# Ohjelmistotuotanto

## Luento 7

16.4.2012

Testaus ketterissä menetelmissä, jatkuu..

# 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 kahdesta 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. TDD:n sivutuotteena syntyy toki kattava joukko testejä
- Tänään tarkastelemme vielä kahta ketterää testauksen käytännettä:
  - **Continuous Integration (CI)** suomeksi jatkuva integraatio
  - **Exploratory testing**, suomeksi tutkiva testaus

# Pois integraatiohelvetistä

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

# Jatkuva integraatio – Continuous Integration

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

# Jatkuva integraatio – Continuous Integration

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

- Kun kehittäjän omalla koneella kaikki testit menevät läpi, eli 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
- Tämän hetken suosituin CI-palvelinohjelmisto on jo ensimmäiseltä viikolta tuttu Jenkins
- Tarkoituksena on, että jokainen kehittäjä integroi tekemänsä työn muuhun koodiin mahdollisimman usein, *vähintään* kerran päivässä
    - CI siis rohkaisee jakamaan työn pieniin osiin, sellaisiin jotka saadaan testeineen ”valmiiksi” yhden työpäivän aikana
    - CI-työprosessin noudattaminen vaatii kurinalaisuutta
- Jotta CI-prosessi toimisi joustavasti, tulee testien ajamisen tapahtua suhteellisen nopeasti, maagisena rajana pidetään usein kymmentä minuuttia
- Jos osa testeistä on hitaita, voidaan testit konfiguroida ajettavaksi kahdessa vaiheessa
    - *commit build*:in läpimeno antaa kehittäjälle oikeuden pushata koodi repositorioon
    - CI-palvelimella suoritetaan myös hitaammat testit sisältävä *secondary build*

# Seuraava askel jatkuvasta integraatiosta

- Usein CI:tä viedään vielä askel pidemmälle ja integraatioprosessiin lisätään myös automaattinen ”deployaus”
  - käännetty koodi paketoidaan ja siirretään suoritettavaksi testipalvelimelle
- Jatkuvaan integraatioon liittyvä koodin automaattinen deployaaminen testiympäristöön on monin tavoin hyödyllinen käytäntö
  - Ohjelmaa on mahdollista demota asiakkaalle milloin vaan
  - Testipalvelimella voidaan järjestelmälle ajaa esim. pitkäkestoisia kuormitustestejä
  - Testaajien on mahdollista tehdä tutkivaa testausta (tästä tarkemmin pian) koko ajan uusimmalle järjestelmän versiolle
- Viime aikoina on ruvettu suosimaan tyyliä jossa web-palveluna toteutettu ohjelmisto julkaistaan tuotantoon jopa useita kertoja päivästä
- Tästä tavasta käytetään nimitystä *continuous deployment* tai *continuous delivery*
  - http://timothyfitz.wordpress.com/2009/02/08/continuous-deployment/
  - http://sna-projects.com/blog/2011/06/continuous-deployment-flickr/

# 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 harkittuja tahansa, 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 User Storyn tai muutaman 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 vain 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 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

# 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 seuraavaksi 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 vaatimismäärittelyn jälkeen ja ohjelmointi aloitetaan vasta kun suunnittelu valmiina ja dokumentoitu
  - Ketterissä menetelmissä suunnittelua tapahtuu 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 mitä ohjelmistoarkkitehtuurilla 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

