

Ohjelmistotuotanto

Luento 5

26.3.2012

Verifiointi ja Validointi

Verifiointi ja Validointi

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

Verifioinnin ja Validoinnin tekniikat

- Perinteisesti verifioinnissa on käytetty kahta tekniikkaa
 - Katselmointeja ja tarkastuksia
 - Testausta
- **Katselmoinneissa** (review) käydään läpi erilaisia ohjelmiston tuotantoprosessin aikana tuotettuja dokumentteja ja ohjelmakoodia ja etsitään näistä erilaisia ongelmia
- **Tarkastukset** (inspection) ovat katselmusten muodollisempi versio
 - Järjestetään formaali kokous, jolla tarkkaan määritelty agenda ja kokouksen osallistujilla ennaltamääritellyt roolit
- Katselmointi on *staattinen tekniikka*, suorituskelpoista ohjelmakoodia ei tarvita ja jos katselmoinnin kohteena on ohjelmakoodi, ei ohjelmaa katselmuksissa ajeta
- **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 jäädytetään ja sen muuttaminen vaatii yleensä monimutkaista prosessia
- Ketterässä prosessissa 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 siis poiketa määritellystä toiminnallisuudesta!
 - Jos ei, on seuraavassa iteraatiossa mahdollista ottaa korjausliike
- On ilmeistä, että ketterän mallin käyttämä vaatimusten validointiprosessi toimii paremmin tuotekehitystyyppisissä tilanteissa joissa ollaan tekemässä jotain 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 parantamiseen
- Katselmoinnin avulla voidaan havaita koodista ongelmia, joita testauksella ei välttämättä havaita, esim.
 - Noudattaako koodi sovittua tyyliä
 - Onko koodi ylläpidettävää
- Koodikatselmoinnissa 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 koodille staattista analyysiä tekeviä työkaluja esim. Javalla PMD ja Checkstyle
 - <http://pmd.sourceforge.net/>
 - <http://checkstyle.sourceforge.net/>

Koodin katselmointi ketterissä menetelmissä

- Toisin kuin Scrum, eXtreme Programming eli XP määrittelee useita käytänteitä joita ohjelmistoa tehdessä on noudatettava
 - Suuri osa XP:n käytänteistä on hyvin tunnettuja ”best practiseja” mutta kuitenkin usein vietyä äärimmäiseen (extreme) muotoon
- Osa käytänteistä tähtää ohjelmiston laadun maksimoimiseen, 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
 - navigoija 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

