Ohjelmistotuotanto

Luento 8

20.11.

Ohjelmiston suunnittelu

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
  - **Olio/komponenttisuunnittelu**
    - 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
nopea kertaus

Ohjelmiston arkkitehtuuri

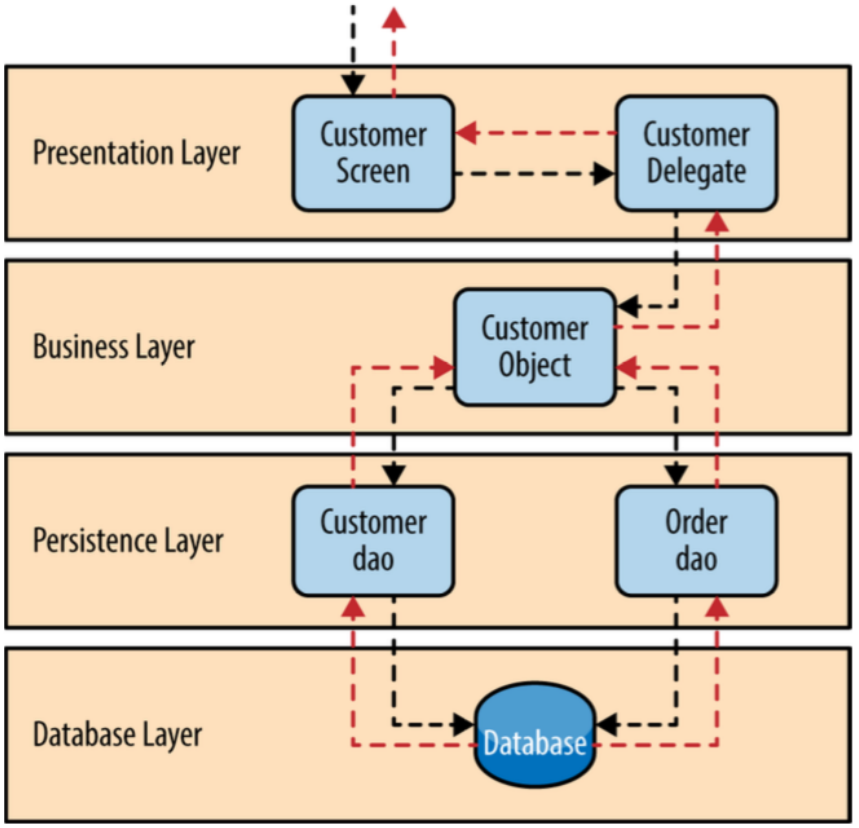
- *Ohjelmiston arkkitehtuuri on järjestelmän perusorganisaatio, joka sisältää järjestelmän osat, osien keskinäiset suhteet, osien suhteet ympäristöön sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua ja evoluutiota (IEEE)*
- Arkkitehtuuri syntyy joukosta arkkitehtoorisia valintoja (...**set of significant decisions about the organization of a software system)**
- Järjestelmälle asetetuilla ei-toiminnallisilla laatuvaatimuksilla (engl. -ilities) on suuri vaikutus arkkitehtuuriin
  - Käytettävyys, suorituskyky, skaalautuvuus, vikasietoisuus, tiedon ajantasaisuus, tietoturva, ylläpidettävyys, laajennettavuus, hinta, time-to-market, …
- Myös toimintaympäristö kuten integraatiot muihin järjestelmiin vaikuttavat arkkitehtuuriin
- Ohjelmiston 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)