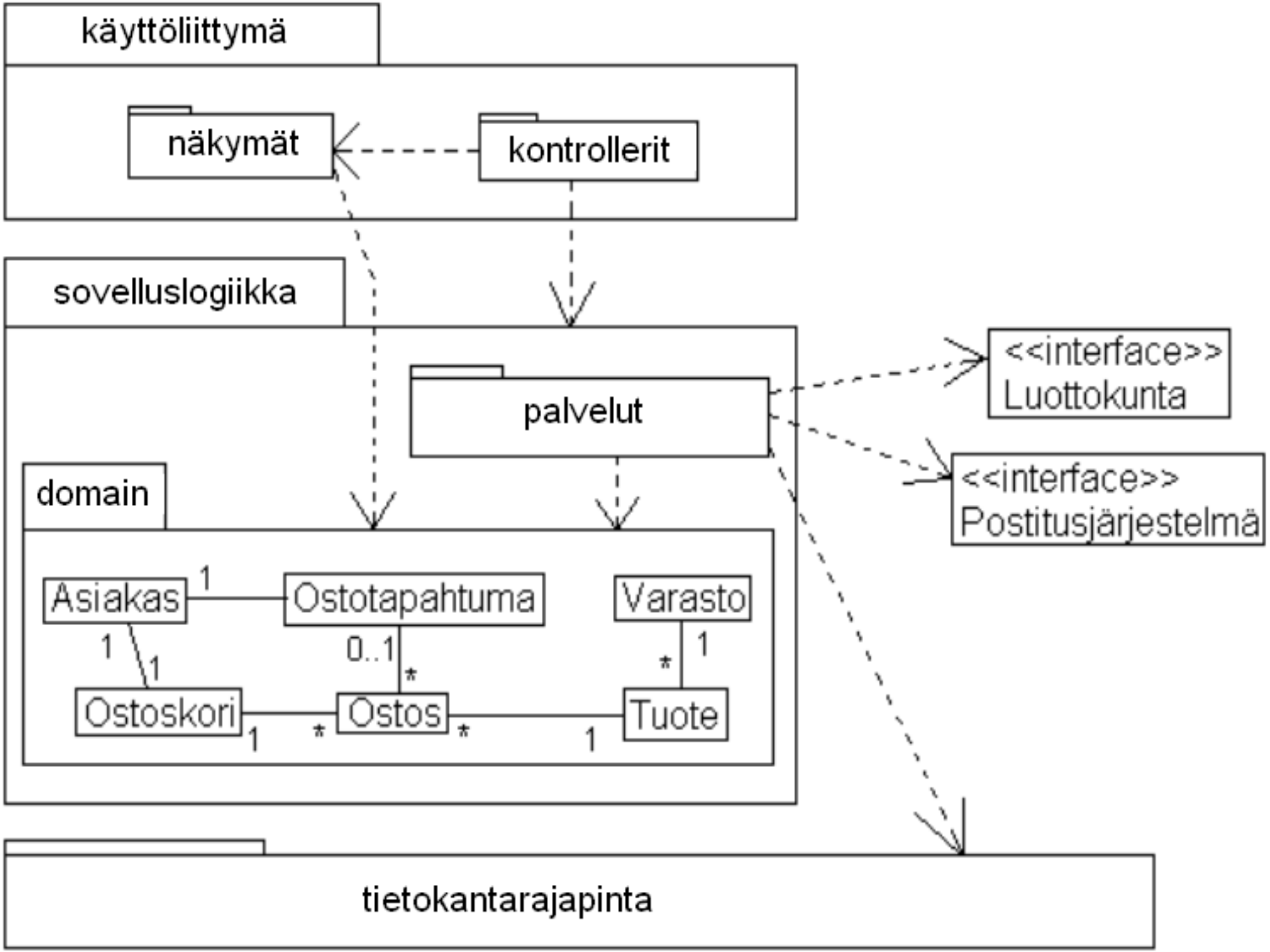
- Vaikka arkkitehtuurin määritelmät hieman vaihtelevat, löytyy määritelmistä joukko samoja teemoja
- Lähes jokaisen määritelmän mukaan arkkitehtuuri määrittelee ohjelmiston rakenteen, eli jakautumisen erillisiin osiin ja komponenttien osien rajapinnat
- Arkkitehtuuri ottaa kantaa rakenteen lisäksi myös käyttäytymiseen
  - Arkkitehtuuritason rakenneosien vastuut ja niiden keskinäisen kommunikoinnin muodot
- Arkkitehtuuri keskittyy järjestelmän tärkeisiin/keskeisiin osiin
  - Tärkeitä osia voivat olla esim. järjestelmän hajautukseen, skaalautuvuuteen tai tietoturvaan liittyvät osat
  - 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. -illities) on suuri vaikutus arkkitehtuuriin
  - Käytettävyys, suorituskyky, skaalautuvuus, vikasietoisuus, 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
- Myös järjestelmän toimintaympäristö muokkaa arkkitehtuuria
  - Organisaation standardit
  - Integraatio olemassaoleviin järjestelmiin
- Järjestelmän arkkitehtuuri perustuu yleensä johonkin arkkitehtuurimalliin
- **Arkkitehtuurimallilla** (architectural pattern) tarkoitetaan hyväksi havaittua tapaa strukturoida tietyntyyppisiä sovelluksia
  - arkkitehtuurimalleja ovat esim: kerrosarkkitehtuuri, MVC, pipes-and-filters, repository, client-server, SOA

