JEGYZŐKÖNYV

Szoftvertesztelés

Féléves feladat

Készítette: **Józsa Róbert**

Neptunkód: **GDVXZW**

1. Készítsünk egy emberek neveit tároló programhoz egységtesztet.

Írjunk egy osztályt, amely emberek tárolására alkalmas. Legyen „add” metódus, amely által egy embert lehet hozzáadni. Legyen metódus, amely által egy embert lehet törölni, ha nincs benne a törlendő ember. Legyen metódusa, amely visszaadja az emberek számát. Legyen metódusa, amely visszaadja, hogy üres-e a tároló vagy sem. Legyen metódusa, amely kiüríti a tárolót. Írjunk egy teszt osztályt, amely az előzőleg implementált osztály metódusait teszteli. Minden teszt lefutása előtt legyen feltöltve a tároló emberekkel. Írjunk a lista méretének ellenőrzésére vonatkozó tesztesetet. Írjunk a lista ürességének ellenőrzésére vonatkozó tesztesetet. Írjunk új elem hozzáadására vonatkozó tesztesetet. Írjunk a lista tartalmát törlő metódusra vonatkozó tesztesetet. Írjunk olyan tesztesetet, amely nem létező ember töltésére vonatkozik. Írjunk olyan metódust, amely minden teszt után lefut és kiüríti a listát.

Az egységteszt

A tesztelés szintjei
A tesztelés szintjei a következők:
• komponensteszt,
• integrációs teszt,
• rendszerteszt,
• 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
1-3
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:
• unit-teszt,
• 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.

wiki

Az egységtesztek általában automatizált tesztek, amelyeket a szoftverfejlesztők írnak és futtatnak annak biztosítása érdekében, hogy egy adott alkalmazás adott szakasza (úgynevezett "egység") megfeleljen a kialakításának és az elvárt, kívánt módon viselkedjen. A procedurális programozásban az egység lehet egy teljes modul, de túlnyomórészt egyéni függvény vagy eljárás. Objektumorientált programozás során az egység gyakran egy teljes felület/interfész, például egy osztály, de lehet egyedi metódus is. Azáltal, hogyha a tesztek készítése először a legkisebb tesztelhető egységekre történik, majd az azok közötti összetett, egész viselkedésre, átfogó tesztek hozhatók létre komplex alkalmazásokhoz.

Az esetlegesen felmerülő problémák elkülönítése érdekében minden tesztesetet egymástól függetlenül, külön kell tesztelni. Egy modul izolált tesztelésének elősegítésére többek között az összetett, bonyolult viselkedéssel bíró objektumokat helyettesítő, ún. álcaobjektumok (mock objektumok) is használatosak.

A fejlesztés során a szoftverfejlesztő kritériumokat vagy ismert eredményeket is kódolhat a tesztbe azért, hogy ellenőrizze az adott egység helyességét. Az egyes tesztesetek végrehajtása során a keretrendszerek naplózzák azokat a teszteket, amelyek bármely kritériumot nem teljesítik és jelentést készítenek róluk egy összefoglalóban.

Az egységtesztek írása és fenntartása gyorsabban kivitelezhető, mint a paraméterezett tesztek esetén. Ezek lehetővé teszik egy teszt többszöri végrehajtását különböző bemeneti halmazokkal, csökkentve ezzel a tesztkód reprodukálását. A hagyományos egységtesztektől eltérően, amik általában zárt metódusok és invariáns körülményeket tesztelnek, a paraméterezett tesztek bármilyen paraméterkészletet felvehetnek. A paraméterezett teszteket támogatja a TestNG, JUnit és ennek .NET-beli párja, az xUnit is. Az egységteszthez tartozó megfelelő paraméterek manuálisan is megadhatók, vagy egyes esetekben a teszt keretrendszere automatikusan generálja őket. Az utóbbi években támogatták az erőteljesebb (egység) tesztek írását, az elméletek koncepciójának kiaknázását, az olyan teszteseteket, amelyek ugyanazokat a lépéseket hajtják végre, de futási idő alatt generált tesztadatokat használnak, ellentétben a szokásos paraméterezett teszttel, amely ugyanazokat a végrehajtási lépéseket használja olyan bemeneti halmazokkal, amelyek előre definiáltak.
Az egységteszt célja a program minden részének elkülönítése és annak igazolása, hogy az egyes részek helyesek. Az egységteszt szigorú, írásbeli „szerződést” biztosít, amelyet az adott kóddarabnak ki kell elégítenie. Ennek eredményeként az egységteszt számos előnyt kínál.

