Pääkirjoitus

Kovasti arvostamani testauskonsultti Ed Kit sanoi kirjassaan “Software Testing in the Real World” testauksesta, että se voi olla paras mahdollinen työ tai kaikista hankalin tapa kuluttaa työpäivääsi. Tämä on totta. Parhaimmillaan testaaja saa käyttää samanaikaisesti kaikki luovat ja tekniset kykynsä auttaessaan asiakasta ennakoimalla järjestelmän heikkoja kohtia ja suunnitellessaan niiden mahdollisimman tehokasta löytämistä. Ja parhaissa organisaatioissa testausroolin merkitys myös tiedostetaan ja se näkyy käytännön arvostuksessa (palkka, koulutus- ja etenemismahdollisuudet) ja testaajien kokemuksen hyödyntämisenä projektin määrittelyvaiheessa ja kehitysprosessin parantamisessa.

Toisaalta, jonkun muun laatiman testijoukon toistuva käsin suorittaminen eli niinsanottu “apinatestaus” voi toimia lyhyehkön ajan vaikkapa osana uuden testaajan perehdytystä, mutta polttaa tekijänsä loppuun varsin pian. On hämmentävää edelleen kohdata organisaatioita, joissa testaajien hahmotetaan olevan joko matkalla ohjelmoijiksi, entisiä epäonnistuneita ohjelmoijia tai sitten muuten vaan projektin kiinteä kulu. Suomessakin on paljon testaajia, joille ei ole järjestetty sitä ammattikoulutusta, mikä perustutkinnosta unohtui.

Onneksi tilanne muuttuu koko ajan parempaan; testauskursseja on lähivuosina puolessakymmenessä yliopistossamme ja testausta kehitetään aktiivisesti osana organisaation ohjelmistoprosessia. Asiakkaiden tiukkenevat laatuvaatimukset ja -mieltymykset näkyvät oikeana rahana (tai sen puutteena) pari kertaluokkaa aiempaa nopeammin. Vai miten käy eBusiness-asiakastyytyväisyyden, kun uusi sähköinen kauppapaikka ei vakuuta luotettavuudellaan, käytettävyydellään tai suorituskyvyllään? Tätä nykyä käyttäjä jaksaa odottaa hermostumatta enimmillään 7-8 sekuntia / sivulataus ja aika lyhenee koko ajan. Samalla jokainen kiireellä tehty ohjelmistomuutos ja jokainen uusi virhe näkyy verkossa heti parille tuhannelle seuraavalle firman sivuille lähitunteina eksyvälle käyttäjälle. Online-myymälän pahin kilpailija on aina vain yhden klikkauksen päässä, vaikka olisikin maapallon toisella puolella (“jos Bokus tökkii, tilaan Amazonista”).

Merkki paremmasta tämäkin: SYTYKE ry perusti Suomen softatestaajille kerhon (ks. Systeemityö 4/2000), jossa tuetaan testaajien ammattikuvaa, osaamista ja verkostoitumista. Lisää kerhosta yhdistyksen sivuilta. Aktiviteetit – ainakin yksi avoin kokous tulossa tämän kevään aikana – ilmoitetaan avoimella sähköpostilistalla (ohjeet kerhon sivuilla). Tässä lehdessä on kerhon liikkeellelähdön kunniaksi asiaa testauksen eri puolilta, Hans Schaefer kertoo selkeitä ajatuksia testauksen parantamisesta, professori Jukka Paakki antaa vaihtoehtoja koko testaukselle ja kokoamani Mark Fewsterin ajatukset testauksen automatisoinnista saanevat jatkoa myös kerhon sivuilla. Kuormitustestaus (ks. edellä) on yllättänyt monet tärkeydellään ja tilannetta selkeyttää Reetta Mansikkamäen artikkeli ja testausalan konferensseista otetaan myös selkoa. Lisää testausasiaa kootaan matkan varrella kerhon sivuille, mm. kokemuksia työkalujen käyttöönotosta, testausprosessin parantamisesta, testauksen menetelmistä ja viitemalleista (TPI, TMAP).

Erkki Pöyhönen
erkki.poyhonen@nokia.com

Viiteet:

- Edward Kit: “Software Testing in the Real World:Improving the Process” (Acm Press Books – Addison-Wesley, 1995)
- SYTYKE ry:n WWW-sivut http://www.pcuf.fi/sytyke/

