Ohjelmistotuotanto

Luento 7

20.11.

Testaus ketterissä menetelmissä, nopea kertaus..

Ketterien menetelmien testauskäytänteitä

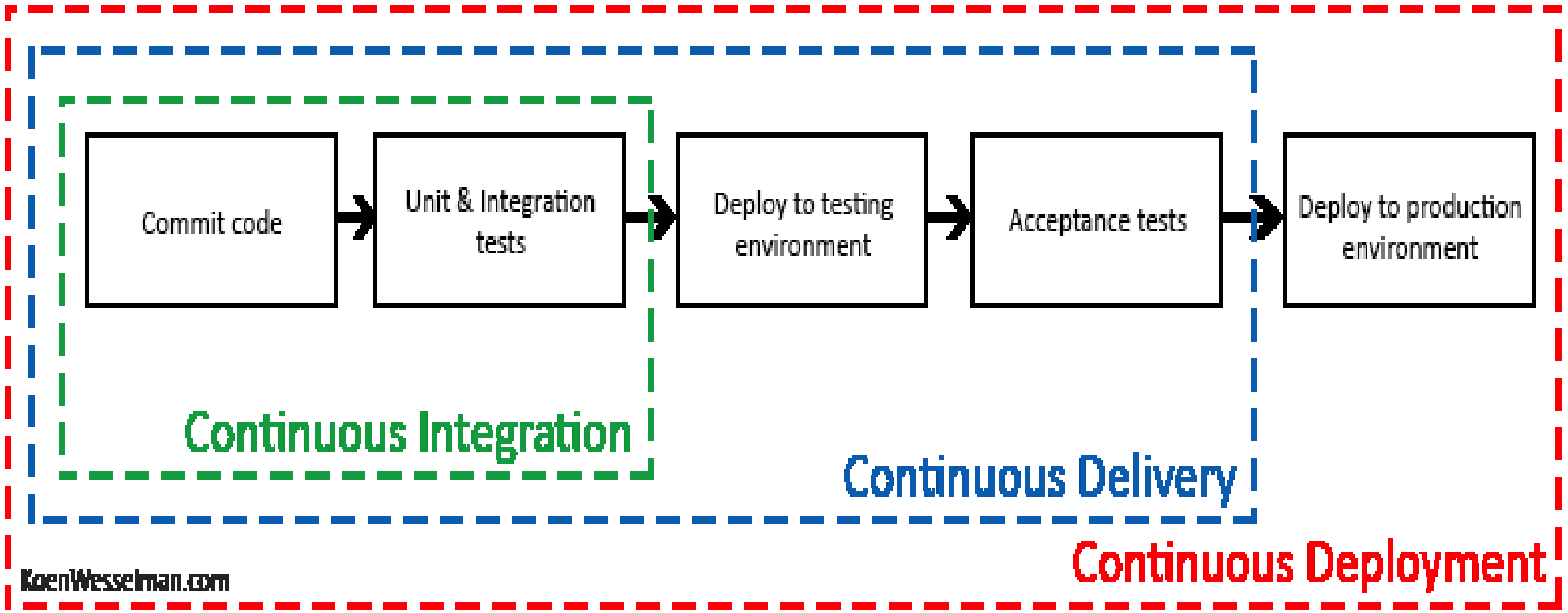
- Testauksen rooli ketterissä menetelmissä poikkeaa huomattavasti vesiputousmallisesta ohjelmistotuotannosta
  - Testaus on integroitu kehitysprosessiin ja testaajat työskentelevät osana kehittäjätiimejä
  - Testausta tapahtuu projektin ”ensimmäisestä päivästä” lähtien
  - Toteutuksen iteratiivisuus tekee regressiotestauksen automatisoinnista erityisen tärkeää
- Viimeksi puhuimme neljästä ketterästä testaamisen menetelmästä:
  - **Test driven development (TDD)**
  - **Acceptance Test Driven Development / Behavior Driven Development**
    - Etenkin TDD:ssä on on kyse enemmän ohjelman suunnittelusta kuin testaamisesta. sivutuotteena syntyy toki kattava joukko testejä
  - **Continuous Integration (CI)** suomeksi jatkuva integraatio
    - Moderni kehitys on kulkenut kohti **Continuous deploymentiä** eli automaattisesti tapahtuvaa jatkuvaa tuotantoonvientiä
  - **Exploratory testing**, suomeksi tutkiva testaus

Continuous deployment ja deployment pipeline

- Jatkuvassa käyttöönotossa siis jokainen sovelluskehittäjän commit voi mahdollisesti johtaa järjestelmän uuden version tuotantoonvientiin
- Commit kulkee **deployment pipelinen** läpi
  - CI-palvelin suorittaa commitille joukon testejä
  - Seuraavassa vaiheessa commitin aikaansaama sovelluksen uusi versio siirtyy **staging-ympäristön**
  - Staging-ympäristössä sovelluksen uudelle versiolle suoritetaan lisää testejä
    - Testit ovat lähinnä järjestelmätason testejä, jotka varmistavat, että sovellus toimii käyttäjän kannalta halutulla tavalla
  - Jos staging-ympäristössä suoritetut testit menevät läpi, siirtyy uusi versio tuotantoympäristöön
- **Staging-ympäristö** on sekä konfiguraatioiltaan, että käsiteltävän datan suhteen mahdollisimman paljon tuotantoympäristön kaltainen ympäristö

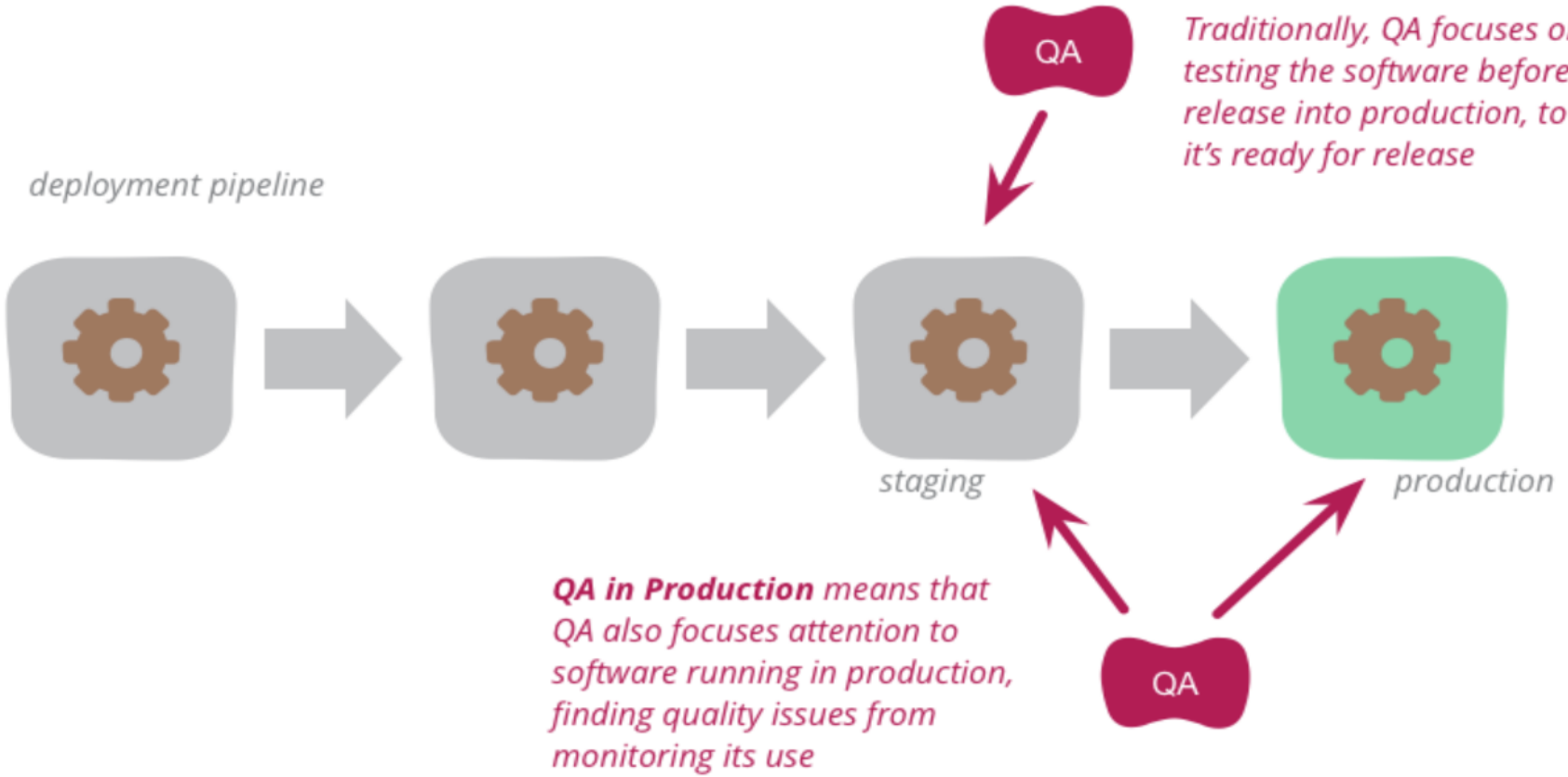
Continuous deployment ja deployment pipeline