Az egységtesztelés a szoftverfejlesztési folyamat korai szakaszában találja meg a problémákat. Ez egyaránt magában foglalja mind a programozó végrehajtásának bugjait, mind az egység specifikációjának hibáit vagy annak hiányzó részeit. Az alapos, részletes tesztkészlet megírásának folyamata arra készteti a fejlesztőt, hogy gondolja át a bemeneteket, a kimeneteket és a hibafeltételeket, ezáltal így pontosabban meghatározza az egység kívánt viselkedését. A hibakeresés költsége a kódolás megkezdése vagy a kód első írása előtt jóval alacsonyabb, mint a hiba későbbi felismerésének, azonosításának és kijavításának költsége. A kiadott kódban szereplő hibák szintén költséges problémákat okozhatnak a szoftver végfelhasználói számára. A kódra lehetetlen vagy bonyolult egységtesztet írni, ha az rosszul van megírva, így az egységtesztelés arra készteti a fejlesztőket, hogy jobban strukturálják a funkciókat és az objektumokat.

A tesztvezérelt fejlesztés során (TDD), amelyet gyakran használnak mind az extrém programozásban, mind a scrumban, az egységteszteket a kód írása előtt hozzák létre. Amikor a tesztek sikeresek, az adott kódot teljesnek tekintik. Ha az egységteszt sikertelen, akkor bugnak tekintik vagy a megváltozott kódot, vagy magukat a teszteket is. Ezután az egységtesztek lehetővé teszik a hiba vagy a balsiker helyének könnyű nyomon követését. Mivel az egységtesztek azelőtt figyelmeztetik fejlesztői csoportot a problémára, mielőtt a kódot azt kiosztanák a tesztelőknek vagy az ügyfeleknek, a lehetséges problémák a fejlesztési folyamat korai szakaszában fedezhetők fel.

Az egységteszt lehetővé teszi a programozó számára a kódrefaktorálást vagy a rendszerkönyvtárak egy későbbi időpontban történő frissítését és a modul továbbra is helyes működéséről való megbizonyosodást(például regressziós teszteléskor). A művelet során tesztesetek írására kerül sor minden függvényre és metódusra azért, hogyha bármilyen változás hibát okozna, az gyorsan azonosítható legyen. Az egységtesztek olyan változásokat észlelnek és derítenek ki, amelyek megsérthetik a szerződésalapú tervezést.

Az egységtesztelés csökkentheti a bizonytalanságot magukban az egységekben és a lentről felfelé irányuló tesztelési stílusú (bottom-up testing style) megközelítésben használható. Ha először teszteljük a program részeit, majd teszteljük a részek összegét, az integrációs tesztelés sokkal könnyebbé válik.

Az egységtesztelés a rendszer egyfajta élő dokumentációját nyújtja. Azok a fejlesztők, akik meg akarják tanulni, hogy egy egység milyen funkcionalitást kínál és hogy hogyan kell ezt használni, megnézhetik az egységtesztelést, hogy alapvetően megismerjék az egység felületét (API).

Az egységteszt olyan jellemzőket testesít meg, amelyek kritikusak az egység sikeréhez. Ezek a jellemzők jelezhetik az egység megfelelő / nem megfelelő használatát, továbbá a negatív viselkedéseket, amelyek az egység által „csapdába" (trapped by the unit) estek. Egy egységteszt önmagában dokumentálja ezeket a kritikus jellemzőket, habár sok szoftverfejlesztő környezet nem kizárólag arra szolgál, hogy a fejlesztés alatt álló terméket dokumentálja a kód alapján.