Arkkitehtuurimalli

- 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

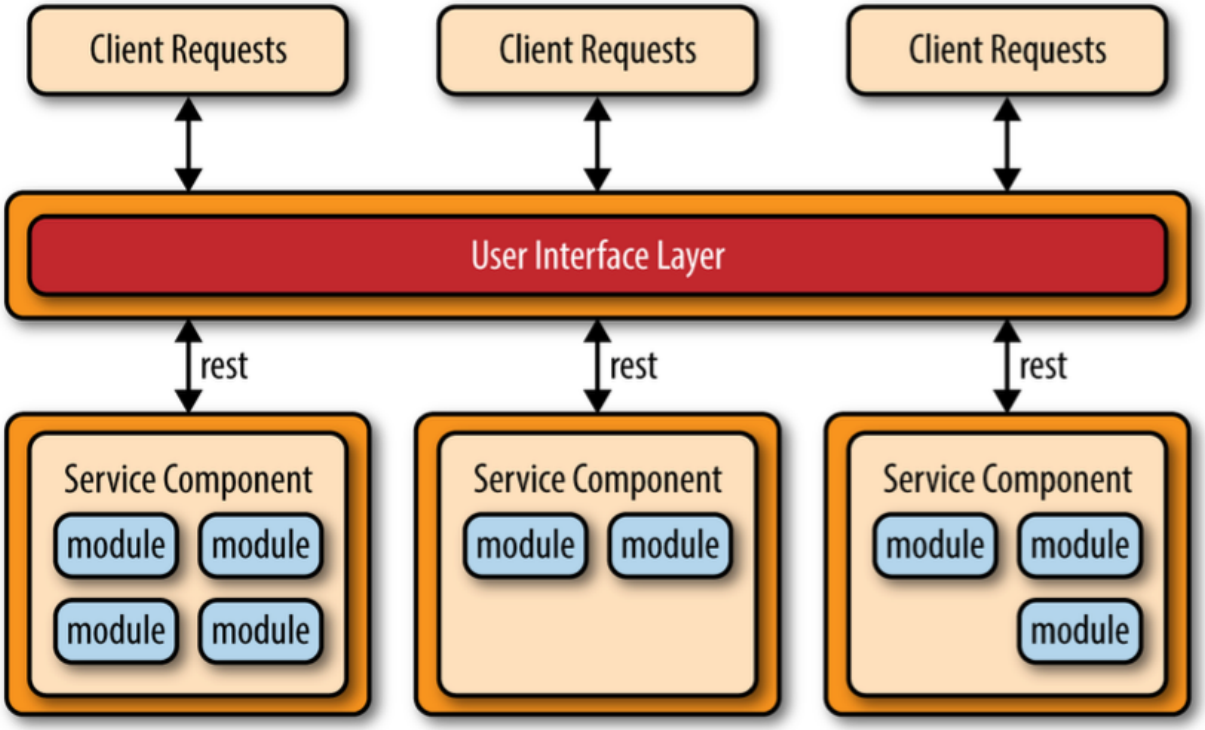
- Eräs tunnetuimmista on *kerrosarkkitehtuuri*
  - Kerros on kokoelma toisiinsa liittyviä olioita tai alikomponentteja, jotka muodostavat toiminnallisuuden suhteen loogisen kokonaisuuden
  - Pyrkimyksenä järjestellä komponentit siten, että ylempänä oleva kerros käyttää ainoastaan alempana olevien kerroksien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

Hieman lisää arkkitehtuurimalleista

- Tarkastellaan vielä hieman paria arkkitehtuurimallia
- Edellisellä kalvolla 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