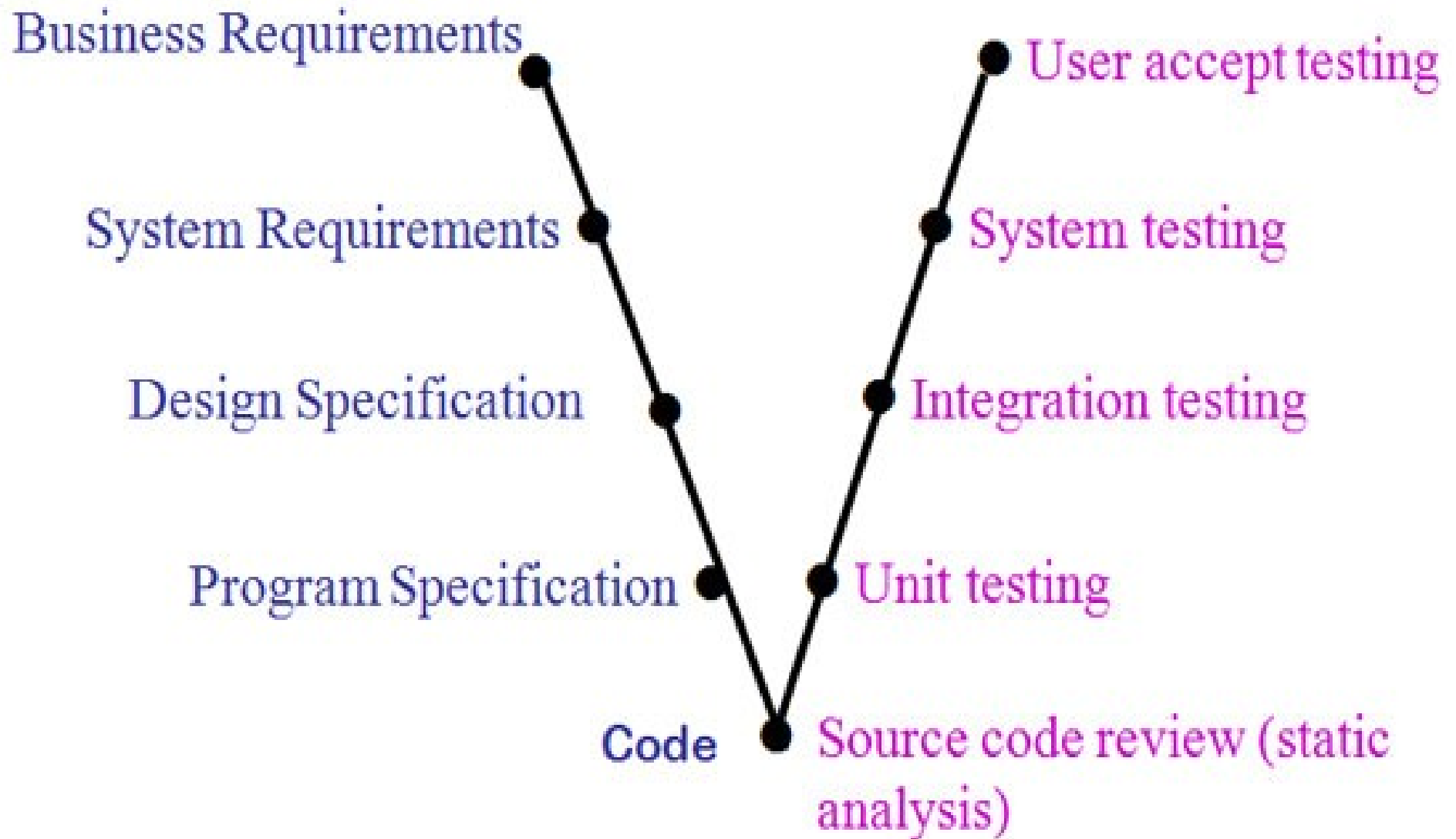
- Lisää pariohjelmoinnista esim. seuraavassa:
 - http://jamesshore.com/Agile-Book/pair_programming.html
- Pariohjelmoinnin ohella ”katselmointimaisia” tavoitteita koodin laadun nostajana voidaan ajatella olevan XP:n yhteisomistajuuden ja ohjelmointistandardien periaatteilla
- Koodin **yhteisomistajuus** (collective code ownership) tarkoittaa periaatetta jossa kukaan yksittäinen ohjelmoija ei omista 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 niitä testauksiin liittyvillä käytänteillä
 - http://jamesshore.com/Agile-Book/collective_code_ownership.html
- Ohjelmointistandardi **coding standards** tarkoittaa, että tiimi määrittelee koodityylin johon kaikki ohjelmoijat sitoutuvat
 - Tyyllillä tarkoitetaan nimeämiskäytäntöä, koodin muotoilua ja myös tiettyjä ohjelman rakenteeseen liittyviä seikkoja
 - http://jamesshore.com/Agile-Book/coding_standards.html

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 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) 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 myös eri tasoihin sen mukaan minkä testauksen 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 tekevä sovelluskehittäjä suorittaa 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, eli kyseessä **black box -testaus**
 - Testaus tapahtuu ilman tietoa järjestelmän sisäisestä rakenteesta
- Yleensä järjestelmätestaus perustuu järjestelmän potentiaalsiin käyttöskenaarioihin
 - jos vaatimukset on ilmaistu User Story -muodossa, 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

Testisyötteiden valinta

- Kattava testaaminen on mahdotonta ja testaus joka tapauksessa työlästä
- onkin tärkeää löytää kohtuullisen kokoinen testisyötteiden joukko jonka avulla on mahdollista löytää mahdollisimman suuri määrä virheitä
- Useat syötteet ovat järjestelmän toiminnan kannalta samalaisia
- Testeissä kannattaakin pyrkiä jakamaan syötteet **ekvivalenssiluokkiin** ja tehdä yksi testitapaus kutakin ekvivalenssiluokkaa kohti
- Ekvivalenssiluokkien edustajien lisäksi kannattaa tehdä myös testitapaukset ekvivalenssiluokkien **raja-arvoille**
- Seuraavalla sivulla on tuttu kaavake, minkälaisia testitapauksia kannattaisi valita, jos oletetaan, että järjestelmä olisi määritelty seuraavasti
 - Oikein täytetty kaavake hyväksytään ja kaavakkeen tiedot talletetaan järjestelmään
 - Väärän syötearvon omaava kaavake hylätään
 - Käyttäjä palautetaan lomakenäkymään ja väärin syötetyn kentän kohdalle annetaan virheilmoitus

Tee uusi kirjaus

Opiskelijanumero		
Etunimi		
Sukunimi		
GitHub-tunnus		
Jenkins-projektisi url		(kopioi selaimen osoitekentästä)
Tehtävät		listaa pilkulla erotettuina, ei välilyöntejä välissä esim: 1,2,4,5
Käytetyt tunnit		
	add	

