Ohjelmistotuotanto

Luento 7

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 kolmesta 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
  - **Continuous Integration (CI)** suomeksi jatkuva integraatio
- Tänään tarkastelemme vielä yhtä ketterää testauksen käytännettä:
  - **Exploratory testing**, suomeksi tutkiva testaus

Tutkiva testaaminen

- Jotta järjestelmä saadaan niin virheettömäksi, että se voidaan ottaa tuotantokäyttöön, on testaus tehtävä erittäin perusteellisesti
- 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 testausdokumentissa 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 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/

Arkkitehtuuriin vaikuttavia tekijöitä

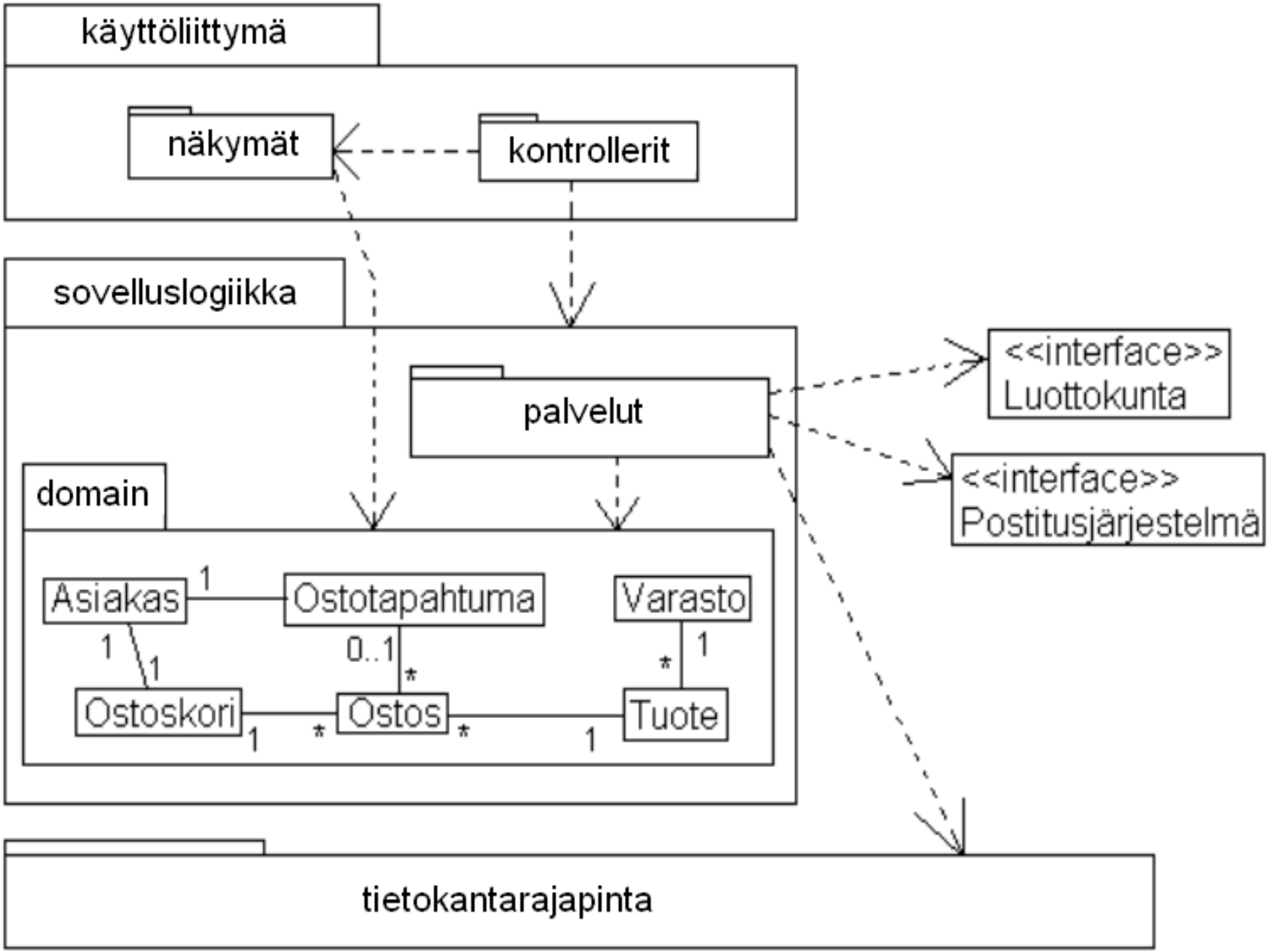
- Järjestelmälle asetetuilla ei-toiminnallisilla laatuvaatimuksilla (engl. -ilities) on suuri vaikutus arkkitehtuuriin
  - Käytettävyys, suorituskyky, skaalautuvuus, vikasietoisuus, tiedon ajantasaisuus, tietoturva, ylläpidettävyys, laajennettavuus, hinta, time-to-market, ...
- Laatuvaatimukset ovat 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 sovelluskehykset