Ha a szoftvert a tesztvezérelt megközelítés alkalmazásával fejlesztik, interfész meghatározására történő egységtesztek, valamint a teszt letelte után elvégzett kódrefaktorálási tevékenységek kombinációja léphet a hivatalos tervezés helyébe. Minden egységteszt tervezési elemnek tekinthető, amely meghatározza az osztályokat, metódusokat és a megfigyelhető viselkedést.
Korlátok és hátrányok
A tesztelés nem fog minden hibát elfogni a programban, mert nem tud minden végrehajtási útvonalat kiértékelni. Ez a döntési probléma (a megállási probléma (halting problem) szülő-halmaza, azaz, hogy az adott program befejezi-e a futását vagy örökre fut,) eldönthetetlen, mivel lehetnek olyan kódrészek, amikre nem írható olyan ellenőrző algoritmus, amely igaz-hamis válasszal szolgálna a kimenetében. Ugyanez igaz az egységtesztelésre is. Ezenkívül, a definíció szerint az egységteszt csak maguknak az egységeknek a funkcionalitását teszteli. Ebből kifolyólag nem fog elkapni integrációs hibákat vagy tágabb, rendszerszintű hibákat (például több egységen végrehajtott függvények vagy nem funkcionális tesztterületek, például a teljesítmény) Az egységtesztelést más szoftvertesztelési tevékenységekkel összefüggésben kell elvégezni, mivel ezek csak bizonyos hibák jelenlétét vagy hiányát mutatják; nem tudják bizonyítani az összes hiba hiányát. Ahhoz, hogy garantálva legyen a végrehajtás minden útjának és minden lehetséges bemenetnek a helyes viselkedése és biztosítva legyen a hibák hiánya, más technikákra is szükség van, nevezetesen formális módszerek alkalmazására annak igazolására, hogy a szoftverkomponensnek nincs váratlan viselkedése.

Egy egységtesztek alkotta bonyolult hierarchia nem egyezik meg az integrációs teszteléssel. A periferikus egységekkel való integráció bele kell, hogy tartozzon az integrációs tesztekbe, de az egységtesztekbe nem. Az integrációs tesztelés még mindig erősen támaszkodik az emberek általi manuális tesztelésre; ugyanis a magasszintű vagy globális körű tesztelést nehéz automatizálni, a manuális tesztelés gyakran könnyebben és olcsóbban kivitelezhetőbbnek tűnik.

Az egységteszt írásával kapcsolatos másik kihívás a reális és hasznos tesztek felállításának nehézsége. Szükséges megteremteni a megfelelő kezdeti, kiinduló feltételeket azért, hogy az alkalmazás tesztelt része a teljes rendszer részeként viselkedjen. Ha ezek a kezdeti feltételek nincsenek helyesen beállítva, akkor a teszt nem gyakorolja a kódot reális kontextusban, ami csökkenti az egységteszt értékét és pontosságát.

Az egységtesztelés előnyeinek eléréséhez szigorú fegyelemre van szükség az egész szoftverfejlesztési folyamat során. Alapvető fontosságú, hogy ne csak az elvégzett teszteket, hanem a szoftver ezen vagy bármely más egységének forráskódjában elvégzett változtatásokat is gondosan rögzítsük. Lényeges egy verziókezelő rendszer használata is. Ha az egység egy későbbi verziója elbukik egy olyan sajátos teszten, amin korábban átment, akkor a verziókezelő szoftver a forráskód-változtatások (ha vannak) listáját képes nyújtani, amelyek az egységre azóta vonatkoznak.

Scott Norberg: Miért értjük félre a unit tesztelést?

„Miért értjük félre a unit tesztelést?
A legtöbb fejlesztői csoport nem találja az arany középutat a unit tesztelésben, pedig van ilyen

A unit tesztelés helyes használata az egyik legjobb módszer lehetne a technológiai fejlesztések hatékonyságának növelésére.