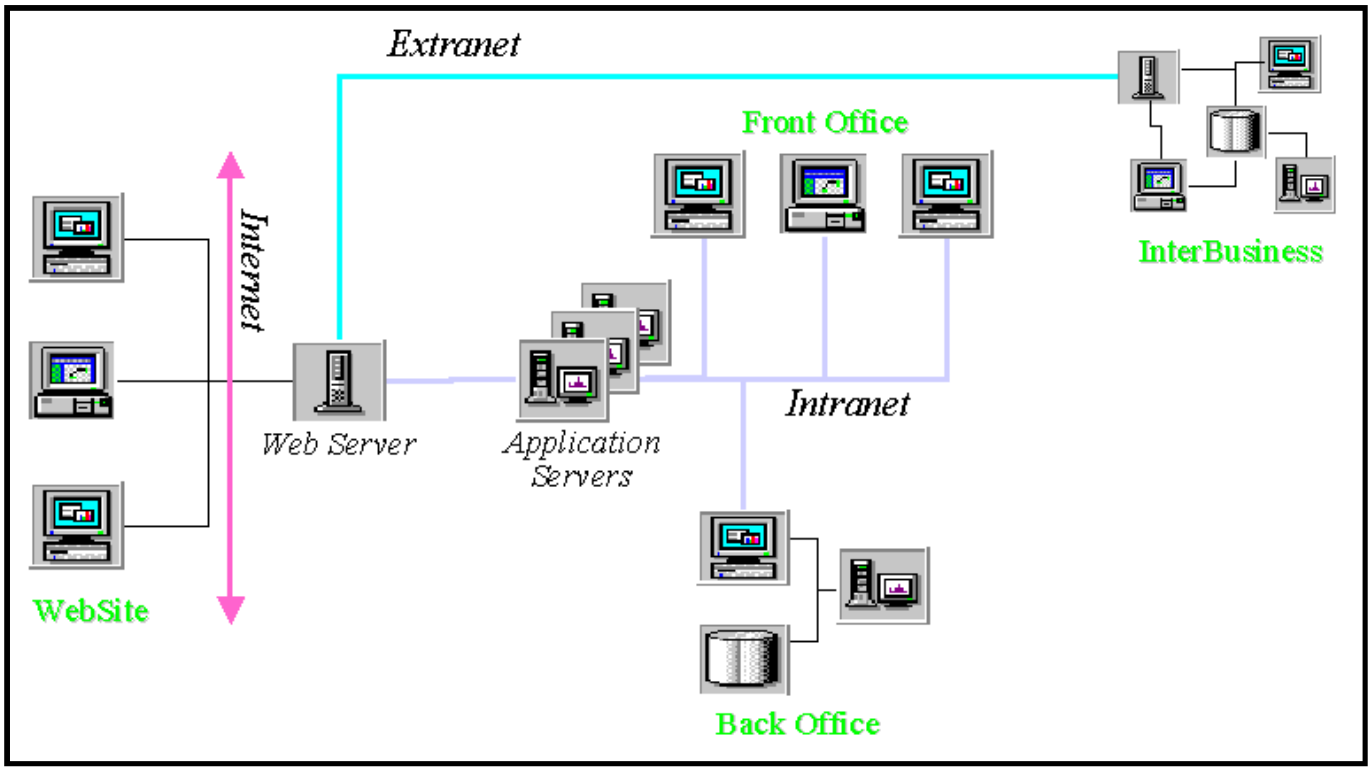
Kuormitustestaus businesvaatimusten varmistajana

Reetta Mansikkamäki,
Software Engineering Center Oy

Konsultti Reetta Mansikkamäki tarkastelee konsultin silmin järjestelmien kuormitustestausta. Kuormitustestaus (eng. *load testing*) on nykyisissä monikerroksisissa järjestelmissä oleellisesti aiempaa tärkeämpää ja se sisältää useita testaustyyppejä ja testauksessa huomioitavia komponentteja. Reetan työnantaja, Software Engineering Center tuo maahan Segue Softwaren testaustyökaluperhettä ja erityisesti sen kuormitustestausvälinettä SilkPerformer.

Uusien teknologioiden myötä, sekä talon sisäisessä että ulkoisessa käytössä, olevat sovellukset ovat muuttuneet. Sovellukset rakentuvat useista kerroksista ja ne voivat olla erilaisilla arkkitehtuureilla toteutettuja. Sovellukset eivät toimi enää yksittäisellä työasemalla, vaan niitä käytetään erilaisista käyttöliittymistä. Edellä esitetystä johtuen palvelimien kuormituskestävyys on muodostunut yrityksen toiminnan kannalta kriittiseksi.

Sovellusta suunniteltaessa tulee huomioida tuleva kuormitettavuus. Bisnesvaatimuksissa pyritään määrittelemään tuleva käyttö, joka varmistetaan testausvaiheessa. Sähköisessä kaupankäynnissä yrityksen kyky palvella asiakkaita on tärkeää ja alkava mobiilibisnes tuo omat haasteensa. Palvelukykyyn vaikuttaa järjestelmän kyky suoriutua asiakassuhteiden hoitamisesta laadullisesti virheettömästi ja riittävän nopeasti. Kuormitustestauksella pyritään selvittämään, miten kyseinen järjestelmä reagoi ja selviytyy suurista käyttäjämääristä sekä paikantamaan mahdolliset ongelmakohdat.



Kuva: Oheisessa kuvassa näkyvät monitasoisten järjestelmien aktiiviset osapuolet, käyttäjäryhmät sekä käytetyt verkkotyypit - - kaikki asettavat omat vaatimuksensa kuormitustestaukselle (Kuva: Segue Software, Inc. USA)

Kuormitustestauksen suoritus

Kuormitustestauksen parametrit saadaan vaatimusmäärittelydokumenteista. Testauksella varmistetaan, että sovellus toteuttaa sille asetetut vaatimukset. Dokumenteista pitää voida johtaa kuormitusmäärät ja profiilit. Suunnitteluvaiheessa tulee olla käsitys, kuinka sovellusta käytetään, ketkä ja milloin sitä käytetään. Näin voidaan rakentaa kuormitustestaustapaukset sellaisiksi, että ne vastaavat tulevia todellisia käyttötilanteita. Kuormitustestit ajetaan virtuaalisilla käyttäjillä, jotka toimivat kuten todelliset sovelluksen käyttäjät.

Kuormitustestausta tulee tehdä sovelluksen elinkaaren jokaisessa vaiheessa – sitä suunniteltaessa, kehitettäessä ja tuotantokäytössä. Tällöin voidaan määritellä miten sovellusta tulee parantaa ja kehittää eteenpäin. Sovelluksen kehitysvaiheessa tulee tehdä suorituskykyanalyyseja, jotta voidaan valita eri sovellusmalleista paras mahdollinen. Suorituskykyanalyysilla voidaan selvittää sovelluksen toiminnallisuuksia ja mitä parannuksia/muutoksia sovellukseen kannattaisi tehdä.

Kuormitustestaustavat

Kuormitustestit suunnitellaan eri tavoin, riippuen siitä mitä sen tulisi mitata. Yleisimmin käytettyjä kuormitustestaustapoja ovat stressitesti, stabiilisuustesti ja eristetyssä ympäristössä tehdyt isolaatiotestit.

Stressitesti

Stressitestauksessa testataan kuinka suurta kuormaa sovellus parhaimmillaan kestää. Siinä ajetaan suuri kuorma niin nopeasti kuin mahdollista, jotta voidaan kuormittaa sovellusta sen äärirajoille. Näin saadaan selville mikä on maksimikuorma jonka sovellus kestää tietyssä ajassa ja missä vaiheessa sovellus ylikuormittuu ja kaatuu. Mitattavina parametreinä ovat transaktioita sekunnissa, kb/s jne.

