Ohjelmistotuotanto

Luento 6

12.4.

Testaus ketterissä menetelmissä

Testauksen automatisointi

Ketterien menetelmien testauskäytänteet

- Testauksen rooli ketterissä menetelmissä poikkeaa huomattavasti vesiputousmallisesta ohjelmistotuotannosta
- Iteraation/sprintin aikana toteutettavat ominaisuudet integroidaan muuhun koodiin ja testataan yksikkö-, integraatio- ja järjestelmätasolla
  - Sykli ominaisuuden määrittelystä siihen että se on valmis ja testattu on erittäin lyhyt, viikosta kuukauteen
- Testausta tehdään sprintin ”ensimmäisestä päivästä” lähtien, testaus ”integroitu” suunnitteluun ja toteutukseen
- Testauksen automatisointi erittäin tärkeässä roolissa, sillä testejä ajetaan usein
  - Regressiotestaus tärkeää
- Ideaalitilanteessa testaajia sijoitettu kehittäjätiimiin, ja myös ohjelmoijat kirjoittavat testejä
  - Testaajan rooli muuttuu virheiden etsijästä virheiden estäjään: testaaja auttaa tiimiä kirjoittamaan automatisoituja testejä, jotka pyrkivät estämään bugien pääsyn koodiin

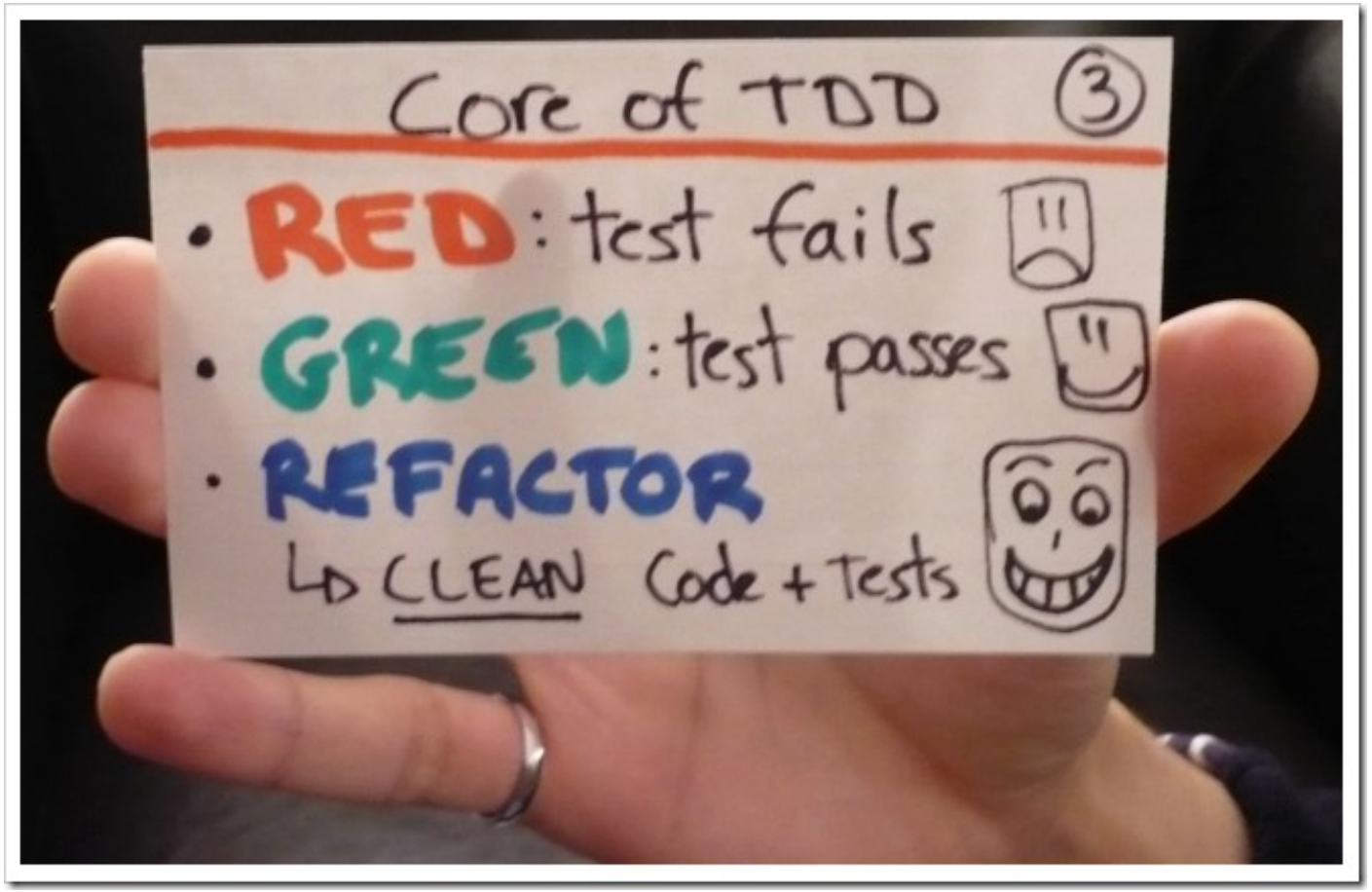
Ketterien menetelmien testauskäytänteitä

- Puhumme tänään neljästä ketterien menetelmien suosimasta testauskäytänteestä
- **Test driven development (TDD)**
  - Nimestään huolimatta kyseessä enemmänkin suunnittelu- ja toteutustason tekniikka
  - ”sivutuotteena” syntyy kattava joukko automaattisesti ajettavia yksikkö- ja integraatiotestejä
- **Acceptance Test Driven Development / Behavior Driven Development**
  - Käyttäjätason vaatimusten tasolla tapahtuva ”TDD”
- **Continuous Integration (CI)** suomeksi jatkuva integraatio
  - Perinteisen integraatio- ja integraatiotestausvaiheen korvaava työskentelytapa
- Kaikista edellisistä käytänteistä seurauksena suuri joukko eritasoisia (eli yksikkö-, integraatio-, järjestelmä-) automatisoituja testejä
- **Exploratory testing**, suomeksi tutkiva testaus
  - Järjestelmätestauksen tekniikka, jossa testaaminen tapahtuu ilman formaalia testisuunnitelmaa, testaaja luo lennossa uusia testejä edellisten testien antaman palautteen perusteella

