelési stratégia és módszertan,** ami segít abban, hogy a közös fejlesztési menetrend gond nélkül támogatható legyen.

- **Egységes fejlesztési és minőségbiztosítási irányelvek,** ami segítenek abban, hogy a különböző projekteket könnyebb legyen integrálni.

- **Kötelezően előírt környezetek,** amik egyértelművé teszik az integrációs pontokat és a használandó környezeteket az egységesített tesztelésnél.

- **Egységesített tesztkörnyezet karbantartás,** amelyek lehetővé teszik az adatok konzisztenciájának megőrzését és a tesztelések adatkövetelményeinek kielégítését.

- **Egységesített verziókontroll,** hogy egyértelműen azonosítható legyen az adott időpontban telepített kód.

A fenti egységesítések egyrészt segítenek kezelni a megnövekedett komplexitást, és egyes részeken nagy mozgásteret engednek, más területeken nagyon megszorítóvá tudnak válni. Amennyiben jó kompromisszumot kötünk, az adott rendszer sokáig kezelhető és megfelelő maradhat, azonban az idővel a prioritások jelentősen változhatnak, és egy ilyen rendszer meglehetősen merev is tud lenni bizonyos kérdésekben. Az itt bemutatott példánál például nehéz lehet a minőségi problémák kezelése, a kisebb menetrendcsúszások kezelése, esetleg más projektmenedzsment módszerek használata. Ami egy kisebb szervezetben jó kompromisszum, egy nagyobban éppen a továbbfejlődés gátjává válhat.

Nagy méretű szervezetek

A nagyobb szervezetek fejlesztési folyamatai bonyolult rendszert alkotnak, hogy a nagy komplexitást kezelni tudják. Egy ilyen rendszer akkor jó, ha az egyének szintjén a végrehajtandó elemek viszonylag egyszerűek és könnyen végrehajthatóak.

A következő néhány fejezetben átnézzük azokat a folyamat területeket, amelyeknek hatásuk van a tesztkörnyezetekre illetve azok menedzsmentjére.

Konfigurácimenedzsment (Configuration Management)

Ez az a terület, ami sok organizációnál nem kap elég figyelmet, pedig nagyon fontos hatékonysági tényező, valamint nagyban segíti, hogy az egyes elemek jól kommunikálhatóak és azonosíthatóak legyenek, valamint a különböző termékelemek története visszakövethető legyen.

A konfigurációmenedzsment „a következő tevékenységek technikai és adminisztratív irányítása: a konfigurációs elemek funkcionális és fizikai karakterisztikáinak meghatározása és dokumentálása, az ezen karakterisztikákhoz képest történő változás irányítása, a változáskezelési és megvalósítási állapot nyomon követése és jelentése, illetve a különböző követelményeknek történő megfelelés”[1].

A könnyebb érthetőség kedvéért nézzük meg, hogy milyen tevékenységek tartozhatnak az adott folyamatokhoz:

- Egy termékelem adott időpontban lévő alapkonfigurációjának meghatározása
- A konfigurációs elemek változásainak kontrollja
- A termékelemek megépítése vagy az építési módszer specifikálása a konfigurációmenedzsment rendszerből
- Az alapkonfigurációk egységének fenntartása
- Megfelelő státusz és konfiguráció információk nyújtása a fejlesztőknek, végfelhasználóknak és ügyfeleknek

A konfigurációba, amit menedzselünk, beletartozhatnak az ügyfeleknek átadott termékek, belső eszközök, termékelemek, beszerzett termékek, eszközök, egyéb elemek, amelyeket ezek fejlesztéséhez vagy leírásukhoz felhasználtunk.

Példák, amit konfigurációmenedzsment alá helyezhetünk:

- Hardver és felszerelések
- Specifikációk
- Eszközkonfigurációk
- Forráskód
- Fordítóprogramok
- Teszteszközök és leírások
- Teszteredmények
- Telepítési feljegyzések
- Adatfájlok
- Termékről megjelent publikációk
- Tervek
- Követelmények, folyamatdefiníciók, üzleti esetek leírása, ezek implementációs állapota
- Architektúra dokumentálása és tervei

Egyrészt jól látható, hogy a tesztkörnyezetek lehetnek konfigurációs elemek és így a konfigurációmenedzsment része (egyébként ez a javasolt eljárás), tehát alkalmazkodnia kell a megfelelő szabályrendszerhez, másrészről ezen keresztül kapcsolhatóak össze konzisztensen ezek az elemek.

Egy adott konfigurációból több változat is jelen lehet egyszerre. Lehetséges konfigurációk lehetnek:

- **Kiadott változat(ok),** amelyek már elérhetőek a felhasználói tábor részére.
- **Kiadás jelölt,** ami a várható kiadásra jelölt. Ebből is létezhet akár több is egyszerre.
- **Tervezett kiadás,** ami ugyan szerepel a tervekben, de még nincs végleges döntés róla.
- **Archivált kiadás,** amely olyan kiadás, amely valamikor elérhető volt a felhasználók számára, de már nem elérhető/támogatott.

A konfiguráción belül az elemeknek különböző státuszuk van, amelyek meghatározzák, hogy mekkora és milyen változtatási kontroll alatt van az adott elem, illetve az adott változtatáshoz kinek az engedélye szükséges, illetve kit kell értesíteni az adott változásról. Ez fontos a megfelelő kommunikáció és

minőségbiztosítás szempontjából is. Mivel a tesztkörnyezeteink is a konfigurációmenedzsment kontrollja alatt vannak, meg kell határozni, hogy azok milyen állapotokat vehetnek fel, és milyen változásmenedzsmentet és kommunikációt kell folytatni az adott környezettel, illetve milyen szabályoknak kell megfelelni.