- Lopullinen tuotantoonvienti voi olla automaattinen, tällöin puhutaan *jatkuvasta tuotantoonviennistä,* **continuous deployment**
- Tai tuotantoonvienti voi myös tapahtua manuaalisesti, tästä käytänteestä käytetään nimitystä *jatkuva toimitusvalmius,* **continuous delivery**
  - Ohjelmiston asiakkaalla voi olla useita syitä miksi sovelluksen uusia versiota ei välttämättä haluta heti käyttöön vaan esim. kahden viikon välein



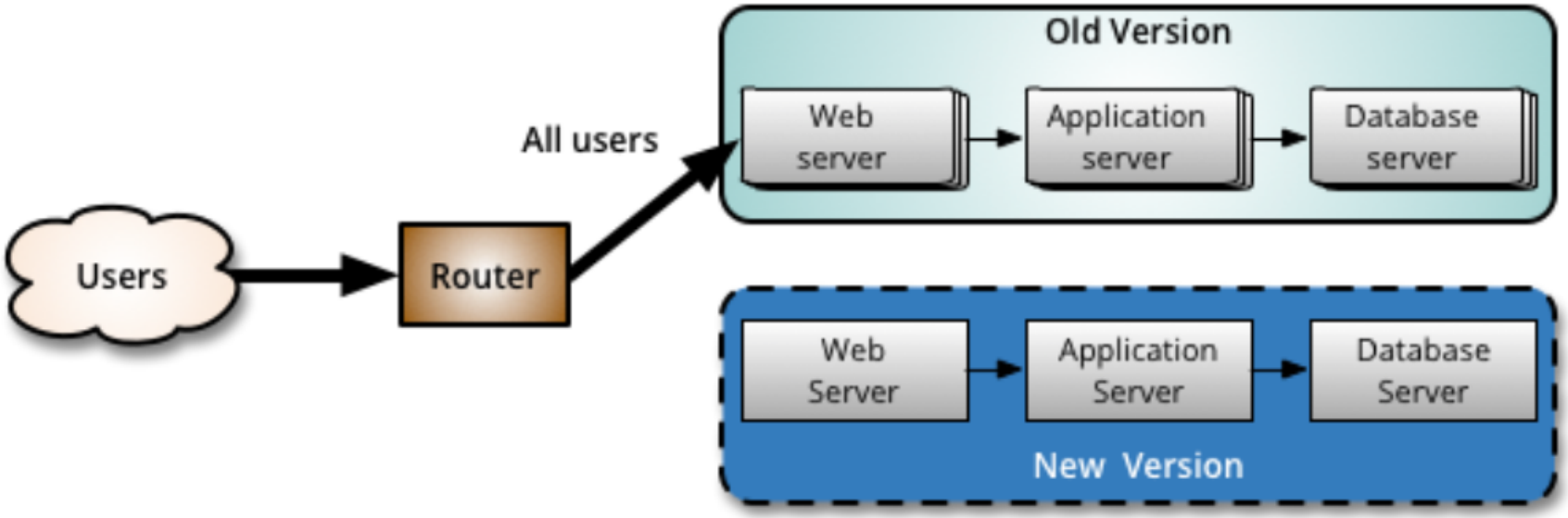
Tuotannossa tapahtuva testaaminen ja laadunhallinta

- 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



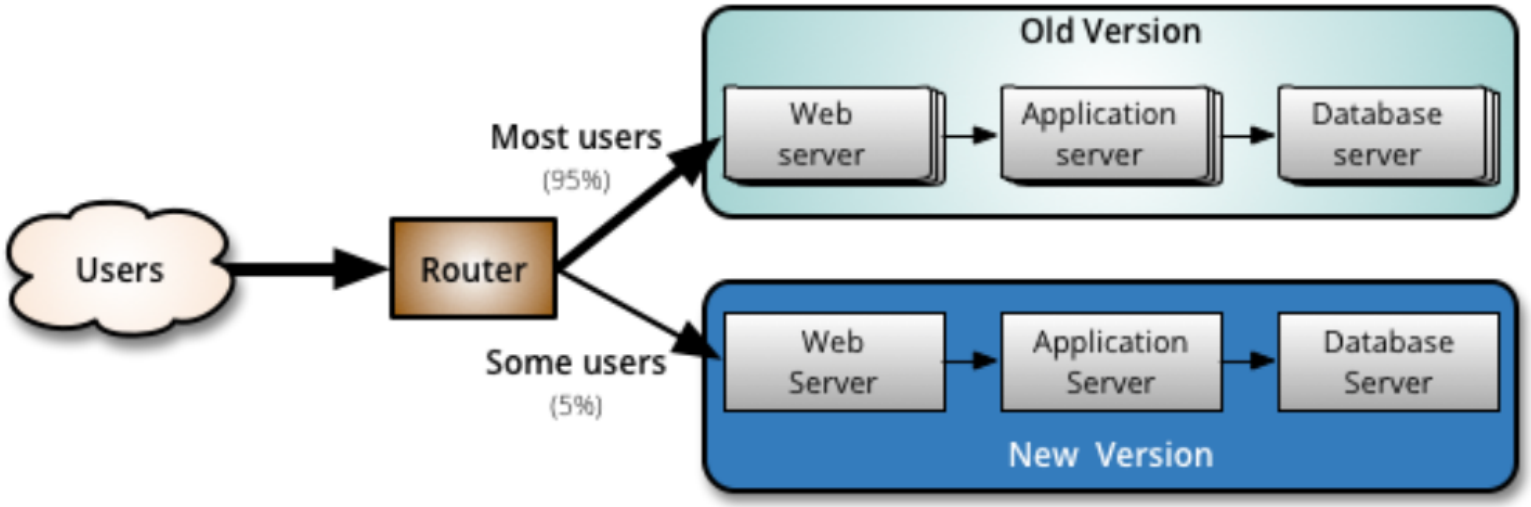
- Kun järjestelmään toteutetaan uusi ominaisuus, deployataan se ensin passiivisena olevaan ympäristöön

Blue-green-deployment

- 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
- 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
- On tarkoituksenmukaista, että kaikki blue-green-deploymentiin liittyvät testit, niiden tulosten varmistaminen, tuotantoympäristön vaihto ja mahdollinen rollback tapahtuvat automatisoidusti

canary release

- Blue-green-deploymentin hieman pidemmälle viedyssä versiossa ***canary-releasessa*** uuden ominaisuuden sisältävään ympäristöön ohjataan osa, esim. 5% järjestelmän käyttäjistä



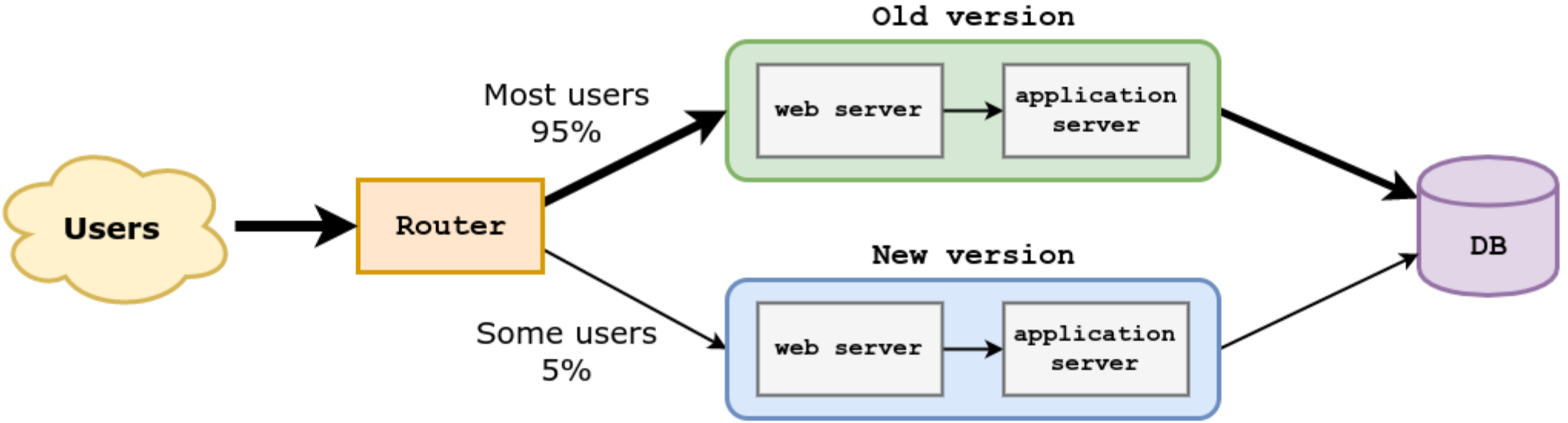
- Uuden ominaisuuden sisältämää versiota monitoroidaan aktiivisesti ja jos ongelmia ei ilmene, vähitellen kaikki liikenne ohjataan uuteen versioon
- Kuten blue-green-deploymentin tapauksessa, ongelmatilanteissa palautetaan käyttäjät aiempaan, toimivaksi todettuun versioon

canary release

- 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 metriikoita** (engl. bussiness level metrics)
  - 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
- Myös canary releasejen yhteydessä testauksen ja kaikkien tuotantoon vientiin liittyvän on syytä tapahtua automatisoidusti

