Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék
Hibatűrő Rendszerek Kutatócsoport

Szolgáltatásbiztonságra tervezés laboratórium
2015/2015 őszi félév

Automatikus tesztfuttatás

Mérési segédlet
(v1.1.2)

Készítette: Oláh János
(janos.olah@mit.bme.hu)
Frissítette: Ujhelyi Zoltán
(ujhelyiz@mit.bme.hu)
October 29, 2014

1 Szoftvertesztelés

A szoftvertesztelésnek nevezzük egy szoftver vagy egy szoftver komponens kiértékelésének **folyamatát**, mely során **kontrollált** körülmények között **figyeljük meg** a szoftver viselkedését **működés közben**. A tesztelés célja annak megállapítása, hogy a szoftverünk az adott fejlesztési életciklusban megfelel-e az általunk (illetve az összes érintett által) támasztott követelményeknek. Ennek érdekében a megfigyeléseket egy **teszt orákulummal (test oracle)** értékeljük (egyszerű esetben összehasonlítjuk a specifikációban rögzített elvárt viselkedéssel), így állapítva meg a szoftver megfelelőségét[1].

A tesztelés napjainkban is a szoftver verifikáció és validáció egyik alapvető technikája. Az általunk használt definíció szerint a tesztelés kizárólag **dinamikus technikákat** alkalmaz, tehát a szoftver **végrehajtását igényli**. A V&V tevékenységek[2] közé azonban további, statikus technikákat is felsorolhatunk, mint például a statikus kódellenőrzés vagy fejlesztői kód review.[3]

Léteznek egyéb, sokszor a tesztelést helyettesítő módszerként emlegetett verifikációs és validációs technikák, mint például a **formális verifikáció**. Ennek során precíz matematikai modellt készítünk a rendszerünkről, és ezen matematikai eszközöket felhasználva bizonyítjuk annak hibamentességét. A módszer előnye, hogy így valóban képesek vagyunk minden kétséget kizáróan *garantálni a hibamentességet* (és kódgenerálás esetén szükségtelenné tenni a tesztelést), míg teszteléssel csak a hibák jelenlétét tudjuk kimutatni, azok hiányát nem (hiszen a teljes tesztelés lehetetlen), illetve legtöbbször a hibák pontos helyét sem tudjuk teszteléssel megállapítani. A formális verifikáció komoly hátrány azonban, hogy ennek alkalmazása nagyon sok valós helyzetben gyakorlatilag lehetetlen, de legalábbis gazdaságilag elképzelhetetlen. Formális verifikációt főként kritikus beágyazott rendszerek esetén szokás alkalmazni (ahol ezek alkalmazását sok esetben a hatósági szabályozások megkövetelik), illetve protokollok verifikációjánál. Ezzel szemben a tesztelés **nagyon hatékonyan** képes felfedni a hibák esetleges jelenlétét.

1.1 A tesztelés lépései

A tesztelés három részfolyamat különíthető el markánsan, melyeket az alábbiakban röviden összefoglalunk.

Teszttervezés A tesztelés tervezési fázisa valójában már a követelményanalízis során elkezdődik. A tervezés elején meg kell határozni egy stratégiát melyet követni fogunk a tesztelés során, illetve létre kell hozni egy olyan platformot, melyen végre lehet majd hajtani a teszteket (bár sok esetben erre nincs is külön szükség). Ezen ismeretek birtokában készíthető egy részletes terv, mely leírja a tesztelés munkafolyamatát minden részletre kiterjedően. Ezt követően már a lehetőség van a tesztesetek elkészítésére, mely ennek a részfolyamatnak az elsődleges terméke.

Tesztvégrehajtás A tervezés során sok esetben nagyon nagy számú teszteset áll elő, melyek közül a végrehajtás elején ki kell választanunk egy számunkra megfelelő készletet, így

[1] A tesztelésnek számos definíciója létezik (IEEE, ISTQB, stb.), melyek olykor jelentősen eltérnek, vagy akár ellent is mondanak egymásnak. Ezen felül a különböző szakterületek művelői (beágyazott rendszerek, SOA, minőségmenedzsment) számára a szoftvertesztelés fókusza más és más. Ezért is mindig fontos előre tisztázni, hogy mit értünk tesztelés alatt az adott kontextusban.

[2] Sajnos a V&V mint folyamat meghatározása sem kevésbé homályos, mint a szoftvertesztelésé. Sok esetben a V&V folyamatokat tekintik komplett minőségbiztosításnak, ami legalábbis vitatható. Minőségbiztosítás során számos, a szoftver **fejlesztése során** alkalmazott technikát is figyelembe kell venni, melynek semmi köze a V&V célkitűzéséhez (például kódmetrikák, belső dokumentáltság, bugtracking rendszer fejlettsége, szabványoknak való megfelelés, stb.).

[3] Egyes szoftvertesztelési definíciók még ezeket a technikákat is a teszteléshez kapcsolják.

például minimalizálva a futtatott tesztesetek számát, költséget csökkentve ezzel. Ez a feladat általában egy nagyon nehezen eldönthető problémát eredményez, ahol komoly optimalizációs eljárásokat szükséges bevetni. A tesztvégrehajtás során végül a kapott teszteseteket hajtjuk végre, és dokumentáljuk az egyes futtatások eredményét.