Lehetséges kontrollszintek:

- **Nem kontrollált:** Korai fázis, amikor bárki által szabadon változtatható.
- **Kidolgozás alatt:** A tervezők szabadon módosíthatnak.
- **Szemlézés alatt:** A tervezők szabadon változtathatnak a kommentek alapján, de a változtatásokat dokumentálni kell, és megosztani a szemlézőkkel.
- **Véglegesített, tesztelés alatt:** Amikor a környezet konfigurációja már fix, de a változtatások egyes köre még engedélyezett külön jóváhagyás nélkül (például a hibajavítások nem engedélykötelesek, de új funkció már csak meghatározott kör engedélyével kerülhet bele).
- **Véglegesített, végső tesztelés alatt:** minden változtatás engedélyköteles.

Amikor tesztelünk, fontos annak az ismerete, hogy milyen elemekből áll adott pillanatban össze a tesztelt termék. Hogy mindezt pontosan követni tudjuk, minden egyes build-et, ami tesztelés alá került tudnunk kell egyedileg azonosítani. Az adott problémára általában a verziószámok bevezetését alkalmazzák. Az alábbiakban egy általánosan elterjedt verziószámozási rendszerrel ismerkedünk meg, amely széles körűen alkalmazott, de a világban számos megoldás létezik. Egy jó összefoglaló található erről a Wikipedia-n[69] azoknak, akik más módszereket is meg kívánnak vizsgálni.

A verziószámozás alapja általában egy olyan séma, ahol a nagyobb számok jelentik az újabb változatot. Mindezen kívül a változás mértékét is jelzi olyan módon, hogy a verziószám több számból áll, és minél nagyobb helyértékű számot változtatunk, a változás annál nagyobb mértékű.

Az itt ismertetett módszer egy négyszintű rendszert vezet be a következő formátummal:

<nagy verzió>.<kiegészítés>.<hibajavítás>.<build>

Helyérték	**Leírás**
Nagy verzió (major)	Jelentős funkcionális változtatások, a központi funkcionalitást érintő gyökeres változtatás.
Kiegészítés (minor)	Apróbb új funkciók, esetlegesen hibajavítások, alapvetően használhatósági változtatások, aktualizálások, de alapjaiban nem változtatja meg az eszközt.
Hibajavítás (bug fix)	Nincs új funkció, csak hibajavítások.

Build	Az egyedi azonosítást, különösen a nem kiadott változatok közötti megkülönböztetést segíti. Egyes rendszerek minden hivatalos kiadásszám után visszaállítják a számlálót, mások csak nagyobb változtatás esetén, vagy sohasem.

Táblázat 11 - Szoftver verziószámozási séma

Egy példaként, ebben a rendszerben a 3.2.1.243-as verzió a harmadik lényeges változat második kiegészítésének első javított kiadásának a 243. megépített változata. Egyes esetekben a negyedik számot a build számot a hivatalos kiadások számozásában nem szerepeltetik, csak a belső kommunikációban esetleg egy olyan átiratban, hogy „v3.2.1 build 243”, ott viszont a megfelelő azonosításhoz mindenképpen szükséges.

Projektek tervezése, nyomon követése és ellenőrzése (Project Planning, Project Monitoring and Control)

A két folyamatterület tipikusan a projektmenedzserek tevékenységi körét fedi le (néha kiegészítve egyéb területekkel), és tipikusan az összekötő kapocs a különböző területekkel. Mivel a tesztkörnyezetek definíciójakor több terület közös munkájára van szükség, ennek megfelelő koordinálása elengedhetetlen.

A tesztkörnyezetek, és azok eszközigénye jelentős beruházási tétel egy projekt számára, amiben hardverelemek és általában szoftverlicencek is szerepelnek.

A tervezésük, beszerzésük, megépítésük, adminisztrációjuk és rendszeres felkészítésük a különböző tevékenységekre általában jelentős emberi erőforrás ráfordítását igényli, amit biztosítani, és a megfelelő helyeken allokálni kell a feladatra. A tevékenységek folyamán jelentős külső és belső függőségi viszonyok és időbeli megkötések jelentkeznek, amelyek a tevékenységek tervezését általában komplex feladattá teszik.

A függőségek problémája nem csak a tervezés folyamán jelent feladatot, hanem a végrehajtási fázisban is folyamatos nyomon követést követel, hogy a különböző változásokat és azok hatását megfelelően kezelni lehessen. A megfelelő kockázatkezelés sokat segíthet a váratlan események kezelésében, amelyek egyik fő forrása éppen a különböző belső és külső függőségek, melyekre kevés behatásunk van, amikor ezért a területért vagyunk felelősek.

Mint minden területen, itt is fontos, hogy nyomon tudjuk követni a státuszt, és amennyiben eltéréseket tapasztalunk, azokra reagálva megfelelő korrekciós lépéseket tegyünk.

Beszállítók menedzsmentje (Supplier Agreement Management)

Ez is egy olyan terület, amelyet tipikusan a projektmenedzser koordinál a projekt szempontjából, de többféle szerepkör is részt vesz benne. Nagyobb szervezetek általában meghatározott beszállítói körből intézik a beszerzéseket, és a meghatározott beszállítókkal szerződésekben rögzítik a beszerzési feltételeket. Amint a technikai megoldás körvonalazódik, illetve a tesztkörnyezetek tervezése is véglegessé kezd válni, a megfelelő eszközöket be kell szerezni. Egyes eszközökre már létezhetnek beszállítói szerződések, és azok alapján már lehet tervezni költségekkel és határidőkkel, más eszközökre viszont elképzelhető módon meg kell keresni a megfelelő beszállítót. Egy nagyobb szervezetben erre a

területre léteznek csoportok, amelyek ezt a folyamatot segítik és felügyelik, ami nagy segítséget nyújthat.

