

Ohjelmistotuotanto

Luento 5

12.11.

Verifiointi ja Validointi

eli ohjelmistojen laadunhallinta

Verifiointi ja Validointi

- Kehitettävän ohjelmiston elinkaareen oleellisena osana kuuluu
 - Verifiointi *"are we building the product right"* ja
 - Validointi *"are we building the right product"*
- Verifioinnissa pyritään varmistamaan, että ohjelmisto toteuttaa vaatimusmäärittelyn aikana sille asetetut vaatimukset
 - Yleensä tämä tapahtuu *testaamalla*, että ohjelma toteuttaa sille asetetut (ja vaatimusmäärittelyyn kirjatut) toiminnalliset ja ei-toiminnalliset vaatimukset
- Validointi pyrkii varmistamaan, että ohjelmisto täyttää käyttäjän odotukset
 - Vaatimusmäärittelyn aikana kirjatut ohjelmiston vaatimukset eivät ole aina se mitä käyttäjä ohjelmalta todella haluaa!

Verifiointi ja Validointi

- Verifiointin ja validoinnin tavoitteena on varmistaa että ohjelma on ”riittävän hyvä” siihen käyttötarkoitukseen, mihin ohjelma on tarkoitettu
 - Hyvyys on suhteellista ja riippuu ohjelman käyttötarkoituksesta
 - Ohjelman ei esim. tarvitse yleensä olla virheetön ollakseen kuitenkin riittävän hyvä käytettäväksi
- Verifiointin ja validoinnin suorittamista käytetään yleisesti nimitystä *laadunhallinta* (engl quality assurance, QA)
- Jos laadunhallinta on erillisen tiimin vastuulla, käytetään tästä usein nimitystä QA-tiimi

Verifioinnin ja Validoinnin tekniikat

- Perinteisesti verifioinnissa on käytetty kahta tekniikkaa
 - Katselmointeja/tarkastuksia
 - Testausta
- **Katselmoinneissa** (review) käydään läpi erilaisia ohjelmiston tuotantoprosessin aikana tuotettuja dokumentteja ja ohjelmakoodia, ja etsitään näistä erilaisia ongelmia
- **Tarkastukset** (inspection) ovat katselmointien muodollisempi versio
 - Järjestetään formaali kokous, jolla tarkkaan määritelty agenda ja kokouksen osallistujilla ennalta määritellyt roolit
- Katselmointi on *staattinen tekniikka*, suorituskelpoista ohjelmakoodia ei tarvita ja jos katselmoinnin kohteena on ohjelmakoodi, ei ohjelmaa katselmuksissa suoriteta
- **Testaus** on *dynaaminen tekniikka*, joka edellyttää aina ohjelmakoodin suorittamista
 - Testauksessa tarkkaillaan miten ohjelma reagoi annettuihin testisyötteisiin

Vaatimusten validointi

- Ohjelmistolle määritellyt vaatimukset on validoitava, eli *varmistettava, että määrittelydokumentti määrittelee sellaisen ohjelmiston, jonka asiakas haluaa*
- Vesiputousmallissa määrittelydokumentin kirjattujen vaatimusten validointi suoritetaan nimenomaan katselmoimalla
 - Vaatimusmäärittely päättyy siihen, että asiakas tarkastaa määrittelydokumenttiin kirjattujen vaatimuksien vastaavan asiakkaan kuvaa tilattavasta järjestelmästä
 - Katselmoinnin jälkeen määrittelydokumentti *jäädyytetään* ja sen muuttaminen vaatii yleensä monimutkaista prosessia
- Ketterässä ohjelmistotuotannossa vaatimusten validointi tapahtuu iteraation päättävien demonstraatioiden (Scrumissa sprint review) yhteydessä
 - Asiakkaalle näytetään ohjelman toimivaa versiota
 - Asiakas voi itse verrata vastaako lopputulos sitä mitä asiakas haluaa
 - Asiakkaan haluama toiminnallisuushan voi poiketa määritellystä toiminnallisuudesta!
 - Jos ei, on seuraavassa iteraatiossa mahdollista ottaa korjausliike
- On ilmeistä, että ketterän mallin käyttämä vaatimusten validointitapa toimii paremmin tuotekehitystyyppisissä tilanteissa, joissa ollaan tekemässä tuotetta, joka on vaikea määritellä tarkkaan etukäteen