Tesztkiértékelés Bár egy-egy futtatás eredménye a legtöbb esetben közvetlenül a futtatás során eldől, el szoktunk különíteni egy kiértékelési fázist, mely során összesítjük a kapott eredményeket (adott esetben felül is bírálhatjuk azokat), és jelentéseket készítünk a menedzsment számára, illetve döntünk a arról, hogy milyen intézkedésekre van szükség.

1.2 Tesztelés csoportosítása

Mint az előzőekből is látható, a szoftvertesztelés nem egyértelmű fogalom, így várhatóan a széles körben elterjed csoportosítások sem feltétlenül jól definiált, jól elkülöníthető fogalmak. Itt a mérési segédletben igyekszünk csak a legfontosabb, a mérés szempontjából is releváns csoportosításokat és azok jellemzőit felsorolni.

1.2.1 Csoportosítás a rendelkezésre álló információ alapján - a doboz megközelítés

A tesztelési módszerek talán legismertebb csoportosítása a szerint történik, hogy a tesztelőnek mennyi információ áll rendelkezésére a tesztelendő rendszererről (SUT - System Under Test).

Fehér doboz - Strukturális tesztelés Strukturális tesztelés esetén a tesztelőnek teljes hozzáférése van a SUT belső szerkezetéhez, az adatstruktúrákhoz, algoritmusokhoz, vagyis a forráskódhoz úgy általában. A teszttervezés folyamatát ebben az esetben ez az ismeret vezérli, vagyis úgy tervezünk, hogy bizonyos strukturális elemeket veszünk figyelembe és szeretnénk lefedni (például a jól ismert döntés lefedettség).

Fekete doboz - Funkcionális tesztelés Funkcionális tesztelés esetén semmilyen információ nem áll rendelkezésre a tesztelendő szoftver belső struktúrájáról sem az alkalmazott algoritmusokról. A teszttervezést teljes egészében a követelmény- illetve specifikáció leírások vezérlik, vagyis úgy állítjuk elő a teszteseteinket, hogy végrehajtásukkal a lehető legjobban megbizonyosodjunk arról, hogy a SUT megfelel a specifikációknak.

Néha szokás használni egy harmadik csoportosítási fajtát is, az ún. szürkedoboz tesztelést. Ahogy a nevéből is látszik, itt valamiféle limitált információ áll rendelkezésre, például a SUT architektúrája vagy egyes algoritmusok, de például a forráskód jellemzően nem elérhető.

1.2.2 Csoportosítás a tesztelés szintje alapján

Következő csoportosítás arra fókuszál, hogy a szoftver, mint végső termék mely szintjén hajtjuk végre a tesztelést.

Modultesztelés A modultesztelés (unit testing) során a tesztelendő szoftver egy specifikus részét, egy logikailag jól elkülöníthető modulját teszteljük egy egységként kezelve. Például elképzelhető, hogy objektum-orientált tesztelés során egy osztály tesztelését szeretnénk külön elvégezni.

A unit tesztelés előnye, hogy egyszerűen elkészíthető és végrehajtható, azonban általában problémát okoz a függőségek kezelése (például tesztcsonkok szükségesek az együttműködő modulok helyettesítéséhez).

Integrációs tesztelés Integrációs tesztelés esetén már több modulunk van, melyek együttműködését szeretnénk leteszteltni. Ilyenkor az integráció teszteléshez készült tesztesetek a modulok kapcsolódási interfészeire fókuszálnak. Attól még, hogy modul szinten alaposan teszteltük az egyes modulokat, még lehet hibás az együttműködés!

A különböző komponensek integrációját rendszerint iteratív módon szokás elvégezni, és minden iteráció során végrehajthatjuk a tesztjeinket. A másik lehetőség az úgynevezett "Big Bang", vagyis az összes modul egyszerre történő összeállítása - mely kis rendszerek esetén indokolható ugyan, azonban általában rossz szoftvermérnöki gyakorlat, kiváltképp tesztelési szempontból.

Rendszertesztelés Rendszertesztelés során a teljes elkészült szoftvert mint terméket vizsgáljuk (esetleg a hardver-szoftver integrációt is), jellemzően már a megrendelő követelményei alapján. Rendszertesztelés során figyelembe kell venni a várható felhasználói profilokat, és meg kell állapítani a rendszer alkalmazhatósági korlátait.

1.2.3 Csoportosítás a tesztelés célja alapján

Végül megkülönböztethetjük a szoftvertesztelést aszerint, hogy mi a célja az adott teszt futtatásának.

Regressziós tesztelés Regressziós tesztelés során az a cél, hogy kiszűrjük az olyan hibákat, melyeket egy vagy több komponens változtatása vezethetett be egy rendszerbe. Itt tehát nem készítünk új teszteket, hanem a már meglévőeket futtatjuk le újra. Például ha két külön fejlesztett komponenst integrációja során az integrációs teszteken túl lefuttatjuk a korábban, a moduloknál használt unit tesztjeinket is, akkor tulajdonképpen regressziós tesztelést végzünk.

Jellemzően azért ennél nagyobb méretben szokás gondolkodni, és a regressziós tesztelést arra a folyamatra alkalmazzák, mikor elkészül egy szoftverből egy új "build" (például mikor egy fejlesztő commitol a közös SVN-be), és egy meglévő tesztkészlet lefuttatásával ellenőrzik, hogy nem romlott-e el semmi az előzőhöz képest. A mérés során mi is regressziós tesztelést fogunk fejleszteni.