Amikor a tesztkörnyezeteket tervezzük és építjük, az itt rögzített feltételek alapján várhatunk el szolgáltatást a beszállítóktól.

Kockázatkezelés

Mint minden területen egy projekten belül, a tesztkörnyezetek felépítésénél és karbantartásánál is vannak kockázatok, amelyeket valamilyen módon érdemes kezelni. Egy-egy kockázatra különböző válaszokat adhatunk:

- **Csökkentjük a veszélyét,** vagy az előfordulását vagy a hatását megelőző tevékenységekkel.
- **Elhárítási tervet alkotunk,** hogy amikor bekövetkezik az esemény, tudjuk, hogy mik a megfelelő teendők, hogy a károkat minimalizáljuk.
- **Áthárítjuk,** egy másik félre a kockázatot, például biztosítást kötünk.
- **Elfogadjuk,** vannak olyan kockázatok, melyeknek az elhárítása illetve kezelése akkora erőforrástöbbletet követelne, ami nincs arányban az okozott negatív következményekkel.

Egy kockázatra nem csak egy válaszunk lehet, hanem használhatjuk ezek kombinációját.

Egy példa: Földrengésveszélyes a terület, ahol a fejlesztés folyik, amire a következő válaszokat adjuk egyszerre:

- **Csökkentjük a veszélyét**
    - Olyan épületben tartjuk a környezeteket, amely ellenáll egy erős földrengésnek is.
    - Az adatvesztés kockázatát megfelelő biztonsági mentéssel csökkentjük, úgy, hogy a mentett állományoknak létezik egy nagyobb földrajzi távolságban mentett kópiája is.
- **Elhárítási tervet alkotunk**
    - A legfontosabb eszközöket egy nyilvános felhőbe költöztetjük
        - Amihez megkötjük a szolgáltatási szerződést,
        - A megfelelő virtuális gép konfigurációt és képet készenlétben tartjuk,
        - A költöztetési eljárást dokumentáltuk és teszteltük,
        - Több ember is megfelelően képzett, hogy az eljárást végre tudja hajtani.
- **Áthárítjuk**
    - Az eszközökre biztosítást kötünk, hogy az eszközöket, amelyek megsérültek anyagi veszteség nélkül vissza tudjuk állítani.

- **Elfogadjuk**
    - Hogy bizonyos tevékenységek csúszni fognak, ezért bizonyos határidők nem lesznek betartva, és újra kell tervezni a projekt menetrendjét.

Minden folyamatnak, területnek saját magának kell gondoskodnia a megfelelő kockázatok kezeléséről. Mivel maga a tesztkörnyezet kezelése nem egy önálló folyamat, hanem több területen is foglalkoznak az adott területtel, annak a kockázatai általában elosztva több kockázati listában is megjelennek.

A kockázatokat mindig az adott szinten kell kezelni, egészen addig, amíg annak előfordulási valószínűsége, illetve más területre való hatásának mértéke nem indokolja, hogy azon a területen is megjelenjen, ahol ennek a hatásait esetlegesen szintén orvosolni szükséges.

A kockázatok kezelésére általában elég egy egyszerű táblázatkezelő is, de vannak olyan, általában nagyon nagy méretű projektek, ahol erre külön kockázatelemző és kezelő eszközöket használnak.

A rögzített jellemzők egy kockázatnál:

- A kockázat egyedi azonosítója
- A kockázat összefoglalója
- A kockázat részletes leírása
- Előfordulási valószínűsége, általában 3-5 elemű lista szerint
- Előfordulása esetén a várható hatás mértéke, általában szintén 3-5 elemű lista szerint
- Alkalmazott tevékenység(ek) (veszély csökkentése, elhárítási terv, áthárítás, elfogadás)
- Az alkalmazott tevékenységek részletes leírása
- A kockázat kezelésének státusza
- Egyes esetekben jelezni szokták azon csoportok, folyamatok körét ahol ennek hatása lehet

Követelménymenedzsment és fejlesztés (Requirement management and Requirement Development)

Egy projekt célja az ügyfél termékkel kapcsolatos követelményeinek kielégítése. Az ilyen követelmények lehetnek a kívánt funkcionalitást meghatározóak illetve technikai jellegűek. Ezek a követelmények határozzák meg a technikai megoldást, ami jelentősen befolyásolja magának a tesztkörnyezeteknek a felépítését.

További követelményeket generálhat maga a projekt, illetve maga a fejlesztési módszer, illetve maga a szervezet is (például előírt tesztmódszerek, kötelezően felépítendő környezetek, és azok követelményei, egyéb minőségi követelmények stb.), amelyeknek további hatása lehet a megfelelő környezetek felépítésére, mennyiségére, azok használatának módszereire. A követelmények fejlesztése maga is előállhat technikai követelményekkel a tesztkörnyezetek számára, mert szükség lehet rájuk demonstrációs, tréning vagy koncepciók ellenőrzése céljából.

Technikai megoldás (Technical Solution)

A technikai megoldás folyamatai arra szolgálnak, hogy kiválasszanak, megtervezzenek és implementáljanak olyan megoldásokat, amelyek kielégítik a követelményeket.

A technikai megoldás szolgáltatja az egyik legjelentősebb részt a tesztkörnyezetek számára. A megoldásnak kulcsszerepe van a tesztkörnyezetek technikai specifikációjában.

Maga a technikai megoldás definiálja a fejlesztési eljárást, illetve szerepe van a megfelelő tesztfolyamat kialakításában is, amelyek együttesen meghatározzák, hogy milyen követelményeknek kell megfelelni a tesztkörnyezeteknek.

