

Ohjelmistotuotanto

Luento 7

11.4.

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 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

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

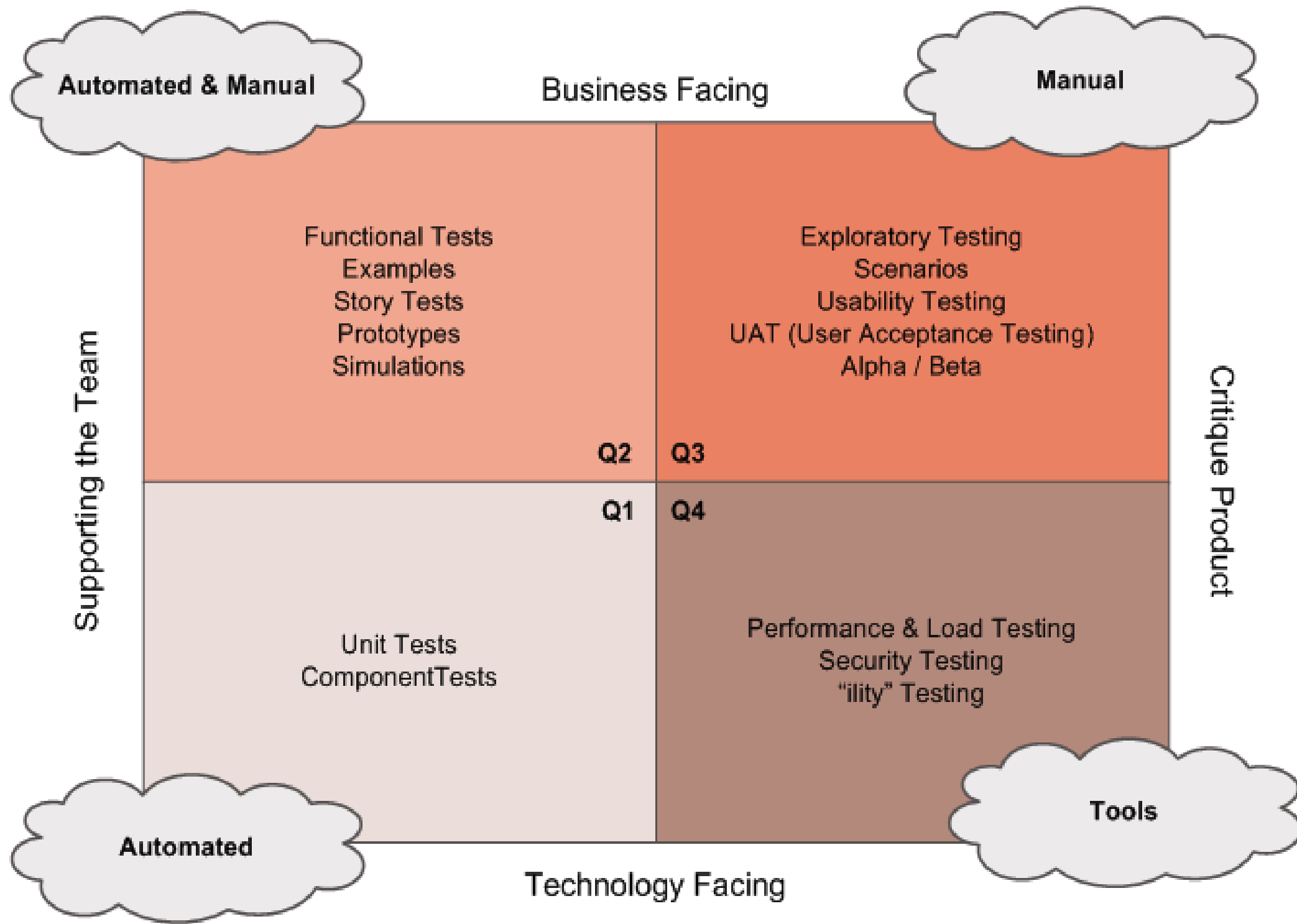
Tutkiva testaaminen

- Tutkivassa testauksessa keskeistä on kaiken järjestelmän tekemien asioiden havainnointi
 - Normaaleissa etukäteen määritellyissä testeissä 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
 - 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