Smoke tesztelés Tulajdonképpen a regressziós tesztelés egy egyszerűbb formája, mely során azt vizsgáljuk, hogy az aktuális szoftverváltozat "életképes-e" egyáltalán. Ez még jellemzően a fejlesztő saját verzióján hajtódig végre, tehát például commit művelet előtt tesztelik le a legfőbb funkciók működését (például elindul-e a szoftver, stb.). Többnyire a fejlesztő hatja végre.

Elfogadás tesztelés Rendszerint a legutolsó tesztelési tevékenység, és sok esetben a megrendelő saját tesztelői, vagy egy kiválasztott független szervezet hajtja végre. A cél annak kiderítése, hogy az elkészült termék valóban megfelel-e megrendelő elvárásainak. Voltaképpen ez a végső validáció: "a jó terméket építjük?"

1.2.4 További megjegyzésk

Sokszor alkalmazzák az **parafunkcionális tesztelés** (vagy a még homályosabb nemfunkcionális tesztelés) gyűjtőfogalmat, mint a tesztek egy külön kategóriáját.

Funkcionális tesztelés esetén a tesztelendő rendszert "függvényként" kezeljük, vagyis úgy tekintjük, hogy az egy adott bementi adatot transzformál át kimeneti adattá. Vagyis a funkcionális teszteseteink egyértelmű leképzést jelentenek, melyek az explicit megrendelői követelmények

alapján állnak elő. Ezzel szemben a parafunkcionális tesztelés az implicit követelmények teljesülését vizsgálja.

Általában ide sorolják a teljesítménytesztelést, a használhatóság tesztelését, a biztonság tesztelését, a karbantarthatóság tesztelését, lokalizációs tesztelést vagy sok esetben a robosztusságtesztelést és a hibainjektálást is. Látható, hogy ezek olyan mértékben eltérő fókuszú és költségigényű tesztelési feladatok, mely értelmetlenné teszi ennek a közös kategóriának a használatát.

1.3 Tesztelési termékek

Ebben a fejezetben röviden felsoroljuk a legfontosabb tesztelési termékeket (testing artifact), melyet tisztában kell lenni a tesztelési folyamat végrehajtásához.

Teszterv (Test plan) Tulajdonképpen egy magas szintű munkafolyamat-leírás. Tartalmazza az elvégzendő teszteket, a szükséges dokumentumokat, archiválási előírásokat, felelősségi köröket, stb.

Teszteset (Test case) Egy teszteset minden információt tartalmaz, mely alapján egy tesztet végre tudunk hajtani és kiértékelni. Tartalmazza a bemeneti adatokat és a kiindulási feltételeket (például rendszerállapotokat, akár csak implicit módon kezdeti lépések formájában, ahogyan elérhetünk a kívánt rendszerállapotba), illetve a kimeneti adatokat és feltételeket.

A teszteset mint dokumentum lehet egy egyszerű szövegfájl egyedi azonosítóval ellátva, melyben szerepel a teszteset kategóriája, a teszteset verziószáma, készítőjének neve, de akár helyet kaphat benne a konkrét futtatások eredményének a leírása is.

Tesztadat (Test data) Az előzőek alapján a tesztadat a teszteset egy része. Nem tartalmazza a szükséges rendszerállapotot, sem az elvárt kimeneti adatot. Csak a tesztadat alapján tehát nem lehetséges tesztelést futtatni. Megkülönböztetése azért indokolt mégis, mert csak tesztadatot sok esetben képesek vagyunk automatikusan generálni.

Teszt orákulum (Test oracle) A teszt orákulum feladata az, hogy eldöntse egy teszteset sikeresen lefutott-e vagy sem. Tehát például amennyiben specifikáció alapján tesztelünk, tudjuk, hogy egy adott bemeneti adatra milyen viselkedést várunk el - vagyis az elvárt viselkedés lesz az orákulum. Ez a döntés sok esetben nem egyértelmű. Ezért általában az orákulumba beleértjük azt a mechanizmust is, mely képes végrehajtani a kiértékelést.

Teszt szkript (Test script) A teszt szkript már eszközfüggő, általában a használt tesztelési keretrendszer saját nyelvén íródik (például Selenium esetén selenese parancsokból 2.1). Ez már értelmezhető a környezet számámra, így az végre tudja hajtani. Minden esetben a teszteset alapján készül.

Tesztkészlet (Test suite) A tesztesetek egy csoportját tesztkészletnek nevezzük. Általában érdemes azonos teszteseteket egy készletbe összefogni, tehát például azokat, melyeknek azonos előkövetelményeik vannak. Egy tesztkészlet tartalmazhat további információkat is, mint például a rendszerkonfigurációs leírása.

2 Automatikus tesztfuttatás

A 1.1 fejezetben felsoroltuk a szoftvertesztelés három legfontosabb lépését. Mindhárom lépés esetén érdemes szót ejteni a lehetséges automatizálásról.