Test driven development

- TDD on yksi XP:n käytänteistä, Kent Beckin lanseeraama
- Joskus TDD:ksi kutsutaan tapaa, jossa testit kirjoitetaan ennen koodin kirjoittamista
  - Tätä tekniikkaa parempi kuitenkin kutsua nimellä *test first programming*
- ”määritelmän mukainen” TDD etenee seuraavasti

1) Kirjoitetaan sen verran testiä että testi ei mene läpi
   - Ei siis luoda heti kaikkia luokan testejä, edetään tekemällä ainoastaan yksi testi kerrallaan
2) Kirjoitetaan koodia sen verran, että testi saadaan menemään läpi
   - Ei heti yritetäkään kirjoittaa ”lopullista” koodia
3) Jos huomataan koodin rakenteen menneen huonoksi (copypastea koodissa, liian pitkiä metodeja, ...) *refaktoroidaan* koodin rakenne paremmaksi
   - Refaktoroinnilla tarkoitetaan koodin sisäisen rakenteen muuttamista sen rajapinnan ja toiminnallisuuden säilyessä muuttumattomana
4) Jatketaan askeleesta 1



- TDD:llä ohjelmoitaessa toteutettavaa komponenttia ei yleensä ole tapana suunnitella tyhjentävästi etukäteen
- Testit kirjoitetaan ensisijaisesti ajatellen komponentin käyttäjää
  - huomio on komponentin rajapinnassa ja rajapinnan helppokäyttöisyydessä, ei niinkään komponentin sisäisessä toteutuksessa
- Komponentin sisäinen rakenne muotoutuu refaktorointien kautta

TDD

- TDD:ssä perinteisen suunnittelu-toteutus-testaus -syklin voi ajatella kääntyneen täysin päinvastaiseen järjestykseen, tarkka oliosuunnittelu tapahtuu vasta refaktorointivaiheiden kautta
- TDD:tä tehtäessä korostetaan yleensä lopputuloksen yksinkertaisuutta, toteutetaan toiminnallisuutta vain sen verran, mitä testien läpimeno edellyttää
  - Ei siis toteuteta ”varalta” ekstratoiminnallisuutta, sillä "You ain't gonna need it" (YAGNI)
- Koodista on vaikea tehdä testattavaa jos se ei ole modulaarista ja löyhästi kytketyistä selkeärajapintaisista komponenteista koostuvaa
  - Tämän takia TDD:llä tehty koodi on *yleensä* laadukasta ylläpidettävyyden ja laajennettavuuden kannalta
- Muita TDD:n hyviä puolia:
  - Rohkaisee ottamaan pieniä askelia kerrallaan ja näin toimimaan fokusoidusti
  - Tehdyt virheet havaitaan nopeasti suuren testijoukon takia
  - Hyvin kirjoitetut testit toimivat toteutetun komponentin rajapinnan dokumentaationa

TDD

- TDD:llä on myös ikävät puolensa
  - Testikoodia tulee paljon, usein suunilleen saman verran kuin varsinaista koodia
    - Toisaalta TDD:llä tehty tuotantokoodi on usein hieman normaalisti tehtyä koodia lyhempi
  - Jos ja kun koodi muuttuu, tulee testejä ylläpitää
  - TDD:n käyttö on haastavaa (mutta ei mahdotonta) mm. käyttöliittymä-, tietokanta- ja verkkoyhteyksistä huolehtivan koodin yhteydessä
    - testauksen kannalta hankalat komponentit kannattaakin eristää mahdollisimman hyvin muusta koodista, näin on järkevää tehdä, käytettiin TDD:tä tai ei
  - Jo olemassaolevan ”legacy”-koodin laajentaminen TDD:llä voi olla haastavaa
- Lisää TDD:stä
  - http://jamesshore.com/Agile-Book/test_driven_development.html
  - http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

Riippuvuudet yksikkötesteissä

- TDD:tä ja muutenkin yksikkötestejä tehdessä on ratkaistava kysymys, miten testeissä suhtaudutaan testattavien luokkien riippuvuuksiin, eli luokkiin, joiden oliota testattava luokka käyttää
- Dependency Injection -suunnittelumalli parantaa luokkien testattavuutta sillä se mahdollistaa riippuvuuksien asettamisen luokille testistä käsin
  - https://github.com/mluukkai/ohtu2016/blob/master/web/riippuvuuksien_injektointi.md
- Yksi mahdollisuus on tehdä testejä varten riippuvuudet korvaavia tynkäkomponentteja eli stubeja, näin tehtiin mm. viikon 2 tehtävässä 3:
  - https://github.com/mluukkai/ohtu2016/wiki/Laskari-2
- Stubeihin voidaan esim. kovakoodata metodikutsujen tulokset valmiiksi
  - Testi voi myös kysellä stubilta millä arvoilla testattava metodi sitä kutsui
- Stubeja on viimeaikoina ruvettu myös kutsumaan **mock-olioiksi**
- Martin Fowlerin artikkeli selvittää asiaa ja terminologiaa
  - http://martinfowler.com/articles/mocksArentStubs.html
- On olemassa useita kirjastoja mock-olioiden luomisen helpottamiseksi, tutustumme viikon 4 laskareissa Javalle tarkoitettuun *Mockito*-kirjastoon

