Ohjelmistotuotanto

Luento 6

10.11.

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

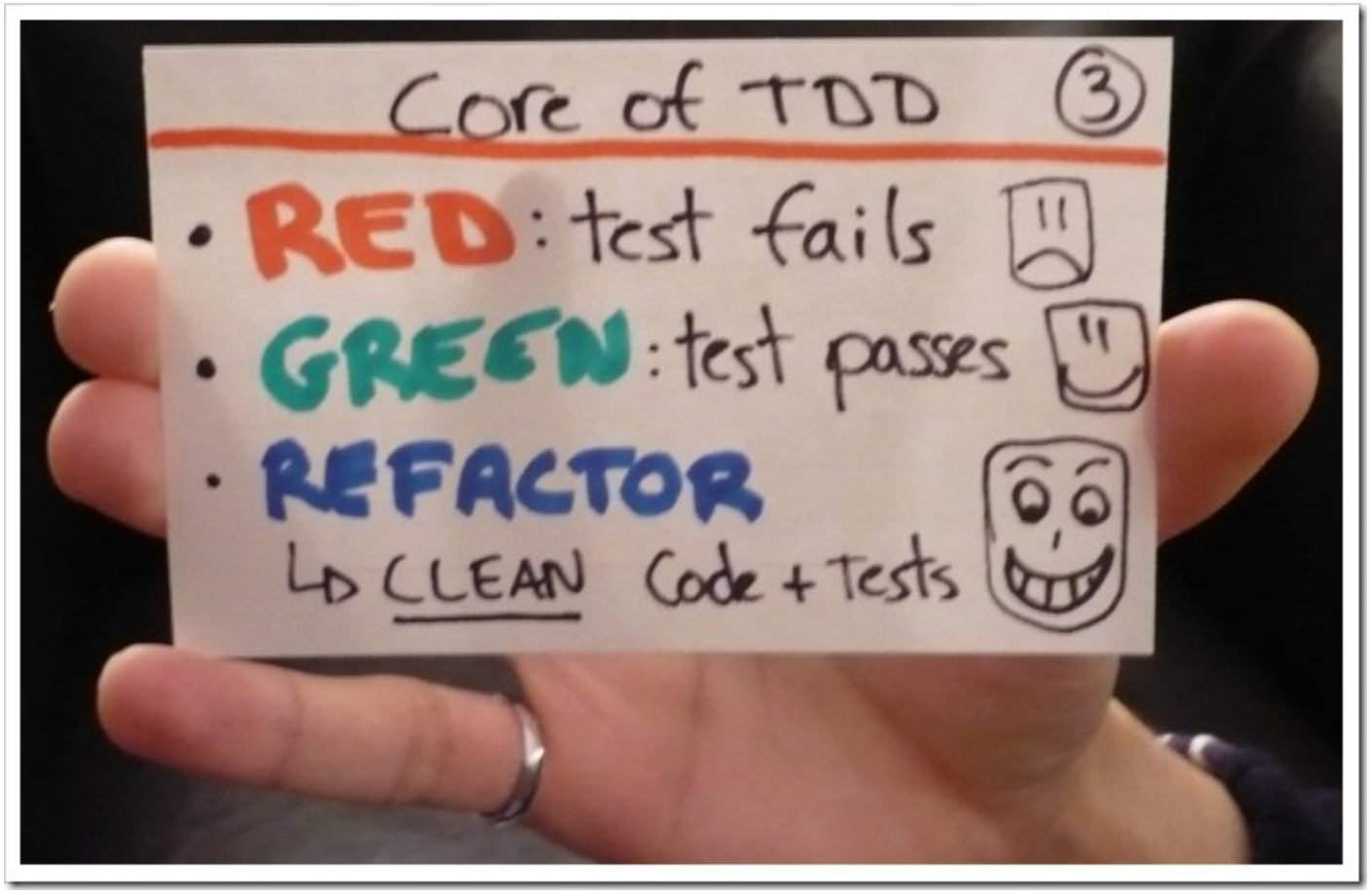
Ketterien menetelmien testauskäytänteitä

- **Test driven development (TDD)**
  - Nimestään huolimatta kyseessä suunnittelu- ja toteutustason tekniikka
  - ”sivutuotteena” syntyy kattava joukko automaattisesti suoritettavia testejä
- **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
- **Tuotannossa tapahtuva testaus**
  - Nouseva trendi on suorittaa uusien ominaisuuksien laadunhallintaa siinä vaiheessa kun osa oikeista käyttäjistä on jo ottanut ne käyttöönsä

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 suunnilleen 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/ohjelmistotuotanto2017/blob/master/web/riippuvuuksien_injektointi.md
- Yksi mahdollisuus on tehdä testejä varten riippuvuudet korvaavia tynkäkomponentteja eli stubeja, näin tehtiin mm. viikon 1 tehtävässä 16:
  - https://github.com/mluukkai/ohjelmistotuotanto2017/blob/master/laskarit/1.md#16--nhlstatistics-ohjelman-yksikkötestaus
- 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 selventää asiaa ja terminologiaa
  - http://martinfowler.com/articles/mocksArentStubs.html
- On olemassa useita kirjastoja mock-olioiden luomisen helpottamiseksi, tutustumme laskareissa Javalle tarkoitettuun *Mockito*-kirjastoon

Riippuvuudet yksikkötesteissä: mockito

- Esimerkki viikon 2 laskareista (Verkkokauppa)
- Ostotapahtuman yhteydessä kaupan tulisi veloittaa asiakkaan tililtä ostosten hinta *kutsumalla luokan pankki metodia maksa*:

```
Pankki myNetBank = new Pankki();
Viitegeneraattori viitteet = new Viitegeneraattori();
Kauppa kauppa = new Kauppa(myNetBank, viitteet);
```

-

```
kauppa.aloitaOstokset();
kauppa.lisaaOstos(5);
kauppa.lisaaOstos(7);
kauppa.maksa("1111");
```

- Miten varmistamme, että maksun suorittavaa metodia on kutsuttu?
- Käytetään *mockito*-kirjastoa

Riippuvuudet yksikkötesteissä: mockito

- Luodaan testissä kaupan riippuvuuksista mock-oliot:

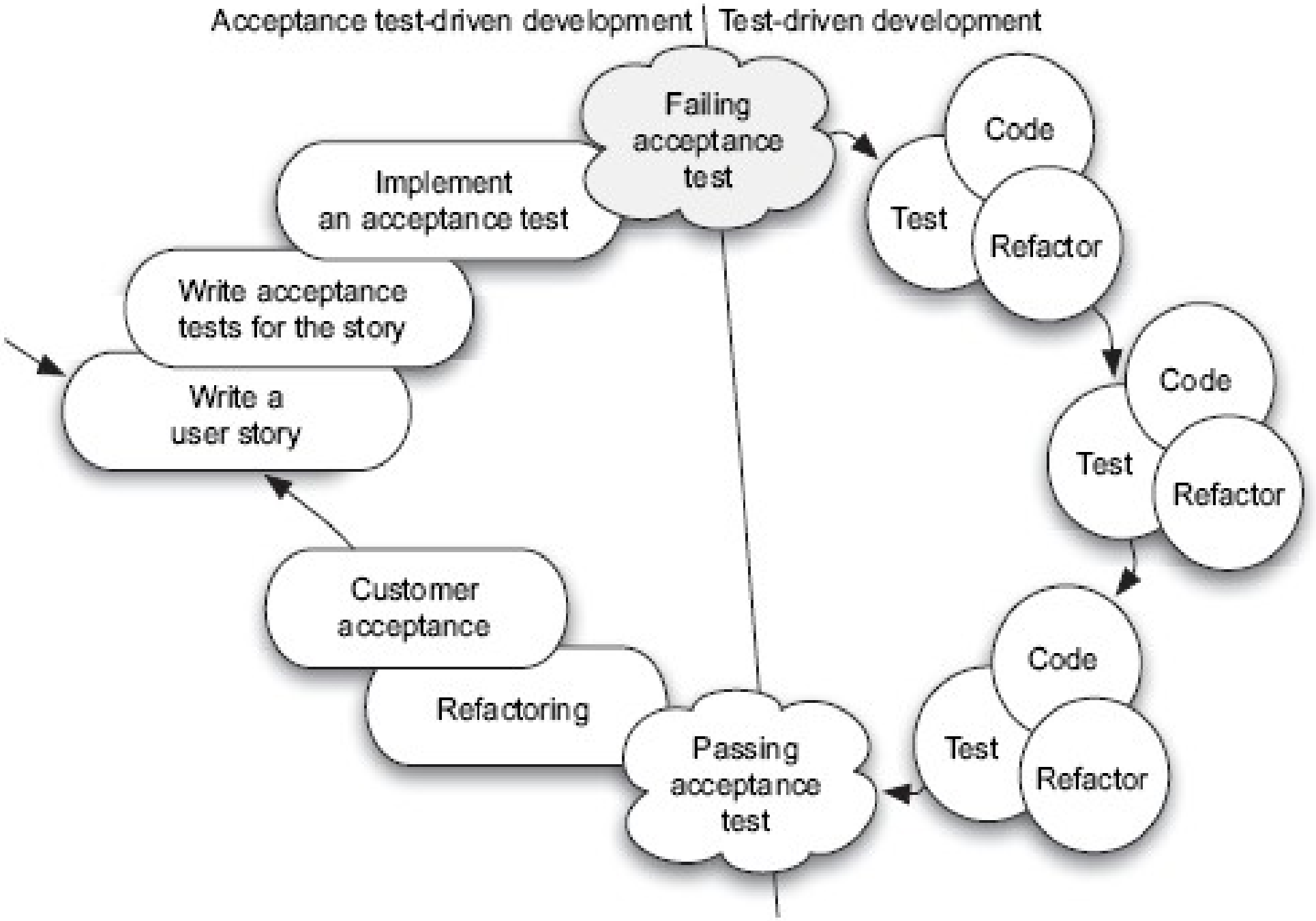
```
@Test
public void kutsutaanPankkiaOikeallaTilinumerollaJaSummalla() {
    Pankki mockPankki = mock(Pankki.class);
    Viitegeneraattori mockViite = mock(Viitegeneraattori.class);
    kauppa = new Kauppa(mockPankki, mockViite);


    kauppa.aloitaOstokset();
    kauppa.lisaaOstos(5);
    kauppa.lisaaOstos(5);
    kauppa.maksa("1111");


    verify(mockPankki).maksa(eq("1111"), eq(10), anyInt());
}
```

- Pankkia edustavalle mock-oliolle on asetettu *ekspektaatio*, eli vaatimus joka varmistaa, että metodia *maksa* on kutsuttu testin aikana sopivilla parametreilla

User Storyjen testaaminen

- Luennon 2 kalvolla 16 mainittiin, että tärkeä osa User Storyn käsitettä ovat Storyn hyväksymätestit (tai hyväksymäkriteerit), 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

- Ideaalitilanteessa hyväksymätesteistä tehdään automaattisesti suoritettavia
- Automaattisen hyväksymätestauksen työkaluja mm:
  - Fitnesse, FIT, Robot (ATDD)
  - Cucumber ja JBehave (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 johtavaan BDD-työkaluun Cucumberiin
  - https://cucumber.io
- Kuten kaikissa ATDD/BDD-työkaluissa, Cucumberissa testit kirjoitetaan asiakkaan kielellä
- Ohjelmoija kirjoittaa testeistä mäppäyksen koodiin, näin testeistä tulee automaattisesti suoritettavia

Cucumber

- 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
- Cucumberissa jokaisesta User Storystä kirjoitetaan oma *.feature*-päätteinen tiedosto, joka sisältää
  - nimen ja
  - joukon storyyn liittyvä hyväksymätestejä joita Cucumber kutsuu *skenaarioiksi*
- Storyn hyväksymätestit eli *skenaariot* kirjoitetaan *Gherkin*-kielellä, muodossa
  - *Given [initial context], when [event occurs], then [ensure some outcomes]*
- Esimerkki seuraavalla sivulla

Feature: User can log in with valid username/password-combination

Scenario: user can login with correct password

Given command login is selected

When username "pekka" and password "akkep" are entered

Then system will respond with "logged in"

Scenario: user can not login with incorrect password

Given command login is selected

When username "pekka" and password "wrong" are entered

Then system will respond with "incorrect username or password"

Scenario: nonexistent user can not login to

Given command login is selected

When username "nonexisting" and password "wrong" are entered

Then system will respond with "incorrect username or password"