A legtöbb fejlesztő azonban vagy túlzásba viszi a használatát, vagy teljesen kerüli.

Nehéz megtalálni tehát az arany középutat, ahol a tesztelés e formája még emeli a minőséget a termelékenység akadályozása nélkül. Azonban, ha sikerül megtalálni az egyensúlyt, az sokkal jobb minőségű szoftver előállítását teszi lehetővé jóval alacsonyabb költségek mellett.

Mit is jelent a unit tesztelés?

Mielőtt nagyon belevágnánk a témába, tisztázzuk, hogy mi is az a unit tesztelés, hiszen a fogalmat sokszor félremagyarázzák. Unit tesztelés alatt azt értjük, amikor a szoftvernek csak egy kisebb komponensét vagy elemét teszteljük.

Mivel minden egyes ilyen teszt működési területe igen behatárolt, az egyetlen járható út, ha írunk egy kódot a kód tesztelésére olyan keretrendszerek segítségével, mint a NUnit vagy a Microsoft Testing Framework. E keretrendszerek működési elvének részletes ismertetése nem témája ennek a cikknek, dióhéjban az unit tesztelés tehát csak annyi, hogy a fejlesztő különböző teszt metódusokat ír, amelyeket lefuttatva ellenőrizni tudja, hogy minden programegység megfelel-e a specifikációjának.

Egyszeri programozó: "Nem lehet csak úgy simán unit teszteket írni."

Elég zavarba ejtő tehát, amikor olyan programozók, akik nem ismerik az unit teszt keretrendszereket, az általuk végzett manuális tesztelést hívják így. Márpedig az nem unit tesztelés, csak tesztelés.

Mégis miért írjunk kódokat más kódok tesztelésére?

Egy olyan ember számára ugyanis, aki nem szoftverfejlesztő, ez elég értelmetlennek tűnhet.

Az automatizált tesztek nélküli szoftverfejlesztés olyan, mitha villával piszkálnák egy elektromos panelt:
"Nem értem, miért ne működhetne..."
Pár másodperccel később: "Áuuu... Ja, ezért"

De azok, akik ezt csinálják, könnyen belátják az előnyeit:

1. Egy rendszer átlagos tesztelése során be kell jelentkezni és egy sor előre meghatározott műveletet kell elvégezni annak érdekében, hogy a rendszer egy bizonyos funkcióját ellenőrizhessük. Ez egyáltalán nem hatékony és borzasztóan időigényes. A unit tesztelés tehát lehetőséget ad a fejlesztőnek arra, hogy célzott ellenőrzést végezzen a rendszer kérdéses területein.

2. Ha valami nem működik, a fejlesztő csapatnak nem kell átfésülni az egész rendszert a hiba megtalálásához, elég, ha lefuttatják a korábban megírt unit teszteket, leszűkítve ezzel a keresési területet.

3. Végül, de nem utolsósorban a kód meghatározott időközönkénti újratervezésével való javítása elengedhetetlen a rendszer hosszú távú fenntarthatósága érdekében.

Amikor azonban már túlzás

A szoftverfejlesztőkkel kapcsolatos tapasztalataim alapján azt vettem észre, hogy kissé, mintha végletekben gondolkodnának. Van helyes és helytelen. Ha a unit tesztek használata „helyes", akkor mindenhez, amit csinálunk, kell ilyen teszteket gyártani. Íme, két közhiedelem a témával kapcsolatban, amelyek több kárt okozhatnak a projektben, mint hasznot.

"Ha a tesztelők teszt eseteket írnak, hogy mások kódját teszteljék..., akkor ki teszteli a tesztelők teszt eseteit?"