A **teszttervezés** során addig kell eljutnunk, hogy rendelkezésünkre álljanak a tesztesetek, illetve egy részletes terv a végrehajtásról mikéntjéről, jellemzően egy munkafolyamat formájában. E lépés leírásából is érezhető, hogy nem egy triviális feladatról van itt szó, itt igenis szükség van az emberi intelligenciára. Ennek a lépésnek az automatizálására rengeteg törekvés, ötlet leírás található az irodalomban (elég rákeresni a "test case generation" szóösszetételre), azonban iparban is elterjed, **általánosan** jól használható sajnos nem állnak rendelkezésre (természetesen vannak speciális alkalmazási területek, ahol az automatikus generálás megoldott). Itt érdemes megemlíteni a formális módszerek és a tesztgenerálás egy "összekötését". A SUT-ről készült precíz modellen formális módszerekkel (például model checking, SAT solving) képesek vagyunk olyan érdekes bementi feltételeket találni, melyek éppen az általunk kívánt kódrész végrehajtását fogják eredményezni. A teszttervezés automatizálását jelen mérés során nem próbáljuk ki.

A **tesztvégrehajtás** a szoftvertesztelés legkönnyebben automatizálható lépése, már bőven rendelkezésre állnak az iparban is elterjedt, valóban jól használható eszközök. Ezek feladata a meglévő tesztesetek lefuttatása kontrollált körülmények között a tesztrendszeren és információk gyűjtése a végrehajtás során. Különböző célú illetve szintű tesztek különböző automatizmust kívánnak meg, de például az elterjed unit tesztelő keretrendszerek, illetve a GUI tesztelő keretrendszerek is automatikus végrehajtást kínálnak. A mérésen a végrehajtás automatizálását fogjuk elvégezni.

Az utolsó lépés, a **tesztkiértékelés** automatizálása is sok esetben megoldott, illetve integrálva van a tesztvégrehajtással. Például sok olyan eszköz létezik, mely az általunk beállított metrikák alapján képes jelentéseket generálni, és elküldeni azokat előre beállított felelősöknek. Mint a mérésen is látni fogjuk, erre a feladatra remekül alkalmazhatóak a folytonos integrációt biztosító eszközök.

Mint azt már említettük, a mérés során a tesztvégrehajtás automatizálását szeretnénk konfigurálni, egy folytonos integrációt támogató szerver támogatásával. A legtöbb ilyen típusú eszköz bármely automatikusan futtatható teszt végrehajtását támogatja, azonban mi a jelen mérés során felhasználói felület tesztelésére fogunk teszteseteket készíteni, és ezen tesztek végrehajtását automatizálni.

2.1 Selenium

A mérés során a Java alapú tartalomkezelő rendszer, a dotCMS felhasználói felületén fogjuk végrehajtani a teszteseteinket.

A Selenium[4] az egyik legnépszerűbb tesztautomatizáló eszköz, mellyel böngészőben futtatható alkalmazások tesztelést lehet könnyedén automatizálni. Lehetőséget nyújt egyrészt egyszerű tesztek nagyon gyors prototipizálására akár nem hozzáértő számára is, ezen túl azonban valóban professzionális eszközként használva lehetővé teszi nagy mennyiségű (például regressziós) teszt futtatását és karbantartását.

A **Selenium IDE**[5] tulajdonképpen egy Firefox bővítmény, mely egyszerű környezetet biztosít Selenium szkriptek készítésére. Segítségével rögzíteni lehet egy tetszőleges felhasználói interakciót és visszajátszani azt, illetve módosítani a szerkesztőben. Egy így elkészített szkript Selenium parancsokból áll (ezeket gyakran **selenese**-eknek nevezik), melyeket felhasználva bármilyen felhasználói interakciót meg lehet valósítani. Ilyenek például az *open*, *click/clickAndWait*, *verifyText*, *waitForPageToLoad*, etc. A teljes listát megtalálhajuk az IDE menüjében, illetve mérés előtt érdemes megismerkedni a Selenium IDE dokumentációjával [5]. Az IDE felülete az 1 ábrán látható.