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 luenolla 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 monimutkaista 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 ohjelmistotuotantomenetelmien 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 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 architect
- 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

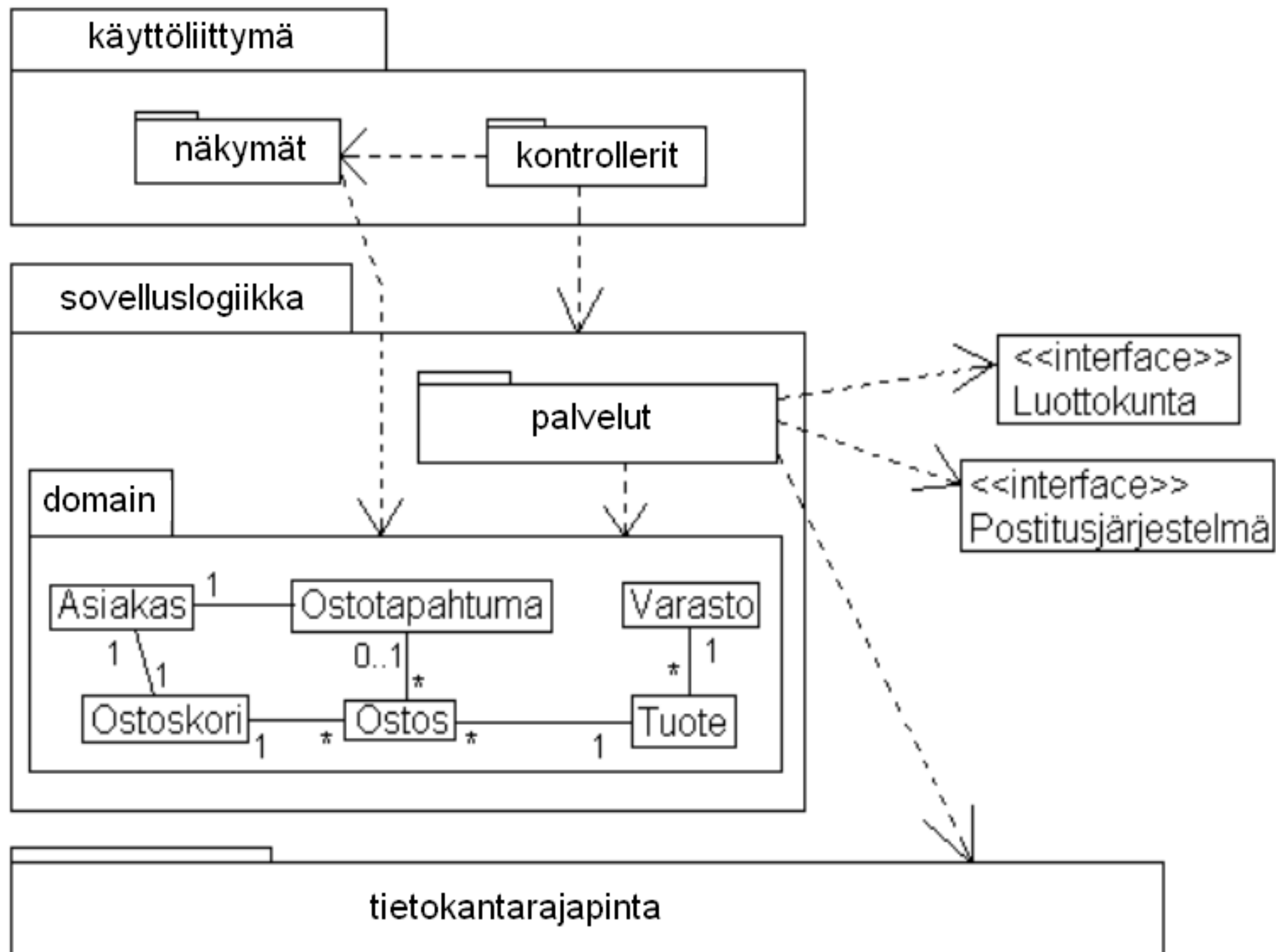
- Vaikka arkkitehtuurin määritelmät hieman vaihtelevat, löytyy määritelmistä joukko samoja teemoja
- Lähes jokaisen määritelmän mukaan arkkitehtuuri määrittelee ohjelmiston rakenteen, eli jakautumisen erillisiin osiin ja osien väliset rajapinnat
- Arkkitehtuuri ottaa kantaa rakenteen lisäksi myös käyttäytymiseen
 - Arkkitehtuurityöskentelyn rakennepuoleen 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/>