Riippuvuudet yksikkötesteissä

- Kalvolla 12 esimerkki Rubyn Rspec-kirjastolla tehdystä testistä, jossa testin riippuvuus mockataan
- Testin kohteena on luokka **Action** ja sen metodi **run**
  - Luokan instanssi luodaan rivillä 6, riveillä 3 ja 4 luodun testisyötteen perusteella
  - Testattavan metodin *run* tarkoitus on saada aikaan järjestelmässä erilaisia asioita Action-olion saamista parametreista riippuen
  - Tällä kertaa testin on tarkoitus saada aikaan se, että järjestelmä lähettää sähköpostin (rivin 3 luoman) testisyötteen määrittelemiin osoitteisiin
- Sähköpostin lähtettämisestä huolehtii järjestelmän komponentti **Engine::Email**, ja sen metodi **perform,** joka saa parametriksi lähetettävän emailin tiedot
- **Testauksen kohteena olevan luokan Action metodin run tulee siis toimiakseen saada aikaa metodin perform kutsu oikeilla parametreilla**

Riippuvuudet yksikkötesteissä

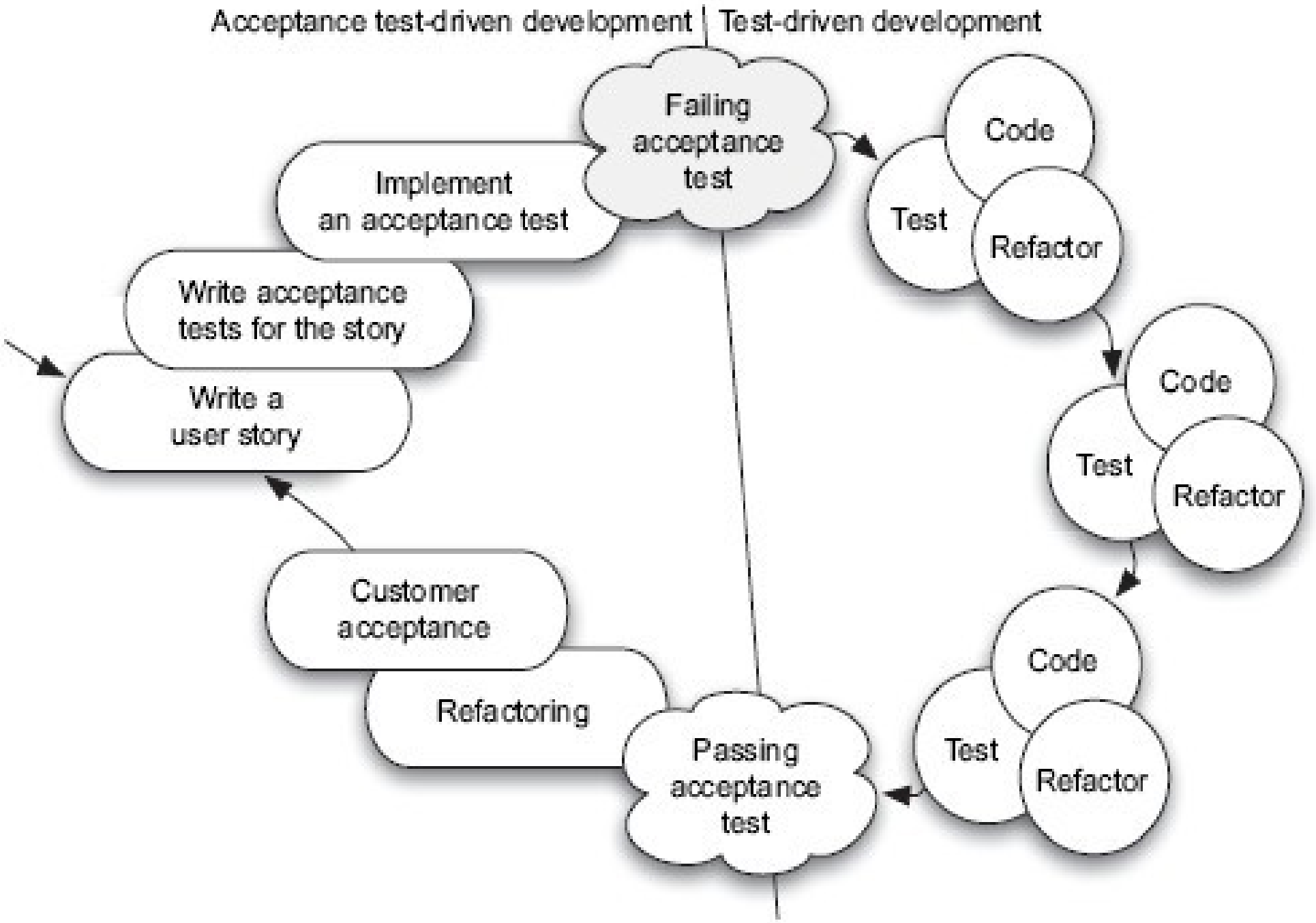
- Riveillä 8-13 asetetaan mock-kirjaston avulla *ekspektaatio eli vaatimus,* jonka mukaan testin suorituksen jälkeen metodia *perform* tulee olla kutsuttu oikeilla parametreilla
  - Määritellään myös, että mockattu metodikutsu palauttaa kutsujalle totuusarvon true
- Testin viimeisellä rivillä kutsutaan testattaavaa metodia *run*
- Testi menee läpi vain jos metodikutsu saa aikaan ekspektaation mukaisen metodikutsun
- Kyseinen esimerkki on eräästä todellisuudessa tuotantokäytössä olevasta järjestelmästä
- Riippuvuuden korvaaminen mockaamalla oli esimerkin tapauksessa erittäin hyödyllistä
  - Testi on nopea sillä se ei riipu emailin lähetyksen kaltaisista hitaista, verkkoyhteyden olemassaolosta ja nopeudesta riippuvasta toimenpiteestä

```
1  it "task creation triggers email sending to recipients" do
2
3      event = build_event("...simulate task creation...")
4      rule = Rule.find_by name:"task_creation_triggers_email_sending"
5
6      action = Action.create(rule, event)
7
8      Engine::Email.should_receive(:perform).with(
9          to: [ "avihavai@cs.helsinki.fi", "josalmi@cs.helsinki.fi" ],
10         from: "mluukkai@iki.fi",
11         subject: "testi",
12         body: "testiviesti"
13     ).and_return( true )
14
15     action.run()
```