[4] `http://seleniumhq.org/`

[5] `http://seleniumhq.org/projects/ide/`

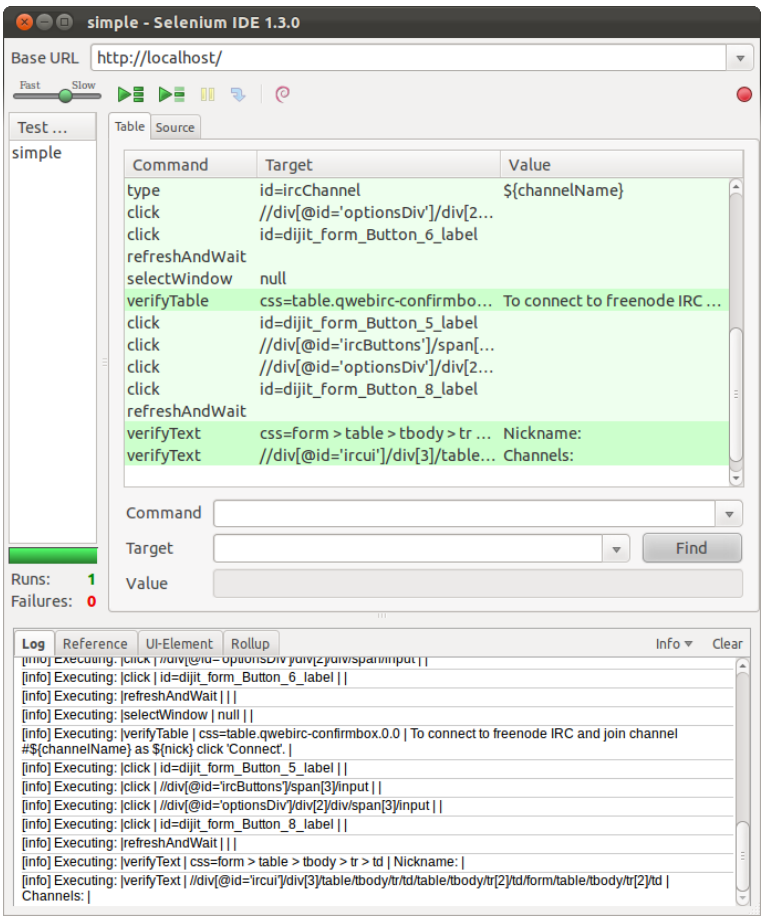

Figure 1: A Selenium IDE egy szkript futtatása után

A Selenium IDE egy nagyon egyszerű eszköz néhány alapszintű teszteset elkészítésére illetve a Seleniummal való megismerkedésre. Éppen ezért a mérésen mi is először ezzel fogunk megismerkedni. Sok teszteset esetén azonban nem tanácsos egy ilyen jellegű eszköz használata, hiszen a tesztek karbantartására valami elterjedtebb nyelvet, esetleg keretrendszert szeretnénk használni, egyszerű, kötegelhető végrehajtással. Szerencsére a Selenium ebben is tud nekünk segíteni.

A **Selenium Webdriver**[6] a Selenium legfrissebb része, mely fokozatosan átveszi a korábbi Selenium Remote Control helyét. A Webdriver lényegében egy olyan könyvtár, mely egy átgondolt API-t nyújt a tesztesetek készítéséhez, és melyet egy általunk választott programnyelven (valójában Java, C#, Python, Ruby vagy PHP közül választhatunk) használhatunk fel. A Webdriver direkt hívásokkal hajtja meg a böngészőt, melyet éppen használunk a teszteléshez (tehát itt már nem csak Firefox használható, lehetővé téve a cross-browser validációt). Ez esetben is érdemes a mérés előtt megismerkedni a Webdriver [6] dokumentációjával.

2.2 További technológiák

Felhasználói felület tesztelésére természetesen több eszköz is létezik, melyek közül kiemelkedik a Froglogic cég Squish [1] nevű megoldása, illetve további ismert szoftver pl. az IBM-től a Rational Robot[4]. Ezek jellemzően bármilyen ablakozóval készül GUI tesztelését végre tudják hajtani (Qt, SWT, MFC, .Net, ...), mivel alacsony szintű paraméterekkel (pl. egérmutató

[6] `http://seleniumhq.org/projects/webdriver/`

pozíciója) dolgoznak. Sajnos ezeket az eszközöket a laborban nem tudjuk kipróbálni.

Más eszközök nem ilyen alacsony szintű paramétereket figyelnek, hanem megjegyzik azokat az objektumokat, melyeket a felhasználó aktivál. Ez a megoldás természetesen nem lehet univerzális az előzőhöz képest (ugyanakkor könnyebben karbantartható szkripteket eredményez), ám így is létezik olyan eszköz [7] mely képes Eclipse pluginekhez, illetve RCP, SWT és Swing alkalmazáshoz GUI teszteket rögzíteni.

3 Folytonos integráció

A szoftverfejlesztés során egy átlagos projektben számos fejlesztő dolgozik együtt a kódbázis különböző 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 előtt szükségképpen frissíteni kell a saját lokális példányukat, amit integrációnak nevezünk. Szélsőséges esetben sajnos a lokális és a központi, up-to-date másolat közötti különbség olyan nagy lehet, hogy jelentősen 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), mely azt jelenti, hogy a fejlesztés során minden fejlesztő 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 tekinthető például egy egyszerű céges előírás, hogy minden reggel a munka megkezdése előtt a fejlesztők kötelesek frissíteni a lokális másolatukat, kikényszerítve ezzel az rendszeres integrációt. Tehát a folytonos integrációs szigorúan véve csak egy verziókezelő rendszer használatát követeli meg. Egy nagyon jó összefoglalást a témáról a [3] weboldalon kaphatunk, érdemes elolvasni!

Azonban az integrációnak nyilvánvalóan része egy fordítás végrehajtása, melyet automatizálva nagyban megkönnyíthetjük a fejlesztők 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 meglévő tesztkészlettel ellenőrizni 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, melynek végén egy riport generálódhat az egyes tesztek eredményeiből.

A legtöbb folytonos integrációt megvalósító szerver támogatja egyedi szkriptek futtatását a fordítás előtt 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, mely 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.

3.1 Jenkins

A Jenkins egy ingyenesen elérhető, nyílt forráskódú folytonos integráció támogató eszköz. Napjainkban az egyik legnépszerűbb ilyen szerver, mely támogatja a legtöbb verziókezelő rend-

szert (CVS, SVN, Git) és képes értelmezni Ant és Maven projekteket valamint tetszőleges shell scriptet illetve windows batch parancsfájlt. Tesztautomatizálást tekintve a Jenkins JUnit teszteket képes végrehajtani.

Azonban a Jenkins architektúrája lehetővé teszi a funkcionalitás kibővítését bővítmények segítségével, így integrálható számos egyéb tesztelő keretrendszerrel is. Természetesen a bővítményeken keresztül a Jenkins összeköthető a népszerű bugtracking rendszerekkel is, de amennyiben valami saját megoldásra van szükségünk, nekünk is van lehetőségünk saját plugint fejleszteni.

