Budapesti Corvinus Egyetem
Gazdálkodástudományi Kar
Számítástudomány Tanszék

VERZIÓVÁLTÁS SZOLGÁLTATÁSFOLYTONOS ALKALMAZÁSOKBAN

Készítette: Oroszi Róbert
Gazdaságinformatikus szak
E-Business szakirány
2013

Konzulens:
Balogh Zoltán

Tartalomjegyzék

1. fejezet

Bevezetés

Az interneten böngészve a felhasználó gyakran észre sem veszi, de egy weboldalon történő kattintása között elképzelhető, hogy az adott rendszer kétszer is verziót váltott, azaz frissítette a rendszerét. De hogyan történik mindez? Hogyan lehetséges leállás nélkül naponta többször verziót váltani, akár földrajzilag elosztott rendszereket populálni? Ezekre a kérdésekre kíván a szakdolgozat válaszokat adni.
Bemutatásra kerül a continuous integration folyamat, ezekhez használt rendszerek, eszközök.
A dolgozat témája alapvetően a webes alkalmazásokat hivatott bemutatni, azonban kitér az asztali alkalmazásokra is, illetve figyelmet szentel a mobil alkalmazásoknak is (mind az iOS, Android és a Windows Phone platformoknak).

2. fejezet

Continuous Integration

> A Continuous Integration – azaz a folyamatos integráció – egy szoftver fejlesztési módszer melyben a fejlesztőcsapat tagjai az általuk írt kódot legalább napi rendszerességgel integrálják a korábbi fejlesztések közé, ez napi többszöri integrálást jelent. Minden új kód integrálása során automatizált tesztek ellenőrzik, hogy a rendszerbe való illesztés során okozott-e valamilyen hibát az új kódrészlet és ennek eredményeként a lehető leghamarabb visszajelzést ad az integráció eredményéről. *Continuous Integration (original version)* 2002

A szoftverfejlesztés során egy átlagos projektben számos fejleszto dolgozik együtt a kódbázis különbözo részein. A fejlesztők a szoftver egy lokális másolatán dolgoznak a saját gépükön. Mikor elkészülnek az adott feladattal (például lefuttatják már a smoke teszteket is), felmásolják a megváltoztatott forrásokat a közös szerverre. Azonban ez elott szükségképpen frissíteni kell a saját lokális példányukat, amit integrációnak nevezünk. Szélsoséges esetben sajnos a lokális és a központi, up-to-date másolat közötti különbség olyan nagy lehet, hogy jelentosen módosítani kényszerülnek az újonnan fejlesztett funkciókat. majd ezután következhet az újabb frissítés, és esetleg az újabb kényszerű változtatás. Ezen ördögi kör elkerülésére alkalmazott szoftverfejlesztési gyakorlat a folytonos integráció (continuous integration), amelynek jelentése, hogy a fejlesztés során minden fejleszto adott rendszerességgel végrehajtja az integrációs lépést, elkerülve ezzel, hogy túl nagy különbség alakuljon ki az egyes lokális másolatok között. Ez a megközelítés egyértelműsége ellenére a 2000-es évek elején született meg, azóta terjed és örvend töretlen népszerűségnek. Bár a folytonos integrációt szinte mindig összekapcsolják az automatikus fordítással („build automation”), maga az alapötlet nem követeli ezt meg. Folytonos integrációs gyakorlatnak tekintheto például egy egyszerű céges előírás, hogy minden reggel a munka megkezdése előtt a fejlesztok kötelesek frissíteni a lokális másolatukat, kikényszerítve ezzel az rendszeres integrációt. Tehát a folytonos integrációs szigorúan véve csak egy verziókezelo rendszer használatát követeli meg. Egy nagyon jó összefoglalást a témáról a [3] weboldalon kaphatunk, érdemes elolvasni! Azonban az integrációnak nyilvánvalóan része egy fordítás végrehajtása, amelyet automatizálva nagyban megkönnyíthetjük a fejlesztok dolgát. Ez történhet például úgy, hogy egy commit művelet végrehajtása a közös tárolóba elindítja a fordítási folyamatot. Ezen felül érdemes ezt a folyamatot továbbgondolni, és egy meglévo tesztkészlettel ellenorizni azt, hogy az újonnan fejlesztett funkciók nem rontottak-e el egy már korábban