Cucumber: 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
- Esimerkki seuraavalla sivulla
- Käytännössä jokaista testin *given, when* ja *then*-askelta vastaa oma metodinsa
  - Metodit kutsuvat ohjelman luokkia simuloiden käyttäjän syötettä
  - varmistaen että ohjelma reagoi käyttäjän toimiin halutulla tavalla
- Palaamme cucumberiin laskareissa

```java
public class Stepdefs {
    App app;
    StubIO io;
    AuthenticationService auth = new AuthenticationService(new InMemoryUserDao());
    List<String> inputLines = new ArrayList<>();

    @Given("^command login is selected$")
    public void command_login_selected() throws Throwable {
        inputLines.add("login");
    }

    @When("^username \"([^\"]*)\" and password \"([^\"]*)\" are entered$")
    public void a_username_and_password_are_entered(String username, String password) {
        inputLines.add(username);
        inputLines.add(password);

        io = new StubIO(inputLines);
        app = new App(io, auth);
        app.run();
    }

    @Then("^system will respond with \"([^\"]*)\"$")
    public void system_will_respond_with(String expectedOutput) {
        assertTrue(io.getPrints().contains(expectedOutput));
    }
}
```

Websovellusten testien automatisointi

- Olemme jo nähneet, miten dependency injectionin avulla on helppo tehdä komentoriviltä toimivista ohjelmista testattavia
- Myös Java Swing, JavaFX 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 tiedon syöttämistä lomakkeeseen
  - Selenium Webdriver -rajapinta on käytettävissä lähes kaikilla ohjelmointikielillä
- Seleniumia käyttävät testit voi tehdä normaalin testikoodin tapaan joko JUnit- tai Cucumber-testeinä
- Katsotaan esimerkkinä käyttäjätunnuksista ja sisäänkirjautumisesta huolehtivan järjestelmän web-versiota
  - Asiaan tutustutaan tarkemmin viikon 3 laskareissa

```java
public class Stepdefs {
    WebDriver driver = new ChromeDriver();
    String baseUrl = "http://localhost:4567";

    @Given("^login is selected$")
    public void login_selected() throws Throwable {
        driver.get(baseUrl);
        WebElement element = driver.findElement(By.linkText("login"));
        element.click();
    }

    @When("^username \"([^\"]*)\" and password \"([^\"]*)\" are given$")
    public void username_and_password_are_given(String username, String password) {
        WebElement element = driver.findElement(By.name("username"));
        element.sendKeys(username);
        element = driver.findElement(By.name("password"));
        element.sendKeys(password);
        element = driver.findElement(By.name("login"));
        element.submit();
    }

    @Then("^system will respond \"([^\"]*)\"$")
    public void system_will_respond(String pageContent) {
        assertTrue(driver.getPageSource().contains(pageContent));
    }
}
```

Ohjelmiston integraatio

- 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
- Suurten projektien integrointivaihe on kestänyt ennakoimattoman kauan
  - integrointivaiheen ongelmat ovat aiheuttaneet ohjelmaan suunnittelutason muutoksia
- Integraatio on ollut perinteisesti niin hankala vaihe, että sitä kuvaamaan on lanseerattu termi *integratiohelvetti*
  - http://wiki.c2.com/?IntegrationHell

Pois integraatiohelvetistä

- 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
- 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

Päivittäisestä jatkuvaan integraatioon

- Mahdollisimman usein tapahtuva integraatiovaihe todettiin hyväksi käytännöksi. Tästä syntyi idea toistaa integraatiota vielä päivittäistä sykliäkin useammin: **jatkuva integraatio** eli **continuous integration**
- Integraatiovaiheen yllätysten minimoinnin lisäksi jatkuvassa integraatiossa on tarkoitus eliminoida ”but it works on my machine”-ilmiö
- Integraatiosta tarkoitus tehdä todella vaivaton operaatio, 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 mahdollisimman läheisesti 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***

Jatkuva integraatio – Continuous Integration

- 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
- 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

Jatkuva integraatio – Continuous Integration

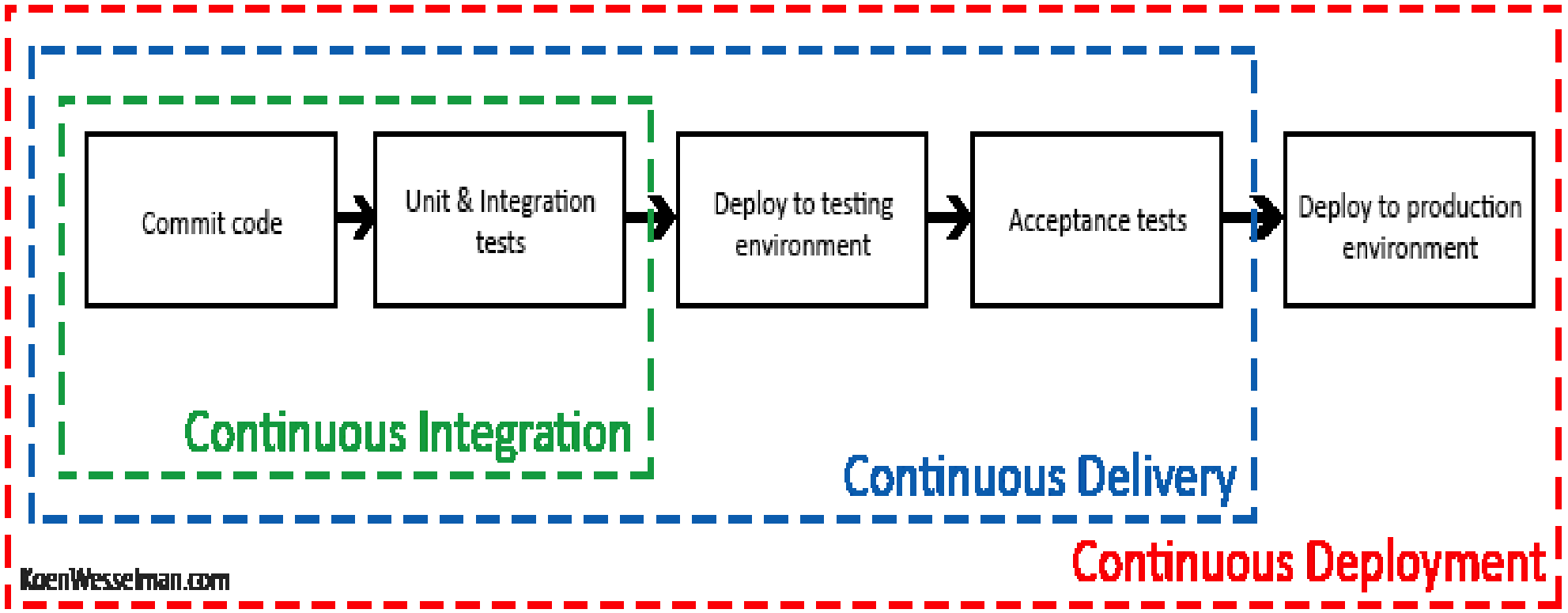
- Jotta CI-prosessi toimisi joustavasti, tulee testien ajamisen tapahtua suhteellisen nopeasti, maagisena rajana pidetään usein kymmentä minuuttia
- Jos osa testeistä on hitaita, voidaan testit konfiguroida ajettavaksi kahdessa (tai tarvittaessa useammassakin) vaiheessa
  - *commit build*:in läpimeno antaa kehittäjälle oikeuden pushata koodi repositorioon
  - CI-palvelimella suoritetaan myös hitaammat testit sisältävä *secondary build*
- Ensimmäisellä viikolla käyttämämme *Travis* on tämän hetken ehkä eniten huomiota saanut CI-palvelinohjelmisto
  - Eräs travisin suurista eduista on se, että ohjelmisto toimii pilvessä ja tarvetta oman CI-palvelimen asentamiselle ei ole
- Travisia paljon vanhempi *Jenkins* lienee edelleen maailmalla eniten käytetty CI-palvelinohjelmisto
  - Tällä hetkellä ei taida olla olemassa yhtään ilmaista verkossa olevaa Jenkins-palvelua. Jenkinsin käyttö siis edellyttää sen asentamista omalle palvelimelle