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 tietokantaa



Tuotannossa testaaminen ja 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 tila synkronoida, jotta järjestelmien vaihtaminen onnistuu saumattomasti
  - yhteen kantaa sovelluksen tekemät päivitykset on siis tavalla tai toisella tehtävä myös toiseen, kenties skeemaltaan jo muuttuneeseen kantaan

- Nimi *canary release* periytyy kaivostyöläisten tavasta käyttää kanarialintuja tutkimaan sitä onko kaivoksessa myrkyllisiä kaasuja, jos kaivokseen viety lintu ei kuole, ilma on turvallista

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

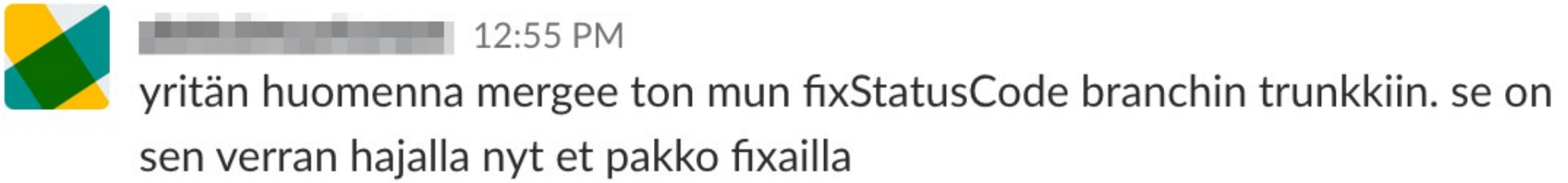
```
List<News> recommendedNews(User user) {
  if ( isInCanaryRelease(user) ) {
    return experimentalRecommendationAlgorith(user)
  } else {
    return recommendationAlgorith(user)
  }
}
```

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

Feature branchit ja merge hell

- Edellisellä kalvolla mainittiin *feature branchit*
- Kyseessä on siis käytäntö, missä uudet uusi ominaisuus, esim user story toteutetaan ensin omaan versionhallinnan haaraansa (branch) ja ominaisuuden valmistuttua haara mergetään pääkehityshaaraan (esim. masteriin)
- Monet pitävät feature brancheja versionhallinnan käytön best practicena
- Viime aikoina on kuitenkin huomattu, että feature branchit aiheuttavat helposti pahoja merge-konflikteja sprintin lopussa
  - Seurauksena pienimuotoinen integraatiohelvetti: *merge hell*
- Arkipäivää ohjelmistotiimissä



trunk based development

- Viime aikaisena suuntauksena noussut esiin ns **trunk based development** missä pitkäikäisiä feature brancheja pyritään välttämään tai ne jopa kielletään
- Kaikki muutokset tehdään suoraan pääkehityshaaraan, josta käytetään nimitystä *trunk*
  - Pääkehityshaara voi olla master tai joku erillinen branch käytännöistä riippuen
  - Ohjelmiston kustakin julkaistusta versiosta saatetaan tarvittaessa tehdä oma *release branch*
- Trunk-pohjainen kehitys pakottaa sovelluskehittäjät tekemään pieniä, nopeasti päähaaraan mergettäviä muutoksia
- Trunk-pohjainen kehitys yhdistetään usein feature toggleihin, näin puolivalmiitakin ominaisuuksia voidaan helposti mergetä päähaaraan ilman sovelluksen olemassaolevan toiminnallisuuden sotkemista
- Trunk-pohjainen kehitysmalli edellyttää sovelluskehittäjiltä kuria ja systemaattisuutta
  - Feature togglejen holtiton käyttö voi johtaa *feature toggle helvettiin…*
- Trunk-pohjaista kehitysmallia noudattavat esim. Google, Facebook ja Netflix
- https://trunkbaseddevelopment.com

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

DevOps

- Toimintamallia missä dev ja ops työskentelevät tiiviisti yhdessä on alettu kutsua termillä *DevOps*
- 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*
- 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)

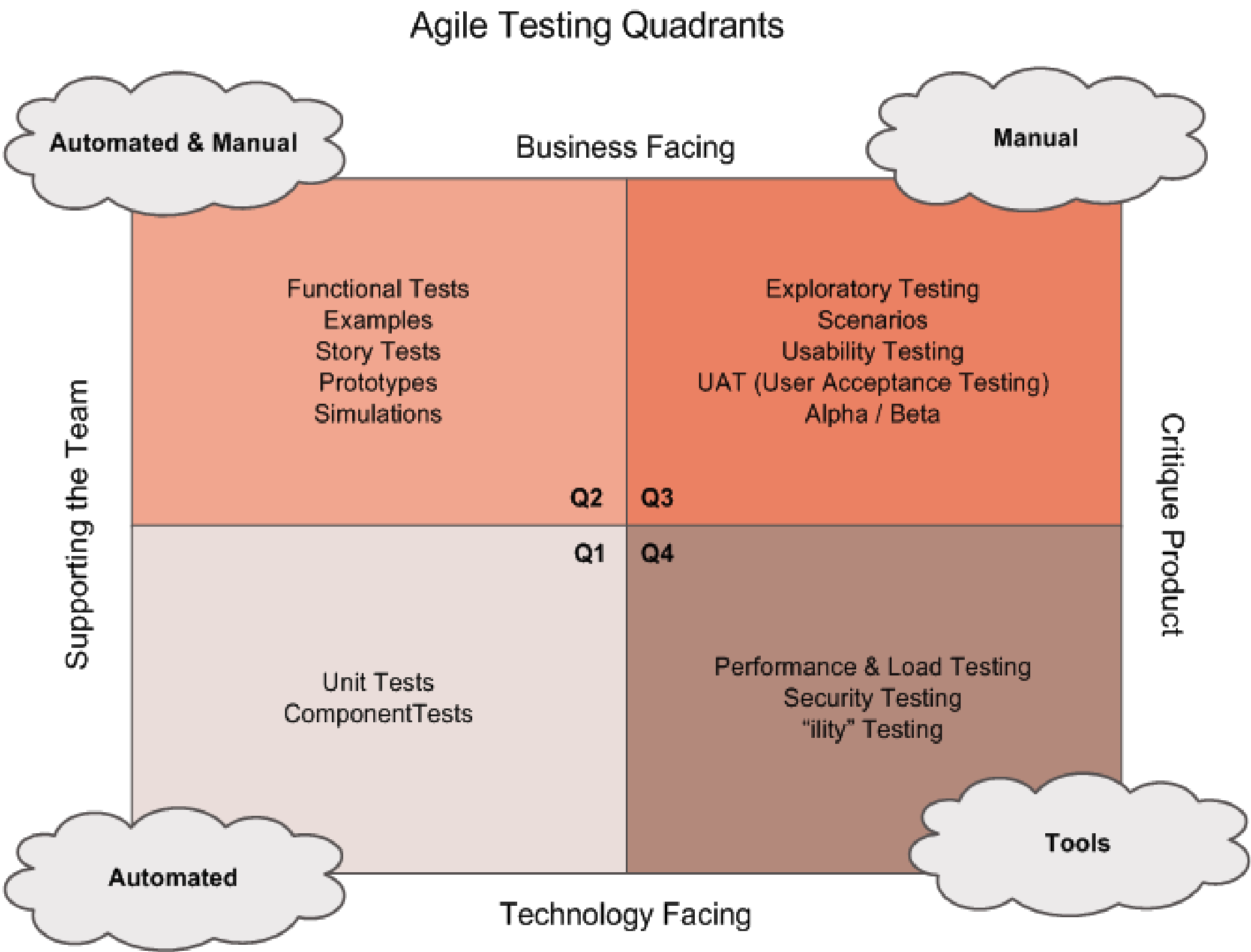
DevOps

- 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”, eli **infrastructure as code**
  - 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 (huonoina) 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 ja operoimaan 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