is meglévő funkcionalitást. Tehát a fordítás végén automatikusan elindulhat egy regressziós tesztelési fázis is, amelynek végén egy riport generálódhat az egyes tesztek eredményeibol. A legtöbb folytonos integrációt megvalósító szerver támogatja egyedi szkriptek futtatását a fordítás elott illetve utána. Így lehetőség van például a telepítést automatizálni (ez az úgynevezett continuous deployment). Ennek segítségével a frissen fordított programverziót képesek vagyunk akár egy webszerverre is telepíteni, emberi beavatkozás nélkül. Látható, hogy a folytonos integrációt támogató szerverek segítségével lényegében egy munkafolyamatot tudunk összeállítani, amely a számunkra érdekes műveleteket végzi el teljesen autonóm módon. Egyes eszközök még akár bugtracking rendszerekkel is integrálhatóak (például Bamboo és Jira). Mi a mérésen a Jenkins Java alapú szervert fogjuk használni automatikus fordításra és tesztfuttatásra.

2.1. Jenkins

A Jenkins egy ingyenesen elérheto, nyílt forráskódú folytonos integráció támogató eszköz. Napjainkban az egyik legnépszerűbb ilyen szerver, ami támogatja a legtöbb verziókezelo rendszert (CVS, SVN, Git) és képes értelmezni Ant és Maven projekteket valamint tetszoleges shell scriptet illetve windows batch parancsfájlt. Tesztautomatizálást tekintve a Jenkins JUnit teszteket képes végrehajtani. Azonban a Jenkins architektúrája lehetové teszi a funkcionalitás kibővítését bővítmények segítségével, így integrálható számos egyéb tesztelő keretrendszerrel is. Természetesen a bővítményeken keresztül a Jenkins összeköttheto a népszerű bugtracking rendszerekkel is, de amennyiben valami saját megoldásra van szükségünk, nekünk is van lehetoségünk saját plugint fejleszteni. A mérés során mi ugyan nem fogunk saját Jenkins plugint fejleszteni, hanem csak egy munkafolyamat konfigurálását fogjuk elvégezni, de azért mérés elott érdemes megismerkedni a Jenkins weboldalával.

2.2. TDD

Test Driven Development

ok esetben alapos szemléletváltozás szükséges ahhoz, hogy a tesztvezérelt fejlesztést előnyeit ki tudja használni a fejlesztő. Legalább annyi támogatója van, mint ellenzője. Sajnos feltételezésekbe kell bocsátkozzak, de szerintem az ellenzők egy jó része nem értette meg mire is való a TDD, és sokszor a lelkes kezdők is hasonló

sorsra jutnak. A TDD kulcsszava a tervezés nem pedig a tesztelés. Járjuk körül egy kicsit, hogy miért is létezik ez az egész, hogy végül (reményeim szerint) használható alapot adjak azok kezébe, akik megpróbálkoznak a TDD-vel. Ez természetesen azzal jár, hogy néhány vitatott kérdésben elkötelezzem magam valamelyik oldal mellett.

Jól működő, könnyen bővíthető alkalmazás, szolgáltatás, stb. kell ahhoz, hogy a cég tulajdonosai élvezhessék a szoftverfejlesztés által megvalósuló (jól megérdemelt) hasznot. Ehhez biztosítani kell a létrehozott szoftver minőségét. Egyik ilyen módszer a tesztvezérlet fejlesztés. Minek és milyen minőségét fogja biztosítani az szerepeljen a következő sorokban, de előtte még oszlassunk el néhány tévhitet.