User Storyjen testaaminen

- Luennon 2 kalvolla 16 mainittiin, että tärkeä osa User Storyn käsitettä ovat Storyn hyväksymätestit, eli Mike Cohnin sanoin:
  - *Tests that convey and document details and that will be used to determine that the story is complete”*
- User Storyt kuvaavat loppukäyttäjän kannalta arvoa tuottavia toiminnallisuuksia, esim:
  - *Asiakas voi lisätä oluen ostoskoriin*
- Myös hyväksymätestit on tarkoituksenmukaista ilmaista käyttäjän kielellä
  - Usein pidetään hyvänä asiana että asiakas on mukana laatimassa hyväksymätestejä
- Edellisen User storyn hyväksymätestejä voisivat olla
  - Ollessaan tuotelistauksessa ja valitessaan tuotteen jota on varastossa, menee tuote ostoskoriin ja ostoskorin hinta sekä korissa olevien tuotteiden määrä päivittyy oikein
  - Ollessaan tuotelistauksessa ja valitessaan tuotteen jota ei ole varastossa, pysyy ostoskorin tilanne muuttumattomana
- Storyn hyväksymätestit on tarkoituksenmukaista kirjoittaa heti Storyn toteuttavan sprintin alussa

- Jos näin tehdään voidaan sprintissä tapahtuva ohjelmistokehitys ajatella hyväksymätestien tasolla tapahtuvana TDD:nä



User Storyjen testaaminen

- Tälläisestä käytännöstä käytetään nimitystä *Acceptance Test Driven Development, ATDD*
  - ATDD:stä käytetään myös nimiä StoryTest Driven Develepment ja Customer Test Driven Development
  - http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf
  - http://www.methodsandtools.com/archive/archive.php?id=23
  - http://www.methodsandtools.com/archive/archive.php?id=72
  - www.industriallogic.com/papers/storytest.pdf
- Osittain sama idea kulkee nimellä Behavior Driven Development (BDD)
  - http://dannorth.net/introducing-bdd/
- ATDD:ssä sovelluskehityksen lähtökohta on User story eli asiakkaan tasolla mielekäs toiminnallisuus
  - Asiakkaan terminologialla yhdessä asiakkaan kanssa kirjoitetut hyväksymätestit määrittelevät toiminnallisuuden ja näin ollen korvaavat perinteisen vaatimusdokumentin
  - Testien kirjoittamisprosessi lisää asiakkaan ja tiimin välistä kommunikaatiota

Hyväksymätestauksen työkalut

- Yleensä hyväksymätesteistä pyritään tekemään automaattisesti suoritettavia, käytössä olevia työkaluja mm:
  - Fitnesse, FIT, Robot (ATDD)
  - Cucumber, JBehave, ja easyB (BDD)
- ATDD:ssä ja BDD:ssä on kyse lähes samasta asiasta pienin painotuseroin
  - BDD kiinnittää testeissä käytettävän terminologian tarkemmin, BDD ei mm. puhu ollenkaan testeistä vaan spesifikaatioista
  - BDD:llä voidaan tehdä myös muita kuin hyväksymätason testejä
  - kurssilla käytämme pääosin BDD:n nimentäkäytäntöjä
- Tutustumme nyt easyB:hen http://www.easyb.org/
  - Melko läheistä sukua Cucumberille (Ruby) ja JBehavelle (Java) kuitenkin hieman kevyempi ja helpompi käyttää
- Kuten kaikissa ATDD/BDD-työkaluissa, testit kirjoitetaan asiakkaan kielellä
- Ohjelmoija kirjoittaa testeistä mäppäyksen koodiin, näin testeistä tulee automaattisesti suoritettavia

EasyB

- Tarkastellaan esimerkkinä käyttäjätunnuksen luomisen ja sisäänkirjautumisen tarjoamaa palvelua
- Palvelun vaatimuksen määrittelevät User Storyt
  - A new user account can be created if a proper unused username and a proper password are given
  - User can log in with a valid username/password-combination
- EasyB:ssä jokaisesta User Storystä kirjoiteaan oma .story-päätteinen tiedosto, joka sisältää
  - Storyn kuvauksen joko vapaamuotoisena tekstinä (description) tai *as a... I want... so that* (narrative) -muodossa
  - Storyyn liittyvä hyväksymätestejä joita EasyB kutsuu *skenaarioiksi*
- Storyn hyväksymätestit eli *skenaariot* kirjoitetaan *Gherkin*-kielellä, muodossa
  - *Given [initial context], when [event occurs], then [ensure some outcomes]*
- Esimerkki seuraavalla sivulla

```
description 'User can log in with valid username/password-combination'


scenario 'user can login with correct password', {
    given 'command login selected'
    when 'a valid username and password are given'
    then 'user will be logged in to system'
}
scenario 'user can not login with incorrect password', {
    given 'command login selected'
    when 'a valid username and incorrect password are given'
    then 'user will not be logged in to system'
}
scenario 'nonexistent user can not login to', {
    given 'command login selected'
    when 'a nonexistent username and some password are given'
    then 'user will not be logged in to system'
}
```

EasyB: skenaarioiden mäppäys koodiksi

- Ideana on että asiakas tai product owner kirjoittaa tiimissä olevien testaajien tai tiimiläisten kanssa yhteistyössä Storyyn liittyvät testit
  - Samalla Storyn haluttu toiminnallisuus tulee dokumentoitua sillä tarkkuudella, että ohjelmoijat toivon mukaan ymmärtävät mistä on kyse
- Skenaariot muutetaan automaattisesti suoritettaviksi testeiksi kirjoittamalla niistä *mäppäys* ohjelmakoodiin
  - Ohjelmoijat tekevät mäppäyksen siinä vaiheessa, kun tuotantokoodia on tarpeellinen määrä valmiina
- EasyB:ssä mäppäys tehdään Groovy:llä
  - ”Javaa ilman puolipisteitä ja tyyppimäärittelyjä”
- Mäppäykset kirjoitetaan skenaarioiden yhteyteen
  - Tässä EasyB poikkeaa raskaammista esikuvistaan Cucumberista ja JBehavesta joissa mäppäykset kirjoitetaan omaan tiedostoon. Molemmilla tavoilla on etunsa
- Esimerkki seuraavalla sivulla, lisää asiasta laskareissa

```
description 'User can log in with valid username/password-combination'

scenario "user can login with correct password", {
    given 'command login selected', {
        auth = new AuthenticationService()
        io = new StubIO("login", "pekka", "akkep")
        app = new App(io, auth)
    }

    when 'a valid username and password are given', {
        app.run()
    }

    then 'user will be logged in to system', {
        io.getPrints().shouldHave("logged in")
    }
}
```