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 muokkavat arkkitehtuuria
 - Organisaation standardit
 - Integraatio olemassaoleviin järjestelmiin
 - Toteutuksessa käytettävät sovelluskehyykset

Arkkitehtuurimalli

- Järjestelmän arkkitehtuuri perustuu yleensä yhteen tai useampaan **arkkitehtuurimalliin** (architectural pattern), jolla tarkoitetaan hyväksi havaittua tapaa strukturoida tietyn tyyppisiä sovelluksia
 - Samasta asiasta käytetään joskus nimitystä **arkkitehturityyli** (architectural style)
- arkkitehtuurimalleja:
 - Kerrosarkkitehtuuri, MVC, Pipes-and-filters, Repository, Client-server, publish-subscribe, event driven, REST, Microservice, SOA...
- Useimmiten sovelluksen rakenteesta löytyy monien arkkitehtuuristen mallien piirteitä
- Lisää tietoa eri arkkitehtuurimalleista internetissä, esim. http://en.wikipedia.org/wiki/Architectural_pattern ja syventäviin opintoihin kuuluvalla kurssilla *Ohjelmistoarkkitehtuurit*
- Seuraavalla sivulla kuvaus Kumpulabiershopin arkkitehtuurista, joka on mukaelma kerrosarkkitehtuuria ja MVC-mallia
 - Kuvaus on UML-pakkauskaaviona, näyttäen myös yhden pakkauksen sisältävät luokat
 - Luokkatasolle ei yleensä arkkitehtuurikuvauksissa mennä