Koodin katselmointi

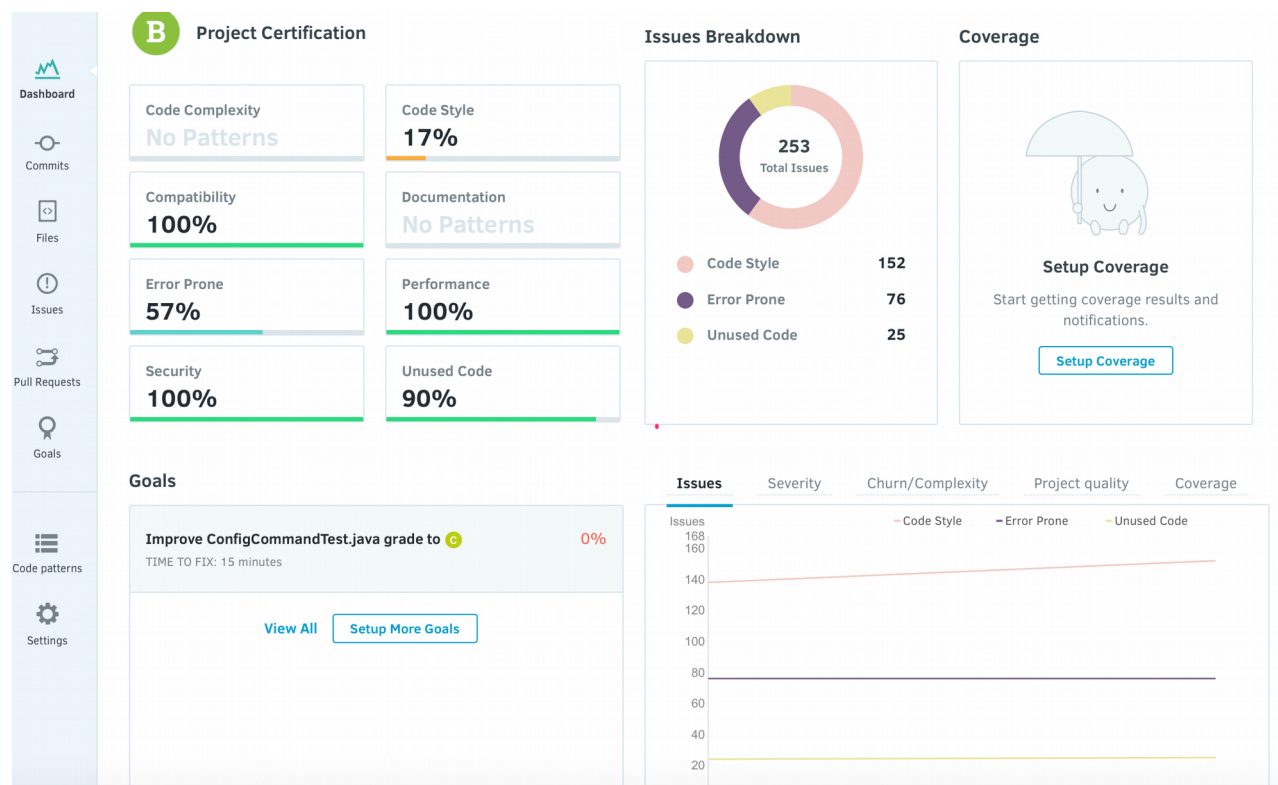
- Koodin katselmointi eli koodin lukeminen jonkun muun kuin ohjelmoijan toimesta on havaittu erittäin tehokkaaksi keinoksi koodin laadun parantamisessa
- Katselmoinnin avulla voidaan havaita koodista ongelmia, joita testauksella ei välttämättä havaita, esim.
 - noudattaako koodi sovittua tyyliä
 - onko koodi ylläpidettävää
- Koodin katselmoinnissa on perinteisesti käyty läpi onko koodissa tiettyjä ”checklisteissä” listattuja riskialttiita piirteitä, ks. esim.
 - http://www.oualline.com/talks/ins/inspection/c_check.html
 - Joissakin kielissä, esim. Javassa kääntäjän tuki tekee osan näistä tarkistuksista turhaksi
- Nykyään on tarjolla paljon katselmointia automatisoivia *staattista analyysiä* tekeviä työkaluja esim. Javalla PMD ja Checkstyle
 - <http://pmd.sourceforge.net/>
 - <http://checkstyle.sourceforge.net/>
- Tutustuimme checkstyleen viikon 2 laskareissa

Automatisoitu staattinen analyysi pilvessä

- Pilvipalvelut ovat helpottaneet sovelluskehittäjien työtä monissa asioissa, esim. GitHubin ansiosta omaa versionhallintapalvelinta ei ole enää tarvetta ylläpitää
- Pilveen on viime aikoina ilmestynyt myös koodille staattista analyysiä tekeviä palveluita, esim. <https://codeclimate.com/>
- CodeClimate analysoi koodista mm. seuraavia asioita:
 - Liian kompleksiset metodit ja luokat
 - Copy paste -koodi
 - Testaamaton koodi
- CodeClimate myös huomauttaa koodin laadun muutoksista, esim. jos koodin kompleksisuus kasvaa muutosten yhteydessä ja antaa parannusehdotuksia liian monimutkaisiin metodeihin
- Minkä tahansa GitHubissa olevan Ruby, Javascript tai PHP-projektin saa konfiguroitua Codeclimaten tarkastettavaksi nappia painamalla
- Codeclimate suorittaa tarkastukset koodille aina kun uutta koodia pushataan GitHubiin
- Labtoolin eli laitoksen harjoitustöiden kirjanpito-ohjelmiston CodeClimate-raportti löytyy osoitteesta <https://codeclimate.com/github/mluukkai/labtool>