Termékintegráció (Product Integration)

A termékintegráció azt a folyamatot specifikálja, hogy hogyan építjük fel a végterméket a részekből.

Maga az integráció folyamata meghatároz követelményeket, karbantartási eljárásokat a tesztkörnyezetekkel kapcsolatban, amelyek egy része magából a folyamatból ered, míg más részei áttételesen abból, hogy az integrációs stratégia több dolgot is meghatározhat a tesztelési folyamatban.

Egyes esetekben az integráció egy inkrementális folyamat, ahol az egyszerűbb modulok felől közelítve egyre nagyobb részeket építünk egybe és az esetlegesen hiányzó részeket csonkokkal és meghajtókkal helyettesítjük, amíg maga a megfelelő komponens nem elérhető.

Agilis módszerek alkalmazása esetén ez a tevékenység gyakori, akár napi rendszerességű. Megfelelő integrációs stratégia elengedhetetlen már a korai fázisban is, amit folyamatosan karban kell tartani a tapasztalatok alapján.

Egyes integrációs stratégiák előírnak rendszeres automatizált egység és egységintegrációs teszteket, amelyeknek általában vannak tesztkörnyezetbeli elvárásai.

Verifikáció és validáció (Verification and Validation)

A verifikáció és a validáció határozza meg egyebek mellett a tesztelés folyamatát. A két terület módszereiben nagyon hasonló, de más jellegű hibákra koncentrál.

Verifikáció: *„az adott követelmények teljesülésének vizsgálata és konfirmálása [ISO 9000]”*[1]

Validáció: *„annak vizsgálata és konfirmálása, hogy a szoftver tervezett felhasználási céljának megfelelő követelmények teljesülnek-e [ISO 9000]”* [1]

Képletesebben a verifikáció azt ellenőrzi, hogy „a megfelelő módon épült meg” a termék, addig a validáció azt vizsgálja, hogy a „megfelelő dolog épült meg”.

A két terület meghatározza a tesztelési folyamatokat és célokat a többi már felsorolt terület eredményei, illetve a szervezet minőségbiztosítási irányelvei, folyamatai alapján. Mindezen folyamatok együttesen meghatározzák azokat a követelményeket, amelyek alapján a tesztkörnyezetek összeállnak, többek között:

- A tesztkörnyezetek számát, elnevezését és azok felhasználási célját,

- Technikai/technológiai felépítést,
- A használat feltételeit és szerepköreit,
- A felépítés menetrendjét,
- A tesztelés menetrendjét,
- A tesztkörnyezetek karbantartásának menetrendjét,
- Egyéb technikai paramétereket.

Általában a tesztelésért felelős személyt nevezzük tesztmenedzsernek, aki koordinálja a tesztkörnyezetekkel kapcsolatos teendőket, bár más megoldások is léteznek, illetve egyes döntési jogkörök más személyekhez, szerepkörökhöz is kerülhetnek.

A folyamat során hibákat találunk, amelyeket megfelelően prioritással és egyéb jellemzőkkel kell ellátni. A hibáknál nagyon fontos a megfelelő konfiguráció pontos megjelölése, amihez szükséges, hogy a hibakezelő eszközt megfelelően fel legyen készítve ennek a jelölésére. Egyes elemek a konfigurációban egységesített logikai névvel szerepelnek (például egyes környezettípusok, lásd DTAP modell), amelyhez ezekben az eszközökben alkalmazkodni kell. Az egységesítés segíti a könnyebb kommunikációt és újabb erőforrások beillesztését, valamint lehetővé teheti több platform együttes teljesítményének elemzését ezekkel a paraméterekkel.

Szerepkörök és felelősségek (Roles & Responsibilities)

Egy fejlesztési folyamatban sok szerepkör vesz részt. Bár többen keverik, a szerepkör nem kell, hogy megegyezzen a titulussal (ami a névjegykártyán és a munkaszerződésen szerepel), hanem az adott projekten betöltött munkát és felelősséget reprezentálja. Egy személynek lehet több szerepköre, és egy szerepkört több ember is betölthet.

A tiszta és jól definiált szerepkörök segítenek annak a megértésében, hogy mit várnak el az adott szerepkörtől és a betöltő személytől. Ez segíti mind a szerepkört betöltő személyt, mind a környezetét.

A szerepkörök definiálására több lehetséges módszer is rendelkezésre áll. A legegyszerűbb a szöveges leírás, ami nem biztos, hogy teljesen egzakt, viszont a legtöbb esetben, ha megfelelően részletes, elégséges, és megfelelően gyors lehet.

Egy pontosabb megoldás a RACI mátrix (az angol Responsible, Accountable, Consulted, Informed szavakból), ahol akár a szöveges definíciót kiegészítve, akár nélküle, a megfelelő szerepkörökhöz és termékelemekhez, mint egy mátrixhoz definiáljuk a felelősségi kört. A négy kategória:

- Responsible – Felelős, aki csinálja, az adott tevékenység végrehajtásáért felel
- Accountable – Felelős, aki jogilag, pénzügyileg felel érte
- Consulted – Bevont, akit aktívan bevonnak a tervezésbe, megvalósításba

- Informed – Informált, akit a változásokról értesíteni kell

Egy kicsit még időzzünk el az első két kategóriánál, mivel a magyar nyelvben azonos szóra fordítódik a két angol szó, egyes esetekben tapasztalható, hogy a különbségtétel nehezen megy számunkra. A legegyszerűbb ezt egy példán keresztül bemutatni.

A tesztkörnyezetek megrendelésének összeállításáért az architekt a felelős, azaz ő végzi el a munkát, és állítja össze, de a megrendelésért jogilag és pénzügyileg a projekt vezetője felel a beszállító felé, mivel az architekt nem kezel költségvetést, és nincs pénzügyi felelőssége. Az ilyen esetekben az elvégzett munkát a projekt vezetője ellenőrzi, és nagy valószínűség szerint ő küldi ki a megrendelést a beszállítónak.