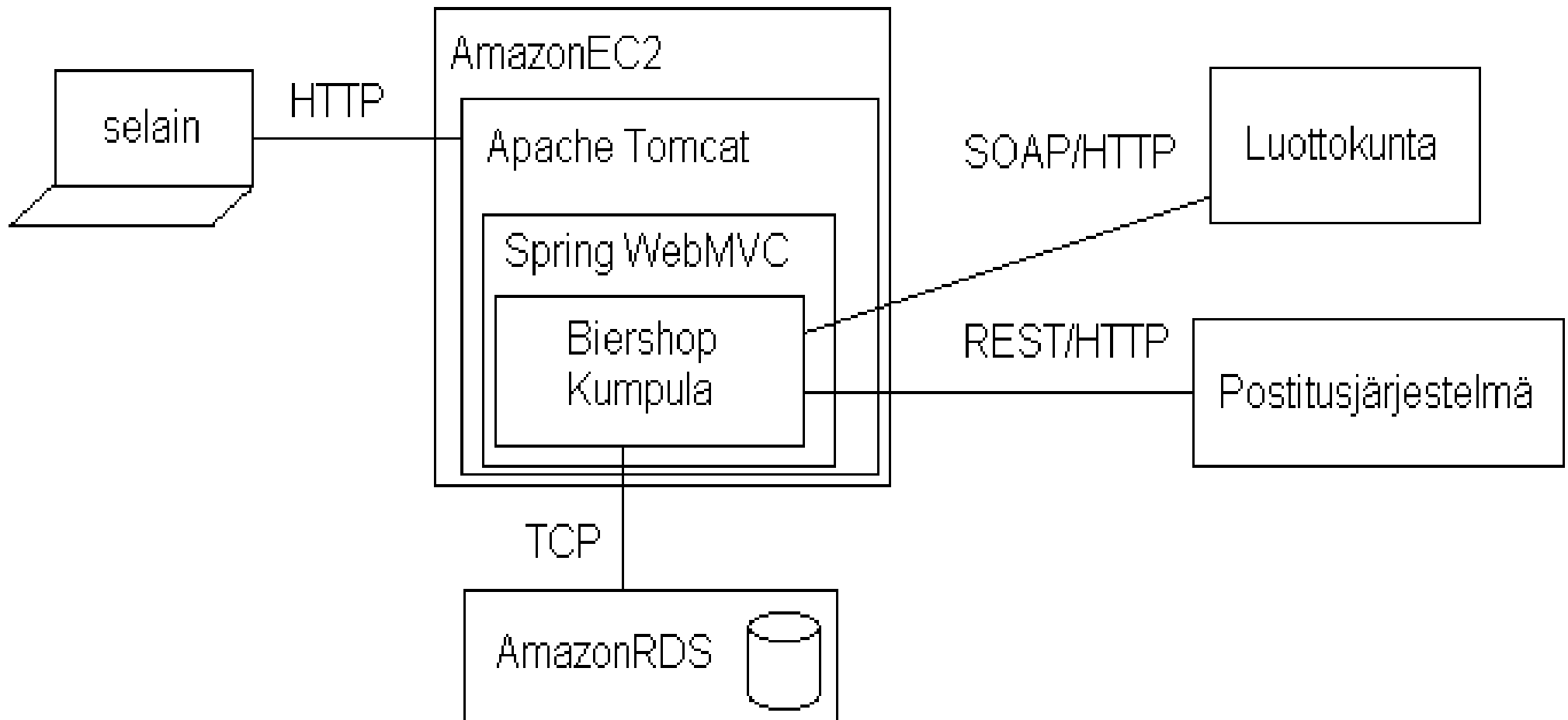


Kumpula biershopin arkkitehtuuri

- Arkkitehtuurikuvaus näyttää järjestelmän jakaantumisen kolmeen kerroksittain järjestettyyn komponenttiin
 - Käyttöliittymä
 - Sovelluslogiikka
 - Tietokantarajapinta
- Ohjelmiston arkkitehtuuri noudattaa **kerrosarkkitehtuurimallia**
 - Kerros on kokoelma toisiinsa liittyviä olioita tai alikomponentteja, jotka muodostavat toiminnallisuuden suhteen loogisen kokonaisuuden
 - Kerrosarkkitehtuurissa pyrkimyksenä järjestellä komponentit siten, että ylempänä oleva kerros käyttää ainoastaan alempana olevien kerroksien tarjoamia palveluita
- Sovelluslogiikkakerros on jaettu vielä kahteen alikomponenttiin, sovellusalueen käsitteistön sisältävään domainiin ja sen olioita käyttäviin sekä tietokantarajapinnan kanssa keskusteleviin palveluihin
- 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