Stabiilisuustesti

Stabiilisuustestauksessa testataan kuinka hyvin sovellus palvelee pitkän ajan kuluessa. Tällaisessa testauksessa sovellusta ajetaan sopivalla kuormalla pitkän aikaa. Riippuen määritellyistä testaustavoitteista stabiilisuustestaus voi kestää 24 tunnista viikkoon tai jopa pidempään. Stabiilisuustesteillä voidaan määritellä ovatko esim. sovelluksen vasteajat tarpeeksi alhaiset, eli pysyykö sovelluksen resurssien kulutus vakaana koko määritellyn ajan ja onko sovelluksessa mitään muita teknisiä virheitä, jotka voivat vaikuttaa sovelluksen stabiilisuuteen.

Isolaatiotesti

Kun tietty määrä kuormitustestejä on tehty ja ne ovat osoittaneet ongelman olemassa olon, auttaa isolaatiotesti testaajaa tunnistamaan kyseisen ongelman ja löytämään virheen.

Isolaatiotestauksessa tietty toimintojen joukko suoritetaan toistuvasti joka kerta täysin samalla tavalla. Testauksessa pyritään eristetyssä ympäristössä toistamaan virhe, jotta sen lähde paikantuisi.

Skaalautuvuus - tärkeä sovelluksen elinkaareen vaikuttava asia

Sovellusta suunniteltaessa tulee huomioida, että sovellusta ei kannata tehdä uudelleen joka kerta saavutettaessa suorituskykyraja. Sovellus tulee rakentaa niin, että kuormituksen kasvaessa sitä voidaan hajauttaa. Hajauttamisella voidaan tarkoittaa palveluiden monistamista joko kokonaisuudessaan tai komponenttitasolla. Toisin sanoen jos sovellus on suunniteltu alun pitäen esim. 100 yhtäaikaiselle käyttäjälle, tulee olla suunnitelma kuinka sovellus voidaan skaalata vaikkapa 1000 yhtäaikaiselle käyttäjälle.

Kuormitustestauksella saadaan hyödyllistä tietoa jo kehityksen alkuvaiheessa siitä kuinka eri komponentit käyttäytyvät ja minne mahdolliset pullonkaulat muodostuvat. Näin voidaan laatia skaalausstrategia; mitä palveluja/komponentteja monistetaan ja miten rautavaatimukset kehittyvät. Palveluiden ollessa tuotannossa tulee kuormituksen kehittymistä seurata ja palvelutason laskiessa tiedetään mihin toimiin tulee ryhtyä.

Kuormitustestauksen tavoitteet

Tavoitteena on varmistaa että sovellus toimii niin kuin se on suunniteltu. Toisena tavoitteena on varmistaa että kaikki käyttäjät voivat käyttää sovellusta vasteaikojen ollessa kohtuullisia kaikista käyttöliittymistä käsin. Usein unohdetaan, että kuormitustauksella ostetaan myös aikaa. Me haluamme tietää kuinka kauan kyseisellä järjestelmällä voidaan tulla toimeen.

Tarkasteltaessa hieman yksityiskohtaisemmin tavoitteita haluamme vastauksia seuraaviin asioihin: millä käyttäjämäärällä (kuormalla) järjestelmä kuormittuu niin paljon, ettei se enää palvele hyväksyttävästi, kuinka kauan järjestelmän käyttäjä joutuu lataamaan sivuja kussakin tilanteessa ja miten suurentunut kuorma vaikuttaa sekä tietokantojen että järjestelmien suorituskykyyn.

Jotta kuormitustestaus olisi tehokasta, tarvitaan hyvät ja oikeanlaiset työkalut sekä mahdollisesti avuksi kokeneita konsultteja. Järjestelmän testauksessa huomioidaan liiketoimintanäkökulma ja tehdään mahdollisimman todenmukaisia käyttötilanteita. Järjestelmän kuormitustestauksessa voidaan arvioida ja tarkentaa pullonkaulojen parantamismahdollisuuksia. Testauksen vaikuttimet ja parantamiskeinot, syy miksi testejä tehdään, tulee olla selvät kaikille osapuolille. Tämä mahdollistaa tehokkaan ja tuloksekkaan testauksen ja lisäksi tarjoaa riittävästi informaatiota kehittäjien tarpeisiin.

Reetta Mansikkamäki,
konsultti,
Software Engineering Center SEC Oy

Systeemityöyhdistys SYTYKE ry:n

TOIMISTO ON MUUTTANUT

uusiin entistä ehompiin tiloihin radan toiselle puolelle Pitäjänmäellä.

Uusi osoite on Henrikintie 7 A, 00370 Helsinki.

Puhelinnumerot ja fax-numero säilyvät ennallaan.

Testauskonferenssit ja seminaarit - kansainvälistä verkostoitumista moneen makuun

Erkki Pöyhönen,
Nokia Research Center,
Software Technology Laboratory

Suomi on softatestauksessa vielä pieni maa. Testaukseen panostetaan joko suurissa yrityksissä, jossa testaajat eivät niinkään tarvitse ulkopuolista vertaisverkostoa tuekseen ja sitten on olemassa kehittyneitä pienempiä yrityksiä, joiden testauksen hyvyydestä ei paljon kuulla talon ulkopuolella.

Helppo ratkaisu

Kun halutaan tavata kollegoja tai oppia, mikä testauksessa on tärkeää juuri nyt, on tehokas kuuri lähteä alan konferenssiin. Sellaisia ovat vuosittaiset “EuroSTAR” ja “QualityWeek Europe” ja uutena ICSTEST. Konferenssiesityksissä saa ajankohtaisiin aiheisiin halutessaan useampiakin näkökulmia. Paras anti kuitenkin on yleensä aulan puolella kahviaikaan, kun on lupa ja mahdollisuus tentata ulkomaisia ja kotimaisia kollegoja testauksen arjesta oman firman ulkopuolella ja havaita ettei ole yksin ongelmiensa parissa.