Websovellusten testien automatisointi

- Olemme jo nähneet, miten dependency injectionin avulla on helppo tehdä komentoriviltä toimivista ohjelmista testattavia
- Myös Java Swing- ja muilla käyttöliittymäkirjastoilla sekä web-selaimella käytettävien sovellusten automatisoitu testaaminen on mahdollista
- Tutustumme laskareissa Web-sovellusten testauksen automatisointiin käytettävään Selenium 2.0 WebDriver -kirjastoon
  - http://seleniumhq.org/docs/03_webdriver.html
- Selenium tarjoaa rajapinnan, jonka avulla on mahdollisuus simuloida ohjelmakoodista tai testikoodista käsin selaimen toimintaa, esim. linkkien klikkauksia ja lomakkeeseen tiedon syöttämistä
  - Selenium Webdriver -rajapinta on käytettävissä lähes kaikilla ohjelmointikielillä
- Seleniumia käyttävät testit voi tehdä normaalin testikoodin tapaan joko JUnit- tai easyB-testeinä
- Katsotaan esimerkkinä käyttäjätunnuksista ja sisäänkirjautumisesta huolehtivan järjestelmän web-versiota
  - Asiaan palataan tarkemmin laskareissa

```
scenario "user can login with correct password", {
    given 'login selected', {
        driver = new HtmlUnitDriver();
        driver.get("http://localhost:8080");
        element = driver.findElement(By.linkText("login"));
        element.click();
    }
    when 'a valid username and password are given', {
        element = driver.findElement(By.name("username"));
        element.sendKeys("pekka");
        element = driver.findElement(By.name("password"));
        element.sendKeys("akkep");
        element = driver.findElement(By.name("login"));
        element.submit();
    }
    then 'user will be logged in to system', {
        driver.getPageSource().contains("Welcome to Ohtu Application!").shouldBe true
    }
}
```

Pois integraatiohelvetistä

- Vesiputousmallissa eli lineaarisesti etenevässä ohjelmistotuotannossa ohjelmiston toteutusvaiheen päättää integrointivaihe
  - yksittäin testatut komponentit integroidaan yhdessä toimivaksi kokonaisuudeksi
  - Suoritetaan integraatiotestaus, joka varmistaa yhteistoiminnallisuuden
- Perinteisesti juuri integrointivaihe on tuonut esiin suuren joukon ongelmia
  - Tarkasta suunnittelusta huolimatta erillisten tiimien toteuttamat komponentit rajapinnoiltaan tai toiminnallisuudeltaan epäsopivia
  - ”integrointihelvetti” http://c2.com/cgi/wiki?IntegrationHell
- Suurten projektien integrointivaihe on kestänyt ennakoimattoman kauan
  - Integrointivaiheen ongelmat ovat aiheuttaneet ohjelmaan suunnittelutason muutoksia
- 90-luvulla alettiin huomaamaan, että riskien minimoimiseksi integraatio kannattaa tehdä useammin kuin vain projektin lopussa
- best practiceksi muodostui päivittäin tehtävä koko projektin kääntäminen *daily/nightly build* ja samassa yhteydessä *ns. smoke test:*in suorittaminen
  - http://www.stevemcconnell.com/ieeesoftware/bp04.htm

Päivittäisestä jatkuvaan integraatioon

- Smoke test:
  - The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems
- Daily buildia ja smoke testiä käytettäessä järjestelmän integraatio tehdään (ainakin jollain tarkkuustasolla) joka päivä
  - Komponenttien yhteensopivuusongelmat huomataan nopeasti ja niiden korjaaminen helpottuu
  - Tiimin moraali paranee, kun ohjelmistosta on olemassa päivittäin kasvava toimiva versio
- Mahdollisimman usein tapahtuva integraatiovaihe todettiin hyväksi käytännöksi. Tästä syntyi idea toistaa integraatiota vielä päivittäistä sykliäkin useamiin: **jatkuva integraatio** eli **continuous integration**
  - http://martinfowler.com/articles/continuousIntegration.html
  - http://jamesshore.com/Agile-Book/continuous_integration.html
- Integraatiovaiheen yllätysten minimoinnin lisäksi jatkuvassa integraatiossa on tarkoitus eliminoida ”but it works on my machine”-ilmiö

Jatkuva integraatio – Continuous Integration

- Integraatiosta tarkoitus tehdä ”nonevent”, ohjelmistosta koko ajan olemassa integroitu ja testattu tuore versio
- Ohjelmisto ja kaikki konfiguraatiot pidetään keskitetyssä repositoriossa
- Koodi sisältää kattavat automatisoidut testit
  - Yksikkö-, integraatio- ja hyväksymätason testejä
- Yksittäinen palvelin, jonka konfiguraatio vastaa tuotantopalvelimen konfiguraatiota, varattu CI-palvelimeksi
- CI-palvelin tarkkailee repositoriota ja jos huomaa siinä muutoksia, hakee koodin, kääntää sen ja ajaa testit
  - Jos koodi ei käänny tai testit eivät mene läpi, seurauksena poikkeustilanne joka korjattava välittömästi: ***do not break the build***
- Sovelluskehittäjän työprosessi etenee seuraavasti
  - Haetaan repositoriosta koodin uusi versio
  - Toteutetaan työn alla oleva toiminnallisuus ja sille automatisoidut testit
  - Integroidaan kirjoitettu koodi suoraan muun koodin yhteyteen
  - Kun työ valmiina, haetaan repositorioon tulleet muutokset ja ajetaan testit