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 viety eli deployattu staging-palvelimelle, suoritetaan sille hyväksymätestit
- Hyväksymätestien suorittamisen jälkeen uusi versio voidaan siirtää tuotantopalvelimelle
- Parhaassa tapauksessa myös staging-ympäristössä tehtävien hyväksymätestien suoritus on automatisoitu, ja ohjelmisto kulkee koko **deployment pipelinen** läpi, eli sovelluskehittäjän koneelta CI-palvelimelle, sieltä stagingiin ja lopulta tuotantoon, automaattisesti
  - Termillä deployment pipeline siis tarkoitetaan niitä ohjelman käännöksen ja testauksen vaiheita, joiden suorittamista edellytetään, että ohjelma saadaan siirrettyä tuotantoympäristöön asiakkaan käyttöön

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-palvelimella 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

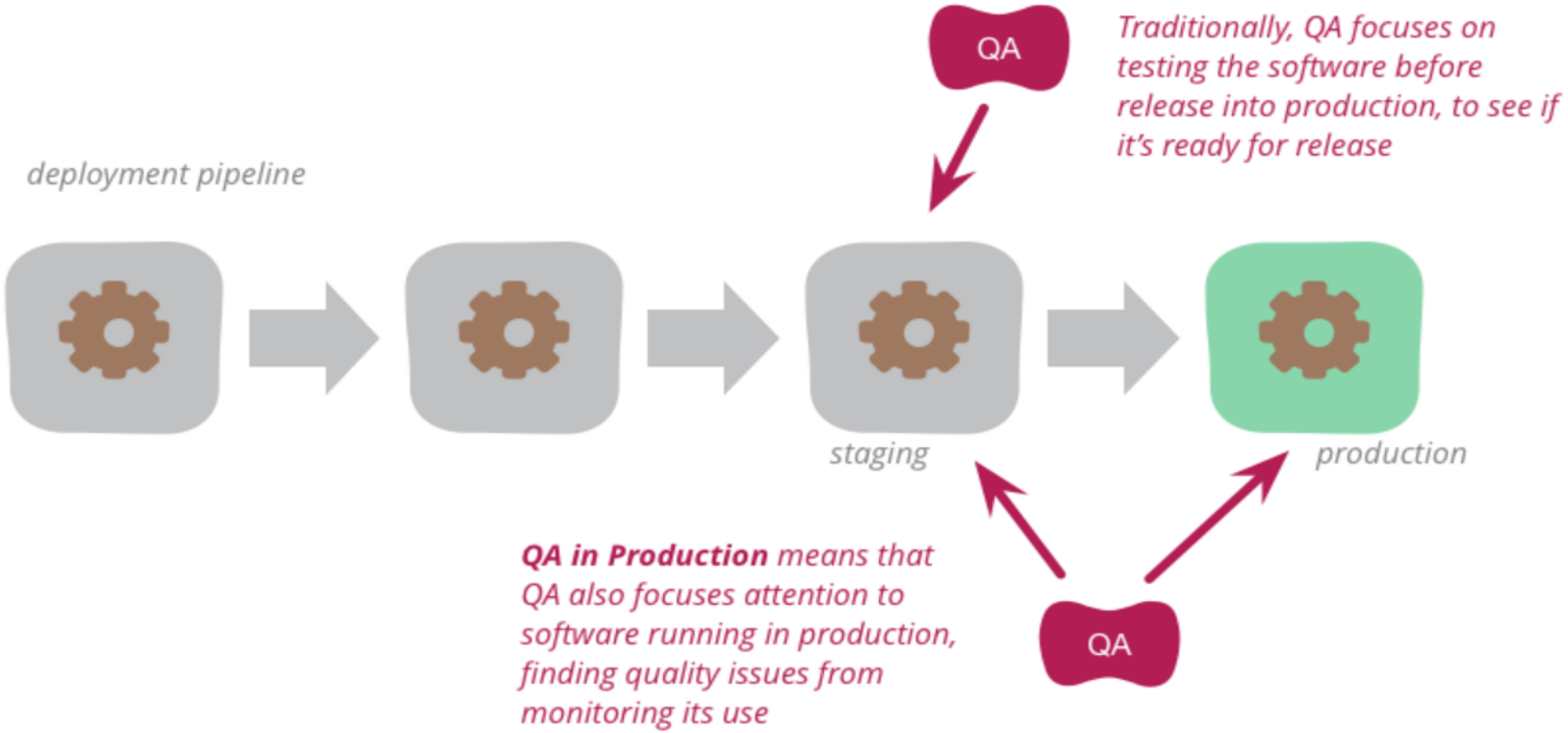
- *Exploratory testing is simultaneous learning, test design and test execution*
  - www.satisfice.com/articles/et-article.pdf
  - http://www.satisfice.com/articles/what_is_et.shtml
- Ideana on, että testaaja ohjaa toimintaansa suorittamiensa testien tuloksen perusteella
- Testitapauksia ei suunnitella 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