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

Arkkitehtuurimalli

- Järjestelmän arkkitehtuuri perustuu yleensä yhteen tai useampaan **arkkitehtuurimalliin** (architectural pattern), jolla tarkoitetaan hyväksi havaittua tapaa strukturoida tietyntyyppisiä sovelluksia
- arkkitehtuurimalleja:
  - Kerrosarkkitehtuuri, MVC, Pipes-and-filters, Repository, Client-server, publish-subscribe, event driven, SOA...
- Useimmiten sovelluksen rakenteesta löytyy monen arkkitehtuuristen mallin 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

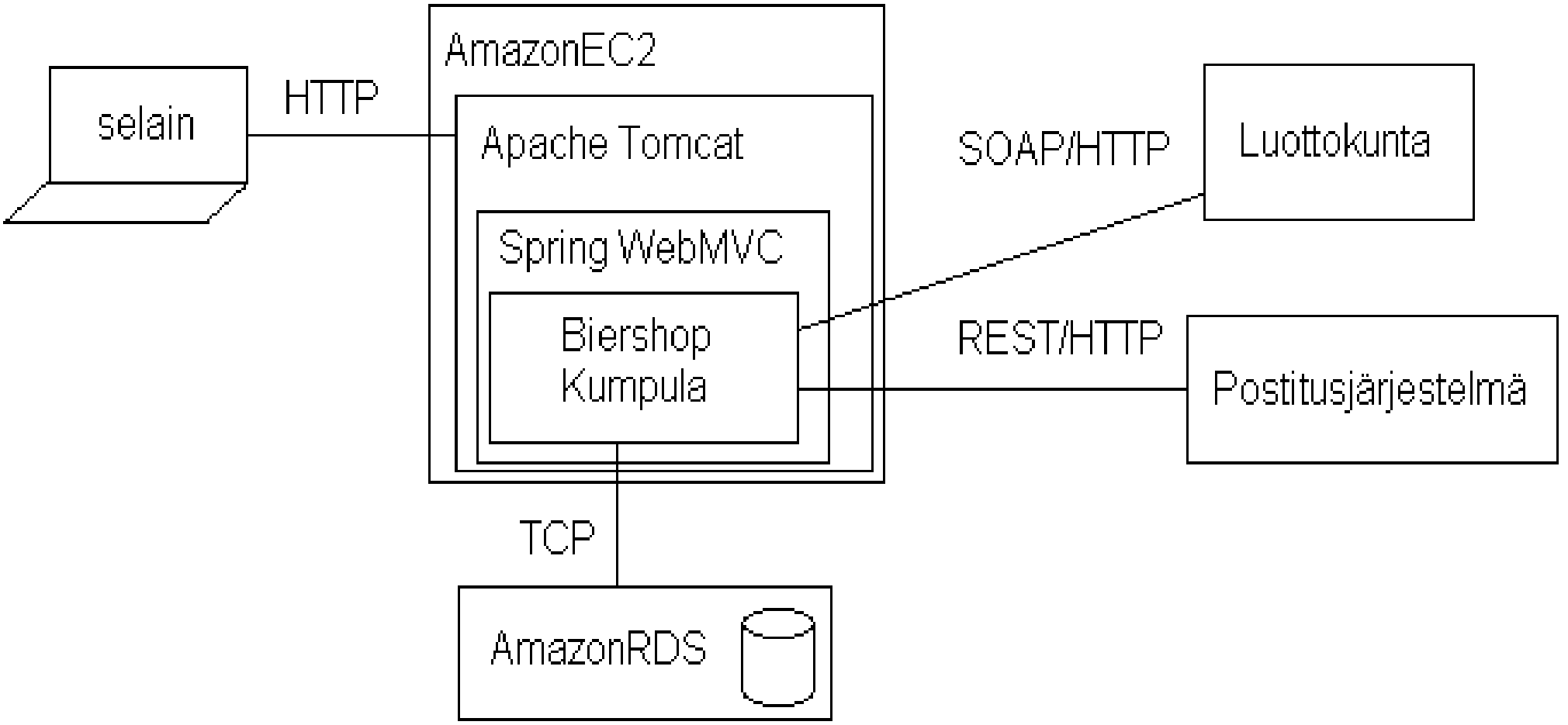

käyttöliittymä
näkymät
kontrollerit
sovelluslogiikka
palvelut
<<interface>>
Luottokunta
<<interface>>
Postitusjärjestelmä
domain
Asiakas
Ostotapahtuma
Varasto
Ostoskori
Ostos
Tuote
1
1
1
1
0..1
*
*
1
*
*
1
tietokantarajapinta