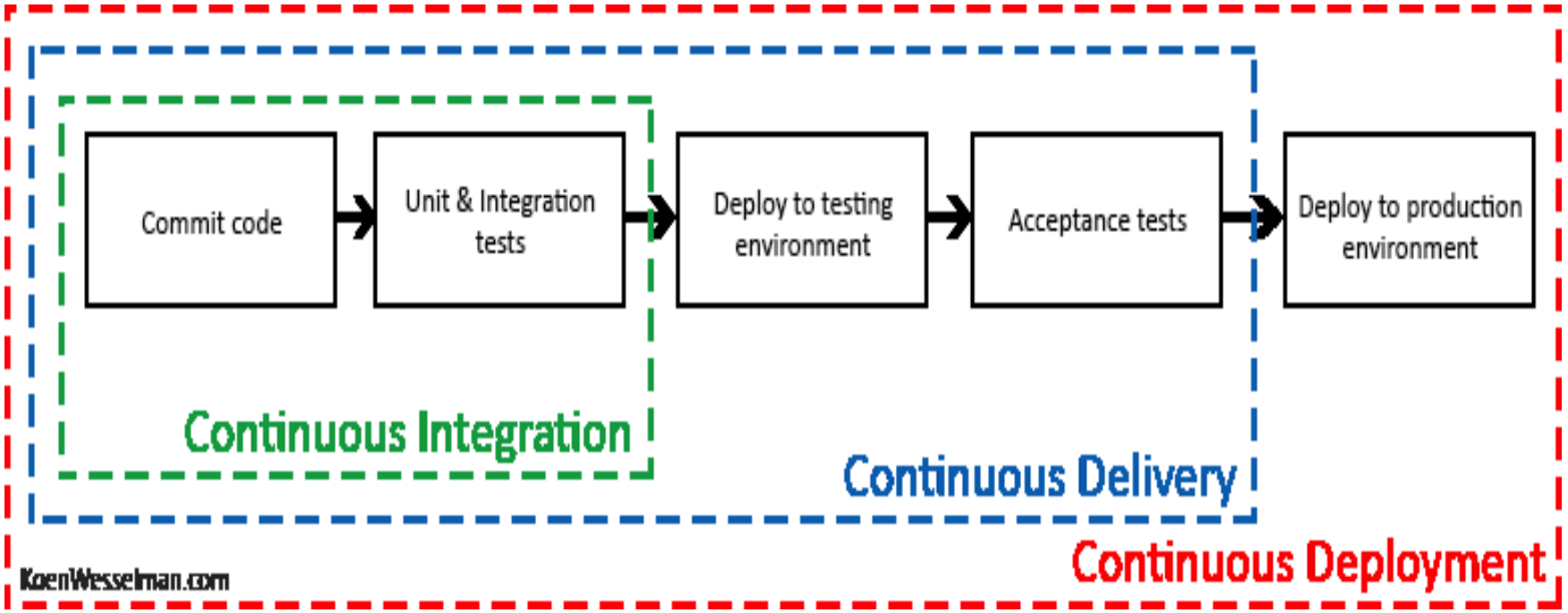
- Kun kehittäjän omalla koneella kaikki testit menevät läpi ja koodi on integroitu muuhun ohjelmakoodiin, pushaa kehittäjä koodin repositorioon
- CI-palvelin huomaa tehdyt muutokset, hakee koodit ja suorittaa testit
- Näin minimoituu mahdollisuus sille, että lisätty koodi toimii esim. konfiguraatioerojen takia ainoastaan kehittäjän paikallisella työasemalla
- Ensimmäisellä viikolla käyttämämme https://travis-ci.org on tämän hetken ehkä eniten huomiota saanut CI-palvelinohjelmisto. Eräs travisin suurista eduista on se, että ohjelmisto toimii pilvessä ja tarvetta omalle CI-palvelimelle ei ole
- Tarkoituksena on, että jokainen kehittäjä integroi tekemänsä työn muuhun koodiin mahdollisimman usein, *vähintään* kerran päivässä
  - CI siis rohkaisee jakamaan työn pieniin osiin, sellaisiin jotka saadaan testeineen ”valmiiksi” yhden työpäivän aikana
  - CI-työprosessin noudattaminen vaatii kurinalaisuutta
- Jotta CI-prosessi toimisi joustavasti, tulee testien ajamisen tapahtua suhteellisen nopeasti, maagisena rajana pidetään usein kymmentä minuuttia
- Jos osa testeistä on hitaita, voidaan testit konfiguroida ajettavaksi kahdessa vaiheessa
  - *commit build*:in läpimeno antaa kehittäjälle oikeuden pushata koodi repositorioon
  - CI-palvelimella suoritetaan myös hitaammat testit sisältävä *secondary build*

Jatkuva toimitusvalmius ja käyttöönotto

- Viime aikoina nousseen trendin mukaan CI:tä viedään vielä askel pidemmälle ja integraatioprosessiin lisätään myös automaattinen ”deployaus”
  - käännetty ja testattu koodi siirretään suoritettavaksi ns. *staging*- eli testipalvelimelle
- **Staging-palvelin**, on ympäristö, joka on konfiguraatioidensa ja myös sovelluksen käsittelemän datan osalta mahdollisimman lähellä varsinaista tuotantoympäristöä
- Kun ohjelmiston uusi versio on deployattu staging-palvelimelle, suoritetaan sille hyväksymätestit
- Hyväksymätestien suorittamisen jälkeen uusi versio voidaan siirtää tuotantopalvelimelle
- Parhaassa tapauksessa myös hyväksymätestien suoritus on automatisoitu, ja ohjelmisto kulkee koko **deployment pipelinen** läpi, eli sovelluskehittäjän koneelta CI-palvelimelle, sieltä stagingiin ja lopulta tuotantoon, automaattisesti