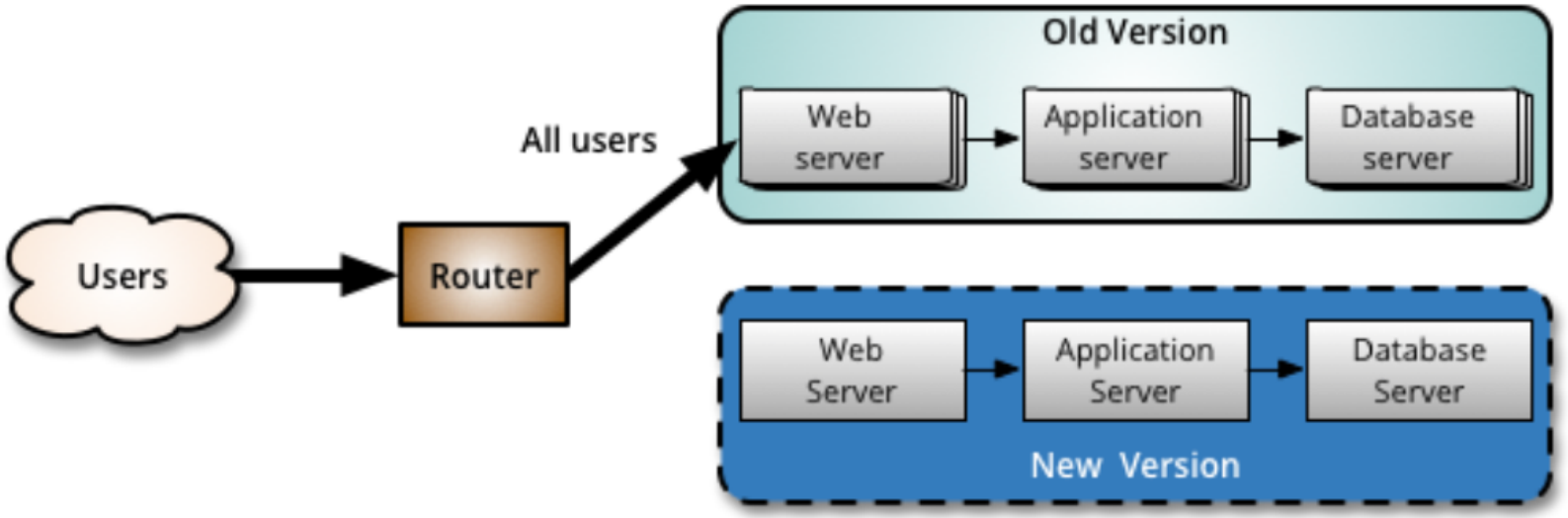
Tuotannossa tapahtuva testaaminen ja laadunhallinta

- Perinteisesti on ajateltu, että kaiken ohjelmiston laadunhallintaan liittyvän testauksen tulee tapahtua ennen kuin ohjelmisto tai sen uudet toiminnallisuudet on otettu käyttöön eli viety tuotantoympäristöön
- Viime aikoina erityisesti web-sovellusten kehityksessä on noussut esiin suuntaus, missä osa laadunhallinnasta tapahtuu *monitoroimalla* tuotannossa olevaa ohjelmistoa



blue-green-deployment

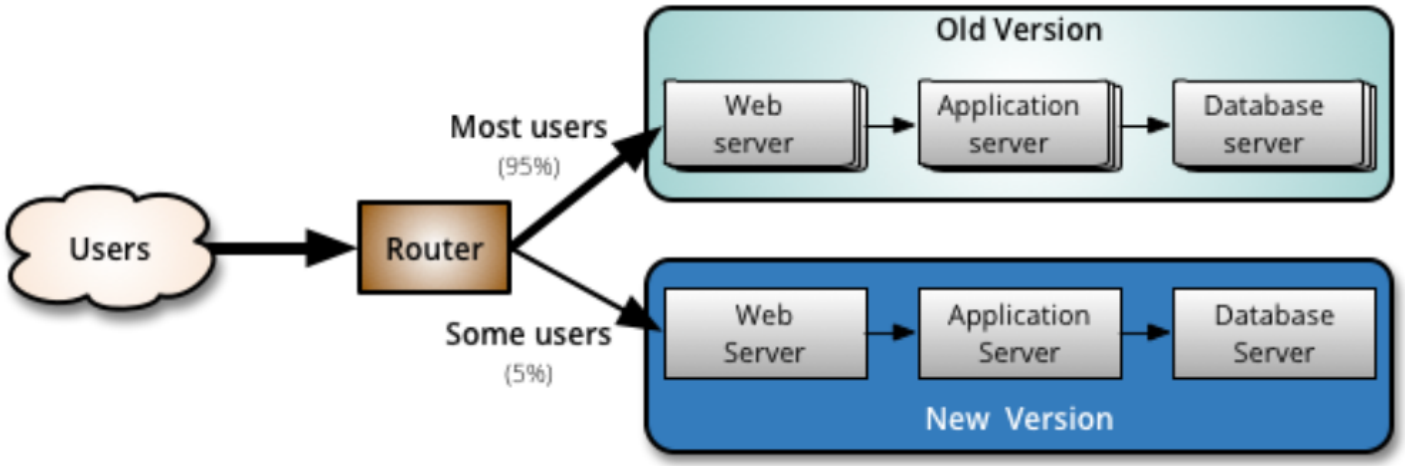
- Eräs tuotannossa tapahtuvan testaamisen tekniikka on *blue-green-deployment*, missä periaatteena on ylläpitää rinnakkain kahta tuotantoympäristöä (tai palvelinta), joista käytetään nimiä *blue* ja *green*
- Tuotantoympäristöistä vain toinen on ohjelmiston käyttäjien aktiivisessa käytössä
- Käyttäjien ja tuotantopalvelinten välissä oleva komponentti esim, ns. reverse proxynä toimiva web-palvelin (kuvassa router) ohjaa käyttäjien liikenteen aktiivisena olevaan ympäristöön