Automatisoitu staattinen analyysi pilvessä

- Hiljattain ilmestynyt pilvipalvelu <https://www.codacy.com> osaa tehdä staattista analyysiä myös Javalla tehdyille ohjelmille
 - Tutustumme viikon 2 laskareissa codacyyn
- Codacy analysoi osin samoja asioita kuin Codelimate, mutta erojakin löytyy. Codacy osaa identifioida koodista mm. tietoturvaan liittyviä ongelmia
- Ote TMC:n komentorivicientin raportista



Koodin katselmointi: GitHub ja pull requestit

- Yhä enenevä määrä ohjelmistotuotantoprojekteja tallettaa lähdekoodinsa GitHubiin
- GitHubin *pull requestit* tarjoavat hyvän työkalun koodikatselmointien tekoon
- Pull requesteja käytettäessä työn kulku on seuraava
 - Sovelluskehittäjä *forkkaa* repositorin itselleen, tekee muutokset omaan repositorioon ja tekee *pull requestin* projektia hallinnoivalle taholle
 - Hallinnoija, esim. tiimin ”senior developer” tai opensource-projektin vastaava tekee katselmoinnin pull requestille
 - Jos koodi ei ole vielä siinä kunnossa että tehdyt muutokset voidaan *mergetä* repositorioon, kirjoittaa hallinnoija pull requestin tekijälle joukon parannusehdotuksia
 - Muutosten ollessa hyväksyttävässä kunnossa, pull request *mergetään* päärepositorioon
- Seuraavalla sivulla esimerkki TMC-projektiin tehdystä pull requestistä ja siihen liittyvistä kommentteista

Koodin katselmointi: GitHub ja pull requestit

 testmycode / tmc-server

 Unwatch ▾ 8

Course participants #201

 Open kennyhei wants to merge 9 commits into `testmycode:master` from `rage:course-participants`

 Conversation 24

 Commits 9

 Files changed 13









kennyhei commented on Oct 27, 2014

Implementing [#185](#)



kennyhei added some commits on Oct 21, 2014

-   Course JSON with participants 9287e10
-   Course knows its students through submissions and vice versa e3e7c03
-   Prettier JSON b1b5dd7



@@ -31,6 +32,17 @@ def course_data(course)

31

32

})

32

33

end

33

34

+ # Course JSON with participants

+ def course_participants_data(course)

+ participants = course.users

+

+ data = {

+ :id => course.id,

+ :name => course.name,

+ :participants => participants.map {|participant| participant_data(participant_data(course_id: course.id, participant_id: participant.id))}



mpartel added a note on Oct 29, 2014

Owner



On my desktop, with the mooc production DB dump, this takes around 30 seconds for the k2014-mooc course. I'd really like to avoid adding more really slow queries to TMC.

Would the following make sense?

- Let this only return a list of participants and their newest submission IDs.
- Load a user's exercise statuses on demand, and cache them either on your side or maybe in TMC until the submission ID changes.
- Consider having the per-user URL support [ETags](#).

Koodin katselmointi ketterissä menetelmissä

- Toisin kuin Scrum, **eXtreme Programming** eli **XP** määrittelee useita käytänteitä, joita pyritään noudattamaan ohjelmistoa tehtäessä
 - Suuri osa XP:n käytänteistä on hyvin tunnettuja ”best practiseja”, mutta kuitenkin usein vietyä äärimmäiseen (extreme) muotoon
- Osa käytänteistä tähtää ohjelmiston laadun maksimoimiseen ja kolmen voidaan ajatella olevan katselmoinnin äärimmilleen vietyjä muotoja
- **Pariohjelmoinnissa** (pair programming) kaksi ohjelmoijaa työskentelee yhdessä yhdellä koneella
 - Koodia kirjoittava osapuoli toimii *ohjaajana* (driver) ja toinen *navigoijana* (navigator), roolia vaihdetaan sopivin väliajoin
 - Navigoiija tekee koodiin **jatkuvaa katselmointia**
 - Etuja:
 - Parantaa ohjelmoijien kuria ja työhön keskittymistä
 - Hyvä oppimisen väline: ohjelmoijat oppivat toisiltaan erityisesti noviisit kokeneimmilta, järjestelmän tietyn osan tuntee aina useampi ohjelmoija
 - Tutkimuksissa todettu vähentävän bugien määrää 15-50%, kokonaisresurssin kulutus kuitenkin nousee hieman