Korkealta tuntuva kustannus saattaa pelästyttää, mutta konferenssi vuoden-parin välein tekisi hyvää ammatti-identiteetille ja työmotivaatiolle. Lisäksi siellä saatu tieto on usein varsin soveltavaa ja käytännöllistä, jolloin muiden kokemukset ovat helpommin sovellettavissa omaan tilantee-

seen. Ja kun puhuja on vielä tavattavissa lisäkysymyksiä varten, voi varmistua ideoiden soveltuvuudesta omaan organisaatioon.

Suomen testausmaailman lama näkyy siinä, miten vähän suomalaisia esityksiä teollisuuden testauskonferensseissa on suhteessa IT-alan nykyiseen merkitykseen. Asia voi olla rohkeudesta kiinni ja ehkäpä oma Sytykkeen testaajakerhomme voisi olla tarjoamassa mahdollisuuksia omien kokemusten ja ideoiden esiintuontiin ja sparraamiseen muiden testausammattilaisten kanssa. Ainakaan puute ei ole tarjolla olevassa asiantuntemuksessa!

Muistoja Kööpenhaminasta

EuroSTAR 2000 vietettiin Kööpenhaminassa viime joulukuun alussa ja se oli tyypillinen suuri konferenssi. Tilaisuuteen saapui viitisensataa testausihmistä useimmista Euroopan maista, parikymmentä Suomestakin. Esityksistä yksikään ei ollut suomalainen ja se hieman harmitti. Viiden päivän kokouksesta kaksi ensimmäistä oli tutoriaaleja, se on konferenssin alkuun istutettuja päivän kursseja alan perusaiheista (Testing Foundations, Test Process Improvement, e-Business testing, ...), joilla oli pääosin puhujina hyvin kokeneita testauskonsultteja.

Varsinaisen konferenssin kolme päivää vietettiin neljässä rinnakkaisella polulla, joissa keskityttiin mm. e-Business-infran testaamiseen, testausprosessiin, testausorganisaation erityisongelmiin ja testauksen automatisointiin. Esitysten rinnalla oli kattava messukokonaisuus, jossa testauspalvelut ja –työkalut olivat hyvin esillä (erityisesti jäivät mieleen Tieto-Enatorin hilpeät testausjulisteet). Kokonaisuuden kruunasi kattava sosiaalinen ohjelma päivällisineen ja vastaanottoineen.

Muu yhteistyö

Tapasin EuroSTARissa myös monen Euroopan maan (Norja, Saksa, Ruotsi, Tanska ja Englanti) testausammattilaisten yhdistysten ja kerhojen vastuuhenkilöitä. Andreas Spillner Bremenistä pitää yllä listaa eri testauskerhoista. Andreas myös kertoi yliopistonsa kunnianhimoisesta testauskoulutushankkeesta ja sai palautetta

myös eri eurooppalaisten kerhojen yhteyshenkilöiltä. Ensi talvena muuten pyritään pitämään useassa suomalaisessa yliopistossa ainakin yksi testausaiheinen kurssi.

Tapahtuma- ja organisaatioviitteet

- EuroSTAR 2001, Tukholma 19.-23.11.2001, "THE Eurooppalainen testauskonferenssi" http://www.testingconferences.com, huom. papereiden jättöpäivä on 6.4.2001!
- STAR East 2001, Orlando 2.-6.4.2001, EuroSTARin "isoveli" http://www.sqe.com/stareast
- ICSTEST, Bonn 2.-6.4.2001, uusi konferenssi, esitykset pääosin saksalaisia http://www.icstest.com
- QualityWeek (San Francisco 29.5.-1.6.2001) & Quality Week Europe (Brysseli marraskuussa) http://www.soft.com/QualWeek/
- EuroSIGIST, eurooppalaisten testauskerhojen löyhä ryhmittymä Andreas Spillnerin koordinoimana http://www. fbe.hs-bremen.de/spillner/eurosigist.html

Erkki Pöyhönen,
Senior Consultant,
Nokia Rersearch Center,
Software Technology Laboratory

Testauksen automatisointi edellyttää selkeitä tavoitteita

Erkki Pöyhönen,
Nokia Research Center,
Software Technology Laboratory

Tunnetut ohjelmistotestauksen asiantuntijat, konsultit Mark Fewster ja Dorothy Graham ovat koonneet testauksen automatisointikokemuksensa kirjaksi "Software Test Automation". Kirja on hyvin käytännöllinen, mutta automatisoinnin kontekstia selvitellessä palataan perusteisiin selvittämällä organisaatioiden syitä ylipäänsä ryhtyä automatisoimaan testaustaan ja kuinka näissä tavoitteissa onnistuttiin. Tässä esitellään myös ytimekäs ohjelma automatisoinnin aloittamiseksi.

Monenlaisia ongelmia ja monenlaisia ratkaisuja

Testauksen automatisointiin pyrkimisessä voi olla monta syytä, jotka eivät kaikki suinkaan ole näkyvissä tai perustu tietoiseen päätökseen. Erilaiset tarpeet johtavat erilaisiin tavoitteisiin ja keinoihin niiden saavuttamiseksi. Kun toiminnan kehittämisen voimavarat ovat organisaatioissa useimmiten varsin rajalliset, olisi prosessin parantamisessa syytä tiedostaa ja sopia tavoitteet heti alkuun. Testaus on toistaiseksi epämuodikasta suomalaisissa yrityksissä. Tämä saattaa vaikuttaa sen, että esimerkiksi testauksen automatisointiin on vain yksi mahdollisuus, jonka olisi syytä onnistua jos halutaan myöhemminkin resursseja testauksen kehittämiseen.