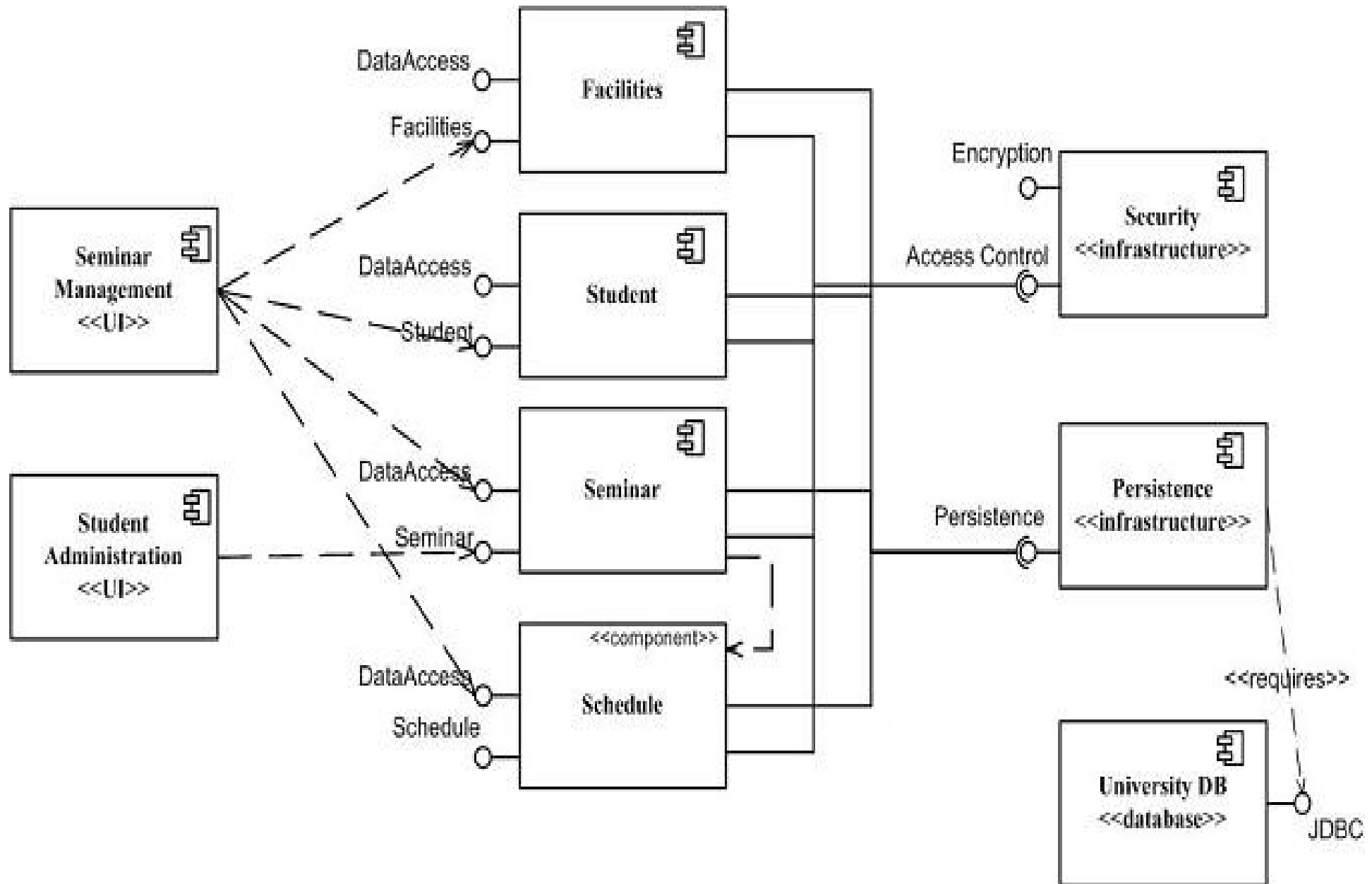


Arkkitehtuurin kuvaamisesta

- UML:n lisäksi arkkitehtuurikuvauksille ei ole vakiintunutta formaattia
 - Luokka ja pakkauskaavioiden lisäksi UML:n komponentti- ja sijoittelukaaviot voivat olla käyttökelpoisia (ks. seuraavat kalvot)
 - Usein 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