A mérés során mi ugyan nem fogunk saját Jenkins plugint fejleszteni, hanem csak egy munkafolyamat konfigurálását fogjuk elvégezni, de azért mérés előtt érdemes megismerkedni a Jenkins weboldalával [2].

3.2 Apache Maven

A legkritikusabb rész folytonos integrációt támogató szerver konfigurálása során, hogy egy egyedülálló parancs kiadásával képesek legyünk elindítani a fordítást ("automate the build"). Szerencsére erre a feladatra számos eszköz létezik, gondoljunk csak az elterjedt make eszközre. Mi mérésen Az **Apache Mavent** fogjuk használni.

Az **Maven** valójában egy átfogó projektmenedzsment eszköz, mely egy saját modellt, az úgynevezett *Project Object Model*-t (POM) használja arra, hogy lefordítsa a szoftvert, riportot illetve dokumentációt (change log, függőségi lista) generáljon.

Azon túl, hogy szükség van egy fordítást automatizáló eszközre a folytonos integrációhoz, a Maven jól használható arra, hogy egységes fordítási szisztémát alakítsunk ki a segítségével. Ezen felül az átgondolt függőség-menedzsment lehetővé teszi például egy központi JAR tároló használatát, így nem szükséges duplikáltan tárolni azonos fájlokat a különböző projektek miatt (illetve projektek közötti kompatibilitási gondokra is gyorsan fény derül).

4 Architektúra a mérésen

A fentebb megismert technológiák együttműködésére mutat egy példa összeállítást a 2 ábra.

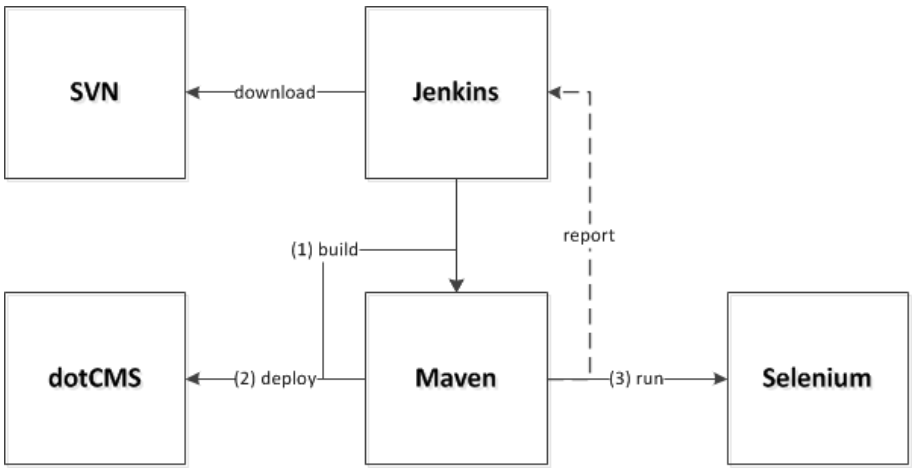

Figure 2: Architektúra ábra

Ezen a konfiguráción túl még számos különböző architektúra összeállítása képzelhető el akár ugyan ezen technológiák használatával (például több külön maven projekt használata is elképzelhető), azonban mi a mérésen egy ilyen architektúrát fogunk összeállítani.

A mérés során elvégzendő feladatok

Megjegyzések a méréshez

A mérés virtuális gépeken zajlik, Eclipse platformon, SVN verziókezelés [8] felhasználásával, mely technológiák ismertetése nem része sem a segédletnek, sem a számonkérésnek. Ezekkel remélhetőleg mindenki találkozott már legalább az eddigi mérések során, de ha valaki mégsem ismeri az alapfogalmakat, és nem tudja ezeket a szoftvereket legalább felhasználói szinten kezelni, az nagyban megnehezíti a mérés végrehajtását a megadott idő alatt.

Az említetteken felül a segédletben bemutatott technológiákat fogjuk használni a mérés során (Selenium, Apache Maven, Jenkins). Ezek segédletben szereplő áttekintését ismerni kell, abból fel kell készülni a számonkérésre. Ezen felül a Seleniumot érdemes kipróbálni a mérésre való felkészülés részeként, megismerkedni a felhasználói felülettel illetve selenese parancsokkal, és röviden átolvasni a Selenium IDE és Webdriver dokumentációt [5], [6].

Honnan indulunk

A mérés célja, hogy teszteseteket hozzunk létre egy szoftver funkcionális, modul-szintű regressziós teszteléséhez, és ezeket automatikus végrehajtását konfiguráljuk egy folytonos integrációt támogató szerver segítségével.

A mérés során a dotCMS Java alapú webes tartalomkezelő rendszert fogjuk használni, pontosabban egy ahhoz fejlesztett plugint fogunk tesztelni. Ez a plugin egy egyszerű IRC (Internet Relay Chat) kliens, mely a tartalomkezelőbe adminként belépett felhasználóknak kínál gyors csatlakozási lehetőséget a freenode IRC hálózatához. A méréshez használt gépeken a dotCMS a tesztelendő pluginnal együtt telepítve van, és használatra kész. A dotCMS ismeretére nincs különösebb szükség a mérés teljesítéséhez.