Koodin katselmointi ketterissä menetelmissä

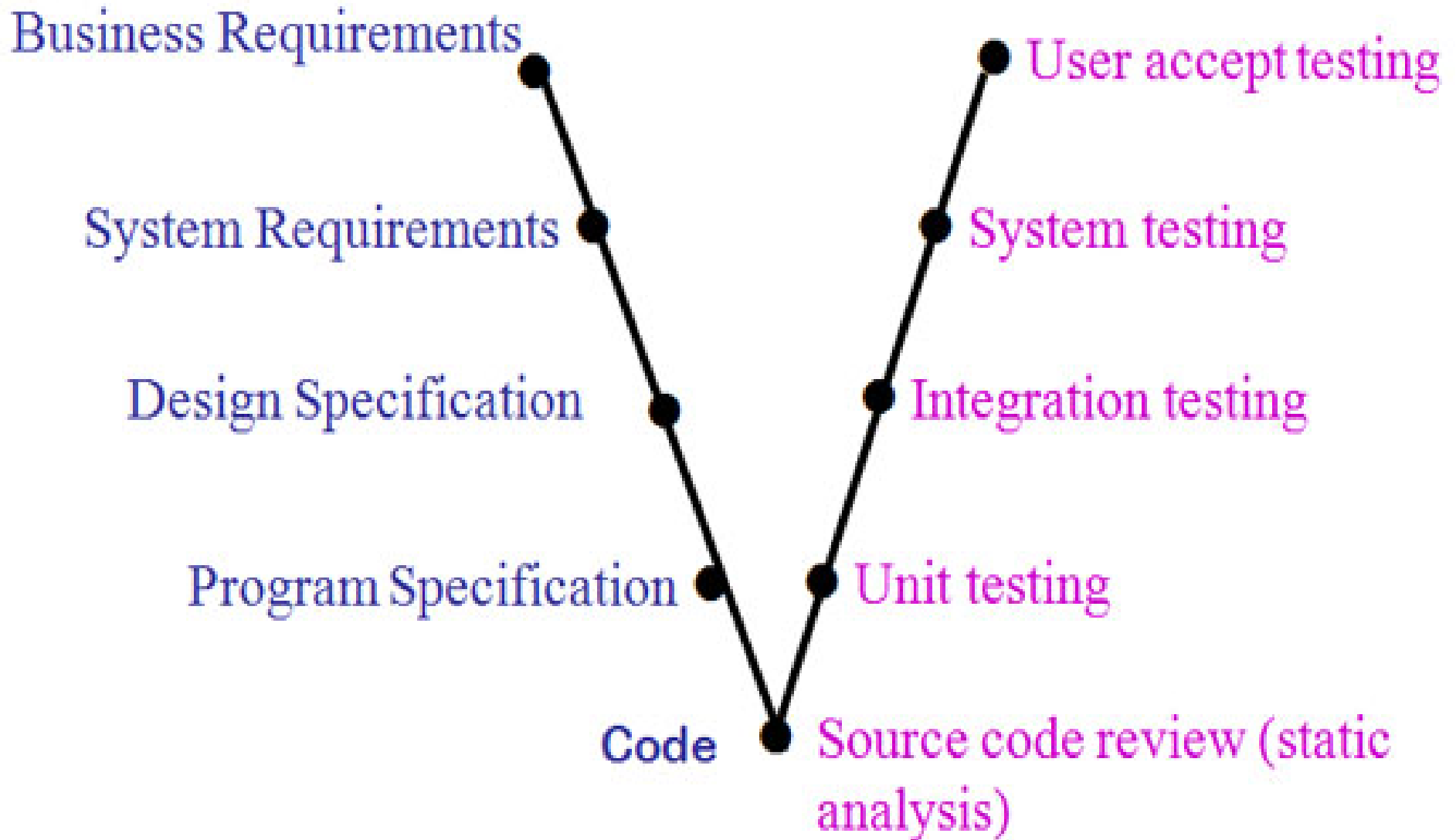
- Pariohjelmoinnin ohella ”katselmointimaisia” tavoitteita koodin laadun nostajana voidaan ajatella olevan XP:n yhteisomistajuuden ja ohjelmointistandardien periaatteilla
- Koodin **yhteisomistajuus** (collective code ownership) tarkoittaa periaatetta, jossa kukaan yksittäinen ohjelmoija ei hallitse yksin mitään kohtaa koodista
 - Kaikilla on lupa tehdä muutoksia mihin tahansa kohtaan koodia
 - Pariohjelmointi tukee yhteisomistajuutta
 - Yhteisomistajuudessa on omat riskinsä: joku koodia kunnolla tuntematon voi saada pahaa jälkeä aikaan. XP eliminoi tämän testauksiin liittyvillä käytänteillä
- Ohjelmointistandardi **coding standards** tarkoittaa, että tiimi määrittelee koodityylin, johon kaikki ohjelmoijat sitoutuvat
 - Tyylillä tarkoitetaan nimeämiskäytäntöä, koodin muotoilua ja myös tiettyjä ohjelman rakenteeseen liittyviä seikkoja
 - Ohjelmointistandardin noudattamista voidaan kontrolloida osittain automaattisesti staattisen analyysin työkaluilla