Grove Consultingin tutkimus: Erilaiset automatisoinnin tavoitteet

Mark ja Dot ovat yrityksensä, Grove Consultingin toimeksiantojen yhteydessä löytäneet useita mahdollisia tavoitteita testauksen automatisoinnille. He suorittivat 1997 kyselyn automatisoinnin parissa toimiville testauksen kehittäjille. Saadun palautteen (noin 100 vastausta, joista puolet Euroopasta ja puolet Yhdysvalloista) perusteella tunnistettiin seuraavat luokat tavoitteille:

Näiden tavoitteiden takana on varmasti koko joukko havaittuja ja tiedostettuja ongelmia, joihin näillä tavoitteilla odotetaan löytyvän ratkaisuja. Pelottavan suuri luku, 26% vastauksista ilman mitään toivottuja tuloksia voidaan ottaa muistutuksena fokusoinnin tärkeydestä.

Sarake "Saavutus" tarkoittaa niiden vastaajien osuutta, jotka kokivat onnistuneensa ko. tavoitteessa. Helpoimpia tuntui olevan listan alkupään tavoitteet – testien toistettavuus, regressiotestauksen tuki ja uudelleentestauksen tiheämmät kierrokset. Lisäksi testien siirrettävyys eri ympäristöihin saavutettiin, mikäli se koettiin erityisen tärkeäksi.

Toisaalta nähdään myös, että esimerkiksi säästö testausbudjetissa ja testausajan lyhenemi-sessä ovat vaikeimmat saavuttaa, jos kaksi kolmannesta niitä tavoitelleista ei koe niitä saavuttaneensa. Helpoin ja todennäköisin selitys on se, että juuri näiden tavoitteiden saavuttaminen edellyttää aikaa ja riittävän monta

testauskierrosta – siinä ajassa koko automatisointitekniikkaan saatetaan ehtiä pettyä (Kemererin (ks. lähteet lopussa) tutkimuksen mukaan 80-90% ohjelmistotyökaluis-ta päätyy hyllyntäytteeksi vuoden kuluessa hankinnasta). Lisäksi testaajien motivaatio ei välttämättä kohennu, mikäli työkalujen käyttöönottoon – koulutukseen, menetelmäkehitykseen ja asian myymiseen ei paneuduta, ennen kuin muutosvastarinta kääntää tuulen vastaiseksi.

Neljä askelta testauksen automatisointiin

Mark ja Dorothy ehdottavat neljää selkeää porrasta testauksen automatisoinnin aloittamiseksi:

1. *Tunnista testauksen todelliset ongelmat* ja niistä etenkin ne, joihin automatisointi parhaiten voi vastata
2. *Valitse työkalu*
3. *Kokeile työkalua* ensin pienessä mittakaavassa
4. *Hyödynnä hyvät kokemukset* laajemmassa käyttöönotossa ja ota oppia pilotoinnin virheistä

1. Ongelma-analyysi

Tyypillisiä testauksen ongelmia, joihin on ratkaisuja sekä testauk-sen automatisoinnin että muun testauksen parantamisen saralla ovat:

- Käsin testaaminen on liian työlästä — *Testauksen sisällön parempi suunnittelu voisi auttaa*
- Regressiotestaukseen ei koskaan ole riittävästi aikaa — *Testausprojektin suunnitteluun tulisi panostaa*
- Käsin testaaminen on epämääräistä — *Testidokumentointi kaipaa varmaan parantamista*
- Epäluotettavat muistiinpanot ajetuista testeistä — *Testauskäytännöissä tulisi kaiketi sopia ajettavien testien dokumentoinnista*
- Testien lopputuloksen arvioinnissa pääsee vikoja läpi huomaamatta — *Sovituista testauskäytännöistä voisi taas olla apua*
- Testausta pidetään tylsänä — *Koulutus auttaa saamaan testaukseen mielekkyyttä*
- Testaus havaitsee liikaa vikoja — *Katselmusten hyödyntäminen ennen testausta vähentää syntyviä vikoja etukäteen*
- Testauksen jälkeen löytyy liikaa vikoja — *Testaustekniikoihin kannattaisi panostaa*

Ongelman tunnistamisen ohessa on tärkeää päätellä eri ongelmien ratkaisujen tärkeysasteet ja kiireellisyys. Lisäksi on syytä havaita vaihtoehtoisia ratkaisutapoja ja valita niistä hinta-tehosuhteel-taan sopivin. Ratkaisua valittaessa tulee myös huomioida organisaation ja prosessin kypsyys – tehokkaimmat lääkkeet edellyttävät tarkempaa tilanneanalyysiä, parempaa kokonaisohjausta ja esim. sisäisiä tukiresursseja toimiakseen.

2. Työkalun valinta

Työkaluvalinta herättää intohimoja konkreettisuutensa (toimittajasuhde, suora investointi, hieno pahvilaatikollinen manuaaleja) vuoksi, vaikka onkin varsin myöhäinen ajankohta kiinnostua asiasta. Valintaprosessi tulee sitoa ongelmien ratkaisuihin ja perustua mahdollisimman objektiivisesti kerättävään tietoon: työkalun ja toimittajan ominaispiirteet ja niiden sopivuus käyttäjien testausprosessiin ja todelliseen tilanteeseen.

3. Pilotointi

“Jos aiot kokea epäonnistumisen, valitse mieluummin pieni sellainen”; Tom Gilbin lause muistuttaa työkalupilotin piirteistä – sen tulee olla hallittu, sisällöltään mielekäs, opettavainen, objektiivinen ja lyhyt (eng. Planned, Important, Learning, Objective, Timely).

4. Käyttöönotto

Onnistuneen pilotin jälkeen on syytä mainostaa onnistumista ja laajentaa käyttö asteittain, sitä mukaa kun ehtii kouluttamaan, tukemaan ja ohjaamaan käyttö prosessein ja standardein parhaaksi koeteltuun suuntaan, välttämään löydetyt sudenkuopat. Jatkuva asteittainen parantaminen on ainut tie automatisoinnin hyödyllisyyteen ja siltä odotettuihin merkittäviin hyötyihin.