A tesztelendő plugin specifikációja

A mérés során egy dotCMS IRC plugin tesztelését fogjuk automatizálni. A plugin egy kliens a freenode IRC hálózatához, mely a dotCMS admin felületén érhető el. A kliens felülete a 3 ábrán látható.

Az IRC kliens funkciói röviden:

- Csatlakozik egy általunk megadott csatornához a választott becenévvel.
- Képes megjegyezni egy default becenevet és csatornát (cookie-ban), valamint képes ezt törölni.
- Képes egy felugró ablakba külön kiemelni a chat ablakot.
- Képes megjegyezni a felugró ablak default méretét.
- Ellenőrzi a csatlakozót reCAPTHA spam blokkolóval.
- Közös munkát támogat egy felugró ablakban megjelenő “collabedit” online editor segítségével.

Alább található a mérés során elvégzendő feladatok rövid leírása. Az egyes feladatokhoz további segédlet lesz elérhető a méréshez használt virtuális gépeken (kiadandó parancsok, konfigurációs fájlok, stb.).

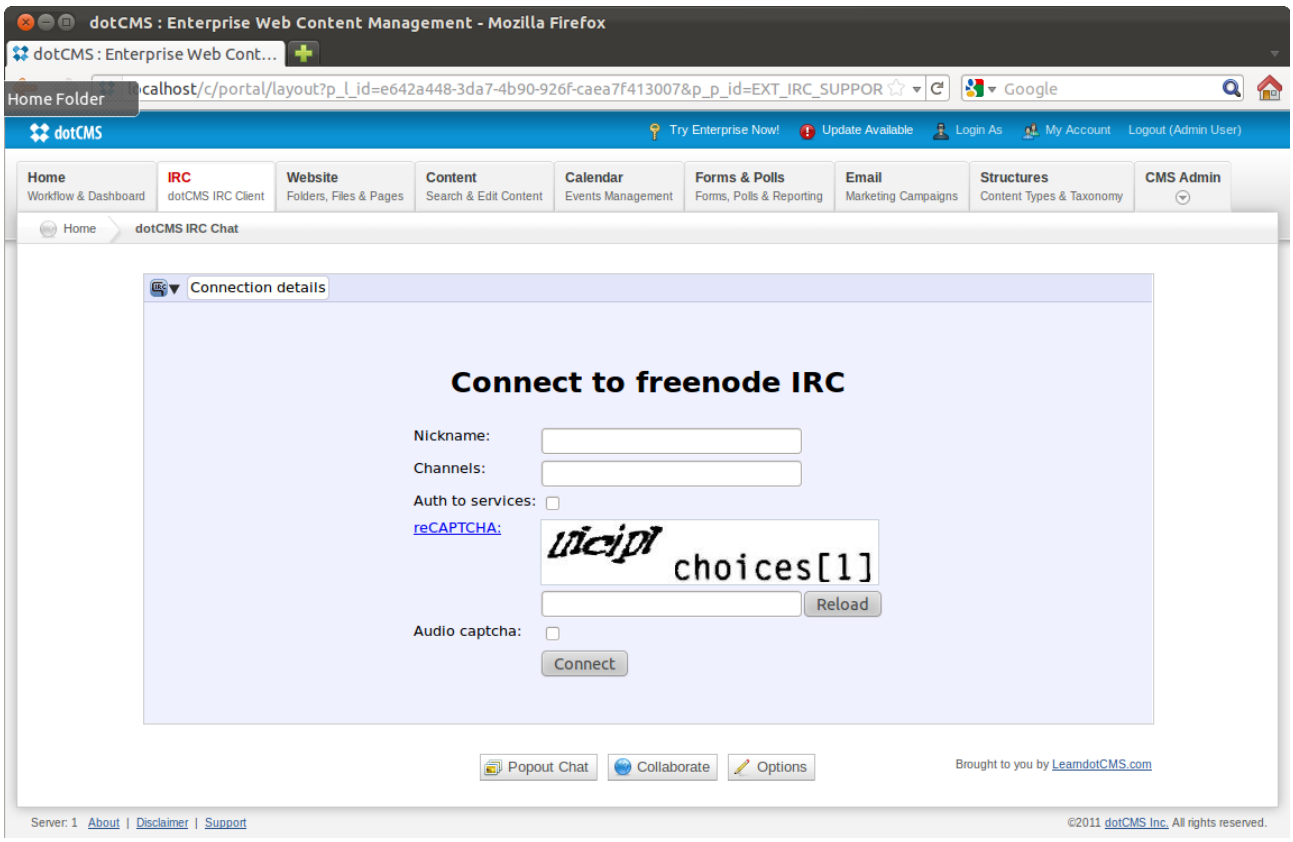

Figure 3: Az freenode IRC kliens a dotCMS admin felületén

1. feladat: Egyszerű Selenium szkript rögzítése

Olvassa át az IRC plugin specifikációját, majd próbálja ki a plugint. Gondolja végig, hogy tudna a funkciók teszteléséhez egy egyszerű tesztkészletet létrehozni (a tesztkészlet jelen esetben néhány tesztesetből álljon, koncentráljon például az "Options" funkcióra).

Indítsa el a Selenium IDE-t, és hozza létre a teszteseteit. Próbálja is ki őket, ismerkedjen meg a különböző beállítási lehetőségekkel! Tapasztalatait dokumentálja a jegyzőkönyvbe.