Bizonyos szabályokat be kell vezetni ahhoz, hogy arra használjuk a TDD-t amire kell. A TDD egy fejlesztési módszer, nem pedig a szoftver tesztelésére használható eszköz. A módszer alkalmazásához teszteket is kell írnunk, azok pedig egység teszek (Unit test) lesznek. Mindenhol futtatható kell hogy legyen. Munkahelyi gépeden, otthoni gépeden, vonaton, Hősök terén. Bármilyen sorrendben lehessen futtatni a különböző teszteket. Külső függőségek nélkül, mint például hálózat, vagy fájlrendszer. A teszteseteknek csak akkor kéne változniuk, ha a kódnak is változnia kell. Nem minden rendszer esetében használható ez a módszer.

A fejlesztési folyamat négy lépésre bontható:

Feladatok meghatározása: Ebben a lépésben az ügyfél igényeknek megfelelő funkcionalitást kell feladat meghatározássá alakítani teszt esetek formájában. Meg kell határozni, hogy mit kellene tennie a rendszernek. Fontos, hogy nem azt kell itt kitalálni, hogy az adott fejlesztő hogyan valósítsa meg az adott feladatot, hanem hogy mit kell majd megvalósítani. A mit meghatározása által a fejlesztő is jobban megérti a feladatot, kisebb a félreértés lehetősége. A teszteset megírása után következik a megvalósítás.

Megvalósítás: Ezután következik a hogyan valósítsuk meg az előre definiált feladatot. Azután meg is kell valósítani. A megvalósításnak kéne a legkönnyebb résznek lennie, ha mégsem könnyű akkor a következő problémák fordulhattak elő. Túl nagy feladatot határoztunk meg az előző lépésben, így az a feladat, hogy kisebb feladatokra bontsuk. Felmerül a kérdés, hogyan lehet könnyű a megvalósítás, ha előtte még egy libet is el kell előtte készíteni? Ilyenkor az adott lib feladatait kell meghatározni úgy, hogy a fejlesztők lesznek az ügyfelek és a ő igényeiket kell kielégíteni, és TDD alapján lefejleszteni. Ha nem nagy lépésről van szó de mégis nehéz megvalósítani, akkor refaktorálni kell az adott részt. Ez lehet azért mert koszos a kód, vagy nincs jól megtervezve, stb. (A refaktorálás viszont egy külön történet, melyet meghagyunk

egyelőre más blogoknak.)

Ellenőrzés: A megfelelő eszközzel le kell ellenőrizni, hogy sikeresek lettek-e a tesztek. Ennek gyorsnak és könnyűnek kell lennie. (A tesztek nem futhatnak néhány mp-nél lassabban, akkor már nagy a gond, ha több idő kell.) Ezáltal folyamatosan ellenőrzött, tervezett és fejlesztett lesz a kódunk.

Tisztítás: Ha elértük, hogy sikeresen lefussanak a tesztek, akkor jön a kód tisztítása. A működő kódot átnézzük, a duplikációkat eltüntetjük. Beszédesebb neveket választhatunk a változóinknak. Mivel az ügyfél számára fontos funkcionalitás már le van tesztelve, ezért bátran megtehetjük mindezt, ugyanis ha például elgépeljük valahol az új változónevet, akkor rögtön jelez a teszt, hogy hiba van. Esetleg túl nagy lett a függvény, akkor több kisebb függvényre szétbonthatjuk, stb. Ettől tisztább és karbantarthatóbb lesz a kód.

Figyeljük meg, hogy az összes lépés megfeleltethető a tervezés egy-egy részének. A meghatározott feladatok automatizálásának köszönhetően lehetséges, hogy kis biztonságos lépésekben történjen a rendszer felépítése és tervezése. Segíti a feladat megértését, rákényszerít a könnyű megvalósíthatóságra a folyamatos tisztítás és újratervezés segítségével.