Lähteet:

- “Software Test Automation”, Mark Fewster & Dorothy Graham (ACM Press & Addison-Wesley, 1999)
- “How to start test automation”, Mark Fewster (Esitys Nokian testaajaverkoston kokouksessa 26.5.1999 Tampereella)
- “How the Learning Curve Affects CASE Tool Adoption”, Chris Kemerer (IEEE Software, Toukokuu 1992)
- Lisää tietoa ja ideoita osoitteessa http://www.grove.co.uk ja myöhemmin myös SYTYKE testauskerhon sivuilta http://www.pcuf.fi/sytyke/

Erkki Pöyhönen,
Senior Consultant,
Nokia Research Center,
Software Technology Laboratoy

Turha testaaminen on turhaa

Jukka Paakki,
Helsingin yliopisto,
tietojenkäsittelytieteen laitos

Vanhoilla kunnon mittareilla mitattuna ohjelmistojen laatu on huono. Ne ovat hankalia käyttää, ne kaatuilevat, jäävät jumiin tai tuottavat yksinkertaisesti vääriä tuloksia. Ohjelmistoissa on siis *virheitä*, jopa melkoisia määriä: useiden eri tutkimusten mukaan jopa pitkään käytössä olleissa ohjelmistotuotteissa on vähintään 3 virhettä (eli “bugia”) jokaista tuhatta riviä kohti riippumatta sovellusalueesta ja tuotteen toimittajasta.

Ohjelmistovirheitä on perinteisesti pidetty pahana asiana ja osoituksena huonosta ammattitaidosta. Varmaankin tästä syystä johtuen ohjelmistoyritykset eivät juurikaan halua julkistaa virhemääriään, toisin sanoen sitä, montako virhettä ne ovat projekteissaan ohjelmistoihinsa koodanneet ja montako virhettä on päässyt asiakkaille ja käyttäjille asti. Niinpä edellä mainitut tutkimukset ovat vain anonyymisti suuntaa antavia eivätkä suoraan kerro, montako ohjelmistovirhettä vaikkapa Nokian puhelinkeskuksessa tai Microsoftin Word-teksturissa oikein luuraa odottamassa varomatonta käyttäjäänsä. Mutta niitä on siis vähintään 3 kappaletta jokaista tuhatta koodiriviä kohti.

Vapaassa akateemisessa lehdistössä on onneksi pohdittu myös tällaisia mielenkiintoisia yksittäistapauksia päätyen mm. yhdessä analyysissa arvioon, jonka mukaan Windows 95:n kehitysprojektin aikana siihen onnistuttiin tunkemaan noin 3 miljoonaa bugia (T. Lewis: Joe Sixpack, Larry Lemming, and Ralph Nader. (IEEE) *Computer* 31, 7, 1998, 107-109). Kunnioitettava määrä, jonka ylittäneitä ei toistaiseksi ole julkisuuteen ilmoittautunut.

Virhemäärillään rehvastelevien yritysten harvalukuisuutta ei toisaalta tarvitse ihmetellä siitäkään syystä, että harvat niistä tuntuvat edes tietävän ohjelmistovirheidensä määrää. Kaikenlainen laadun *mittaaminen* on nimittäin ohjelmistoyrityksissä varsin tuntematonta sarkaa – siihen kun kuluisi aikaa, joka olisi pois “varsinaisesta tuottavasta työstä”. Teollisen laadunvalvonnan ja mittaamisen retuperä paljastuu mm. niistä graduista, joita opiskelijat työpaikoillaan tekevät. Monessa niistä käydään nimittäin läpi erilaisia ohjelmistoprosessimalleja ja ohjelmistojen mittaamistekniikoita sekä suodatetaan niistä jonkinlainen ehdotus, jonka vailla minkäänlaista kunnollista prosessia toimiva yritys voisi ehkä joskus ottaa käyttöön, jos vain ajat muuttuisivat vähemmän kiireisiksi. Sen sijaan pöydälleni ei toistaiseksi ole lätkäisty ensimmäistäkään gradua, jossa kerrottaisiin jossakin yrityksessä todellisessa käytössä olevasta kattavasta mittaus- ja laadunvalvontaprosessista.

Ohjelmistoissa on siis karmean paljon virheitä. Mutta miten virheet syntyvät, ja mitä niille oikein pitäisi tehdä? Ohjelmistoon tulee bugi sen takia, että joku onneton ohjelmoija sen sinne kirjoittaa. Ensi kädessä virheiden syynä on siis ohjelmoijien huono ammattitaito. Ammattitaidon huonoa tasoa ei kannata kauaa ihmetellä, onhan yrityksissä työskentelevien koodaajien joukossa suuri joukko aivan noviiseja ensimmäisen vuoden opiskelijoita tai jopa suoraan lukiosta rekrytoituja. Toinen syy on se, että ohjelmistoilla ratkotaan nykyisin niin vaikeita ongelmia, ettei kokeneimmankaan ohjelmistoammattilaisen kapasiteetti riitä niitä kaikkia hallitsemaan. Ja ellei itse ratkaistava tehtävä olisikaan vaikea, halutaan tuotteeseen tunkea niin monta turhaa piirrettä, että ohjelmisto tulee jo kokonsa takia mahdottomaksi hallita. Kolmanneksi virhelähteeksi voi nostaa ohjelmointikielten ja –työkalujen epäinhimillisyyden: esimerkiksi C++ on tutkimuksissa havaittu pikemminkin kiroukseksi kuin siunaukseksi, vaikka olio-

mekanismien käyttö muutoin onkin ohjelmistoprojekteissa hyväksi.