2. feladat: Szkript exportálása és futtatása JUnit tesztként

Nézze át az IDE által felajánlott exportálási lehetőségeket. Exportálja az elkészített teszteseteket JUnit tesztként, és vizsgálja meg a fájlokat (függőségek, Webdriver API használata). Futtassa le a JUnit fájlokat az Eclipse segítségével.

Ehhez létre kell hozni egy Maven (Java) projektet (a projekt függőségeinek beállításához segítséget nyújt egy már létező projekt a workspace-ben), és hozzá kell adni az exportált fájlt, majd JUnit tesztként futtatni.

Az elkészült tesztesetek érdekesnek ítélt részeit illetve tapasztalatait, valamint a tesztfuttatás eredményét dokumentálja a jegyzőkönyvbe!

3. feladat: Folytonos integrációt biztosító szerver konfigurálása

A méréshez használt virtuális gépen már rendelkezésre áll egy Maven projekt, mely segítségével elvégezhető az automatikus fordítás és telepítés. Nézze át ezt a már meglévő Maven projektet!

A következő feladat, hogy kiegészítse ezt a Maven projektet oly módon, hogy az a tesztfuttatást is automatizálja. Ehhez létre kell hozni egy pom.xml fájlt a megfelelő tartalommal (ez elérhető lesz a virtuális gépeken) a egy mappába a projekten belül, majd a "mvn clean install" parancsot kiadva létrejön a projekt. Végül a Selenium IDE-ből exportált teszteseteket felhasználva elkészíthetőek az új tesztesetek.

Az elkészült teszteseteket illetve a Maven projekt készítése során szerzett tapasztalatait, valamint a futtatás eredményét dokumentálja a jegyzőkönyvbe!

4. feladat: Regressziós tesztelés futtatása

A következő lépés, hogy a Maven projekteket felhasználva konfiguráljuk a Jenkinst az automatikus fordításra és tesztfuttatásra. A virtuális gépen már telepítve van és fut a Jenkins (localhost:8080), itt egy "New Job" felvételével kezdhetünk neki a munkafolyamat összeállításának. (Mivel már rendelkezésünkre áll egy Maven projekt a fordításhoz, ezért itt érdemes egy Maven build jobot létrehozni.) Nézze át, milyen lehetőségei vannak a konfigurációra, majd végezze el a szükségesnek ítélt beállításokat!

Az Jenkins job készítése során szerzett tapasztalatait, valamint a job lefutásának eredményét dokumentálja a jegyzőkönyvbe!

Ellenőrző kérdések[7]

- Hogyan definiálná saját szavaival a szoftvertesztelést?
- Mik a jellemző lépések egy szoftver tesztelése során?
- Milyen kategóriákat ismert meg a szoftvertesztelés csoportosítására?
- Mit nevezünk tesztesetnek?
- Mit nevezünk tesztadatnak?
- Mi a különbség a teszteset és a tesztadat között?
- Mit nevezünk teszt orákulumnak?
- Miért van szükség folytonos integrációra?
- Mire szolgál egy folytonos integrációt támogató szerver?
- A felkészülés részeként gondolja végig a következő szituációt!

  Új munkahelyre kerül szoftvertesztelőként. A cég egy olyan szoftvertermékkel van jelen régóta a piacon, melyre nem jellemzőek a gyors változások. Évente egy-két új verzió kerül kiadásra, a megrendelők igényei szerint azonban a visszafelé kompatibilitás kiemelkedően fontos. Tegyük fel, hogy a cégnél nem végeznek szervezetten tesztelést, és önt azért vették fel, hogy kialakítson egy tesztelési eljárást, mely tartalmaz irányelveket, technológiai ajánlásokat, képzési terveket. Milyen javaslatai lennének?

- A felkészülés részeként gondolja végig a következő szituációt!

  Új munkahelyre kerül szoftvertesztelőként. A cég profilja, hogy tanácsadóként, illetve szükség esetén kivitelezőként segíti illetve átveszi más cégek bajba jutott projektjeit, ily módon mindig csak néhány hónapot töltve el egy-egy projekttel. Tegyük fel, hogy a cégnél eddig nem foglalkoztak külön egységes tesztelés megszervezésével, és önt azért vették fel, hogy kialakítson egy tesztelési eljárást, mely tartalmaz irányelveket, technológiai ajánlásokat, képzési terveket. Milyen javaslatai lennének?

- Milyen ötletei lennének az előbbi szituációkban, ha nem csupán teszteléssel, hanem az egész minőségbiztosítás kialakításával bíznák meg?

[7] Az itt felsorolt kérdések az önellenőrzést hivatottak szolgálni, a beugrón további kérdések is elképzelhetőek!

References

[1] FrogLogic Squish. http://www.froglogic.com/.

[2] Jenkins Continuous Integration Server. http://jenkins-ci.org/.

[3] Martin Fowler: Continuous integration. http://martinfowler.com/articles/continuousIntegration.html.

[4] Rational Robot. http://www-01.ibm.com/software/awdtools/tester/robot/.

[5] Selenium IDE. http://seleniumhq.org/docs/02_selenium_ide.html.

[6] Selenium Webdriver. http://seleniumhq.org/docs/03_webdriver.html.

[7] WindowTester Pro. http://code.google.com/webtoolkit/tools/wintester/html/index.html.

[8] Ben Collison-Sussman, Brian W. Fitzpatrik, and C. Michael Pilato. *Version Control with Subversion*. http://svnbook.red-bean.com/, 2011.