- Kenttien syötemuodot on määritelty seuraavasti
 - Opiskelijanumero koostuu 8 numerosta ja alkaa numerolla 1
 - Etu- ja sukunimi ovat epätyhjiä kirjaimista koostuvia merkkijonoja
 - GitHub-tunnus on epätyhjä alfanumeerisista merkeistä koostuva merkkijono
 - Jenkins-url on muotoa http://jenkins.staff.cs.helsinki.fi/job/*-viikko1/ oleva merkkijono
 - Tehtävät on merkkijono joka koostuu pilkulla erotetuista numeroista, kentän arvo voi myös olla tyhjä
 - Käytetyt tunnit on kokonaisluku

Viikkopalautealomakkeen testitapaukset

- Esim. kentän opiskelijanumero ekvivalenssiluokat voisivat olla
 - Kelvollinen syöte
 - 8 lukua sisältävä muulla kuin 1:llä alkava numerosarja
 - Vähemmän kuin 8 lukua sisältävä numerosarja
 - Yli 8 lukua sisältävä numerosarja
 - Jonkin muun merkin kuin numeron sisältävä syöte
- Kaikkien syötekenttien ekvivalenssiluokkien kombinaatio olisi jo tässä yksinkertaisessa tapauksessa todella suuri
- Järkevä testistrategia lienee valita testitapaukset seuraavasti
 - testataan jokaista kenttää siten että kaikissa muissa kentissä on joku validi syöte ja kokeillaan yksittäiselle kentälle kaikkia sen syötteen ekvivalenssiluokkia
 - Tämä testistrategia olettaa että erillisten kenttien validointi on toisistaan riippumatonta, ja että esim. kahden kentän yhtäaikainen virheellinen syöte ei saa aikaan mitään kummallista (esim. kaada järjestelmää)

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 keskiviikkona
- Pelkän sisäisen laadun kontrollimekanismi yksikkötestaus ei toki ole
- kattavalla yksikkötesteillä saadaan parannettua myös ohjelman ulkoista, eli asiakkaan näkemää 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 järjestelmätestauksessa tai integraatiotestauksessa löytyvä virhe
- Koska yksikkötestejä joudutaan ajamaan moneen kertaan, kannattaa niiden suorittaminen ja testien tulosten raportointi automatisoida, ja nykyinen hyvä työkalutuki tekeekin automatisoinnin helpoksi
 - xUnit eniten käytetty, uudempia tulokkaita mm TestNG ja RSpec

Mitä ja miten paljon tulee testata?

- Mitä tulisi testata yksikkötestein?

"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>]

- Vastaus ei siis ole helppo. Ainakin tulisi olla testitapaukset
 - Kaikkiin metodien (ja loogisten metodikombinaatioiden) toiminta parametrien hyväksyttävillä arvoilla
 - Ja virheellisillä parametrien arvoilla
- Parametrien mahdolliset arvot kannattaa jakaa ekvivalenssiluokkiin (ks. kalvo 13) 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. Varaston metodi otaVarastosta, mitä testitapauksia tulisi generoida jotta kaikki edelläolevat ohjeet täyttyvät?
 - http://www.cs.helsinki.fi/u/wikla/ohjelmointi/materiaali/02_oliot

```

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 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 Cobertura mittaavat testien suorituksen yhteydessä testauskattavuuden
 - Coberturassa haarautumakattavuus on nimellä "conditionals"
- 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**

Integraatiotestaus

- Oldschool-ohjelmoistotuotannossa toimintatapa oli se, että kaikki ohjelman yksittäiset komponentit ohjelmoitiin ja testattiin erikseen ja tämän jälkeen ne integroitiin yhteen
- Tässä yhteydessä suoritettiin **integroititestausta** jonka tarkoituksena on varmistaa että integroidut komponentit toimivat yhdessä halutulla tavalla
- Testauksen painopiste osien välisten rajapintojen toimivuuden tutkimisessa
- Perinteisesti järjestelmän integraatiota ja integroititestausta voidaan toteuttaa kahdella lähestymistavalla
 - **Bottom up:** lähdetään liikkeelle yksittäisistä komponenteista, liitetään ja testataan niitä yhteen kunnes kaikki on yhdistetty ja järjestelmä on valmiina
 - **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
- *Big bang -integraatio* eli perinteinen tapa integroida valmiit komponentit kerralla on osoittautunut todella riskialttiiksi ja sitä ei enää kukaan täysijärkinen suosittelen käyttäväksi
 - Tutustumme keskiviikkona modernimpaan tapaan suorittaa integraatio

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 ajettava 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 keskiviikkona