Vaikka ohjelmistovirheistä ei koskaan päästäkään kokonaan eroon, voisiko niiden määrää jollakin tavoin vähentää? Alan kirjallisuudessa tällaisia keinoja on esitelty pilvin pimein, mutta valitettavasti tieto ei näytä levinneen yrityksiin asti. Koska tätä lehteä lukee toivottavasti myös joukko ohjelmistoammattilaisia, kerrataanpa jälleen kerran muutamia kokemusperäisesti osoitettuja ohjelmistotekniikan perustotuuksia (B. Boehm, V.R. Basili: Software Defect Reduction Top 10 List. (IEEE) *Computer* 34, 1, 2001, 135-137):

1. Tuotantokäytössä olevassa ohjelmistossa havaitun virheen löytäminen ja korjaaminen maksaa usein 100 kertaa niin paljon kuin sen korjaaminen jo ohjelmiston suunnitteluvaiheessa. Kannattaa siis (jopa rahallisesti) poistaa virheet mahdollisimman aikaisessa vaiheessa.
2. Ohjelmistoprojektien ajasta 40-50% kuluu vältettävissä olevan työn uudelleen tekemiseen. Sama neuvo kuin edellä: poista virhe välittömästi, muuten törmäät siihen jatkossa moneen kertaan.
3. Noin 80% turhasta uudelleen tekemisestä aiheutuu 20%:sta virheitä. Kannattaa siis keskittyä erityisesti vakavien virheiden poistamiseen (väärät vaatimukset, väärä arkkitehtuuri).
4. Noin 80% virheistä sijaitsee 20%:ssa ohjelmistomoduuleita, ja noin puolet moduuleista on (lähes) virheettömiä. Erityisesti suuret ja mutkikkaat moduulit on siis syytä käydä läpi mahdollisimman huolellisesti.
5. Noin 90% järjestelmän seisonta-ajasta (”downtime”) syntyy enintään 10%:sta virheitä. Toistamiseen: keskity käytön kannalta vakaviin virheisiin.
6. Tarkastukset ja katselmoinnit (”inspections”, ”reviews”) löytävät 60% virheistä. Nämä tekniikat ovat sikälikin rautaisia, että niitä voi käyttää jo projektin alkuvaiheissa, jolloin erityisesti testausta ei koodin puuttumisen takia voi tehdä lainkaan.
7. Erilaisten käyttötilanteiden mukaan suunnatut tarkastukset ja katselmoinnit löytävät 35% enemmän virheitä kuin tasapaksut vastineensa. Kannattaa siis keskittyä käyttäjien kannalta oleellisiin näkökulmiin.
8. Ohjelmistokehittäjän oma kurinalainen työskentely vähentää hänen tekemiensä virheiden määrää 75%:lla. Tämä tarkoittaa sitä, että olipa prosessi millainen tahansa, ohjelmistokehittäjien henkilökohtainen ammattitaito ja moraali ovat lopulta ratkaisevat laatutekijät.
9. Ns. kriittisten ohjelmistojen kehittäminen maksaa 50% enemmän kuin bulkkisoftan kehittäminen (mikäli kaikki muut projektiin vaikuttavat tekijät ovat samalla tasolla). Luotettavien ohjelmistojen tekeminen on siis hankalaa, mutta onneksi laadukkaan työn aiheuttamat lisäkustannukset kompensoituvat myöhemmin samalla mitalla vähentyvinä ylläpito- ja korjauskustannuksina.
10. Noin 40-50% käyttäjien tekemistä ”ohjelmista” sisältää hankalia virheitä. Tällä viitataan lähinnä taulukkolaskimilla tehtyihin tutkimuksiin, joiden mukaan niiden käyttäjät eivät osaa kirjoittaa kaavojaan oikein. Tällaisten työkaluohjelmien kehittäjien tulisikin auttaa surkeita loppukäyttäjiä tuottamalla käytettävämpiä ja tarkemmin käyttäjän tekemisiä valvovia työkaluja.

Listaan voi vielä lisätä ainakin seuraavan, erityisesti akateemisen yhteisön korostaman faktan:

11. Ohjelmistoissa olevien virheiden määrä vähenee 100%:lla, mikäli ohjelmistot todistetaan oikeiksi. Tällainen absoluuttinen laatu vaatii kuitenkin ohjelmistokehittäjiltä vankkoja matemaattisia taitoja, jotka eivät tunnetusti ole ohjelmistoyritysten valttiässiä.

Tiivistetty yhteenveto yllä olevasta listasta on, että virheet pitää poistaa mahdollisimman aikaisessa ohjelmistoprojektin vaiheessa, siis paljon ennen kuin ne pääsevät koodiin saakka. Tällöin saavutetaan sekä merkittäviä kustannussäästöjä että laadukkaampi tuote. Lisäksi pitää keskittyä erityisesti niihin ohjelmiston osiin, jotka ovat joko erityisen virhealttiita tai käytön kannalta kriittisiä.

Valitettavasti sanoma ei ole mennyt perille, vaan ohjelmistoyritykset poistavat edelleen virheitä mieluummin joten kuten testaamalla kuin tarkastamalla, katselmoimalla ja oikeaksi todistamalla. Esimerkiksi ohjelmistoyritysten suvereenilla ykkösellä, Microsoftilla, on palveluksessaan yhtä paljon testaajia kuin ohjelmoijia. Microsoftin kehitysprosessissa ei juuri muuta tapahdukaan kuin koodaamista ja testaamista: koodaajat tekevät päiväsaikaan virheitä, joita testaajat sitten öisin etsivät. ”Laatu” hoidetaan siis kuntoon joukkovoimalla.

Köyhässä Suomessa ei ohjelmistoyrityksillä ole varaa palkata samaa määrää testaajia kuin Microsoftilla, joten täällä pitää joukkovoiman sijasta käyttää muita menetelmiä. Kannattaa aloittaa lukemalla uudelleen läpi yllä oleva 11 kohdan lista. Sen jälkeen voi yritys aloittaa työntekijöiden kouluttamisen ja ohjelmistoprosessien viilaamisen laadukkaaseen kuntoon. Tätä kautta päästään eroon valtavasti aikaa ja rahaa vievästä testauksesta, se kun on pääosin täysin turhaa työtä – etenkin huonosti hoidettuna.