TDD előnyei: - Refaktorálást segíti. - Bátrabban módosítjuk az elkészült kódot. - Könnyebb egy tesztelhető kódot refaktorálni (a tervezés miatt). - Megmutatja, hogy hol hasalt el a kód. - Az is segítség lehet, hogy hol nincs hiba. - Segít tesztelhetővé tenni az alkalmazást. - Rákényszerít, hogy egy függvény ne foglalkozzon sok mindennel. - Csak olyan kódot írsz, ami a teszthez kell. - Előre kell tervezni. - Elvileg jobban is lesz ettől megtervezve. :) - Gyors, folyamatos visszajezést kapsz a funkció állapotáról. - Az utolsó változtatásodnál ha érintett egy régebben megírt függvény, akkor azt pontosan jelzi. - Jobban ellenőrizhető a munka. - Van, hogy egy funkciónak nincs látható eredménye egy hétig. Ezzel szemben a TDD-nél naponta meg tudod mondani, hogy mennyi sikeres tesztet tudtál írni. - Segít megérteni a feladatot a példákon keresztül. - Időcsökkentő tényező a hibajavításnál és a refaktorálásnál. - Hibajavításnál segíthet pontosabban megjelölni a hiba helyét. - Akár dokumentációként is szolgálhat a teszt. Példakódnak tekinthető. (Python doctest erre az elképzelésre épül.) - Biztosítja, hogy az új kód nem érint más tesztelt egységet. - Vannak emberek akiknek ez lelkisegély. - Örül, hogy de jó, sikeresen lefutott a teszt. - Bizonytalanoknak mentsvár. :) - Vannak akik ettől lesznek motiváltak és büszkék lesznek a munkájukra. - Vitatkoznak arról, hogy melyik a hatékonyabb módszer, a TDD vagy a kód felülvizsgálat. - Szerintem a kettő együtt hatékony, egyik nem he-

lyettesítheti a másikat. - Ha nincs TDD, akkor gyorsabban kész lesz, de nehezebben módosítható később. - A TDD tisztítás része pl. kódfelülvizsgálatnak is tekinthető. - Stabilitást elősegítheti.

Hátrányok: - Nő a fejlesztési idő. (Refaktornál csökken.) - Bevezetésekor drasztikusan. - Ha nem tiszta, hogy mit kell tenni az adott feladattal könnyen írhatunk rossz tesztet, amit át kell majd írni. (Újabb időnövelő tényező.) - Menet közben történt koncepcióváltásnál (létezik ilyen), könnyen kukába mehet az egész. (Újabb időnövelő tényező.) - Sokan azt hiszik ettől hibamentes lesz a program működése. - Nem csodafegyver. A rendszer tesztelésének (minőségbiztosításának) csak egy kis részét kéne, hogy képezze. (Accaptance test, integration test stb. mellett) - A unit test csak annyit jelent, hogy önálló kis részek jól működnek. - Csak tapasztalt fejlesztőkkel érdemes használni. - A tervezést nem mindig lehet úgy alakítani, hogy az megfeleljen a TDD-nek. - Adatstruktúrák (közvetett módon) és b̈lack box algoritmusokẗesztelésére jó. - Hálózattal, fájlrendszerrel kapcsolatos dolgok kilőve. - Páros programozásban lenne érdemes használni. - A tesztek írása unalmas lehet egyesek számára. Nagy fegyelemre lenne szükség. - Nehéz belerázódni, ezért arra a következtetésre jutnak, hogy semmi értelme. - Nehezebben történik meg, hogy egy feladat megoldásának hevében megírod az összes feladatot egy teszt segítségével. :) - TDD-ben tapasztalt párral lenne a legideálisabb. - A páros programozás előnyeit pedig még nehezebben lehet megértetni a managerekkel. :) - Nagy lelkesedés szükséges az elején. - Ha írsz hat tonnányi tesztet, amiről kiderül, hogy semmit nem ér akkor biztosan szükség lesz rá. - Nehéz megmagyarázni a managereknek, hogy az elején miért tart ennyi ideig a fejlesztés.

2.3. QA

(Software) Quality Assurance

Software quality assurance (SQA) consists of a means of monitoring the software engineering processes and methods used to ensure quality.[citation needed] The methods by which this is accomplished are many and varied, and may include ensuring conformance to one or more standards, such as ISO 9000 or a model such as CMMI. SQA encompasses the entire software development process, which includes processes such as requirements definition, software design, coding, source code control, code reviews, change management, configuration management, testing, release management, and product integration. SQA is organized into goals, commit-

ments, abilities, activities, measurements, and verifications.[1] The American Society for Quality offers a Certified Software Quality Engineer (CSQE) certification with exams held a minimum of twice a year.