Egy RACI mátrix összeállítása nagy odafigyelést, tervezést és a szervezet folyamatainak ismeretét igényli, általánosan ilyen mátrix összeállítása nem lehetséges, de a leggyakrabban előforduló szerepköröket és azok felelősségi körét egy táblázatban foglaljuk össze. Mivel minden projekt és szervezet más, az itt felsorolt szerepkörök és felelősségek nem minden szervezetben vannak meg, egyes szerepkörök határai akár jelentősen máshol is húzódhatnak, illetve más elnevezés is alkalmazható egyes esetekben.

Szerepkör	**Felelősség**
Projektvezető (Project Manager)	A projekt kivitelezéséért, céljának eléréséért a megadott költségkeret betartásáért felel. Megtervezi, nyomon követi és ellenőrzi az elvégzendő feladatokat, kezeli a projekt kockázatait és kommunikál az ügyfelekkel az adott státuszinformációk alapján, kezeli a változásokat.
Tesztelési vezető (Test Manager)	Felelős a verifikációs és validációs stratégia és terv létrehozásáért és végrehajtásáért. A stratégia lefordítása konkrét lépésekre, a lépések végrehajtásának megszervezése és ellenőrzése. Definiálja a validációs és verifikációs környezeteket, ellenőrzi azok megvalósulását. Kezeli a tesztelési kockázatokat. Definiálja a hibamenedzsment folyamatokat. Jelentést készít a megfelelő személyeknek az aktuális státuszról. Egyes esetekben egyéb minőségbiztosítási feladatokat is ellát (például auditok).
Tesztmérnök (Test Engineer)	A tesztesetek megtervezéséért felelős. A tesztesetekből adódó követelményeket (például tesztadat), specifikálja, és a megfelelő helyre továbbítja. Segítséget nyújthat a tesztkörnyezetek előkészítésében, amennyiben szükséges.
Tesztelő (Tester)	Végrehajtja a teszteket és jelenti az eredményeket. Hiba esetén jelenti a hibát a megfelelő helyen az előírt módon.
Teljesítménytesztelő (performance tester)	Megtervezi, előkészíti és végrehajtja a teljesítménytesztet. Specifikálja a teljesítménytesztelés folyamatát, és annak követelményeit, amit átad a tesztelési vezetőnek, aki gondoskodik a megfelelő külső követelmények kielégítéséről.
Automatikus tesztelő (Automation Tester)	Megalkotja az automatikus tesztelési stratégiát, meghatározza az automatikusan végrehajtandó tesztesetek körét a tesztmérnökökkel közösen, kifejleszti és lefuttatja az automatikus teszteket.
Architekt (Architect)	A technikai megoldás definiálásáért felelős személy. Segítséget nyújt a projekt minden területén technikai kérdésekben, így például a tesztkörnyezetek technikai paramétereinek meghatározásában is.

Fejlesztő (Developer)	A követelmények és a technikai terv alapján elkészíti a megfelelő programokat, azokat a megfelelő irányelvek szerint hozzárendeli a megfelelő konfigurációkhoz. Javítja a tesztelés folyamán megtalált hibákat, egyes esetekben felelős az egységtesztek tervezéséért és végrehajtásáért.
Rendszerszervező (Business Analyst)	Összegyűjti, analizálja és dokumentálja az ügyfelek követelményeit, megfelelő prioritást állít fel, összekötőként funkcionál az ügyfelek, a fejlesztők és a tesztelők között.
Kiadásmenedzser (Release Manager)	Általánosan felelős a kiadási folyamatért, meghatározza az általánosan használandó minőségi és szoftverfejlesztés módszertani követelményeket, ellenőrzési pontokat, konfigurációmenedzsment alapelveket, amelyeket a projekteknek be kell tartaniuk, hogy ne legyenek konfliktusok a különböző fejlesztések között
Szolgáltatásmenedzser (Service Manager)	Felelős az adott üzleti terület számítástechnikai képességeiért, a szolgáltatás üzemeltetéséért és minőségéért. Kapcsolatot tart az ügyfelekkel, és a visszajelzések alapján felállítja a szolgáltatásfejlesztési stratégiát. Döntési joga van a fejlesztési prioritások meghatározásánál, illetve az adott fejlesztés használatba vételéről.

Táblázat 12 - Szoftverfejlesztési szerepkörök

Eszközök

Ebben a fejezetben áttekintjük azokat az eszközöket, amelyeket teszteléskor, illetve a tesztkörnyezetek felépítése karbantartása folyamán használunk, vagy kapcsolódunk hozzájuk.

Tesztelési eszközök

A tesztelési eszközök segítségével a tesztek végrehajtása tervezhető, menedzselhető, az eredmények nyomon követhetőek, vagy egyes eszközöket a tesztelés végrehajtására használjuk.

A leggyakrabban használt tesztelési eszközök:

- **Tesztmenedzsment eszközök,** amikkel a tesztelések tervezését, a feladatok elosztását lehet elvégezni, illetve a feladatok végrehajtásának eredményeit tudjuk jelenteni illetve nyomon követni. Egyes ilyen eszközök az automatikus teszteszközök vezérlését is el tudják látni, és az eredményeket is feldolgozzák és láthatóvá teszik. Egyes eszközök a teljes életciklust lefedik, és komplett projektmenedzsment eszközként a követelmények kezelésétől, azok megfelelő kiadáshoz való hozzárendeléséig, a megfelelő tesztesetekkel való összekapcsolástól azok eredményéig nagyon sokrétű funkcionalitást tudnak lekezelni, akár a tesztkörnyezetekkel szemben támasztott követelményeket, és a hozzájuk kapcsolódó teendők menedzsmentjét is. Ezek az eszközök rendkívül sokrétűek, viszont a megfelelő használhatóságukhoz nagyfokú szabványosítás lehet szükséges a fejlesztési folyamatokban. Mivel ezek az eszközök egyéb eszközök vezérlését is elláthatják, gondoskodni kell a teszteszközök, és esetlegesen a tesztkörnyezetek kapcsolatáról, és a megfelelő vezérlő komponensek telepítéséről.