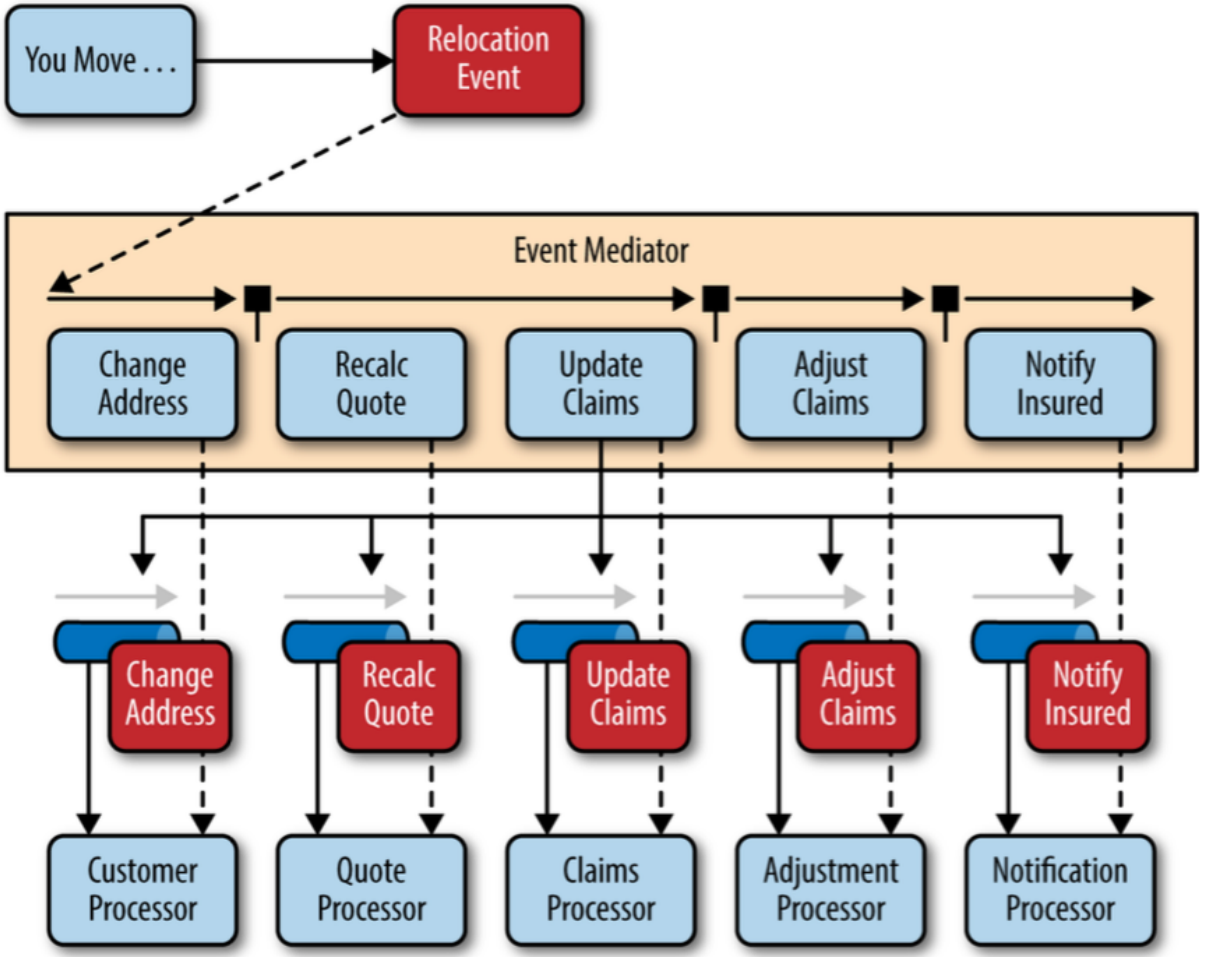
- Mikropalveluiden käyttö mahdollistaa sen, että sovellus voidaan helposti koodata ”monella kielellä”, toisin kuin monoliittisissa projekteissa, mikään ei edellytä, että kaikki mikropalvelut olisi toteutettu samalla kielellä
- Sovelluksen jakaminen järkeviin mikropalveluihin on haastavaa
- Kymmenistä tai jopa sadoista mikropalveluista koostuvan ohjelmiston operoiminen eli ”käynnistäminen” tuotantopalvelimilla on haastavaa ja vaatii pitkälle menevää automatisointia
  - Sama koskee sovelluskehitysympäristöä ja jatkuvaa integraatiota
  - Mikropalveluiden menestyksekäs soveltaminen edellyttää vahvaa devops-kulttuuria
- 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…
  - Mutta pian käynnistyy asiaa käsittelevä kurssi *Devops with Docker*

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 sanotaan että mikropalvelut tarjoavat kommunikointia varten REST-rajapinnan
  - Viikon 3 laskareissa haettiin suorituksiin liittyvää dataa palautusjärjestelmän tarjoamasta REST-rajapinnasta
- Vaihtoehtoinen, huomattavasti joustavampi kommunikointikeino on ns. *viestinvälityksen* (message queue/bus) käyttö
- Palvelut eivät lähetä viestejä suoraan toisilleen, vaan käytössä on verkossa toimiva *viestinvälityspalvelu*, joka hoitaa viestien välityksen eri palveluiden välillä