- Kun järjestelmään toteutetaan uusi ominaisuus, deployataan se ensin passiivisena olevaan ympäristöön
- Passiiviselle, uuden ominaisuuden sisältämälle ympäristölle voidaan sitten tehdä erilaisia testejä, esim. osa käyttäjien liikenteestä voidaan ohjata aktiivisen lisäksi passiiviseen ympäristöön ja varmistaa, että se toimii odotetulla tavalla

Blue-green-deployment ja canary release

- Kun uuden ominaisuuden sisältävän passiivinen ympäristön todetaan toimivan ongelmattomasti myös tuotantoympäristössä, voidaan palvelinten rooli vaihtaa, uuden ominaisuuden sisältämästä palvelimesta tulee uusi aktiivinen tuotantoympäristö
  - Aktiivisen tuotantoympäristön vaihto tapahtuu määrittelemällä web-palvelin ohjaamaan liikenne uudelle palvelimelle
- Jos uuden ominaisuuden sisältämässä ympäristössä havaitaan aktivoinnin jälkeen jotain ongelmia, on mahdollista suorittaa erittäin nopeasti *rollback*-operaatio, ja vaihtaa vanha versio jälleen aktiiviseksi
- Blue-green-deploymentin hieman pidemmälle viedyssä versiossa ***canary-releasessa*** uuden ominaisuuden sisältävään ympäristöön ohjataan osa, esim. 5% järjestelmän käyttäjistä

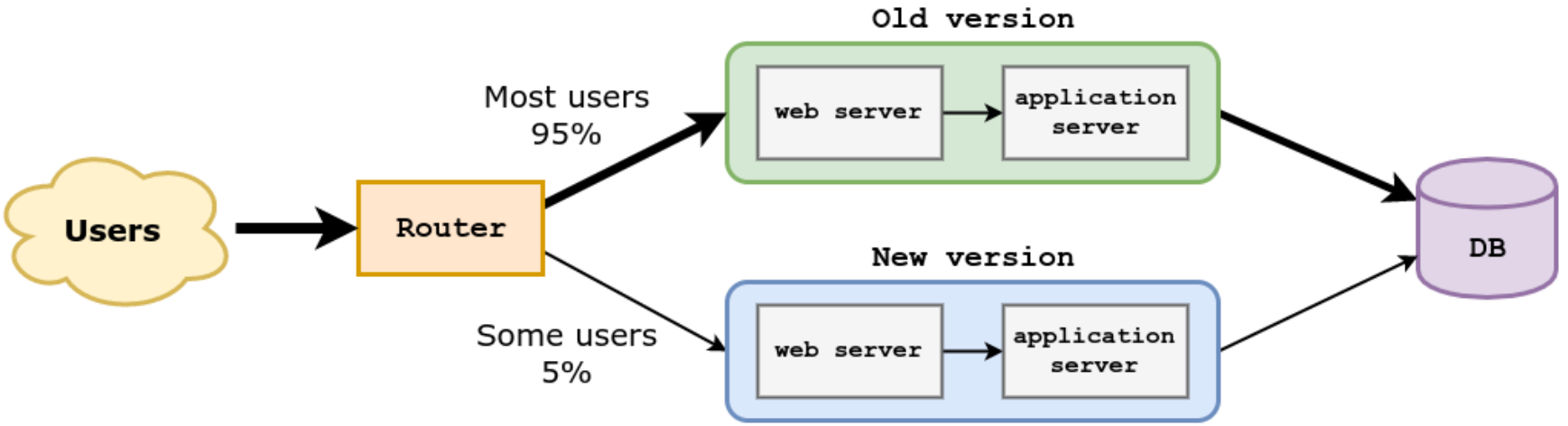


canary release

- Uuden ominaisuuden sisältämää versiota monitoroidaan aktiivisesti ja jos ongelmia ei ilmene, vähitellen kaikki liikenne ohjataan uuteen versioon
- Kuten blue-green-deploymentin tapauksessa, ongelmatilanteissa palautetaan käyttäjät aiempaan, toimivaksi todettuun versioon
- Uuden version toimivaksi varmistaminen siis perustuu järjestelmän monitorioitiin Jos kyseessä olisi esim. sosiaalisen median palvelu, monitoroinnissa voitaisiin tarkastella esim:
  - Palvelun muistin ja prosessoriajan kulutusta sekä verkkoliikenteen määrää
  - Sovelluksen eri sivujen vasteaikoja eli latautumiseen menevää aikaa
  - Kirjautuneiden käyttäjien määrää
  - Luettujen ja lähetettyjen viestien määriä per käyttäjä
  - Kirjautuneen käyttäjän sovelluksessa viettämää aikaa
- Monitoroinnissa tulee siis palvelimen yleisen toimivuuden lisäksi seurata käyttäjätason (engl. bussiness level) metriikoita
  - Jos niissä huomataan eroja aiempaan (esim. kirjautuneet käyttäjät eivät lähetä viestejä samaa määrää kuin keskimäärin normaalisti), voidaan olettaa, että sovelluksen uudessa versiossa saattaa olla joku ongelma ja voi olla tarpeen tehdä rollback vanhaan järjestelmäversioon ja analysoida vikaa tarkemmin
- Nimi *canary release* periytyy kaivostyöläisten tavasta käyttää kanarialintuja tutkimaan sitä onko kaivoksessa myrkyllisiä kaasuja, jos kaivokseen viety lintu ei kuole, ilma on turvallista

Tuotannossa testaaminen ja tietokanta

- Edellisillä kalvoilla oli merkitty järjestelmän vanhalle ja uudelle versiolle erillinen tietokantapalvelin (database server)
- Tilanne ei välttämättä ole tämä ja erityisesti canary releasejen yhteydessä järjestelmän molemmat versiot käyttävät yleensä samaa tietokanta:



- Tämä taas asettaa haasteita, jos järjestelmään toteutetut uudet ominaisuudet edellyttävät muutoksia tietokannan skeemaan
  - Canary releasejen yhteydessä tarvitaan periaatteessa yhtä aikaa sekä tietokannan uutta että vanhaa versiota
- Jos järjestelmän uusi ja vanha versio joutuvat jostain syystä käyttämään eri tietokantaa, täytyy kantojen tilanne synkronoida (eli yhteen kantaa sovelluksen tekemät päivitykset on tavalla toi toisella tehtävä myös toiseen, kenties skeemaltaan jo muuttuneeseen kantaan), jotta järjestelmien vaihtaminen onnistuu saumattomasti

feature toggle

- Jos hyödynnetään *feature toggleja* voidaan Canary releaseja toteuttaa myös käyttämällä pelkästään yhtä tuotantopalvelinta
  - Sama asia kulkee myös nimillä feature flag, conditional feature, config flag. Nimi feature toggle alkaa kuitenkin vakiintua
- Feature togglejen periaate on erittäin yksinkertainen. Koodiin laitetaan ehtolauseita, joiden avulla osa liikenteestä ohjataan vanhan toteutuksen sijaan uuteen tutkimuksen alla olevaan toteutukseen
  - Esim. sosiaalisen median palvelussa voitaisiin käyttäjälle näytettävien uutisten listaan asettaa feature toggle jonka avulla tietyin perustein valituille käyttäjille näytettäisiinkin uuden algoritmin perusteella generoitu lista uutisia

```
List<News> recommendedNews(User user) {
  if ( isInCanaryRelease(user) ) {
    return experimentalRecommendationAlgorith(user)
  } else {
    return recommendationAlgorith(user)
  }
}
```

feature toggle

- Canary releaset eivät ole feature togglejen ainoa sovellus, niitä käytetään yleisesti myös eliminoimaan tarve pitkäikäisille *feature brancheille*
  - Eli sen sijaan, että uusia ominaisuuksia toteutetaan erilliseen haaraan versionhallinnassa joka ominaisuuksien valmistumisen yhteydessä mergetään pääkehityshaaraan, uudet ominaisuudet tehdään suoraan pääkehityshaaraan, mutta ne piilotetaan käyttäjiltä feature toggleilla
  - Käytännössä feature toggle siis palauttaa aina vanhan version normaaleille käyttäjille, sovelluskehittäjien ja testaajien taas on mahdollista valita kumman version feature toggle palauttaa
  - Kun ominaisuus on valmis testattavaksi laajemmalla joukolla, voi feature togglen avulla sitten esim. julkaista ominaisuuden ensin kehittäjäyrityksen omaan käyttöön ja lopulta osalle käyttäjistä canary releasena
  - Lopulta feaature toggle ja vanha toteutus voidaan poistaa
- Suuret internetpalvelut kuten facebook, netflix, google ja flickr soveltavat laajalti canary releaseihin ja feature flageihin perustuvaa kehitysmallia
- Aiheesta löytyy internetistä suuret määrät kiinnostavaa materiaalia, hyvän yleiskuvan antaa https://martinfowler.com/articles/feature-toggles.html

Dev vs ops

- Jatkuvan toimitusvalmiuden (Continuous delivery), käyttöönoton (Continuous deployment) ja tuotannossa testaamisen soveltaminen ei useimmiten ole ollenkaan suoraviivaista
- Perinteisesti yrityksissä on ollut tarkka erottelu sovelluskehittäjien (developers, dev) ja tuotantopalvelimista vastaavan järjestelmäylläpitäjien (operations, ops) välillä
- On erittäin tavallista, että sovelluskehittäjät eivät pääse edes kirjautumaan tuotantopalvelimille ja sovellusten tuotantoonvienti ja esim. tuotantotietokantaan tapahtuvat skeeman päivitykset tapahtuvat ylläpitäjien toimesta
- Tälläisessä ympäristössä esim. continuous deploymentin harjoittaminen on erittäin haastavaa, tilanne ajautuukin helposti siihen, että tuotantopalvelimelle pystytään viemään uusia versioita vain harvoin, esim 4 kertaa vuodessa
- Joustavammat toimintamallit uusien ominaisuuksien tuotantoon saattamisessa vaativatkin täysin erilaista kulttuuria, sellaista, missä kehittäjät (dev) ja ylläpito (ops) työskentelevät tiiviissä yhteistyössä
    - Sovelluskehittäjille tulee antaa tarvittava pääsy tuotantopalvelimelle, scrum-tiimiin sijoitetaan ylläpitovastuilla olevia ihmisiä
- Toimintamallia missä dev ja ops työskentelevät tiiviisti yhdessä on alettu kutsua termillä *DevOps*