A tesztvezérelt fejlesztés (TDD) mögött meghúzódó elképzelés az, hogy a unit teszteket a szoftver kifejlesztése előtt írják meg. Ezután úgy írják meg a szoftvert, hogy az a unit teszteken átmenjen. Ha pedig valamilyen funkciót módosítani kell, netalán hozzáadni, akkor először a teszteket változtatják meg, majd addig csiszolgatják a szoftvert, amíg az újra át nem megy mindegyiken.

Ez elméletben jól hangzik, gyakorlatilag viszont a legtöbb fejlesztő által írt kódoknak nincs is szüksége unit tesztelésre. Egy komplex üzleti logikát megvalósító programhoz feltétlenül kell, ám az egyszerű logikai egységekhez több idő teszteket írni, mint magát az egységet legyártani.

100%-os kódlefedettség

Egy, a fejlesztők által használt közös mérési rendszer a kódlefedettség, azaz, hogy a szoftverhez írt kód hány százalékán futtattak le unit tesztet. Sok fejlesztési vezető vélekedik úgy, hogy a 100%-os kódlefedettség feltétlenül szükséges, hiszen így biztosítható a szoftver megfelelő működése. Ugyanakkor az ilyen programokon nagyon nehéz módosításokat végrehajtani. Ha túl sok unit tesztet használunk, az a kód módosításakor sok fejtörést okozhat a fejlesztőknek a magas költségek miatt.

Akkor tehát hol van a határ?

Sajnos nincsen általános érvényű szabály arra, hogy mely unit teszteket érdemes legyártani és melyeket nem, de itt van néhány irányelv.

Érdemes tesztet írni:
· ha a metódus mögött meghúzódó logika annyira komplex, hogy szükséges egységenként is megvizsgálni, vajon tényleg működik-e.

· ha a kód egy bizonyos funkciója meghibásodik és a javítása egy percnél hosszabb időt vesz igénybe

· ha a teszt megírása kevesebb időt vesz igénybe, mint a program lefuttatása, a bejelentkezés vagy a modell újratervezése

Nem érdemes tesztet írni:
· ha bonyolult keretrendszereket kell készíteni vagy telepíteni azért, hogy az unit tesztek működjenek (pl. szimulált objektumok, függőségi befecskendezés (DI))

· ha a meghibásodott kódon lefuttatott unit teszt csak kis mértékben javít az egész program minőségén

· ha az egységtesztek karbantartása magasabb költségekkel jár, mint maga a szoftver karbantartása

Összefoglalva tehát, az unit tesztek funkciója az, hogy segítse a fejlesztőcsapatokat a költségek leszorításában, a tesztidő lerövidítésében, a regressziós tesztek lecsökkentésében és a karbantartás megkönnyítésében.

A unit tesztelés fontos folyamat, ha azt akarjuk, hogy a program sikeres legyen.

Azonban, azok a fejlesztők, akik túl sok unit tesztet tárolnak és kezelnek, sokkal inkább előidézik azokat a problémákat, amelyeket e tesztek megoldani voltak hivatottak."

Egy egységteszt a JUnit

A JUnit egy egységteszt-keretrendszer Java programozási nyelvhez. A tesztvezérelt fejlesztés (TDD) szabályai szerint ez annyit tesz, hogy a kód írásával párhuzamosan fejlesztjük a kódot tesztelő osztályokat is (ezek az egységtesztek). Ezeken egységtesztek karbantartására, csoportos futtatására szolgál ez a keretrendszer. A JUnit-teszteket gyakran a build folyamat részeként szokták beépíteni. Pl. napi buildek esetén ezek a tesztek is lefutnak. A release akkor hibátlan, ha az összes teszt hibátlanul lefut.

A JUnit a egységteszt-keretrendszerek családjába tartozik, melyet összességében xUnit-nak hívunk, amely eredeztethető a SUnitból.

JUnit keretrendszer fizikailag egy JAR fájlba van csomagolva. A keretrendszer osztályai következő csomag alatt található:

JUnit 3.8-as ill. korábbi verzióiban a junit.framework alatt találhatók
JUnit 4-es ill. későbbi verzióiban org.junit alatt találhatók