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

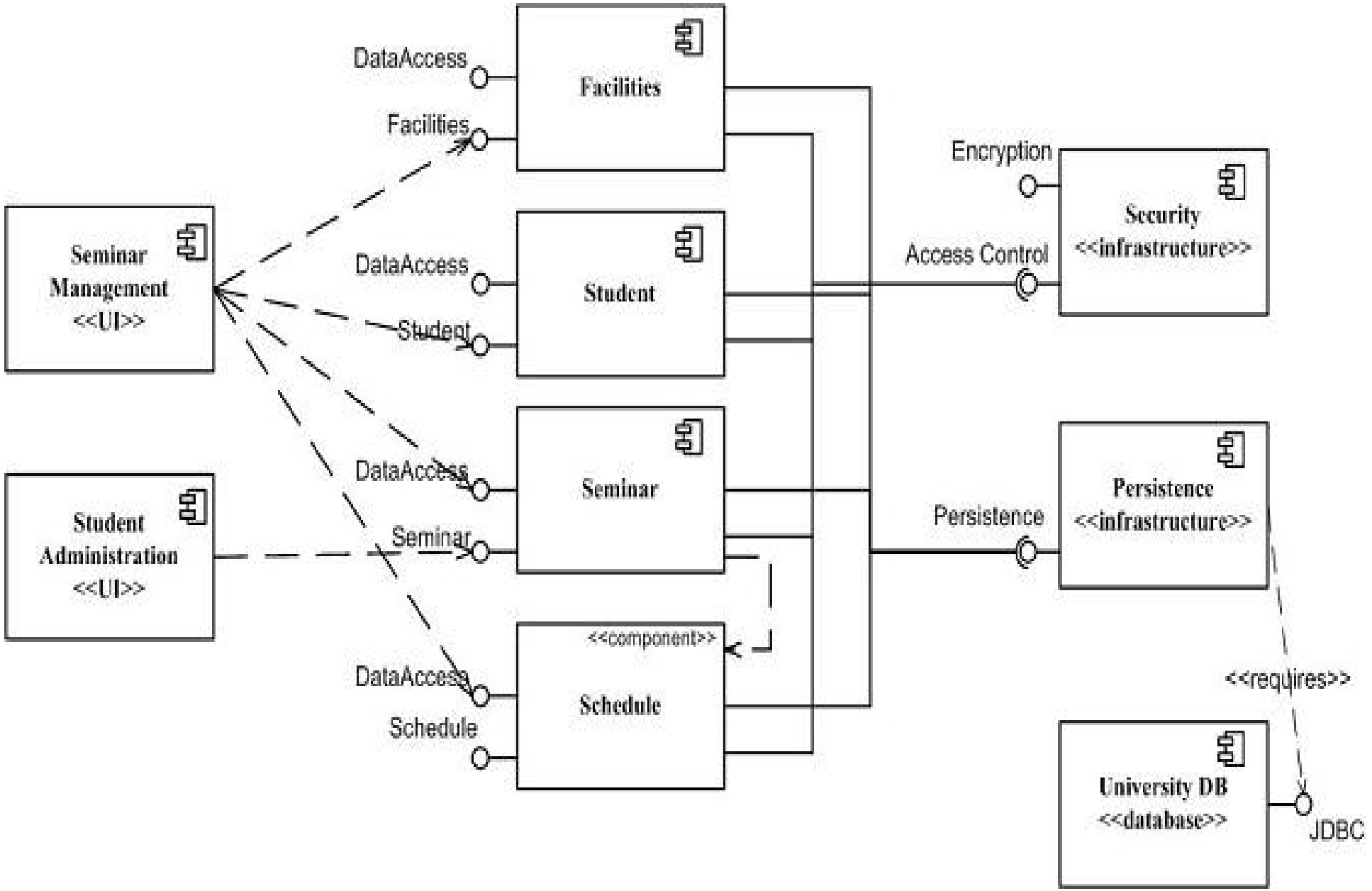


Arkkitehtuurin kuvaamisesta

- 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

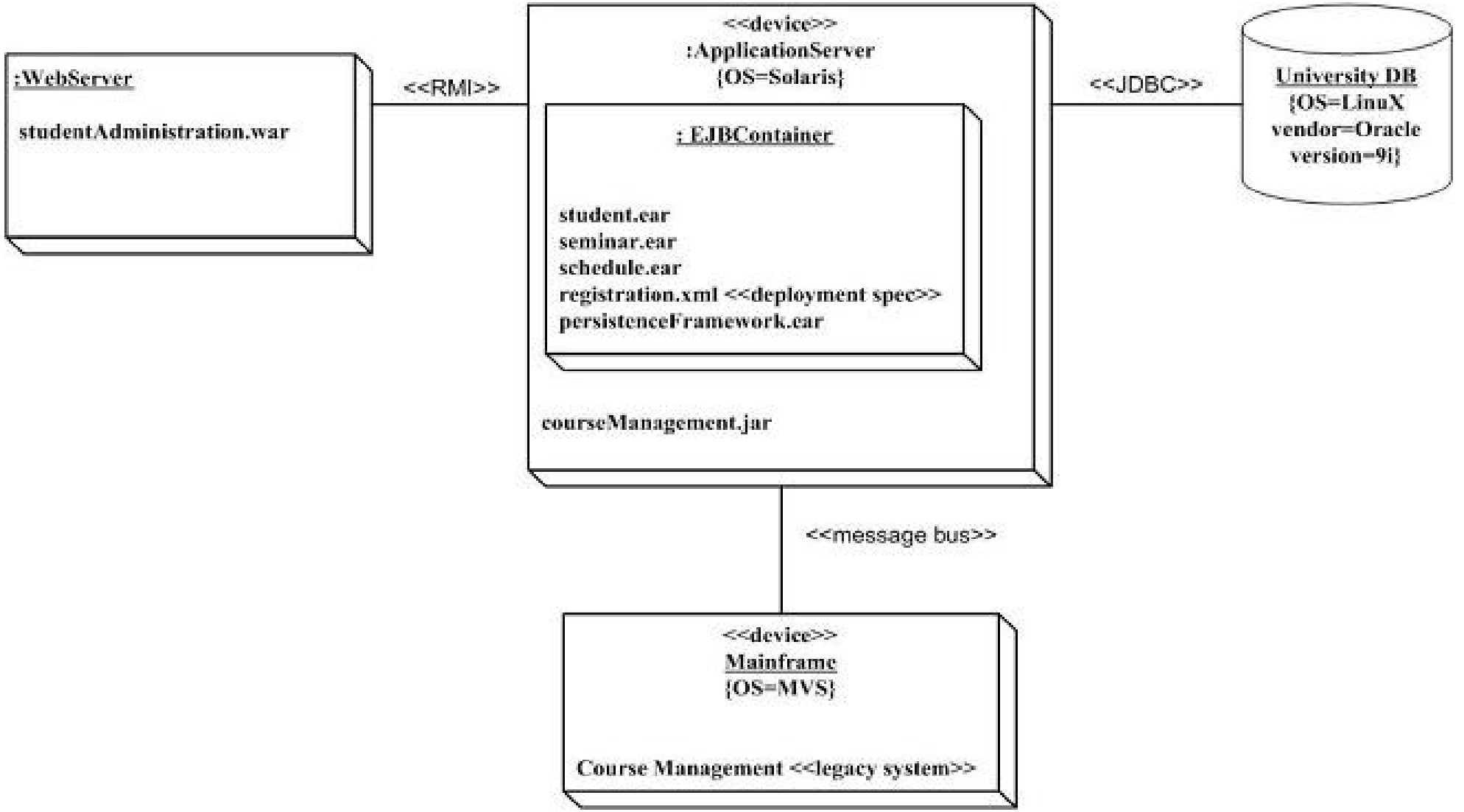
UML komponenttikaavio

- 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 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 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