- **Automatikus tesztelő eszközök,** amikkel automatikusan lehet bizonyos teszteket lefuttatni. Ezeknek az eszközöknek speciális környezeti igényeik lehetnek, amelyeket figyelembe kell venni

a környezetek tervezésénél. Megfelelő automatikus tesztelési stratégia kialakítása szükséges lehet, amiben szerepelnek a megfelelő eszközök, és azok tesztkörnyezetekhez köthető igényei

- **Teljesítménytesztelő eszközök,** amelyekkel a megfelelő teljesítménytesztet végre lehet hajtani. Egy ilyen eszköz általában több komponensből állhat, amelyek megfelelő elhelyezéséről gondoskodni kell. Az eszköz további komponensek telepítését is megkövetelheti, amivel nyomon követhetőek a megfigyelt eszközök teljesítmény paraméterei. Magának a tesztelési módszernek is vannak követelményei a tesztkörnyezetekkel kapcsolatban (általában azonossági paraméterek szempontjából).

- **Hibakezelő rendszerek,** amiben a talált hibákat és azok állapotát tartjuk nyilván. Egyes tesztmenedzsment eszközök integráltan tartalmazzák ezt a modult. Az automatikus tesztelő eszközök integráltak lehetnek az eszközzel, és jelenthetik a hibát, amit találnak, ilyenkor gondoskodni kell a megfelelő kapcsolatról. Meg kell jegyezni, hogy bár az ötlet alapvetően vonzó, a tapasztalatok vegyesek az alkalmazással kapcsolatban. Előnye, hogy az eredmények gyorsan elérhetőek, és gyorsabban megkezdődhet a hibajavítás, azonban a hibajelentések minősége a tapasztalatok szerint elmarad attól, amit egy manuális átnézés eredményeként kapunk, és lassíthatja a fejlesztést. A hibakezelő rendszereket általában az adott folyamatok szerint megfelelően be kell állítani. A talált hibák nagyon értékes statisztikai alapadatok. Egyes elemek általában a szervezet szintjén meghatározottak, hogy megfelelő statisztikákat elő lehessen állítani, és a stratégiai céloknak megfelelő méréseket végre tudjuk hajtani, amíg más mezők nagyobb szabadságot élveznek a fejlesztés folyamán. Általában elmondható, hogy egyes környezeteket szabványos konvenciók alapján neveznek el, és azok használata kötelező, aminek egyik indoka pont a megfelelő statisztikák igénye.

Egyéb eszközök

Rengeteg egyéb eszköz is hasznos lehet ahhoz, hogy a projektet, és azon belül a tesztkörnyezeteket létrehozzuk és menedzseljük. Néhány ilyen a teljesség igénye nélkül:

- **Projekttervezési és -menedzselési eszközök,** amelyek segítenek abban, hogy a tesztkörnyezetekhez kapcsolódó tevékenységeket és erőforrásokat megtervezzük és kontrolláljuk, azok státuszát nyilvántartsuk, azt megosszuk.

- **Vállalatirányítási rendszer (ERP),** egy tesztkörnyezet felépítése több beszerzési folyamatot, és azok pénzügyi nyilvántartását is megköveteli. Ezeket a folyamatokat egy nagyobb szervezet esetén valamilyen szoftvertámogatással hajtjuk végre, ami gondoskodik a megfelelő tranzakciók végrehajtásáról, a megfelelő személyekhez való hozzárendeléséről és annak könyveléséről.

- **Konfigurációmenedzsment eszközök,** amelyek segítenek a megfelelő konfigurációk és konfigurációs elemek nyilvántartásában, az adott pillanatban összetartozó elemek azonosításában. Az ilyen rendszerek lehetnek komplett rendszerek, vagy részfeladatot ellátó eszközök, mint például a forráskódokat nyilvántartó rendszer, ami az alapja lehet a tesztkörnyezeteket ellátó automatikus buildkészítő rendszernek. Ezeket a részfeladatokat egyes esetekben integrálni kell a tesztkörnyezetekbe.

TÁBLÁZATOK JEGYZÉKE

ÁBRÁK JEGYZÉKE

FELHASZNÁLT SZAKIRODALOM

[1] Kapros Gábor, *Szoftvertesztelés egységesített kifejezéseinek gyűjteménye,* Hungarian Testing Board, http://www.masterfield.hu/docs/htb_glossary_hun.pdf, 2015.01.20

[2] Black, Rex: *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. New York: Wiley, 2007. - ISBN 978-0-470-12790-2 p. 240.

[3] James Bach, Patrick J. Schroeder: *Pairwise Testing: A Best Practice That Isn't,* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.3811&rep=rep1&type=pdf, 2015.04.27.

[4] Royce, Winston: , *Managing the Development of Large Software Systems,* http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf, 2015.02.21.

[5] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas: *Manifesto for Agile Software Development,* http://agilemanifesto.org/, 2015.01.19.

[6] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas: *Principles behind the Agile Manifesto,* http://agilemanifesto.org/principles.html, 2015.01.19.