# Kaikilla ohjelmilla on arkkitehtuuri

- Jokaisella ohjelmistolla on arkkitehtuuri riippumatta siitä onko arkkitehtuuria suunniteltu tai dokumentoitu tai ollaanko siitä tietoisia
- Valitettavan yleinen arkkitehtuurinen ratkaisu on ns. ”big ball of mud”
  - While much attention has been focused on high-level software architectural patterns, what is, in effect, the de-facto standard software architecture is seldom discussed. This paper examines this most frequently deployed of software architectures: the Big Ball of Mud...
  - Brian Foote and Joseph Yonderin vuonna 1999 kirjoittamasta artikkelista Big Ball of Mud
  - http://www.laputan.org/mud/mud.html
  - http://www.infoq.com/news/2010/09/big-ball-of-mud
- Seuraavalla sivulla kuvaus Kumpulabiershopin arkkitehtuurista
  - Kuvaus on UML-pakkauskaaviona, näyttäen myös yhden pakkauksen sisältävät luokat
  - Luokkatasolle ei yleensä arkkitehtuurikuvauksissa mennä

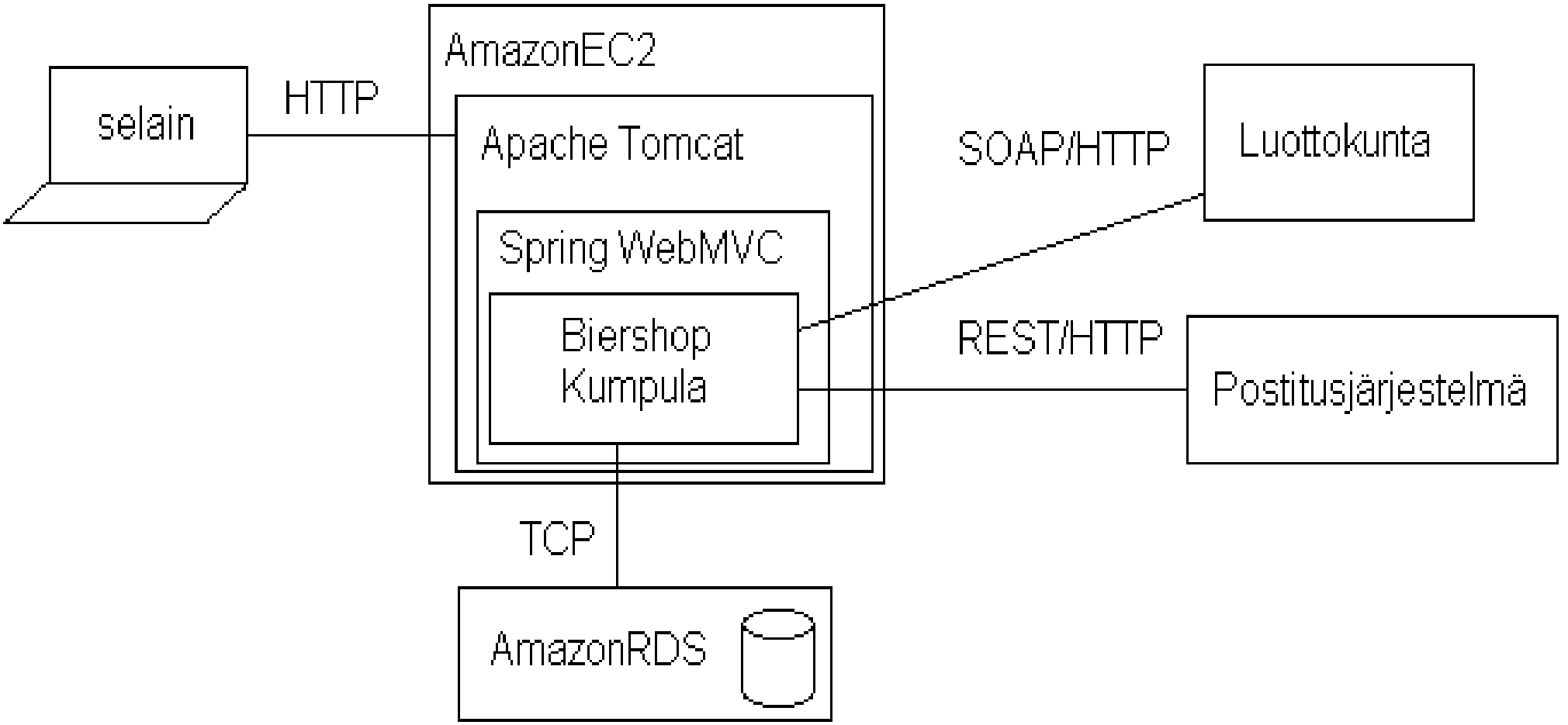

käyttöliittymä
näkymät
kontrollerit
sovelluslogiikka
palvelut
<<interface>>
Luottokunta
<<interface>>
Postitusjärjestelmä
domain
Asiakas
Ostotapahtuma
Varasto
Ostoskori
Ostos
Tuote
1
1
1
1
0..1
*
*
1
*
*
1
tietokantarajapinta