2.4. Webes alkalmazások

A webes alkalmazásoknál a folyamatos integráció több feladatot is ellát:

- tesztek futtatása (unit, tdd, bdd, acceptance, integration)
- a fő verzióba való olvasztás
- adatbázis migrációs fájlok létrehozása
- kliensoldali statikus tartalmak konkatenálása, minimalizálása

A közösségi kódmegosztó github integrációs folyamata teljesen megfelel a felsorolásnak (Douglas, 2012). A különbség mindössze annyi, hogy miután sikeresek a continuous integration lépései egyből élesítésre kerülnek, azaz a fejlesztők a felelősek, ha valami hibát vétenek egy-egy funkció implementálásában.

Viszont a Facebook a saját PHP-ban írt alkalmazásának performancia javításának céljából további feladatokat is végez a folyamatos integrálás során, ez pedig a buildelés. A buildelés során a Facebook saját kódbázisát transzformálja amelynek köszönhetően hatszoros gyorsulást sikerült elérniük (a PHP kódot optimalizált C kódra alakítják, az átalakítás során használt szoftver ingyenes elérhető). Paul, 2012

Tesztek futtatása

A tesztek futtatása a webes rendszerekben ugyanolyan fontos, mint bármelyik másik platformon. Viszont míg az asztali és mobil alkalmazásoknál, ismerhető a kliensek rendszer tulajdonságai (például ha egy alkalmazás csak Windows 7 operációs, akkor tudni lehet az azon elérhető futtathatósági lehetőségeket), azonban a webes alkalmazásoknál az nem csak különböző operációs rendszerekre kell optimalizálni, hanem az eltérő böngészőkre (melyek a lehető legkülönbözőbb módon implementálták az egyébként is laza HTML és ECMAScript szabványokat) és azok különböző verzióra. Ezeknek az okoknak köszönhetően a tesztelés, illetve a tesztautomatizálás rendkívül fontos a webes alkalmazásoknál (Johansen, 2010).

Kliensoldali tartalmak

A kliensoldali tartalmak konkatenálása és minimalizálás rendkívül fontos webes alkalmazásoknál, ugyanis ezek a statikus tartalmak - a HTML mellett - az alkalmazás minden betöltésekor letöltésre kerülnek, ami pedig sok különálló fájl esetén a felhasználó élmény rovására mehet.

2.5. Asztali alkalmazások

Roben, 2012

2.6. Okostelefon alkalmazások

Chow, 2012

2.6.1. iOS

Ervine, 2011

2.6.2. Android

Bael, 2012

Felhasznált irodalom

Bael Steven Van (árp. 2012). *Building an Android App With Jenkins*. Letöltve: 2012. december 22. URL: http://androiddevresources.com/blog/2012/04/01/building-an-android-app-with-jenkins/.

Chow Joseph (2012). *LinkedIn: Continuous Integration for Mobile*. Letöltve: 2012. december 28. URL: http://engineering.linkedin.com/testing/continuous-integration-mobile.

Continuous Integration (original version) (2002). Letöltve: 2012. december 25. URL: http://martinfowler.com/articles/originalContinuousIntegration.html.

Douglas Jake (aug. 2012). *Deploying at GitHub*. Letöltve: 2012. november 20. URL: https://github.com/blog/1241-deploying-at-github.

Ervine Shaun (jún. 2011). *Continuous Deployment of iOS Apps with Jenkins and TestFlight*. Letöltve: 2012. december 20. URL: http://blog.shinetech.com/2011/06/23/ci-with-jenkins-for-ios-apps-build-distribution-via-testflightapp-tutorial/.

Johansen Christian (2010). *Test-Driven JavaScript Development*. Developer's Library.

Paul Ryan (árp. 2012). *Exclusive: a behind-the-scenes look at Facebook release engineering*. Letöltve: 2012. december 27.
URL: http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/3/.

Roben Adam (szept. 2012). *How we ship GitHub for Windows*. Letöltve: 2012. november 30. URL: https://github.com/blog/1271-how-we-ship-github-for-windows.