[7] Agile Aliance, *What is Agile Software Development?,* http://www.agilealliance.org/the-alliance/what-is-agile/, 2015.01.19

[8] Ambler Scott: Agile *Testing and Quality Strategies: Discipline Over Rhetoric,* http://www.ambysoft.com/essays/agileTesting.html, 2015.01.19

[9] Elisabeth Hendrickson, *Agile Testing, Nine Principles and Six Concrete Practices for Testing on Agile Teams*, http://testobsessed.com/wp-content/uploads/2011/04/AgileTestingOverview.pdf, 2015.01.19

[10] Barry W. Boehm: *A Spiral Model of Software Development and Enhancement*, http://csse.usc.edu/csse/TECHRPTS/1988/usccse88-500/usccse88-500.pdf, 2015.01.19.

[11] IT Process Maps: *Release and Deployment Management*, http://wiki.en.it-processmaps.com/index.php/Release_and_Deployment_Management, 2015.02.11

[12] Wikipedia: *Configuration management*, http://en.wikipedia.org/wiki/Configuration_management, 2015.04.27.

[13] Software Engineering Institute (SEI), *A Framework for Software Product Line Practice, Version 5.0, Configuration Management,* http://www.sei.cmu.edu/productlines/frame_report/config.man.htm, 2015.04.27

[14] Cory Lueninghoener: *Getting Started with Configuration Management*, https://www.usenix.org/system/files/login/articles/105457-Lueninghoener.pdf, 2015.04.27

[15] TechNet: *Sybase/UNIX to SQL Server 2000: Database Engine Migration and Application Interoperation; Chapter 6 — Planning Phase: Building the Development*

and Test Environments, Microsoft, https://technet.microsoft.com/en-us/library/bb497048.aspx, 2015.01.20
[16] WIPRO: *Critical success factors for a successfull test environment management*, https://www.wipro.com/documents/critical-success-factors-for-a-successful-test-environment-management.pdf, 2015.04.10.
[17] David Lile Brown: *The Five Essentials for Software Testing*, http://www.isixsigma.com/industries/software-it/five-essentials-software-testing/, 2015.02.05.
[18] Sabin Pilipautanu: 5 rules to the road for test planning in Agile, http://www.3pillarglobal.com/insights/5-rules-to-the-road-for-test-planning-in-agile, 2012.02.01.
[19] Jerry Richardson, Sohail Thaker: *Lessons in Estimating,* http://www.ethier.ca/library_docs/lessons_in_estimating.pdf, 2015.03.01.
[20] Barry Boehm, Bredford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, Richard Selby: *Cost models for future software life cycle processes: COCOMO 2.0*, http://moosehead.cis.umassd.edu/cis365/reading/Cocomo2.pdf, p. 276 2015.02.15.
[21] Wikipedia: *Multitier architecture*, http://en.wikipedia.org/wiki/Multitier_architecture, 2015.03.11.
[22] Cal Evans: *Professional Programming: DTAP – Part 1: What is DTAP?,* http://www.phparch.com/2009/07/professional-programming-dtap-%E2%80%93-part-1-what-is-dtap/, 2015.01.19.
[23] Cal Evans: *Professional Programming: DTAP – Part 2 : Other moving Pieces,* http://www.phparch.com/2009/07/professional-programming-dtap-%E2%80%93-part-2-other-moving-pieces/, 2015.01.19.
[24] Wiggers, Steef-Jan, *DTAP Strategy: Pricing and Licensing*, http://soa-thoughts.blogspot.hu/2009/03/dtap-strategy-pricing-and-licensing.html, 2015.01.19
[25] Peter Murray: *Traditional Development/Integration/Staging/Production Practice for Software Development*, http://dltj.org/article/software-development-practice/, 2015.01.19
[26] Amir Shevat: *Effective Development Environments – Development, Test, Staging/Pre-prod and Production Environments,* http://spacebug.com/effective_development_environments/, 2015.01.19
[27] F. John Reh: *Pareto's Principle - The 80-20 Rule,* http://management.about.com/cs/generalmanagement/a/Pareto081202.htm, 2015.04.20.
[28] Henrik Kniberg, Mattias Skarin: *Kanban and Scrum making the most of both*, http://www.infoq.com/resource/minibooks/kanban-scrum-minibook/en/pdf/KanbanAndScrumInfoQVersionFINAL.pdf, 2015.04.07.
[29] Thraka: *How to Create and Deploy a Cloud Service,* http://azure.microsoft.com/hu-hu/documentation/articles/cloud-services-how-to-create-deploy/, 2015.04.02.
[30] Wikipedia: *Cloud computing*, http://en.wikipedia.org/wiki/Cloud_computing, 2015.04.02.
[31] Qusay F. Hassan: *Demystifying Cloud Computing*, http://static1.1.sqspcdn.com/static/f/702523/10181434/1294788395300/201101-Hassan.pdf?token=wjtAsV%2Fmx%2BEZb3PavlwftrvXKmY%3D, 2015.04.02.
[32] SalesForce: Development Lifecycle Guide Enterprise Development on the Force.com Platform,