Ohjelmiston suunnittelu

Ohjelmiston suunnittelu ja toteutus

- Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekeminen sisältää
  - vaatimusten analysoinnin ja määrittelyn
  - suunnittelun
  - toteuttamisen
  - testauksen ja
  - ohjelmiston ylläpidon
- Olemme käsitelleet vaatimusmäärittelyä ja testaamista erityisesti ketterien ohjelmistotuotannon näkökulmasta
- Siirrymme kurssin ”viimeisessä osassa” käsittelemään ohjelmiston *suunnittelua ja toteuttamista*
  - Osa suunnittelusta tapahtuu vasta toteutusvaiheessa, joten suunnittelun ja toteuttamisen käsittelyä ei ole järkevä eriyttää
- **Ohjelmiston suunnittelun tavoitteena on määritellä miten saadaan toteutettua vaatimusmäärittelyn mukaisella tavalla toimiva ohjelma**

Ohjelmiston suunnittelu

- Suunnittelun ajatellaan yleensä jakautuvan kahteen vaiheeseen:
  - **Arkkitehtuurisuunnittelu**
    - Ohjelman rakenne karkealla tasolla
    - Mistä suuremmista rakennekomponenteista ohjelma koostuu?
    - Miten komponentit yhdistetään, eli komponenttien väliset rajapinnat
  - **Oliosuunnittelu**
    - yksittäisten komponenttien suunnittelu
- Suunnittelun ajoittuminen riippuu käytettävästä tuotantoprosessista:
  - Vesiputousmallissa suunnittelu tapahtuu vaatimusmäärittelyn jälkeen ja ohjelmointi aloitetaan vasta kun suunnittelu valmiina ja dokumentoitu
  - Ketterissä menetelmissä suunnittelua tehdään tarvittava määrä jokaisessa iteraatiossa, tarkkaa suunnitteludokumenttia ei yleensä ole
- Vesiputousmallin mukainen suunnitteluprosessi tuskin on enää juuri missään käytössä, ”jäykimmissäkin” prosesseissa ainakin vaatimusmäärittely ja arkkitehtuurisuunnittelu limittyvät
  - Tarkkaa ja raskasta ennen ohjelmointia tapahtuvaa suunnittelua (BDUF eli Big Design Up Front) toki edelleen tapahtuu ja tietynlaisiin järjestelmiin (hyvin tunnettu sovellusalue, muuttumattomat vaatimukset) se osittain sopiikin

Arkkitehtuurisuunnittelu

Ohjelmiston arkkitehtuuri

- Termiä ohjelmistoarkkitehtuuri (software architecture) on käytetty jo vuosikymmeniä
- Termi on vakiintunut yleiseen käyttöön 2000-luvun aikana ja on siirtynyt mm. ”tärkeää työntekijää” tarkoittavaksi nimikkeeksi
  - Ohjelmistoarkkitehti engl. Software architech
- Useimmilla alan ihmisillä on jonkinlainen kuva siitä, mitä ohjelmiston arkkitehtuurilla tarkoitetaan
  - Kyseessä ohjelmiston rakenteen suuret linjat
- Termiä ei ole kuitenkaan yrityksistä huolimatta onnistuttu määrittelemään siten että asiantuntijat olisivat määritelmästä yksimielisiä
- IEEE:n standardi *Recommended practices for Architectural descriptions of Software intensive systems* määrittelee käsitteen seuraavasti
  - *Ohjelmiston arkkitehtuuri on järjestelmän perusorganisaatio, joka sisältää järjestelmän osat, osien keskinäiset suhteet, osien suhteet ympäristöön sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua ja evoluutiota*

Ohjelmiston arkkitehtuuri, muita määritelmiä

- Krutchten:
  - An architecture is the **set of significant decisions about the organization of a software system**, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these elements into progressively larger subsystems, and the **architectural style** that guides this organization -- these elements and their interfaces, their collaborations, and their composition.
- McGovern:
  - The software architecture of a system or a collection of systems consists of all the important design decisions about the software structures and the interactions between those structures that comprise the systems. **The design decisions support a desired set of qualities that the system should support to be successful**. The design decisions provide a conceptual basis for system development, support, and maintenance.

Arkkitehtuuriin kuuluu

- Vaikka arkkitehtuurin määritelmät hieman vaihtelevat, löytyy määritelmistä joukko samoja teemoja
- Lähes jokaisen määritelmän mukaan arkkitehtuuri määrittelee ohjelmiston rakenteen, eli jakautumisen erillisiin osiin ja osien väliset rajapinnat
- Arkkitehtuuri ottaa kantaa rakenteen lisäksi myös käyttäytymiseen
  - Arkkitehtuuritason rakenneosien vastuut ja niiden keskinäisen kommunikoinnin muodot
- Arkkitehtuuri keskittyy järjestelmän tärkeisiin/keskeisiin osiin
  - Arkkitehtuuri ei siis kuvaa järjestelmää kokonaisuudessaan vaan on isoihin linjoihin keskittyvä abstraktio
  - Tärkeät osat voivat myös muuttua ajan myötä, eli arkkitehtuuri ei ole muuttumaton
  - http://www.ibm.com/developerworks/rational/library/feb06/eeles/

”Arkkitehtuuri on ne asiat joita on vaikea vaihtaa”

- Artikkelissa ”Who needs architect” Martin Fowler toteaa seuraavasti
  - you might end up defining architecture as **things that people perceive as hard to change**
  - https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Melkein sama hieman toisin ilmaistuna oli Krutchtenin määritelmässä
  - **set of significant decisions about the organization of a software system**
- Konkreettinen esimerkki tälläisestä arkkitehtoonisesta päätöksestä tuli esiin miniprojektien ensimmäisellä viikolla
- Monissa projekteissa oli seuraava User story
  - *Ohjelmaa tulee voida käyttää useilta eri koneilta*
- Story ei varmasti ole millään projektilla priorisoitu kovin korkealle, ja ei ole ainakaan ensimmäisen sprintin tavoitteissa

”Arkkitehtuuri on ne asiat joita on vaikea vaihtaa”

- Kaikki ryhmät tekevät kuitenkin jo ensimmäisellä viikolla tärkeän ratkaisun **(significant decisions about …)** sen suhteen miten ohjelma toteutetaan:
  - Konsolisovelluksena
  - Java Swingillä/FX:llä
  - Web-sovelluksena
- Tällä ”arkitehtoonisella päätöksellä” on erittäin suuri vaikutus miten story

  Ohjelmaa tulee voida käyttää useilta eri koneilta

  voidaan toteuttaa siinä vaiheessa kun sen toteututuksen aika tulee
- Tämä ensimmäisen viikon tärkeä arkitehtooninen päätös siis johtaa tilanteeseen, joka on myöhemmin **hard to change**
  - Esim. siirtyminen konsolisovelluksesta WebMVC-sovellukseen ei ole suinkaan mahdoton, mutta se edellyttää paljon töitä