Testaus

- Ohjelmien osoittaminen virheettömäksi on käytännössä mahdotonta
- Testauksen tarkoituksena onkin **vakuuttaa asiakas ja järjestelmän kehitystiimi siitä, että ohjelmisto on tarpeeksi hyvä käytettäväksi**
- Testauksella on kaksi hieman eriävää tavoitetta
 - osoittaa, että ohjelmisto täyttää sille asetetut vaatimukset
 - käytännössä tämä tarkoittaa vaatimusdokumenttiin/vaatimukseen kirjattujen asioiden toteutumisen demonstroimista
 - löytää ohjelmistosta virheitä
 - yritetään rikkoa tai saattaa ohjelma jollain tavalla epäkonsistenttiin tilaan
- Molemmat näistä tavoitteista tähtäävät oikeastaan ohjelman **ulkoisen laadun** (external quality) eli käyttäjän kokemuksen laadun parantamiseen
 - **External Quality is the fitness for purpose of the software.** It's most obvious measure is the Functional Tests, and some measure of the bugs that are still loose when the product is released.
 - <http://c2.com/cgi/wiki?InternalAndExternalQuality>

Testauksen tasot

- Testausta jakaantuu eri *tasoihin* sen mukaan mukaan, mikä testauksen ensisijaisena kohteena on
- Vesiputousmallia laajentava *testauksen V-malli* havainnollistaa testauksen eri tasoja



Testauksen tasot

- **Yksikkötestaus (unit testing)**
 - Yksittäisten luokkien, metodien ja moduulien testaus erillään muusta kokonaisuudesta
 - Yleensä ohjelmoijat suorittavat
- **Integraatiotestaus (integration testing)**
 - Yksittäin testattujen komponenttien liittäminen yhteen eli integrointi ja kokonaisuuden testaus
 - Integroinnin tekevät sovelluskehittäjät suorittavat yleensä myös testauksen
- **Järjestelmätestaus (system testing)**
 - Varmistetaan että järjestelmä toimii vaatimuksiin kirjatulla tavalla
 - Testataan järjestelmää saman rajapinnan kautta, jonka kautta järjestelmää käytetään
 - Kehittäjäorganisaatio suorittaa
 - Jakautuu useisiin alalajeihin joista kohta lisää
- **Käyttäjän hyväksymistestaus (user acceptance testing)**
 - Loppukäyttäjän tuotteelle suorittama testaus

Järjestelmätestaus

- Tarkoitus siis varmistaa, että järjestelmä toimii vaatimuksiin kirjatulla tavalla
- Testataan järjestelmää saman rajapinnan kautta, jonka kautta järjestelmää käytetään
- Testaus tapahtuu ilman tietoa järjestelmän sisäisestä rakenteesta eli kyseessä **black box -testaus**
- Yleensä järjestelmätestaus perustuu järjestelmän potentiaalsiin käyttöskenaarioihin
 - jos vaatimukset on ilmaistu user storyina, on niistä melko helppo muotoilla testejä, joiden avulla voidaan varmistaa että järjestelmällä on storyjen kuvaamat vaatimukset sekä tyypilliset virheskenaariot
- ”perusmuotonsa” eli vaatimuksiin kirjattujen toiminnallisten vaatimuksien testaamisen lisäksi järjestelmätestaukseen kuuluu mm:
 - Käytettävyytestaus
 - Suorituskykytestaus tai stressitestaus
 - Tietoturvan testaus
 - lisää http://en.wikipedia.org/wiki/System_testing

Testitapausten valinta

- Kattava testaaminen on mahdotonta ja testaus joka tapauksessa työlästä
- Onkin tärkeää löytää kohtuullisen kokoinen testitapausten joukko, jonka avulla on kuitenkin mahdollista löytää mahdollisimman suuri määrä virheitä
- Testitapaus testaa järjestelmän toiminnallisuutta joillakin **syötteillä**
- Useat syötteen ovat järjestelmän toiminnan kannalta samanlaisia
- Testeissä kannattaakin pyrkiä jakamaan syötteen **ekvivalenssiluokkiin** ja tehdä yksi testitapaus kutakin ekvivalenssiluokkaa tai syötteiden ekvivalenssiluokkien kombinaatiota kohti
 - Testin kannalta samalla tavalla käyttäytyvät syötteen siis muodostavat oman ekvivalenssiluokkansa
- Ekvivalenssiluokkien edustajien lisäksi kannattaa tehdä myös testitapaukset ekvivalenssiluokkien **raja-arvoille**

Testisyötteiden valinta

- Mitä testitapauksia kannattaisi valita seuraavalla sivulla olevalle **tekstiv:n sivun valintaikkunaan**?
 - Tekstiv:n *sivu* vastaa lukua väliltä 100-899
 - Osaa välin luvuista vastaavaa sivua ei ole olemassa
- Testisyötteen ekvivalenssiluokkia olisivat ainakin seuraavat
 - Olemassaolevaa sivua vastaavat luvut
 - Validit luvut jotka eivät vastaa mitään sivua
 - Liian pienet ja liian suuret luvut
 - Syötteet jotka sisältävät kiellettyjä merkkejä
 - Tyhjä syöte
- Jokaisesta ekvivalenssiluokasta olisi siis hyvä valita ainakin yksi testattava syötearvo
- Olemassaolevaa sivua vastaavan ekvivalenssin rajatapaukset, eli luvut 100 ja 899 kannattaisi ehkä valita testisyötteiksi
- Samoin luvut 99 ja 900 jotka ovat oman ekvivalenssiluokkansa rajatapauksia