Mikropalveluiden kommunikointi

- Palvelut voivat lähettää tai **julkaista** (publish) viestejä viestinvälityspalveluun
- 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*
- Viestinvälityspalvelu välittää vastaanottamansa viestit kaikille, jotka ovat aiheen tilanneet
- Kaikki viestien (tai joskus puhutaan myös tapahtumista, *event*) välitys tapahtuu viestinvälityspalvelun (seuraavan kalvon 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
- Viestien lähetys lähettäjän kannalta asynkronista, eli palvelu lähettää viestin, jatkaa se heti koodissaan eteenpäin siitä huolimatta onko viesti välitetty sen tilanneille palveluille



- Asynkronisten viestien (tai eventtien) välitykseen perustuvaa arkkitehtuureja kutsutaan myös *event-driven*-arkkitehtuureiksi
  - kaikki event-driven-arkkitehtuurit eivät välttämättä ole mikroarkkitehtuureja, esim. Java Swing/FX -sovelluksessa käyttöliittymä kommunikoi sovelluksen kanssa asynkronisten tapahtumien avulla
- Viestinvälitykseen perustuvat mikropalvelut eivät ole ilmainen lounas, erityisesti debuggaus voi olla välillä melko haastavaa

Arkkitehtuuri ketterissä menetelmissä

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

Walking skeleton

- Yleinen lähestymistapa inkrementaaliseen arkkitehtuuriin on *walking skeletonin* käyttö
  - A Walking Skeleton is a **tiny implementation of the system that performs a small end-to-end function**. It need not use the final architecture, **but it should link together the main architectural components**.
  - The architecture and the functionality can then evolve in parallel.
- What constitutes a walking skeleton varies with the system being designed
  - For a layered architecture system, it is a working connection between all the layers
- The walking skeleton is not complete or robust and it is missing the flesh of the application functionality. Incrementally, over time, the infrastructure will be completed and full functionality will be added
- A walking skeleton, is permanent code, built with production coding habits, regression tests, and is intended to grow with the system
- Eli tarkoitus on toteuttaa arkkitehtuurin rungon sisältävä Walking skeleton jo ensimmäisessä sprintissä ja kasvattaa se pikkuhiljaa projektin edetessä
- Katso lisää esim http://alistair.cockburn.us/Walking+skeleton

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 arkkitehtonisia 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 sisäistä 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 olio/komponenttisuunnittelua

Olio/komponenttisuunnittelu

Olio/komponenttisuunnittelu

- Käytettäessä ohjelmiston toteutukseen olio-ohjelmointikieltä, on suunnitteluvaiheen tarkoituksena löytää sellaiset oliot, jotka pystyvät yhteistoiminnallaan toteuttamaan järjestelmän vaatimuksen
  - Jos käytössä jotain muuta paradigmaa käyttävä kieli, tässä suunnittelun vaiheessa suunnitellaan kielen paradigman tukevat rakennekomponentit, esim. funktiot, aliohjelmat, moduulit...
- Komponenttisuunnittelua 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
- Ohjelmistosuunnittelu onkin ”enemmän taidetta kuin tiedettä”, kokemus ja hyvien käytänteiden opiskelu toki auttaa
- Erityisesti ketterissä menetelmissä tarkka suunnittelu tapahtuu yleensä vasta ohjelmoitaessa

Olio/komponenttisuunnittelu

- Emme keskity kurssilla mihinkään yksittäiseen suunnittelumenetelmään, vaan tutustumme eräisiin tärkeisiin menetelmäriippumattomiin teemoihin:
- Laajennettavuuden ja ylläpidettävyyden suhteen laadukkaan koodin/oliosuunnittelun tunnusmerkkeihin ja *laatuattribuutteihin* 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 nähneet muutamia suunnittelumalleja, ainakin seuraavat: dependency injection, singleton, data access object
  - Suuri osa tällä kurssilla kohtaamistamme suunnittelumalleista on syntynyt olio-ohjelmointikielten parissa. Osa suunnittelumalleista on relevantteja myös muita paragigmoja, kuten funktionaalista ohjelmointia käytettäessa
  - Muilla paradigmoilla on myös omia suunnittelumallejaan, mutta niitä emme kurssilla käsittele