Arkkitehtuuriin vaikuttavia tekijöitä

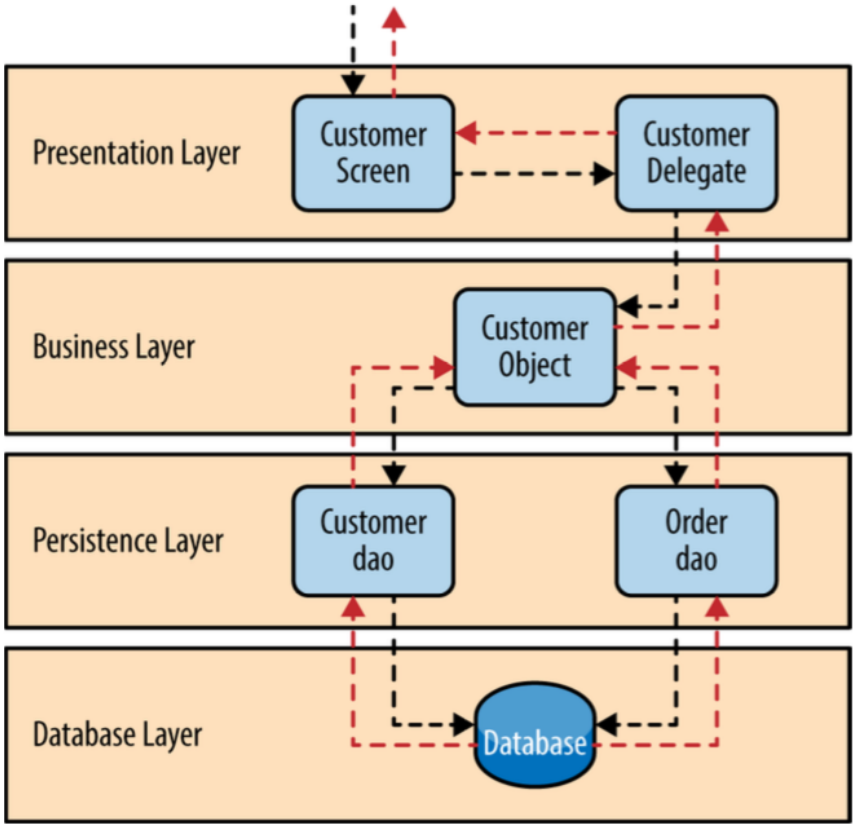
- Järjestelmälle asetetuilla *ei-toiminnallisilla laatuvaatimuksilla* (engl. -ilities) on suuri vaikutus arkkitehtuuriin
  - Käytettävyys, suorituskyky, skaalautuvuus, vikasietoisuus, tiedon ajantasaisuus, tietoturva, ylläpidettävyys, laajennettavuus, hinta, time-to-market, ...
- Laatuvaatimukset ovat usein ristiriitaisia, joten arkkitehdin tulee hakea kaikkia sidosryhmiä tyydyttävä kompromissi
  - Esim. time-to-market lienee ristiriidassa useimpien laatuvaatimusten kanssa
  - Tiedon ajantasaisuus, skaalautuvuus ja vikasietoisuus ovat myös piirteitä, joiden suhteen on pakko tehdä kompromisseja, kaikkia ei voida saavuttaa ks. http://en.wikipedia.org/wiki/CAP_theorem
- Myös järjestelmän toimintaympäristö ja valitut toteutusteknologiat muokkaavat arkkitehtuuria
  - Organisaation standardit
  - Integraatio olemassaoleviin järjestelmiin
  - Toteutuksessa käytettävät sovelluskehykset

Arkkitehtuurimalli

- Järjestelmän arkkitehtuuri perustuu yleensä yhteen tai useampaan **arkkitehtuurimalliin** (architectural pattern), jolla tarkoitetaan hyväksi havaittua tapaa strukturoida tietyntyyppisiä sovelluksia
  - Samasta asiasta käytetään joskus nimitystä **arkkitehtuurityyli** (architectural style)
- Arkkitehtuurimalleja on suuri määrä, esim:
  - Kerrosarkkitehtuuri
  - MVC
  - Pipes-and-filters
  - Repository
  - Client-server
  - publish-subscribe
  - event driven
  - REST
  - Microservice
  - SOA
- Useimmiten sovelluksen rakenteesta löytyy monien arkkitehtuuristen mallien piirteitä

Kerrosarkkitehtuuri

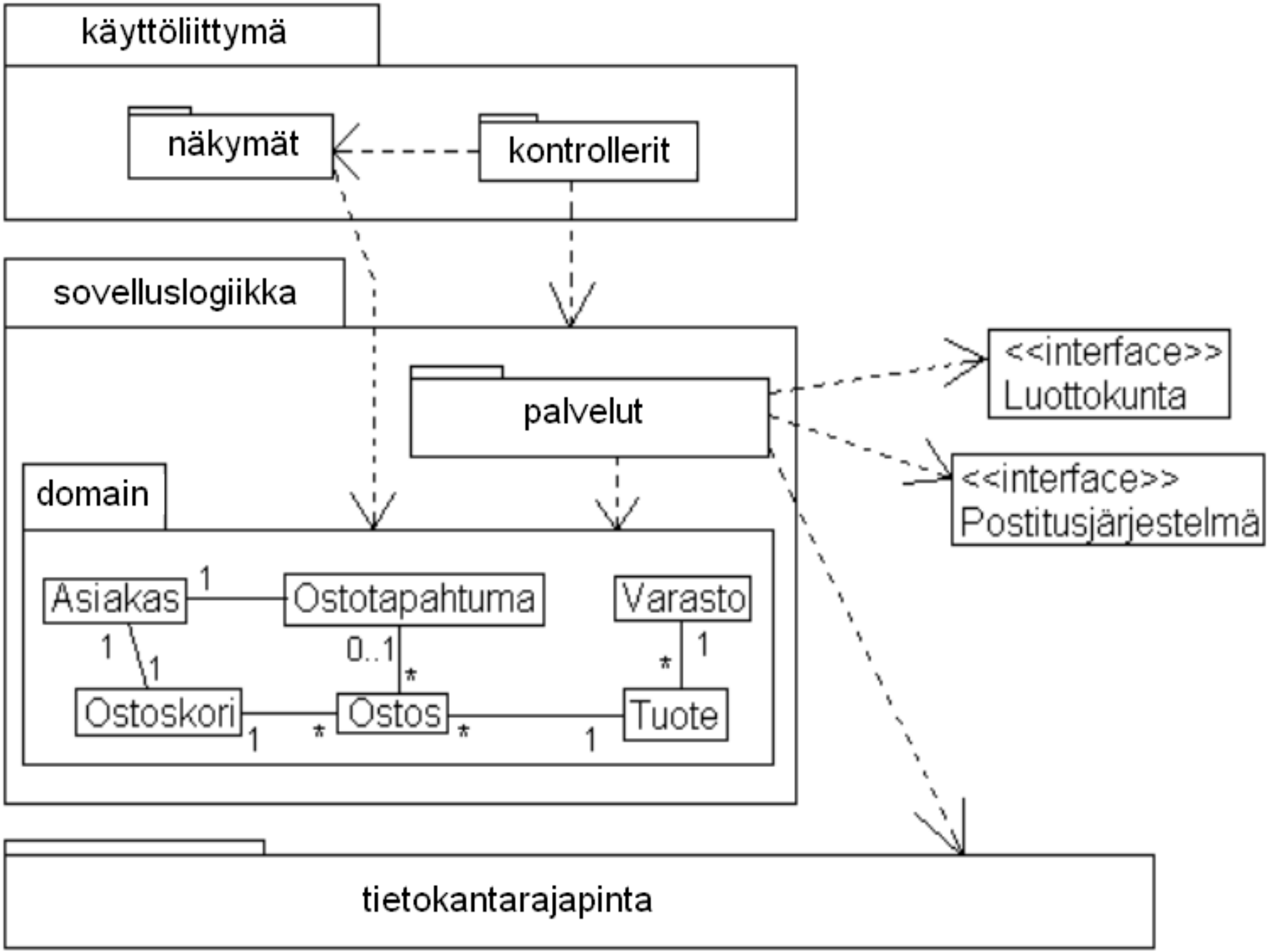
- Eräs tunnetuimmista on *kerrosarkkitehtuurimalli*
  - Kerros on kokoelma toisiinsa liittyviä olioita tai alikomponentteja, jotka muodostavat toiminnallisuuden suhteen loogisen kokonaisuuden
  - Pyrkimyksenä järjestellä komponentit siten, että ylempänä oleva kerros käyttää ainoastaan alempana olevien kerroksien tarjoamia palveluita



- Kerrosarkkitehtuuri on sovelluskehittäjän kannalta selkeä mutta saattaa johtaa massiivisiin monoliittisiin sovelluksiin, joita on lopulta vaikea laajentaa ja joiden skaalaaminen suurille käyttäjämäärille voi muodostua ongelmaksi

Kumpulabeershop

- Seuraavalla sivulla kuvaus Kumpulabiershopin arkkitehtuurista
- Arkkitehtuuri on mukaelma kerrosarkkitehtuuria (layered architecture) ja MVC-mallia
  - Kuvaus on UML-pakkauskaaviona, näyttäen osin myös pakkausten sisäisiä luokkia
  - Luokkatasolle ei yleensä arkkitehtuurikuvauksissa mennä
- Koodi osoitteessa https://github.com/mluukkai/BeerShop
- *Koodin tasolla arkkitehtuuri ilmenee luokkien sijoittelusta pakkauksiin*
- Ohjelman kaikki koodi on nyt yhdessä projektissa
- Laajempien ohjelmien tapauksessa voi olla tarkoituksenmukaista jakaa koodi useampaan eri projektiin, erityisesti jos sovelluksessa on komponentteja, joita hyödynnetään useimmissa ohjelmissa
- Jos sovellus koostuu eri projekteissa toteutetuista komponenteista, ”pääprojekti” sisällyttää silloin aliprojekteissa toteutetut komponentit esim. gradle-riippuvuuksina