Tekstiv:n testitapaukset

[Edellinen sivu](#) | [Edellinen alisivu](#) | [Seuraava alisivu](#) | [Seuraava sivu](#)

Teksti-TV

[yle.fi/tekstiv](#) [199 PÄÄHAKEMISTO](#)

[106](#) Taksisääntelyn purkaminen etenee

[107](#) Tieto irtisanoo lähes [180](#)

[162](#) Reuters: Kreikan kasvu yllätti

[651](#) MM-karsinta: Turkki nujersi Suomen

[221](#) Janne Keränen jälleen Kalpa-sankari

101	UUTISET	160	TALOUS	190	ENGLISH
201	URHEILU	350	RADIOT	470	VEIKKAUS
300	OHJELMAT	400	SÄÄ	575	TEKSTI-TV
799	SVENSKA	500	ALUEET	890	KALENTERI
Sää paikkakunnittain				406-408	
Nopea piiras				811	

[Edellinen sivu](#) | [Edellinen alisivu](#) | [Seuraava alisivu](#) | [Seuraava sivu](#)

[Kotimaa](#) | [Ulkomaat](#) | [Talous](#) | [Urheilu](#) | [Svenska sidor](#) | [Teksti-TV Yle.fi](#) [YLE Uutiset](#)

Sivun valinta:

OK

Yksikkötestaus

- Kohteena siis yksittäiset metodit ja luokat
- Ohjelmoijat suorittavat yksikkötestauksen
- Testattavan koodin rakenne otetaan huomioon testejä laatiessa, eli kyseessä **lasilaatikkotestaus** (white box testing)
- Yksikkötestauksella ei testata suoranaisesti sitä täyttääkö ohjelmisto vaatimuksensa, pikemminkin tavoitteena on ohjelman **sisäisen laadun** (internal quality) kontrollointi
 - Internal quality is about the design of the software
 - This is purely the interest of development
 - If Internal quality starts falling the system will be less amenable to change in the future
 - Hence the need for refactoring, clear coding, relentless testing, and the like
 - You need to be very careful about letting internal quality slip
 - <http://c2.com/cgi/wiki?InternalAndExternalQuality>

Yksikkötestaus

- Ohjelman sisäinen laatu siis vaikuttaa erityisesti siihen, miten ohjelmaa voidaan laajentaa ja jatkokehittää
- Ketterissä menetelmissä sisäisellä laadulla onkin todella suuri merkitys, tähän palataan tarkemmin torstaina
- Pelkän sisäisen laadun kontrollimekanismi yksikkötestaus ei toki ole
- Kattavilla yksikkötesteillä saadaan parannettua myös ohjelman ulkoista, eli asiakkaan kokemaa laatua
 - Yksikkötestit voivat eliminoida joitain asiakkaalle näkyviä virheitä, joita järjestelmätestauksen testitapaukset eivät löydä
- Bugit on taloudellisesti edullista paikallistaa mahdollisimman aikaisessa vaiheessa, eli yksikkötestauksessa löydetty virhe on halvempi ja nopeampi korjata kuin järjestelmä- tai integraatiotestauksessa löytyvä virhe
- Koska yksikkötestejä joudutaan ajamaan moneen kertaan, tulee niiden suorittaminen ja testien tulosten raportointi automatisoida, ja nykyinen hyvä työkalutuki tekeekin automatisoinnin helpoksi
 - xUnit eniten käytetty, uudempia tulokkaita mm. rspec, jest, jasmine

Mitä ja miten paljon tulee testata?

- Mitä tulisi testata yksikkötestein? JUnitin kehittäjän Kent Beckin vastaus:
"Do I have to write a test for everything?"
"No, just test everything that could reasonably break"
[\[http://junit.sourceforge.net/doc/faq/faq.htm\]](http://junit.sourceforge.net/doc/faq/faq.htm)
- Vastaus ei siis ole helppo. Ainakin tulisi olla testitapaukset
 - kaikkien metodien (ja loogisten metodikombinaatioiden) toiminta parametrien hyväksyttävillä arvoilla
 - ja virheellisillä parametrien arvoilla
- Parametrien mahdolliset arvot kannattaa jakaa ekvivalenssiluokkiin (ks. kalvo 19) ja jokaisesta luokasta valita yksi arvo testiä varten, myös ekvivalenssiluokkien raja-arvot kannattaa valita mukaan
- Koska yksikkötestejä tehtäessä ohjelmakoodi on nähtävillä, on testattavien arvojen parametrien ekvivalenssiluokat ja raja-arvot pääteltävissä koodista
- Esim. 1 viikon laskareista tutun Ohtuvaraston metodi *otaVarastosta*
 - mitä testitapauksia tulisi generoida jotta kaikki edelläolevat ohjeet täyttyvät?


```

public class Varasto {
    private double tilavuus;
    private double saldo;

    public double otaVarastosta(double maara) {
        if (maara < 0) return 0.0;

        if (maara > saldo) {
            double kaikkiMitaVoidaan = saldo;
            saldo = 0.0;
            return kaikkiMitaVoidaan;
        }

        saldo = saldo - maara;
        return maara;
    }
}