Jatkuva toimitusvalmius ja käyttöönotto

- Käytännöstä, jossa jokainen CI:n läpäisevä ohjelmiston uusi versio viedään staging-palvelimelle ja siellä tapahtuvan hyväksymätestauksen jälkeen tuotantoon, käytetään nimitystä **jatkuva toimitusvalmius** engl. **continuous delivery**
- Jos staging-pavelimella ajettavat testit ja siirto tuotantopalvelimelle tapahtuvat automattisesti, puhutaan **jatkuvasta käyttöönotosta** engl. **continuous deployment**
- Viime aikoina on ruvettu suosimaan tyyliä, jossa web-palveluna toteutettu ohjelmisto julkaistaan tuotantoon jopa useita kertoja päivästä



Tutkiva testaaminen

- Jotta järjestelmä saadaan niin virheettömäksi, että se voidaan laittaa tuotantoon, on testauksen oltava erittäin perusteellinen
- Perinteinen tapa järjestelmätestauksen suorittamiseen on ollut laatia ennen testausta hyvin perinpohjainen testaussuunnitelma
  - Jokaisesta testistä on kirjattu testisyötteet ja odotettu tulos
  - Testauksen tuloksen tarkastaminen on suoritettu vertaamalla järjestelmän toimintaa testitapaukseen kirjattuun odotettuun tulokseen
- Automatisoitujen hyväksymätestien luonne on täsmälleen samanlainen, syöte on tarkkaan kiinnitetty samoin kuin odotettu tuloskin
- Jos testaus tapahtuu pelkästään etukäteen mietittyjen testien avulla, ovat ne kuinka tarkkaan tahansa harkittuja, ei kaikkia yllättäviä tilanteita osata välttämättä ennakoida
- Hyvät testaajat ovat kautta aikojen tehneet ”virallisen” dokumentoidun testauksen lisäksi epävirallista ”ad hoc”-testausta
- Pikkuhiljaa ”ad hoc”-testaus on saanut virallisen aseman ja sen strukturoitua muotoa on ruvettu nimittämään **tutkivaksi testaamiseksi** (exploratory testing)

Tutkiva testaaminen

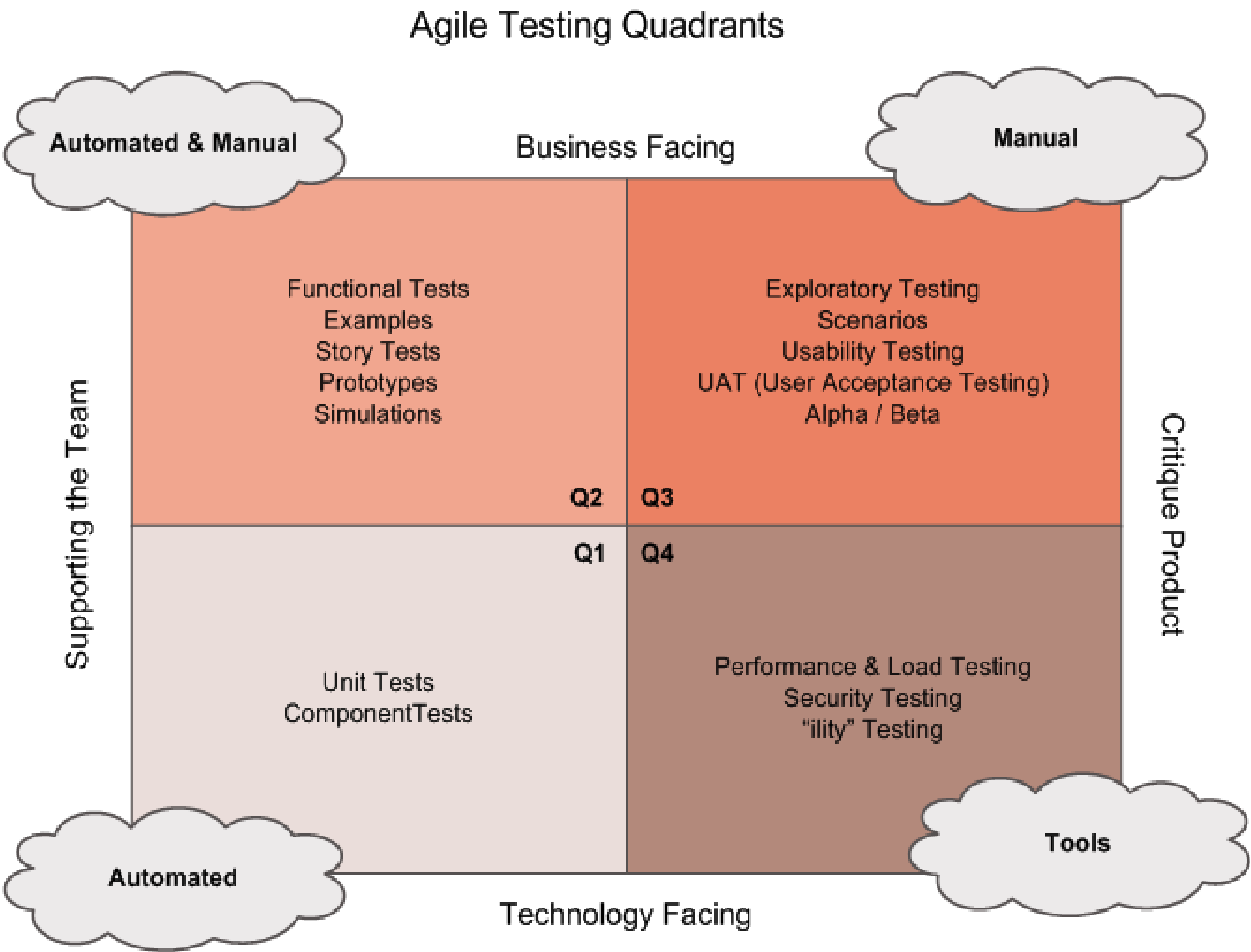
- *Exploratory testing is simultaneous learning, test design and test execution*
  - www.satisfice.com/articles/et-article.pdf
  - http://www.satisfice.com/articles/what_is_et.shtml
- Ideana on, että testaaja ohjaa toimintaansa suorittamiensa testien tuloksen perusteella
- Testitapauksia ei suunnitella kattavasti etukäteen, vaan testaaja pyrkii kokemuksensa ja suoritettujen testien perusteella löytämään järjestelmästä virheitä
- Tutkiva testaus ei kuitenkaan etene täysin sattumanvaraisesti
- Testaussessiolle asetetaan jonkinlainen tavoite
  - Mitä tutkitaan ja minkälaisia virheitä etsitään
- Ketterässä ohjelmistotuotannossa tavoite voi hyvin jäsentyä yhden tai useamman User storyn määrittelemän toiminnallisuuden ympärille
  - Esim. testataan ostosten lisäystä ja poistoa ostoskorista