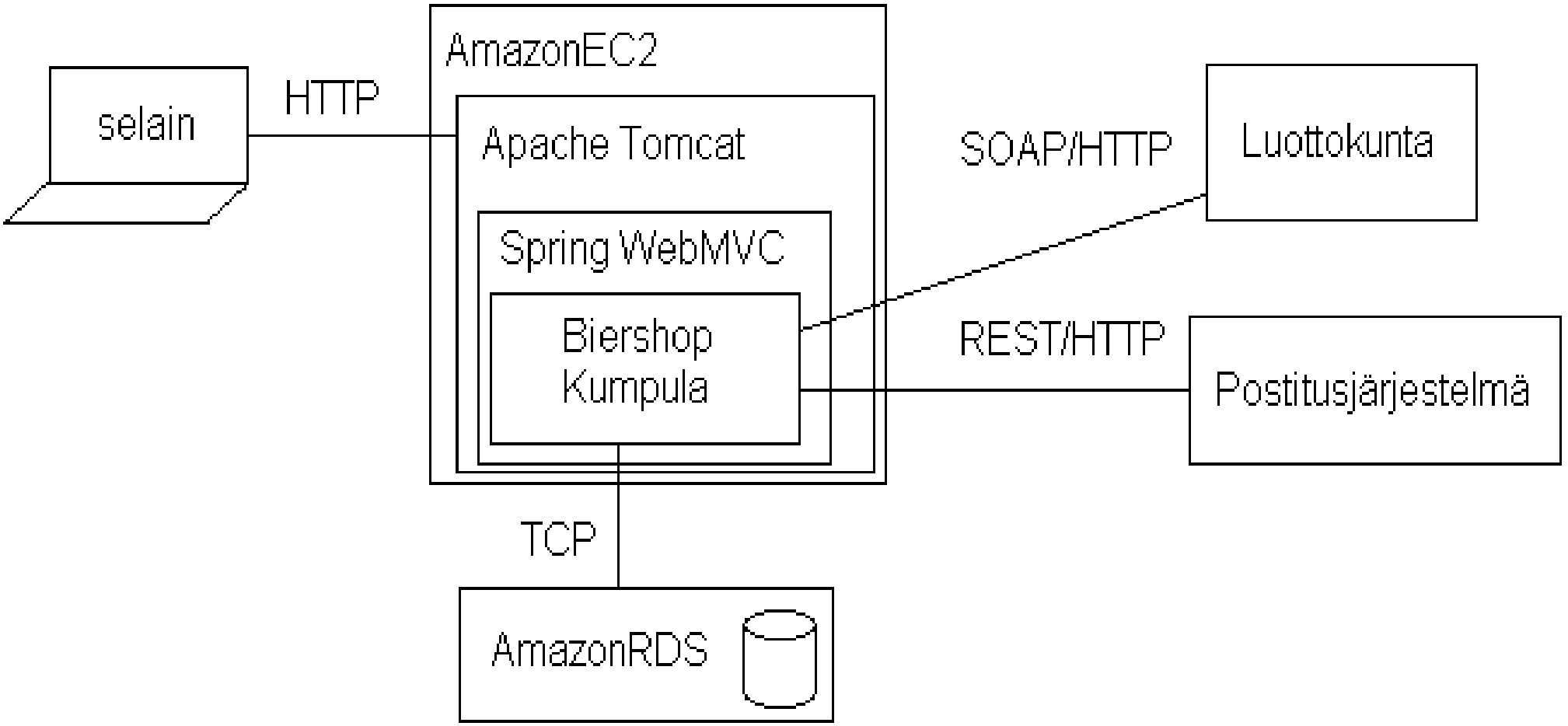

käyttöliittymä
näkymät
kontrollerit
sovelluslogiikka
palvelut
<<interface>>
Luottokunta
<<interface>>
Postitusjärjestelmä
domain
Asiakas
Ostotapahtuma
Varasto
Ostoskori
Ostos
Tuote
1
1
1
1
0..1
*
*
1
*
*
1
tietokantarajapinta

Kumpula biershopin arkkitehtuuri

- Arkkitehtuurikuvaus näyttää järjestelmän jakaantumisen kolmeen kerroksittain järjestettyyn komponenttiin
  - Käyttöliittymä
  - Sovelluslogiikka
  - Tietokantarajapinta
- Sovelluslogiikkakerros on jaettu vielä kahteen alikomponenttiin, sovellusalueen käsitteistön sisältävään *domainiin* ja sen olioita käyttäviin sekä tietokantarajapinnan kanssa keskusteleviin palveluihin
- Käyttöliittymäkerros on myös jakautunut kahteen osaan, näkymään ja kontrollereihin
  - Käytännössä näkymällä tarkoitetaan HTML-tiedostoja
  - Kontrollereilla taas tarkoitetaan main-metodin sisällä olevia selaimen tekemien pyyntöjen käsittelymetodeja
- Kuva tarjoaa **loogisen näkymän** arkkitehtuuriin mutta ei ota kantaa siihen mihin eri komponentit sijoitellaan, eli toimiiko esim. käyttöliittymä samassa koneessa kuin sovelluksen käyttämä tietokanta

Kumpula biershopin arkkitehtuuri

- Alla **fyysisen tason kuvaus,** josta selviää että kyseessä on selaimella käytettävä, SpringWebMVC-sovelluskehyksellä tehty sovellus, jota suoritetaan AmazonEC2-palvelimella ja tietokantana on AmazonRDS
  - Myös kommunikointitapa järjestelmän käyttämiin ulkoisiin järjestelmiin (Luottokunta ja Postitusjärjestelmä) selviää kuvasta