Oliosuunnittelu

Oliosuunnittelu

- Käytettäessä ohjelmiston toteutukseen olio-ohjelmointikieltä, on suunnitteluvaiheen tarkoituksena löytää sellaiset oliot, jotka pystyvät yhteistoiminnallaan toteuttamaan järjestelmän vaatimuksen
- Oliosuunnittelua ohjaa ohjelmistolle suunniteltu arkkitehtuuri
- Ohjelman ylläpidettävyyden kannalta on suunnittelussa hyvä noudattaa ”ikiaikaisia” hyvän suunnittelun käytänteitä
  - Ketterissä menetelmissä tämä on erityisen tärkeää, sillä jos ohjelman rakenne pääsee rapistumaan, on ohjelmaa vaikea laajentaa jokaisen sprintin aikana
- Ohjelmiston suunnitteluun on olemassa useita erilaisia menetelmiä, mikään niistä ei kuitenkaan ole vakiintunut
  - Kurssilla Ohjelmistojen mallintaminen / Ohjelmistotekniikan menetelmät tutustuttiin ohimennen ns. *vastuupohjaiseen oliosuunnitteluun* (Responsibility driven object design)
- Ohjelmistosuunnittelu onkin ”enemmän taidetta kuin tiedettä”, kokemus ja hyvien käytänteiden opiskelu toki auttaa
- Erityisesti ketterissä menetelmissä tarkka oliosuunnittelu tapahtuu yleensä ohjelmoinnin yhteydessä

Oliosuunnittelu

- Emme keskity tällä kurssilla mihinkään yksittäiseen oliosuunnittelumenetelmään
- Sen sijaan tutustumme muutamiin tärkeisiin menetelmäriippumattomiin teemoihin:
- Laajennettavuuden ja ylläpidettävyyden suhteen laadukkaan koodin/oliosuunnittelun tunnusmerkkeihin ja *laatuattribuutteihin*
  - kapselointi, koheesio, riippuvuuksien vähäisyys, toisteettomuus, selkeys, testattavuus
- ja niitä tukeviin ”ikiaikaisiin” hyvän suunnittelun periaatteisiin
- *Koodinhajuihin* eli merkkeihin siitä että suunnittelussa ei kaikki ole kunnossa
- *Refaktorointiin* eli koodin rajapinnan ennalleen jättävään rakenteen parantamiseen
- Erilaisissa tilanteissa toimiviksi todettuihin geneerisiä suunnitteluratkaisuja dokumentoiviin *suunnittelumalleihin*
  - Olemme jo nähdeen muutamia suunnittelumalleja, ainakin seuraavat: dependency injection, singleton, data access object

Helposti ylläpidettävän koodin tunnusmerkit

- Ylläpidettävyyden ja laajennettavuuden kannalta tärkeitä seikkoja
  - Koodin tulee olla luettavuudeltaan selkeää, eli koodin tulee kertoa esim. nimennällään mahdollisimman selkeästi mitä koodi tekee, eli tuoda esiin koodin alla oleva ”design”
  - Yhtä paikkaa pitää pystyä muuttamaan siten, ettei muutoksesta aiheudu sivuvaikutuksia sellaisiin kohtiin koodia, jota muuttaja ei pysty ennakoimaan
  - Jos ohjelmaan tulee tehdä laajennus tai bugikorjaus, tulee olla helppo selvittää mihin kohtaan koodia muutos tulee tehdä
  - Jos ohjelmasta muutetaan ”yhtä asiaa”, tulee kaikkien muutosten tapahtua vain yhteen kohtaan koodia (metodiin tai luokkaan)
  - Muutosten ja laajennusten jälkeen tulee olla helposti tarkastettavissa ettei muutos aiheuta sivuvaikutuksia muualle järjestelmään
- Näin määritelty koodin *sisäinen laatu* on erityisen tärkeää ketterissä menetelmissä, joissa koodia laajennetaan iteraatio iteraatiolta
- Jos koodin sisäiseen laatuun ei kiinnitetä huomiota, on väistämätöntä että pidemmässä projektissa kehitystiimin velositeetti alkaa tippua ja eteneminen alkaa vaikeutua iteraatio iteraatiolta
  - Koodin sisäinen laatu on siis usein myös asiakkaan etu