http://www.salesforce.com/us/developer/docs/dev_lifecycle/salesforce_development_lifecycle.pdf, 2015.04.15.
[33] Neha Thakur: *Performance Testing in Cloud: A pragmatic approach*, http://www.diaspark.com/images/pdf/performance-testing-in-cloud-a-pragmatic-approach.pdf, 2015.04.15
[34] Salesforce: *Sandbox Overview*, https://help.salesforce.com/HTViewHelpDoc?id=create_test_instance.htm&language=en_US, 2015.04.15.
[35] Salesforce: *Creating or Refreshing a Sandbox*, https://help.salesforce.com/apex/HTViewHelpDoc?id=data_sandbox_create.htm, 2015.04.15.
[36] Kim-Mai Cutler: *How Do Top Android Developers QA Test Their Apps?*, http://techcrunch.com/2012/06/02/android-qa-testing-quality-assurance/, 2015.04.15.
[37] Daniel Knott: *How to choose the right mobile test devices*, https://devblog.xing.com/qa/how-to-choose-the-right-mobile-test-devices/, 2015.04.15.
[38] Bart Jacobs: *How to Test Your App on an iOS Device,* http://code.tutsplus.com/tutorials/how-to-test-your-app-on-an-ios-device--mobile-13861, 2015.04.15.
[39] Apple: *Which Developer Program is for you?*, https://developer.apple.com/programs/which-program/, 2015.04.15.
[40] Apple: *Launching Your App on Devices*, https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/LaunchingYourApponDevices/LaunchingYourApponDevices.html, 2015.04.15.
[41] Apple: *Beta Testing iOS Apps*, https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/TestingYouriOSApp/TestingYouriOSApp.html, 2015.04.15.
[42] Apple: *TestFlight Beta Testing (Optional)*, https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/BetaTestingTheApp.html#//apple_ref/doc/uid/TP40011225-CH35, 2015.04.15.
[43] Apple: *Managing Your Team in Member Center*, https://developer.apple.com/library/mac/documentation/IDEs/Conceptual/AppDistributionGuide/ManagingYourTeam/ManagingYourTeam.html, 2015.04.15
[44] Apple: *Managing Accounts,* https://developer.apple.com/library/mac/documentation/IDEs/Conceptual/AppDistributionGuide/ManagingAccounts/ManagingAccounts.html, 2015.04.15
[45] Apple: *Testing Your Game Center-Aware Game*, https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/GameKit_Guide/TestingYourGameCenter-AwareGame/TestingYourGameCenter-AwareGame.html, 2015.04.15.
[46] Apple: *About iTunes Connect*, https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/About.html, 2015.04.15.

[47] Apple: *Setting Up User Accounts*, https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/SettingUpUserAccounts.html, 2015.04.15
[48] Apple: *Testing In-App Purchase Products*, https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnectInAppPurchase_Guide/Chapters/TestingInAppPurchases.html, 2015.04.15.
[49] Apple: *Retrieving Product Information*, https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/StoreKitGuide/Chapters/ShowUI.html, 2015.04.15.
[50] Android: *Android Studio Overview*, http://developer.android.com/tools/studio/index.html, 2015.04.16.
[51] Android: *Using Hardware Devices*, http://developer.android.com/tools/device.html, 2015.04.16.
[52] Google: *Az alfa-/bétatesztelés és a fokozatos közzététel használata*, https://support.google.com/googleplay/android-developer/answer/3131213?hl=hu, 2015.04.16.
[53] Google: *Google Play developer console sign up,* https://play.google.com/apps/publish/signup/, 2015.04.16.
[54] Bernardo Zamora: *Options for publishing Windows Phone beta apps and testing in-app purchase*, http://blogs.windows.com/buildingapps/2013/11/27/options-for-publishing-windows-phone-beta-apps-and-testing-in-app-purchase/, 2015.04.16.
[55] MSDN: *How to deploy and run an app for Windows Phone 8*, https://msdn.microsoft.com/en-us/library/windows/apps/ff402565(v=vs.105).aspx, 2015.04.16.
[56] MSDN: *Registration Info*, https://msdn.microsoft.com/en-us/library/windows/apps/jj206719(v=vs.105).aspx, 2015.04.16.
[57] MSDN: *How to register your phone for development for Windows Phone 8*, https://msdn.microsoft.com/en-us/library/windows/apps/ff769508(v=vs.105).aspx, 2015.04.16,
[58] MSDN: Testing apps for Windows Phone 8, https://msdn.microsoft.com/en-us/library/windows/apps/jj247547(v=vs.105).aspx, 2015.04.16.
[59] MSDN: *Testing Windows Phone Apps*, https://msdn.microsoft.com/en-us/library/windows/apps/dn629256.aspx, 2015.04.16.
[60] MSDN: *Register your Windows Phone device for development*, https://msdn.microsoft.com/en-us/library/windows/apps/dn614128.aspx, 2015.04.16.
[61] Microsoft: *Register as an app developer*, http://dev.windows.com/en-us/join, 2015.04.16.
[62] MSDN: *App Developer Agreement*, https://msdn.microsoft.com/en-us/library/windows/apps/hh694058.aspx, 2015.04.16.
[63] MSDN: *Windows and Windows Phone Store Policies,* https://msdn.microsoft.com/en-us/library/windows/apps/dn764944.aspx, 2015.04.16.
[64] Global Certification Forum: *Homepage*, http://www.globalcertificationforum.org/, 2015.04.10.

[65] Enov8: *8 Dimensions of TEM (Test Enviornment Maturity),* http://enov8.com/docs/Enov8-TEMMi%20Lite-MAY-2014.pdf, 2015.04.06.
[66] Michael Diaz and Jeff King*, How CMM Impacts Quality, Productivity, Rework, and the Bottom Line*, http://static1.1.sqspcdn.com/static/f/702523/9339264/1289317444323/200203-Diaz.pdf?token=g4126PkrsRZ1TZ7A7ngvjtYMcjw%3D, 2014.12.11.
[67] CMMI Product Team, *CMMI® for Development, Version 1.3,* SEI, http://resources.sei.cmu.edu/asset_files/TechnicalReport/2010_005_001_15287.pdf, 2015.04.29.
[68] ITIL: *Homepage*, http://www.itil.org.uk/, 2015.04.29.
[69] Wikipedia: *Software versioning*, http://en.wikipedia.org/wiki/Software_versioning, 2015.04.29
[70] Boda Béla, Bodrogközi László, Gál Zoltán, Illés Péter: Szoftvertesztelés a gyakorlatban, tananyag, 2014.