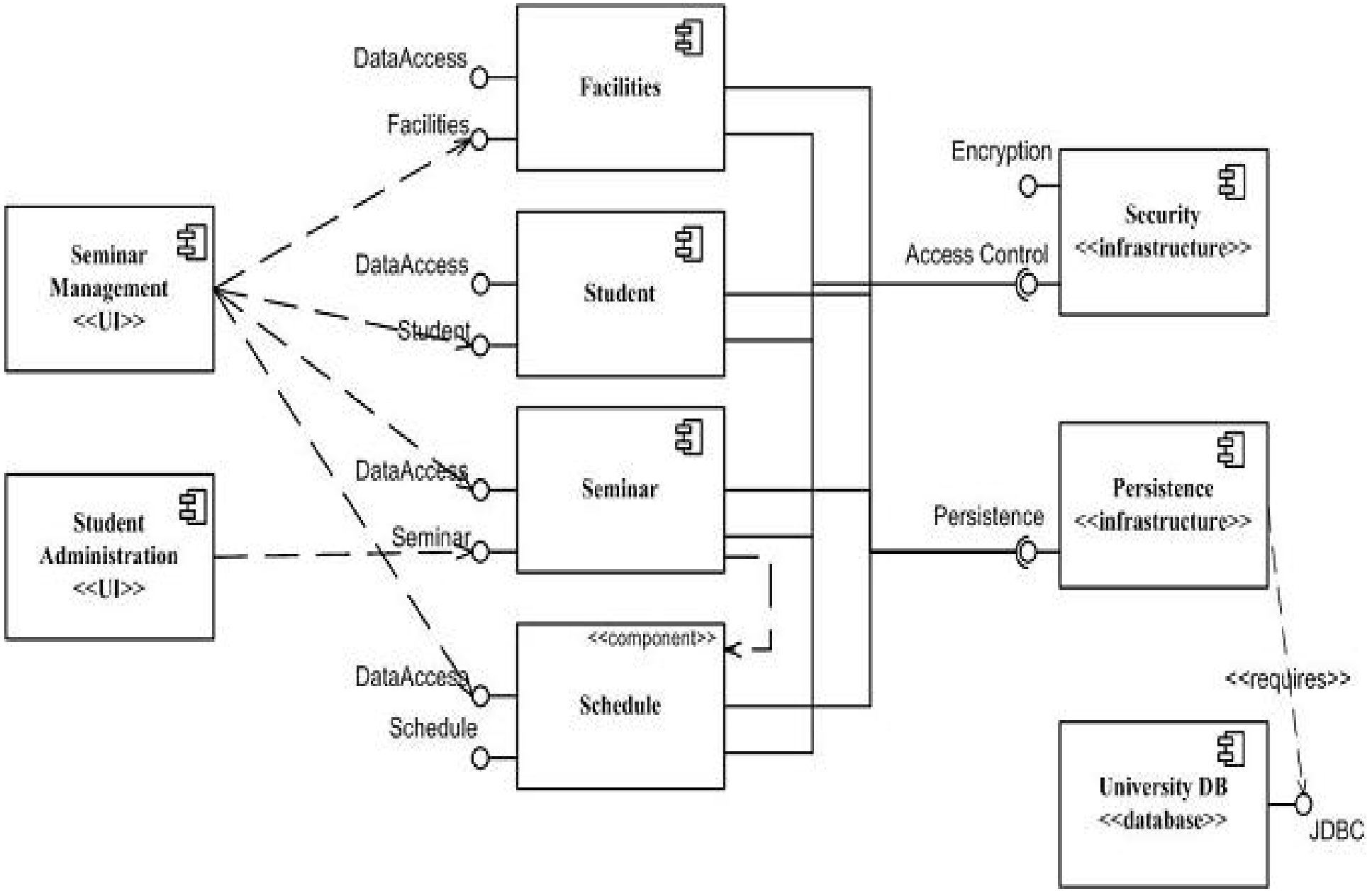


Arkkitehtuurin kuvaamisesta

- UML:n lisäksi arkkitehtuurikuvauksille ei ole vakiintunutta formaattia
  - Luokka ja pakkauskaavioiden lisäksi UML:n komponentti- ja sijoittelukaaviot voivat olla käyttökelpoisia (ks. seuraavat kalvot)
  - Useimmiten käytetään epäformaaleja laatikko/nuoli-kaavioita
- Arkkitehtuurikuvaus kannattaa tehdä useasta eri *näkökulmasta*, sillä eri näkökulmat palvelevat erilaisia tarpeita
  - Korkean tason kuvauksen avulla voidaan strukturoida keskusteluja eri sidosryhmien kanssa, esim.:
    - Vaatimusmäärittelyprosessin jäsentäminen
    - Keskustelut järjestelmäylläpitäjien kanssa
  - Tarkemmat kuvaukset toimivat ohjeena järjestelmän tarkemmassa suunnittelussa ja ylläpitovaiheen aikaisessa laajentamisessa
- Arkkitehtuurikuvaus ei suinkaan ole pelkkä kuva: mm. komponenttien vastuut tulee tarkentaa sekä niiden väliset rajapinnat määritellä
  - Jos näin ei tehdä, kasvaa riski sille että arkkitehtuuria ei noudateta
  - Hyödyllinen kuvaus myös perustelee tehtyjä arkkitehtuurisia valintoja

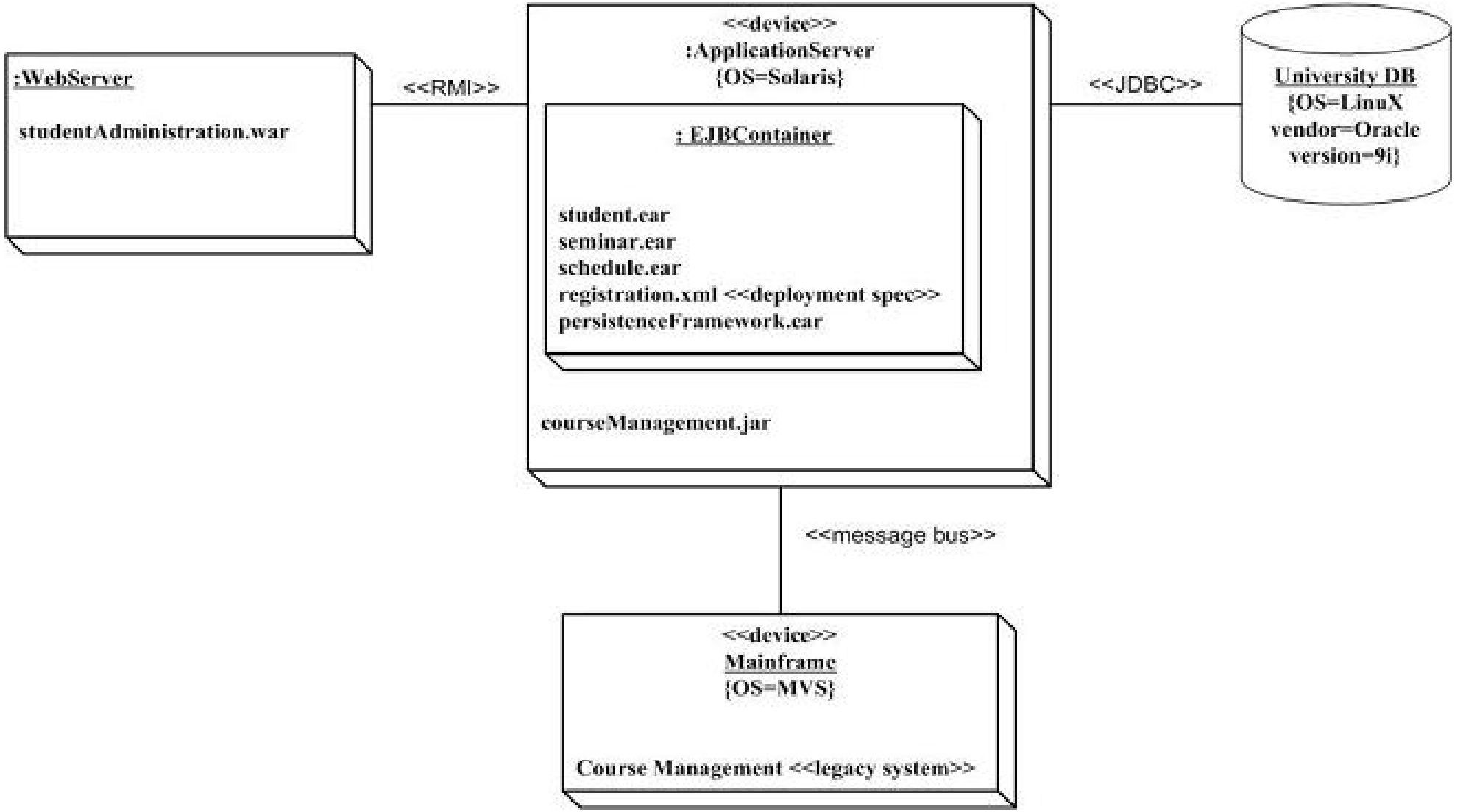
UML komponenttikaavio

- http://www.agilemodeling.com/artifacts/componentDiagram.htm



UML:n sijoittelukaavio

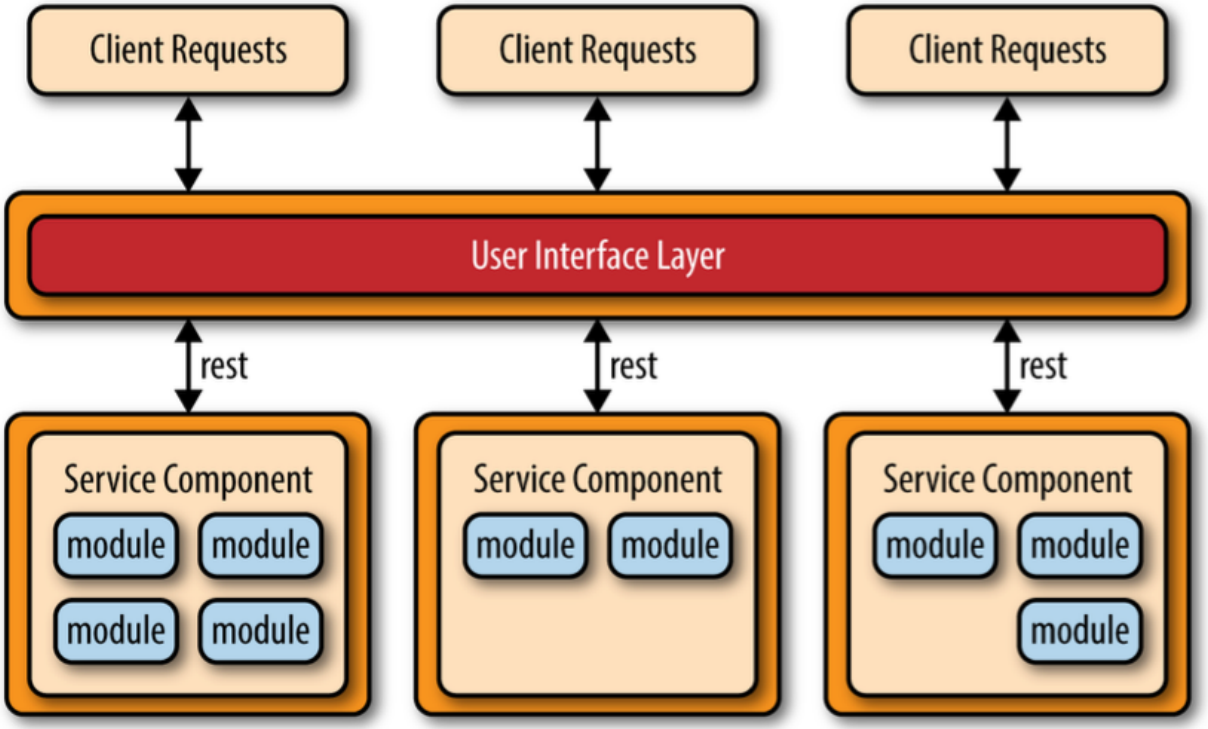
- http://www.agilemodeling.com/artifacts/deploymentDiagram.htm