# Kumpula biershopin arkkitehtuuri

- Arkkitehtuurikuvaus näyttää järjestelmän jakaantumisen kolmeen kerroksittain järjestettyyn komponenttiin
  - Käyttöliittymä
  - Sovelluslogiikka
  - Tietokantarajapinta
- 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

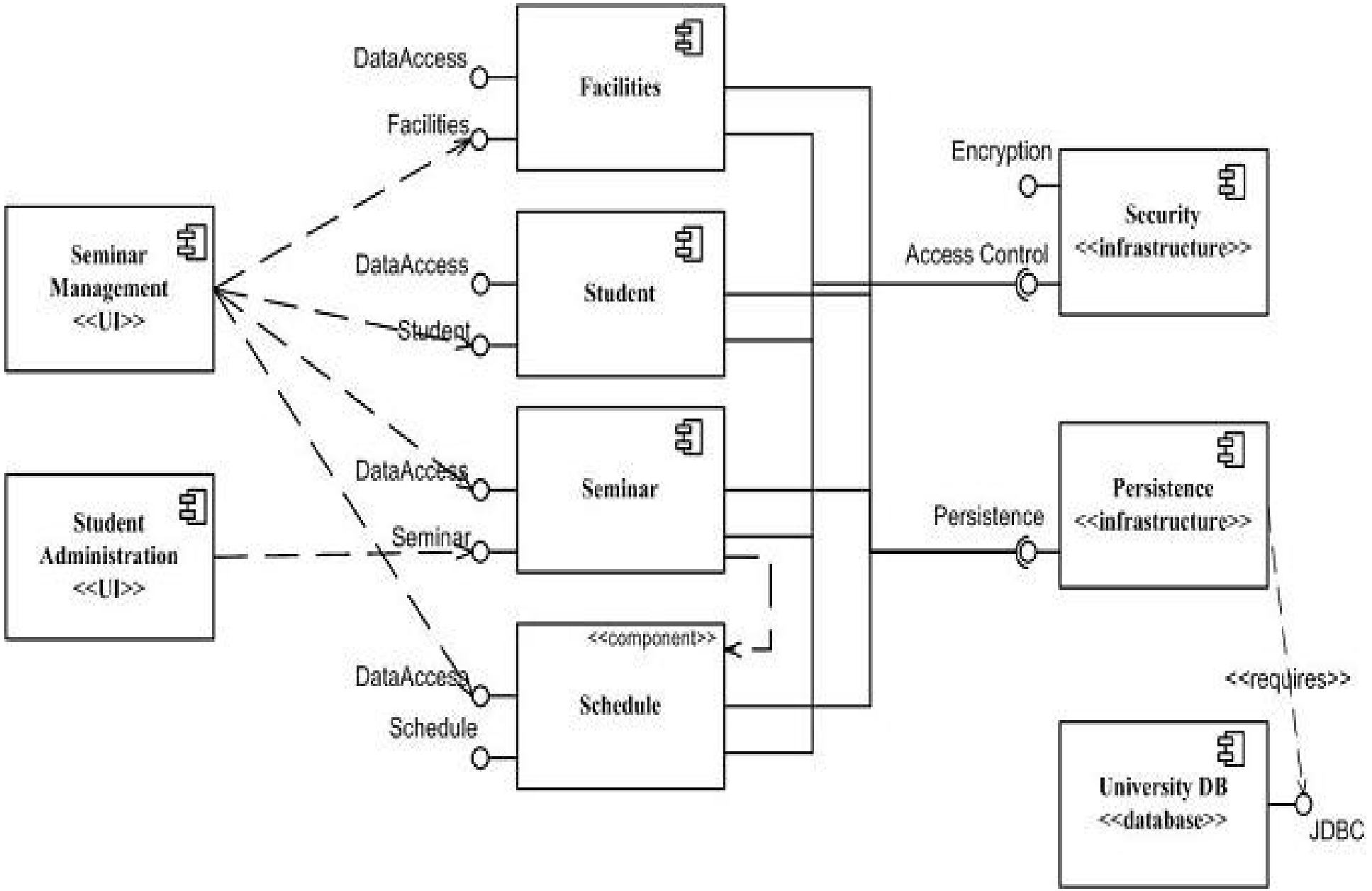


# Arkkitehtuurin kuvaamisesta

- 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

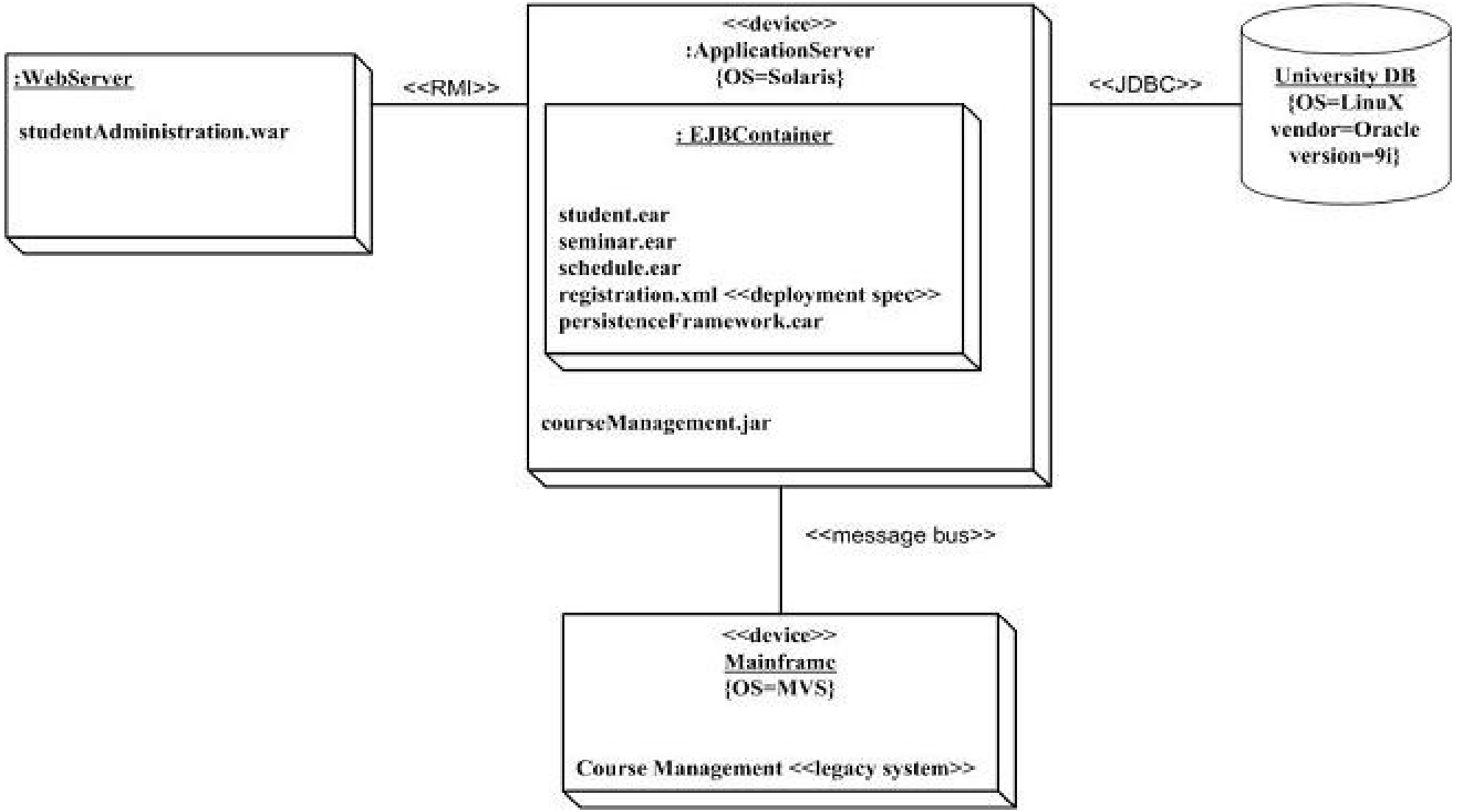
# UML komponenttikaavio

- 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 periaattita):
  - 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 apreference to the shorter timescale.
- Ketterät menetelmät suosivat yksinkertaisuutta suunnitteluratkaisuissa
  - Simplicity--the art of maximizing the amountof work not done--is essential.
  - YAGNI ei you are not going to need it -periaate
- Arkkitehtuuriin suunnittelu ja dokumentointi on perinteisesti ollut melko pitkäkestoinen vaihe joka on edeltänyt ohjelmoinnin aloittamista
  - 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 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. valkotaululle 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