devops-kulttuuri

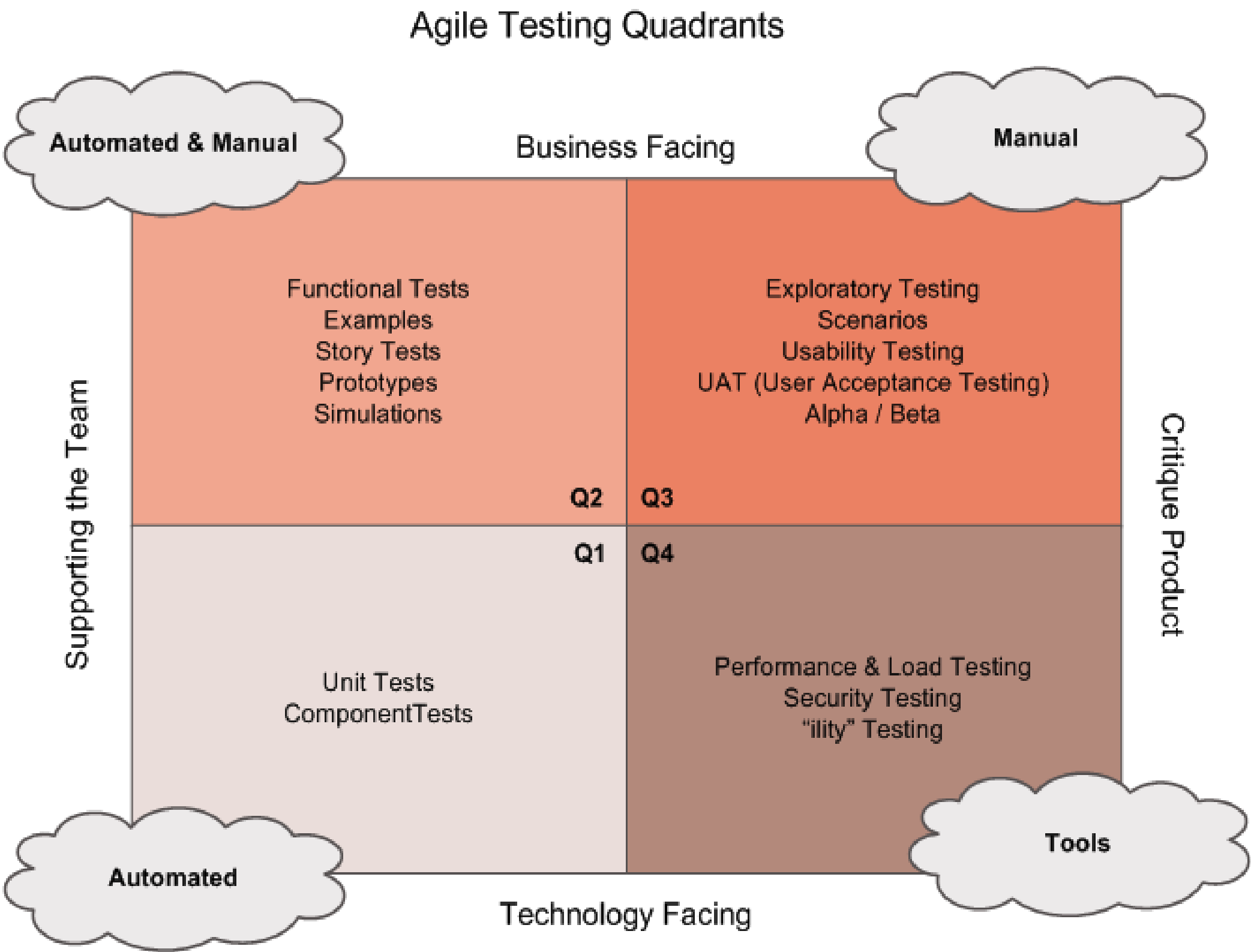
- DevOps on termi joka on nykyään monin paikoin esillä
  - esim. työpaikkailmoituksissa voidaan arvostaa DevOps-taitoja tai jopa etsiä ihmistä DevOps-tiimiin
  - On myös myynnissä mitä erilaisempia DevOps-työkaluja
- On kuitenkin erittäin epäselvää mitä kukin tarkoittaa DevOps:illa
- Suurin osa (järkevistä) määritelmistä tarkoittaa DevOpsilla nimenomaan kehittäjien ja järjestelmäylläpidon yhteistä työnteon tapaa, ja sen takia onkin hyvä puhua *DevOps-kulttuurista*
  - https://martinfowler.com/bliki/DevOpsCulture.html
- On olemassa joukko käsitteellisiä ja teknisiäkin työkaluja, jotka usein liitetään DevOps-tyyliseen työskentelyyn, esim.
  - Automatisoitu testaus
  - Continuous deployment
  - Virtualisointi ja kontainerisointi (docker)
  - infrastructure as code
  - pilvipalveluna toimivat palvelimet ja sovellusympäristöt (PaaS, IaaS, SaaS)

devops-kulttuuri

- Monet edellisistä ovat kehittyneet vasta viimeisen 5-10 vuoden aikana ja täten mahdollistaneet devops:in helpomman soveltamisen
- Eräs tärkeimmistä devops:ia mahdollistavista asioista on ollut siirtyminen yhä enemmän käyttämään fyysisten palvelinten sijaan virtuaalisia ja pilvessä toimivia palvelimia, tällöin raudastakin on tullut ”koodia”
  - Tämä on tehnyt palvelinten ohjelmoinnillisen hallinnoinnin mahdolliseksi
  - Palvelinten konfiguraatioita voidaan tallettaa versionhallintaan ja jopa testata
  - Sovelluskehitys ja ylläpito ovat alkaneet muistuttaa enemmän toisiaan kuin vanhoina aikoina
- Työkalujen käyttöönotto ei kuitenkaan riitä, DevOps:in ”tekeminen” lähtee pohjimmiltaan kulttuurisista tekijöistä, tiimirakenteista, sekä asioiden sallimisesta
- Scrumin ja agilen eräs tärkeimmistä periaatteista on tehdä kehitystiimeistä itseorganisoituvia ja ”cross functional”, eli sellaisia että ne sisältävät kaiken tietotaidon uusien ominaisuuksien Definition of Donen tasolla valmiiksi saattamista varten
- DevOps onkin eräs keino viedä ketteryyttä askeleen pitemmälle, mahdollistaa se, että ketterät tiimit ovat todella cross functional ja että ne pystyvät viemään vaivattomasti toteuttamansa uudet toiminnallisuudet tuotantoympäristöön asti ja jopa testaamaan niitä tuotannossa

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
  - Business facing … technology 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

- Seuraavassa esitettävät asiat ovat osin omia, kokemuksen ja kirjallisuuden perusteella hankittuja testaukseen liittyviä mielipiteitä
- 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

Loppupäätelmiä testauksesta

- Jos ohjelmistossa on komponentteja, jotka tullaan ehkä poistamaan tai korvaamaan pian, saattaa olla järkevää olla automatisoimatta niiden testejä
  - Esim. luennolla 3 esitelty *MVP* eli *Minimal Viable Product* on karsittu toteutus, jonka avulla halutaan nopeasti selvittää, onko jokin ominaisuus ylipäätään käyttäjien kannalta arvokas
  - Jos MVP:n toteuttama ominaisuus osoittautuu tarpeettomaksi, se poistetaan järjestelmästä
- Ongelmallista kuitenkin usein on, että tätä ei tiedetä yleensä ennalta ja pian poistettavaksi tarkoitettu komponentti voi jäädä järjestelmään pitkäksikin aikaa
- Kokonaan uutta ohjelmistoa tai komponenttia tehtäessä voi olla järkevää antaa ohjelman rakenteen ensin stabiloitua ja tehdä kattavammat testit vasta myöhemmin
  - Komponenttien testattavuus kannattaa kuitenkin pitää koko ajan mielessä vaikka niille ei heti testejä tehtäisikään

Loppupäätelmiä testauksesta

- 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 käyttöliittymään tehtävistä muutoksista