- UML:n komponentti- ja sijoittelukaavio ovat jossain määrin käyttökelpoisia mutta melko harvoin käytännössä käytettyjä

Hieman lisää arkkitehtuurimalleista

- Tarkastellaan vielä hieman paria arkkitehtuurimallia
- Muutama kalvo sitten todettiin, että kerrosarkkitehtuuri saattaa johtaa massiivisiin monoliittisiin sovelluksiin, joita on lopulta vaikea laajentaa ja joiden skaalaaminen suurille käyttäjämäärille voi muodostua ongelmaksi
- Viime aikoina nopeasti yleistynyt **mikropalvelumalli** (microservices) pyrkii vastaamaan näihin haasteisiin koostamalla sovelluksen useista (jopa sadoista) pienistä verkossa toimivista autonomisista palveluista jotka keskenään verkon yli kommunikoiden toteuttavat järjestelmän toiminnallisuuden



Mikropalvelut

- Mikropalveluihin perustuvassa sovelluksessa yksittäisistä palveluista pyritään tekemään mahdollisimman riippumattomia
  - Palvelut eivät esim. käytä yhteistä tietokantaa
  - Palvelut on toteutettu omissa koodiprojekteissaan ja ne eivät jaa koodia
  - Palvelut eivät kutsu toistensa metodeja vaan ne keskustelevat keskenään verkon välityksellä
- Mikropalveluiden on tarkoitus olla pieniä ja huolehtia vain ”yhdestä asiasta”
  - Kun järjestelmään lisätään toiminnallisuutta, se yleensä tarkoittaa uusien palveluiden toteuttamista tai ainoastaan joidenkin palveluiden laajentamista
  - Sovelluksen laajentaminen voi olla helpompaa kuin kerrosarkkitehtuurissa
- Mikropalveluja hyödyntävää sovellusta voi olla helpompi skaalata
  - suorituskyvyn pullonkaulan aiheuttavia mikropalveluja voidaan suorittaa useita rinnakkain

Mikropalvelut

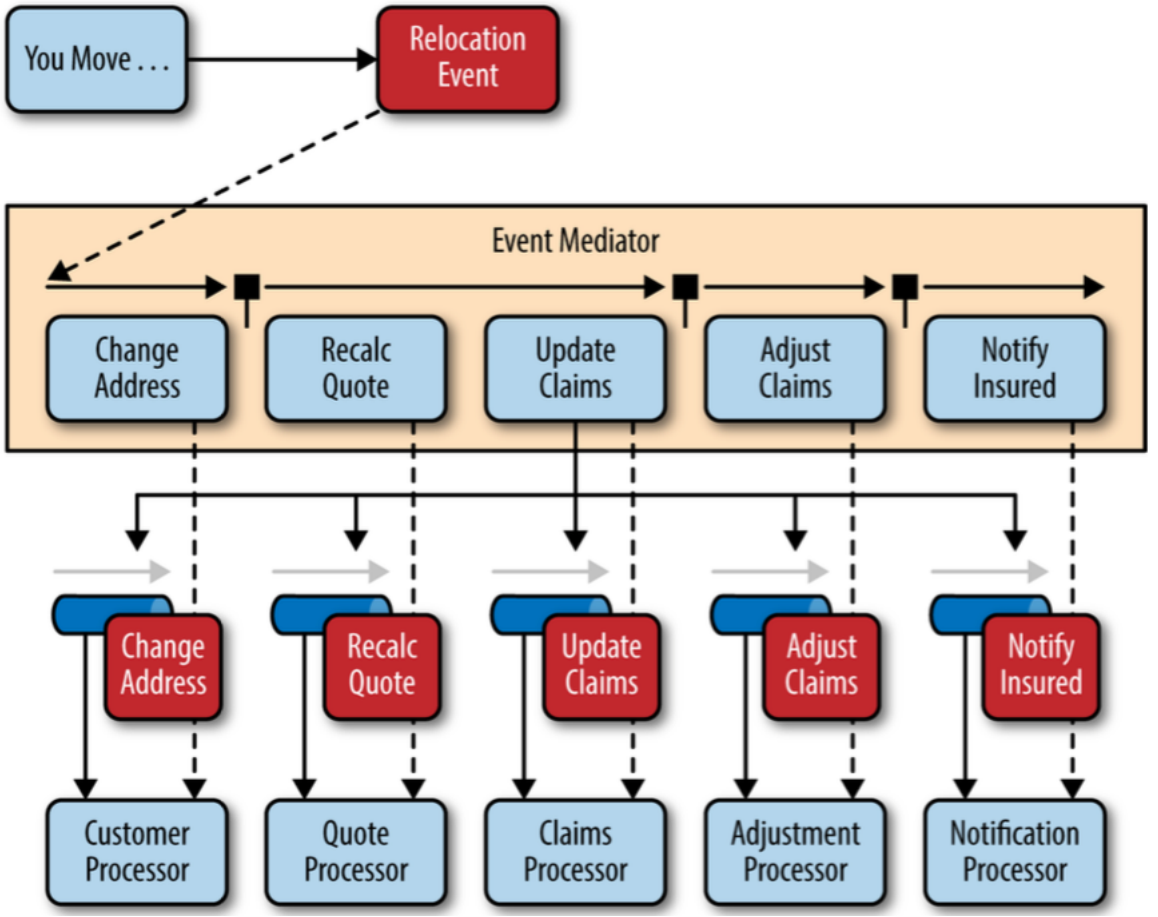
- Mikropalveluiden käyttö mahdollistaa sen, että sovellus voidaan helposti koodata ”monella kielellä”, toisin kuin monoliittisissa projekteissa, mikään ei edellytä, että kaikki mikropalvelut olisi toteutettu samalla kielellä
- Sovelluksen jakaminen järkeviin mikropalveluihin on haastavaa
- Kymmenistä tai jopa sadoista mikropalveluista koostuvan ohjelmiston operoiminen eli ”käynnistäminen” tuotantopalvelimilla on haastavaa ja vaatii pitkälle menevää automatisointia
  - Sama koskee sovelluskehitysympäristöä ja jatkuvaa integraatiota
- Mikropalveluiden yhteydessä käytetäänkin paljon ns *kontainereja* eli käytännössä dockeria
  - Kontainerit ovat hieman yksinkertaistaen sanottuna kevyitä virtuaalikoneita, joita voi suorittaa yhdellä palvelimella suuren määrän rinnakkain
  - Jos mikropalvelu on omassa kontainerissaan, vastaa se käytännössä tilannetta, jossa mikropalvelua suoritettaisiin omalla koneellaan
  - Aihe on tärkeä, mutta emme valitettavasti voi mennä siihen tämän kurssin puitteissa ollenkaan...

Mikropalveluiden kommunikointi

- Mikropalvelut kommunikoivat keskenään verkon välityksellä
- Kommunikointimekanismeja on useita. Yksinkertainen vaihtoehto on käyttää kommunikointiin HTTP-protokollaa, eli samaa mekanismia, jonka avulla web-selaimet keskustelevat palvelimien kanssa
  - Tällöin puhutaan usein että mikropalvelut tarjoavat kommunikointia varten REST-rajapinnan
  - Viikon 2 laskareissa haettiin suorituksiin liittyvää dataa palautusjärjestelmän tarjoamasta REST-rajapinnasta
- Vaihtoehtoinen, huomattavasti joustavampi kommunikointikeino on ns. viestikanavien käyttö, tällöin voi ajatella, että mikropalveluita on höystetty *event-driven*-arkkitehtuurilla
- Tällöin verkkoon käynnistetään eräänlainen viestinvälityspalvelu, johon muut palvelut voivat lähettää tai **julkaista** (publish) viestejä
  - Viesteillä on tyypillisesti joku aihe (topic) ja sen lisäksi datasisältö
  - Esim: topic: new_user, data: (username: Arto Hellas, age: 31, education: PhD)
- Palvelut voivat **tilata** (subscribe) viestipalvelulta viestit joista ne ovat kiinnostuneita
  - Esim. käyttäjähallinnasta vastaava palvelu tilaa viestit joiden aihe on *new_user*
- Viestinvälitysjärjestelmä välittää vastaanottamansa viestit kaikille, jotka ovat aiheen tilanneet

Mikropalveluiden kommunikointi viestien välityksellä

- Kaikki viestien (tai joskus puhutaan myös tapahtumista, *event*) välitys tapahtuu viestinvälityspalvelun (kuvassa *event mediator*) kautta
- Näin mikropalveluista tulee erittäin löyhästi kytkettyjä, ja muutokset yhdessä palvelussa eivät vaikuta mihinkään muualle, niin kauan kuin viestit säilyvät entisen muotoisina



- Viestinvälitykseen perustuvat mikropalvelut eivät ole ilmainen lounas, erityisesti debuggaus voi olla välillä melko haastavaa