Koodin laatuattribuutteja

- Edellä lueteltuihin hyvän koodin tunnusmerkkeihin päästään kiinnittämällä huomio seuraaviin *laatuattribuutteihin*
  - Kapselointi
  - Koheesio
  - Riippuvuuksien vähäisyys
  - Toisteettomuus
  - Testattavuus
  - Selkeys
- Tutkitaan nyt näitä laatuattribuutteja sekä periaatteita, joita noudattaen on mahdollista kirjoittaa koodia, joka on näiden mittarien mukaan laadukasta

- **HUOM** seuraaviin kalvoihin liittyvät koodiesimerkit löytyvät osoitteesta https://github.com/mluukkai/ohtu2014/blob/master/web/luento8.md lue koodiesimerkkejä sitä mukaa kun kalvoissa viitataan niihin

Kapselointi

- Ohjelmointikursseilla on määritelty kapselointi seuraavasti
  - ”Tapaa ohjelmoida olion toteutuksen yksityiskohdat luokkamäärittelyn sisään – piiloon olion käyttäjältä – kutsutaan kapseloinniksi. Olion käyttäjän ei tarvitse tietää mitään olioden sisäisestä toiminnasta. Eikä hän itse asiassa edes saa siitä mitään tietää vaikka kuinka haluaisi! ”
- Aloitteleva ohjelmoija assosioi kapseloinnin yleensä seuraavaan periaatteeseen:
  - Oliomuuttujat tulee määritellä privaateiksi ja niille tulee tehdä tarvittaessa setterit ja getterit
- Tämä on kuitenkin aika kapea näkökulma kapselointiin
- Jatkossa näemme esimerkkejä monista kapseloinnin muista muodoista. Kapseloinnin kohde voi olla mm.
  - Käytettävän olion tyyppi, algoritmi, olioiden luomistapa, käytettävän komponentin rakenne
- Monissa suunnittelumalleissa on kyse juuri eritasoisten asioiden kapseloinnista

Koheesio

- Koheesiolla tarkoitetaan sitä, kuinka pitkälle metodissa, luokassa tai komponentissa oleva ohjelmakoodi on keskittynyt tietyn toiminnallisuuden toteuttamiseen
- Hyvänä asiana pidetään mahdollisimman korkeata koheesion astetta
- Koheesioon tulee siis pyrkiä kaikilla ohjelman tasoilla, metodeissa, luokissa, komponenteissa ja jopa muuttujissa
- **Metoditason koheesiossa** pyrkimyksenä että metodi tekee itse vain yhden asian
- Metoditason koheesiota ilmentävä Kent Beckin ”Composed method”-suunnittelumalli ohjeistaa seuraavasti
- *The composed method pattern* defines three key statements:
  - Divide your programs into methods that perform one identifiable task.
  - Keep all the operations in a method at the same level of abstraction.
  - This will naturally result in programs with many small methods, each a few lines long.
  - http://www.ibm.com/developerworks/java/library/j-eaed4/index.html

Koheesio ja Single responsibility -periaate

- Metoditason koheesioon päästään jakamalla ”koheesioton” metodi useisiin metodeihin, joita alkuperäinen metodi kutsuu
  - Alkuperäinen metodi alkaa toimia korkeammalla abstraktiotasolla, koordinoiden kutsumiensa yhteen asiaan keskittyvien metodien suoritusta
  - esimerkki https://github.com/mluukkai/ohtu2014/blob/master/web/luento8.md kohdassa ”koheesio metoditasolla”
- **Luokkatason koheesiossa** pyrkimyksenä on, että luokan **vastuulla** on vain yksi asia
- Ohjelmistotekniikan menetelmistä tuttu **Single Responsibility** (SRP) -periaate tarkoittaa oikeastaan täysin samaa
  - www.objectmentor.com/resources/articles/srp.pdf
  - Uncle Bob tarkentaa yhden vastuun määritelmää siten, että *luokalla on yksi vastuu jos sillä on vain yksi syy muuttua*
- esimerkejä https://github.com/mluukkai/ohtu2014/blob/master/web/luento8.md kohdassa ”single responsibility -periaate eli koheesio luokkatasolla”
- Vastakohta SRP:tä noudattavalle luokalle on *jumalaluokka/olio*
  - http://blog.decayingcode.com/post/anti-pattern-god-object.aspx