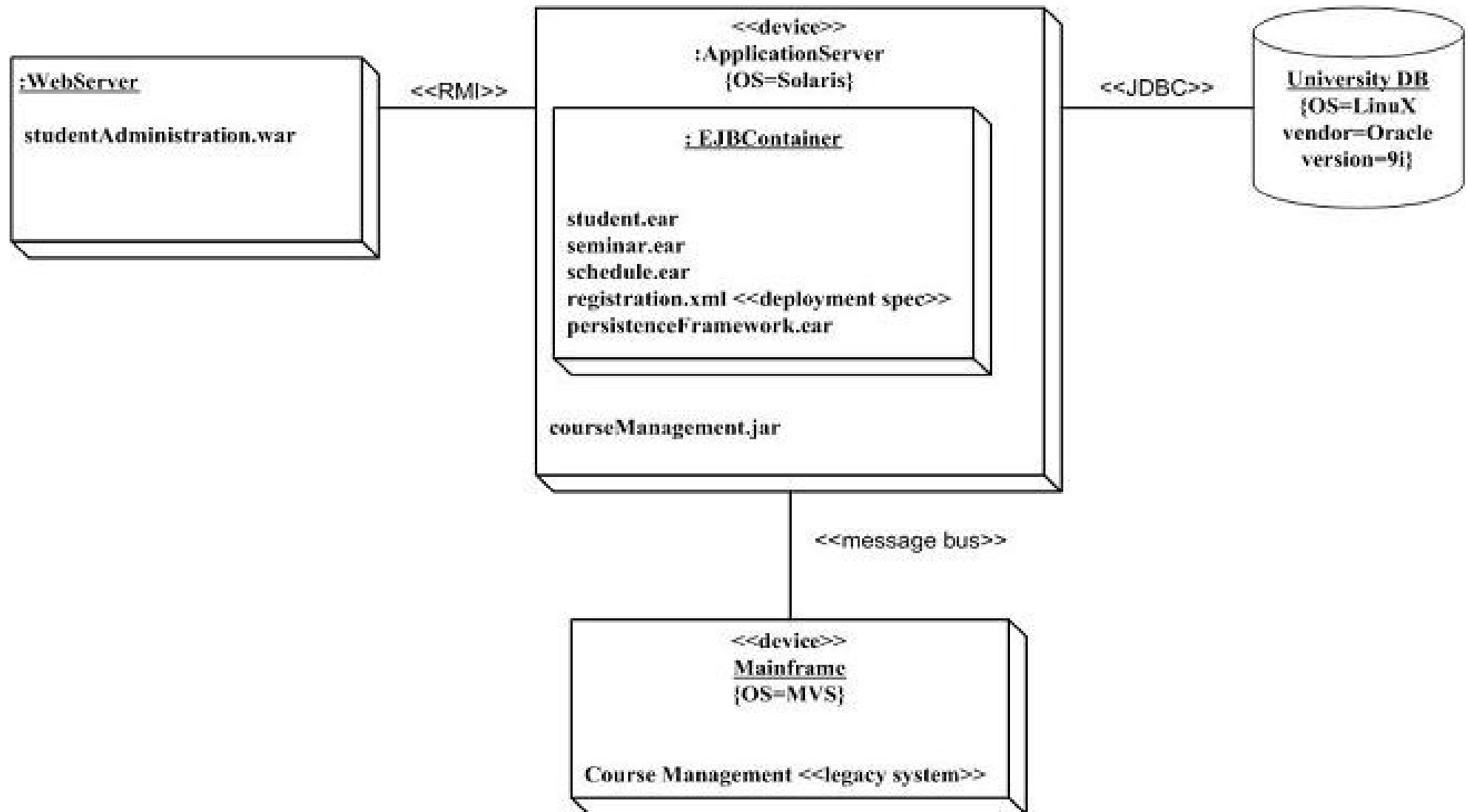
UML komponenttikaavio

- <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

Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen (agile manifestin periaatteita):
 - Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
 - Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Ketterät menetelmät suosivat yksinkertaisuutta suunnitteluratkaisuissa
 - Simplicity--the art of maximizing the amount of work not done--is essential.
 - YAGNI eli "you are not going to need it"-periaate
- Arkkitehtuuriin suunnittelu ja dokumentointi on perinteisesti ollut melko pitkäkestoinen, ohjelmoinnin aloittamista edeltävä vaihe
 - BUFD eli Big Up Front Design
- Ketterät menetelmät ja "arkkitehtuurivetoinen" ohjelmistotuotanto ovat siis jossain määrin keskenään ristiriidassa

Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien yhteydessä puhutaan *inkrementaalisesta tai evolutiivisesta suunnittelusta ja arkkitehtuurista*
- Arkkitehtuuri mietitään riittävällä tasolla projektin alussa
- Jotkut projektit alkavat ns. nollasprintillä ja alustava arkkitehtuuri määritellään tällöin
 - Scrumin varhaisissa artikkeleissa puhuttiin "pre game"-vaiheesta jolloin mm. alustava arkkitehtuuri luodaan
 - Sittemmin koko käsite on hävinnyt Scrumista ja Ken Schwaber (Scrumin kehittäjä) jopa eksplisiittisesti kieltää ja tyrmää koko "nollasprintin" olemassaolon: <http://www.scrum.org/assessmentdiscussion/post/1317787>
- Ohjelmiston "lopullinen" arkkitehtuuri muodostuu iteraatio iteraatiolta samalla kun ohjelmaan toteutetaan uutta toiminnallisuutta
 - Esim. kerrosarkkitehtuurin mukaista sovellusta ei rakenneta "kerros kerrallaan"
 - Jokaisessa iteraatiossa tehdään pieni pala jokaista kerrosta, sen verran kuin iteraation toiminnallisuuksien toteuttaminen edellyttää
 - <http://msdn.microsoft.com/en-us/architecture/ff476940>

Arkkitehtuuri ketterissä menetelmissä

- Perinteisesti arkkitehtuurista on vastannut ohjelmistoarkkitehti ja ohjelmoijat ovat olleet velvoitettuja noudattamaan arkkitehtuuria
- Ketterissä menetelmissä ei suosita erillistä arkkitehdin roolia, esim. Scrum käyttää kaikista ryhmän jäsenistä nimikettä *developer*
- Ketterien menetelmien ideaali on, että kehitystiimi luo arkkitehtuurin yhdessä, tämä on myös yksi agile manifestin periaatteista:
 - The best architectures, requirements, and designs emerge from self-organizing teams.
- Arkkitehtuuri on siis koodin tapaan tiimin *yhteisomistama*, tästä on muutamia etuja
 - Kehittäjät sitoutuvat paremmin arkkitehtuurin noudattamiseen kuin ”norsunluutornissa” olevan tiimin ulkopuolisen arkkitehdin määrittelemään arkkitehtuuriin
 - Arkkitehtuurin dokumentointi voi olla kevyt ja informaali (esim. valkotalulle piirretty) sillä tiimi tuntee joka tapauksessa arkkitehtuurin hengen ja pystyy sitä noudattamaan

Inkrementaalinen arkkitehtuuri

- Ketterissä menetelmissä oletuksena on, että parasta mahdollista arkkitehtuuria ei pystytä suunnittelemaan projektin alussa, kun vaatimuksia, toimintaympäristöä ja toteutusteknologioita ei vielä tunneta
 - Jo tehtyjä arkkitehtoonisia ratkaisuja muutetaan tarvittaessa
- Eli kuten vaatimusmäärittelyn suhteen, myös arkkitehtuurin suunnittelussa ketterät menetelmät pyrkii välttämään liian aikaisin tehtävää ja myöhemmin todennäköisesti turhaksi osoittautuvaa työtä
- Inkrementaalinen lähestymistapa arkkitehtuurin muodostamiseen edellyttää koodilta hyvää laatua ja toteuttajilta kurinalaisuutta
- Martin Fowler <http://martinfowler.com/articles/designDead.html>:
 - Essentially evolutionary design means that the design of the system grows as the system is implemented. Design is part of the programming processes and as the program evolves the design changes.
 - **In its common usage, evolutionary design is a disaster.** The design ends up being the aggregation of a bunch of ad-hoc tactical decisions, each of which makes the code harder to alter
- Seuraavaksi siirrymme käsittelemään oliosuunnittelua