Tutkiva testaaminen

- Tutkivassa testauksessa keskeistä on kaiken järjestelmän tekemien asioiden havainnointi
  - Normaaleissa etukäteen määritellyissä testeissähän havainnoidaan ainoastaan reagoiko järjestelmä odotetulla tavalla
  - Tutkivassa testaamisessa kiinnitetään huomio myös varsinaisen testattavan asian ulkopuoleisiin asioihin
- Esim. jos huomattaisiin selaimen osoiterivillä URL
  http://www.kumpulabiershop.com/ostoskori?id=10
  voitaisiin yrittää muuttaa käsin ostoskorin id:tä ja yrittää saada järjestelmä epästabiiliin tilaan
- Tutkivan testaamisen avulla löydettyjen virheiden toistuminen jatkossa kannattaa eliminoida lisäämällä ohjelmalle sopivat automaattiset regressiotestit
  - Tutkivaa testaamista ei siis kannata käyttää regressiotestaamisen menetelmänä vaan sen avulla kannattaa ensisijaisesti testata sprintin yhteydessä toteutettuja uusia ominaisuuksia
- Tutkiva testaaminen siis ei ole vaihtoehto normaaleille tarkkaan etukäteen määritellyille testeille vaan niitä täydentävä testauksen muoto

Loppupäätelmiä testauksesta

- Seuraavalla sivulla alunperin Brian Maricin ketterän testauksen kenttää jäsentävä kaavio *Agile Testing Quadrants*
  - http://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/
  - http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2
  - Kaavio on jo hieman vanha, alunperin vuodelta 2003
- Ketterän testauksen menetelmät voidaan siis jakaa neljään luokkaan (Q1...Q4) seuraavien dimensioiden suhteen
  - Busness facing … techonology facing
  - Supporting team … critique to the product
- Testit ovat suurelta osin automatisoitavissa, mutta esim. tutkiva testaaminen ja käyttäjän hyväksymätestaus ovat luonteeltaan manuaalista työtä edellyttäviä
- Kaikilla ”neljänneksillä” on oma roolinsa ja paikkansa ketterissä projekteissa, ja on pitkälti kontekstisidonnaista missä suhteessa testaukseen ja laadunhallintaan käytettävissä olevat resurssit kannattaa kohdentaa


Agile Testing Quadrants
Automated & Manual
Business Facing
Manual
Functional Tests
Examples
Story Tests
Prototypes
Simulations
Exploratory Testing
Scenarios
Usability Testing
UAT (User Acceptance Testing)
Alpha / Beta
Q2
Q3
Supporting the Team
Critique Product
Q1
Q4
Unit Tests
ComponentTests
Performance & Load Testing
Security Testing
"ility" Testing
Automated
Tools
Technology Facing

Loppupäätelmiä testauksesta

- Ketterissä menetelmissä kantavana teemana on arvon tuottaminen asiakkaalle
- Tätä kannattaa käyttää ohjenuorana myös arvioitaessa mitä ja miten paljon projektissa tulisi testata
- Testauksella ei ole itseisarvoista merkitystä, mutta testaamattomuus alkaa pian heikentää tuotteen laatua liikaa
- Joka tapauksessa testausta ja laadunhallintaa on tehtävä paljon ja toistuvasti, tämän takia testauksen automatisointi on yleensä pidemmällä tähtäimellä kannattavaa
- Testauksen automatisointi ei ole halpaa eikä helppoa ja väärin, väärään aikaan tai väärälle ”tasolle” tehdyt automatisoidut testit voivat tuottaa enemmän harmia ja kustannuksia kuin hyötyä
- Jos ohjelmistossa on komponentteja, jotka tullaan ehkä poistamaan tai korvaamaan pian, saattaa olla järkevää olla automatisoimatta niiden testejä
  - Vrt luennolla 3 esitelty Minimal Viable Product
  - Ongelmallista kuitenkin usein on, että tätä ei tiedetä yleensä ennalta ja pian poistettavaksi tarkoitettu komponentti voi jäädä järjestelmään pitkäksikin aikaa

- Kattavien yksikkötestien tekeminen ei välttämättä ole mielekästä ohjelman kaikille luokille, parempi vaihtoehto voi olla tehdä integraatiotason testejä ohjelman isompien komponenttien rajapintoja vasten
    - Testit pysyvät todennäköisemmin valideina komponenttien sisäisen rakenteen muuttuessa
- Yksikkötestaus lienee hyödyllisimmillään kompleksia logiikkaa sisältäviä luokkia testattaessa
- Oppikirjamääritelmän mukaista TDD:tä sovelletaan melko harvoin
- Välillä kuitenkin TDD on hyödyllinen väline, esim. testattaessa rajapintoja, joita käyttäviä komponentteja ei ole vielä olemassa. Testit tekee samalla vaivalla kuin koodia käyttävän ”pääohjelman”
- Testitapauksista kannattaa aina tehdä mahdollisimman paljon testattavan komponentin oikeita käyttöskenaarioita vastaavia, pelkkiä testauskattavuutta kasvattavia testejä on turha tehdä
- Automaattisia testejä kannattaa kirjoittaa mahdollisimman paljon etenkin niiden järjestelmän komponenttien rajapintoihin, joita muokataan usein
- Liian aikaisessa vaiheessa projektia tehtävät käyttöliittymän läpi suoritettavat testit saattavat aiheuttaa kohtuuttoman paljon ylläpitovaivaa, sillä testit hajoavat helposti pienistäkin käyttöliittymiin tehtävistä muutoksista