```













- Metodia *otaVarastosta* testatessa testitapauksessa on huomioitava parametrin maara lisäksi varaston tilanne
- Varastotilanteita on kolmea ”ekvivalenssiluokkaa”
 - Tyhjä (esim. saldo 0, tilavuus 10)
 - Ei tyhjä eikä täysi (saldo 5, tilavuus 10)
 - Täysi (saldo 10, tilavuus 10)
- Näitä kutakin kohti on metodin parametrilla *maara* omat ekvivalenssiluokkansa
 - Esim. täysi varasto:
maara = -1, 0, 5, 10, 11
- *Tarvitseeko esim. nollan tai negatiivisen määrän ottamista tarkastaa kaikkien varastotilanteiden yhteydessä?*

Testauskattavuus

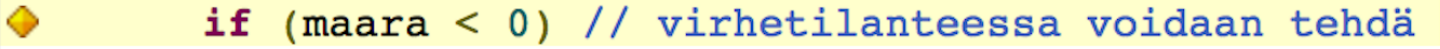

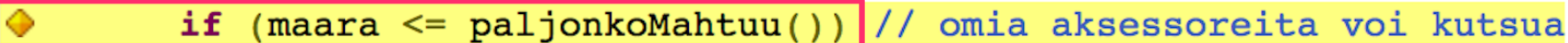
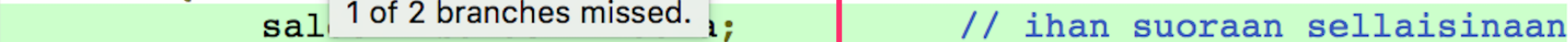

- Yksikkötestien (ja toki myös muunkinlaisten testien) hyvyyttä voidaan mitata **testauskattavuuden** (test coverage) käsitteellä
- Testauskattavuutta on muutamaa eri tyyppiä
- **Rivikattavuudella** (line coverage) tarkoitetaan kuinka montaa prosenttia testattavan metodin/luokan koodirivejä testimetodit suorittavat
 - Vaikka rivikattavuus olisi 100% ei tämä tietenkään tarkoita, että kaikki oleellinen toiminnallisuus olisi tutkittu
- **Haarautumakattavuudella** (branch coverage) tarkoitetaan kuinka montaa prosenttia testattavan metodin/luokan sisältävistä ehtolauseiden haaroista testit ovat suorittaneet
- Monet työkalut, esim. käyttämämme JaCoCo mittaavat testien suorituksen yhteydessä testauskattavuuden
- Muitakin kattavuuden tyyppejä on olemassa, mm. *ehtokattavuus* ja *polkukattavuus*, useat työkalut eivät niitä kuitenkaan testaa
- **Hyvät yksikkötestit siis saavuttavat mahdollisimman suuren kattavuuden ja ottavat huomioon edellisen sivun ohjeistuksen**

Testauskattavuus JaCoCossa

- JaCoCo ilmoittaa sekä rivi- (instruction) että haaraumakattavuuden (branches)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
Varasto(double, double)		0%		0%	4	4	10	10	1	1
toString()		0%		n/a	1	1	1	1	1	1
otaVarastosta(double)		62%		50%	2	3	4	8	0	1
lisaaVarastoon(double)		77%		50%	2	3	2	6	0	1
Varasto(double)		82%		50%	1	2	1	6	0	1
paljonkoMahtuu()		100%		n/a	0	1	0	1	0	1
getSaldo()		100%		n/a	0	1	0	1	0	1
getTilavuus()		100%		n/a	0	1	0	1	0	1
Total	66 of 126	47%	11 of 16	31%	10	16	18	34	2	8

- Epäkattavasti testattu haarautumiskohta esim. if ilmaistaan keltaisella

```
51.     public void lisaaVarastoon(double maara) {
52.          if (maara < 0) // virhetilanteessa voidaan tehdä
53.         {
54.              return; // tällainen pikapoistuminenkin!
55.         }
56.          if (maara <= paljonkoMahtuu()) // omia aksessoreita voi kutsua
57.         {
58.              saldo = 1; // ihan suoraan sellaisinaan
59.         } else {
60.              saldo = tilavuus; // täyteen ja ylimäärä hukkaan!
61.         }
62.     }
63.
```

1 of 2 branches missed.