Mikäli edellä kerrottu totuus on liian raskas uskottavaksi saati sitten toteutettavaksi, niin ei syytä huoleen: ohjelmistolaadun käsite on onneksi muuttunut! Asiaa käsitteli mm. tietokirjailija *Petteri Järvinen* Tietokone-lehden tammikuun 2001 numerossa (s. 31-32). Järvisen mukaan käyttäjät eivät enää odotakaan tuotteen kestävän kovin pitkään vaan ovat valmiita hankkimaan uuden mallin (ohjelmistoversion) heti kun se tulee markkinoille. Eipä siis ohjelmistoyrityksenkään kannata satsata tuotteensa laatuun, koska tuote on joka tapauksessa kohta virtuaalisessa roskakorissa. Järvisen mukaan nykyään ratkaisee, ei niinkään klassinen ja virheettömyyteen pyrkivä laatu, vaan hyvä brändi ja uudet elämykset. Mitäpä siis turhia testailemaan tai katselmoimaan, kunhan pidetään hauskaa.

Jukka Paakki,
Professori,
Helsingin yliopisto,
tietojenkäsittelytieteen laitos

Muistathan ilmoittautua:

Systeemityöyhdistys SYTYKE ry:n

KEVÄTKOKOUS
JA
JÄSENTILAISUUS

tiistaina 20.3.2001 klo 15.00 jäsentilaisuus,
17.30 kevätkokous.

Katso tarkemmat tiedot jäsenpostista!

Cheap Ways to Improve Your Testing

Hans Schaefer,
Hans Schaefer Software Test Consulting

Hans Schaefer on jo pitkään kouluttanut suomalaisia testaajia ja on tuttu kansainvälisistä konferensseistakin. Hans antaa tässä oivia vinkkejä testauksen parantamisen alkuun pääsemisessä ja oikeassa asenteessa.

Your software is not good enough? It fails too often? Testing takes too long time? What to do to get it better?

Some companies know how to produce reliable software. What do they do, how do they succeed? Part of the truth is real hard work. Part of the truth is intelligence. Let¹s concentrate on the intelligent ways.

The first you do is look into your errors. What are the customer complaints? What is wrong? Get a list of all failures reported during the last month. If they are too many, get them from the last week, or whatever period. Sort them by cause, by cost, by seriousness, and/or by subsystem causing them. Putting the failure data into a spreadsheet, database, or a failure reporting system, or a statistics package helps you draw nice histograms. Now pick the worst ones. The worst being the most idiotic, most annoying, most expensive or even some randomly selected problems. Or the subsystem having most trouble. Now call your developers to a meeting. You call it ³This week¹s failure meeting². Present the chosen failures, their cost, and their causes, and then ask people to discuss. They should try to find what could be done in order to prevent this kind of problem, or find it earlier. People learn this way, that problems must be prevented, and that they are important. And people have creative ideas. And they support their own suggestions.

Then introduce whatever solution came up. Make sure it is really introduced, and measure of the corresponding defects disappear.

This method may be implemented using quality circles, or using a process improvement group, or in whatever way, and it is your main tool to find the really important improvements.

What about other cheap ways?

Have you heard about ³The buddy system²? It turns you individual workers into groups of two people. Every software worker shall choose another software worker as his or her ³buddy². For every work product, require that this product is reviewed, checked and tested by the buddy. Every programmer has a buddy and is buddy himself or herself. It is a very informal and not threatening way of reviewing documents. It is a good choice if there is no tradition in technical reviews or inspections. This way of working in a very consequent form is advocated by the people using the newly documented technique of ³Extreme Programming² (www.extremeprogramming.org). With follow up in the form of seminars, defect counting etc., it will sooner or later get people moving into more formal reviews of the most critical documents and subsystems. People also learn from each other. They soon adopt the good ideas they see others apply. And documents become more standardized. As others must understand what is written, the authors will make them more

understandable. Authors also put more care into them, because they do not want the buddy to find obvious ³idiotic² errors. How does this pay back? It pays by reducing debugging time. Debugging often destroys system structure as everything is ³fixed². So it pays by delivering a cleaner system. And it pays off because people know more about what others do in the project. There will be less interface errors. And there is one more benefit: Is someone is sick or leaves the project, there will still be the buddy to take over the job . You always have a backup person.

Sounds logical and easy? So why don¹t people use this idea? It is partly because programmers think of their modules as ³my² module, not ³our² module. Partly because nobody has time to help others check their job. Everyone has enough to do from before. Partly it is because debugging time is never accounted for. But it works. Data from real projects show it costs 20% extra work time, but you produce things 40% faster. Quality is improved in addition, and this pays off in system testing or after release.

As a project leader, you have to require it, and support it during the first two to three weeks. Because then it costs MORE. After that period it costs LESS. As a programmer you can still introduce it even without your managers knowing. Find someone and start sharing the workstation. You will turn a into much more productive team. My own experience is: 7 hours extra work where a colleague reviewed my code saved the equivalent of 21 hours debugging. How I measured it? I first debugged my code and measured the time: 7 hours test preparation and running, 21 hours debugging and retesting. The I gave the original version to my colleague and he found the same defects and more in 7 hours. Fixing them took 30 minutes...

Some cheap ways to test better?

There are a few ground rules for better testing. Get better test examples that cover more of your code and specification. You may spend a week to learn more formal methods. Half of the time in a seminar, half of the time experimenting. But the ground rules are the following:

1 - Always test the boundaries: Maximum + or - 1, minimum + or - 1, zero, forbidden or abnormal inputs. Look at the first and last elements in tables, lists and files.

2 - If your specifications or design contain any kind of diagrams (data structure, data flow, state transition, control flow, petri net, you name it), then test every box and every connection. Your program must have executed every box, and every connection between boxes. This translates to every state and state transition, every statement in the code, every branch in the code, the variation in data relationships, and access to every record type in the database. Every box, every connection. You may continue if you have the resources by combining connections, or different concepts.

3 - If something can be repeated, test the zero, the one and the more than one case. If a maximum is given, test at and beyond the maximum.

4 - If you test with a correct input, always try a wrong input, too, and try with missing input, and the default.

These are the ground rules. Apply then on any level of test where you find them applicable, and your number of errors will decrease. Measure the effect. How many customer reported defects per week? How many failures in acceptance testing? The cost for it?

What to do later on?

The effect of the cheap methods is limited. Sure you increase both quality and productivity, especially if you never did it before, but it is not enough to do so. Really high reliability requires more formal methods. Formal