Mutaatiotestaus

- Pelkkä testikattavuus ei vielä kerro paljoakaan testien hyvyydestä, Hyvien testien tulisi olla sellaisia, että jos ohjelmaan tulee bugi, huomaavat testit virheen
- **Mutaatiotestauksen** (engl. mutation testing) idea on nimenomaan testata testitapausten hyvyyttä generoimalla koodiin systemaattisesti *mutantteja* eli pieniä "bugeja" ja katsoa havaitsevatko testit koodiin tulleet bugit
- Erilaisia mutanttityyppejä, joita mutaatiotestauksessa koodiin generoidaan on paljon erilaisia, mm.
 - Manipuloidaan ehtolausetta: $\text{if} (x < 0) \rightarrow \text{if} (x \leq 0)$ tai $\text{if} (\text{true})$
 - Vaihdetaan operaattoria: $x += 1 \rightarrow x -= 1$
 - Kovakoodataan paluuarvo: $\text{return } x; \rightarrow \text{return true};$
 - Korvataan konstruktorikutsu: $\text{olio} = \text{new Olio}() \rightarrow \text{olio} = \text{null};$
- Mutaatiotestauksen ongelmana on mutaatioiden suuri määrä ja ns. *ekvivalentit mutantit*, joiden takia mutaatiotestauksen tulos vaatii aina ihmisen tulkintaa
 - Ekvivalentti mutantti tarkoittaa muutosta koodissa, joka ei kuitenkaan muuta ohjelman toiminnallisuutta. Eli mutantin lisäämistä koodiin ei voi mikään testi havaita. Mutantin toteaminen ekvivalentiksi algoritmisesti on mahdotonta
- Lisätietoa
 - http://en.wikipedia.org/wiki/Mutation_testing ja <http://pitest.org/>

Integraatiotestaus

- Järjestelmän yksittäiset, erillään yksikkötestatut luokat tulee **integroida** toimivaksi kokonaisuudeksi
- Integroinnin yhteydessä tai sen jälkeen suoritetaan **integroititestausta**
- Integraatiotestauksen painopiste on ohjelman komponenttien välisten rajapintojen toimivuuden tutkimisessa sekä komponenttien yhdessä tuottaman toiminnallisuuden oikeellisuuden varmistamisessa
- Järjestelmän integrointi voi edetä joko järjestelmän rakenteeseen perustuen tai järjestelmän toteuttamien ominaisuuksien mukaan
 - **Rakenteeseen perustuvassa** integraatiossa keskitytään kerrallaan sovelluksen yksittäisten rakenteellisten komponenttien integrointiin
 - Esim. olutkaupassa integroitaisiin sovelluslogiikan luokat, käyttöliittymän toteutus ja tietokantarajapinta omina kokonaisuuksinaan
 - **Ominaisuuksiin perustuvassa** integroinnissa, taas liitetään yhteen alikomponentit, jotka toteuttavat järjestelmän loogisen toimintakokonaisuuden
 - Olutkaupassa voitaisiin esim. integroida kerrallaan kaikki toiminnallisuuteen ”lisää tuote ostoskoriin” liittyvät luokat

Integraatiotestaus

- Sekä rakenteeseen, että ominaisuuksiin perustuva integrointi voi tapahtua joko ylhäältä alas tai alhaalta ylös:
 - **Bottom up:** lähdetään liikkeelle yksittäisistä komponenteista, liitetään niitä yhteen ja suoritetaan testejä kunnes kaikki integroitavat komponentit on yhdistetty
 - **Top-down:** ensin kehitetään järjestelmän korkean tason rakenteet siten, että yksittäisten komponenttien paikalla on *tynkäkomponentteja* (stub). Tyngät korvataan sitten yksi kerrallaan todellisilla komponenteilla koko ajan kokonaisuutta testaten
- Oldschool-ohjelmistotuotannossa toimintatapa oli se, että kaikki ohjelman yksittäiset komponentit ohjelmoitiin ja yksikkötestattiin erikseen ja tämän jälkeen ne integroitiin (yleensä rakenteeseen perustuen) kerralla yhteen
 - *Tämän tyylinen big bang -integraatio* on osoittautunut todella riskialttiiksi (seurauksena usein ns. integraatiohelvetti) ja sitä ei enää kukaan täysijärkinen suosittelen käyttäväksi
- Moderni ohjelmistotuotanto suosii ns. **jatkuvaa integraatiota**, joka on hyvin tiheässä tahdissa tapahtuvaa *ominaisuuksiin perustuvaa integrointia*
 - Palaamme aiheeseen tiistaina

Regressiotestaus

- Iteratiivisessa ja ketterässä ohjelmistotuotannossa, jossa jokainen iteraatio tuottaa ohjelmistoon uusia ominaisuuksia, on oltava tarkkana, että lisäykset eivät hajota ohjelman jo toimivia osia
- Testit siis on suoritettava uudelleen aina kun ohjelmistoon tehdään muutoksia
- Tätä käytäntöä sanotaan **regressiotestaukseksi**
- Regressiotesteinä ei välttämättä tarvitse käyttää kaikkia ohjelmiston testejä, sopiva osajoukko voi taata riittävän luottamuksen
 - Regressiotestijoukko koostuu siis yksikkö-, integraatio- ja järjestelmätesteistä
- Testaus on erittäin työlästä ja regressiotestauksen tarve tekee siitä entistä työläämpää
- Tämän takia on erittäin tärkeää pyrkiä automatisoimaan testit mahdollisimman suurissa määrin
- Käsitlemme muutamia järjestelmätason testauksen automatisoinnin menetelmiä seuraavalla luennolla