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
  - 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 kalvojen asioihin liittyy joukko koodiesimerkkejä, jotka löytyvät osoitteesta

  https://github.com/mluukkai/ohjelmistotuotanto2018/blob/master/web/oliosuunnittelu.md
- Koodiesimerkkejä ei käsitellä luennoilla, mutta on tarkoituksena, että luet ne viikojen 5 ja 6 laskareihin valmistautuessasi

Kapselointi

- Ohjelmointikursseilla on määritelty kapselointi seuraavasti
  - ”Tapaa ohjelmoida olion toteutuksen yksityiskohdat luokkamäärittelyn sisään – piiloon olion käyttäjältä – kutsutaan kapseloinniksi. Olion käyttäjän ei tarvitse tietää mitään olioiden sisäisestä toiminnasta. Eikä hän itse asiassa edes saa siitä mitään tietää vaikka kuinka haluaisi!” (vanha ohpen materiaali)
- 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
- Itseopiskelumateriaalissa on paljon 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 ja Single responsibility -periaate

- Koheesiolla tarkoitetaan sitä, kuinka pitkälle metodissa, luokassa tai komponentissa oleva ohjelmakoodi on keskittynyt tietyn toiminnallisuuden toteuttamiseen
- Hyvänä asiana pidetään mahdollisimman korkeaa 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
- **Luokkatason koheesiossa** pyrkimyksenä on, että luokan **vastuulla** on vain yksi asia
- Ohjelmistotekniikan menetelmistä tuttu **Single Responsibility** (SRP) -periaate tarkoittaa oikeastaan täysin samaa
  - Uncle Bob tarkentaa yhden vastuun määritelmää siten, että *luokalla on yksi vastuu jos sillä on vain yksi syy muuttua*
- *Vastakohta SRP:tä noudattavalle luokalle on jumalaluokka/olio*

Riippuvuuksien vähäisyys

- Single responsibility -periaatteen hengessä tehty ohjelma koostuu suuresta määrästä oliota/komponentteja, joilla on suuri määrä pieniä metodeja
- Olioiden on siis oltava vuorovaikutuksessa toistensa kanssa saadakseen toteutettua ohjelman toiminnallisuuden
- **Riippuvuuksien vähäisyyden** (engl. low coupling) periaate pyrkii eliminoimaan luokkien ja olioiden välisiä riippuvuuksia
- Koska olioita on paljon, tulee riippuvuuksia pakostakin, miten riippuvuudet sitten saadaan eliminoitua?
- Ideana on eliminoida *tarpeettomat* riippuvuudet ja välttää riippuvuuksia *konkreettisiin* asioihin
  - Riippuvuuden kannattaa kohdistua asiaan joka ei muutu herkästi, eli joko rajapintaan tai abstraktiin luokkaan
- Sama idea kulkee parillakin eri nimellä
  - **Program to an interface, not to an Implementation**
  - **Depend on Abstractions, not on concrete implementation**

Riippuvuuksien vähäisyys

- Konkreettisen riippuvuuden eliminointi voidaan tehdä rajapintojen (tai abstraktien luokkien) avulla
  - Olemme tehneet näin kurssilla usein, mm. Verkkokaupan riippuvuus Varastoon, Pankkiin ja Viitegeneraattoriin korvattiin rajapinnoilla
  - *Dependency Injection* -suunnittelumalli toimi usein apuvälineenä konkreettisen riippumisen eliminoinnissa
- Osa luokkien välisistä riippuvuuksista on tarpeettomia ja ne kannattaa eliminoida muuttamalla luokan vastuita
- Perintä muodostaa riippuvuuden perivän ja perittävän luokan välille, tämä voi jossain tapauksissa olla ongelmallista
- Yksi oliosuunnittelun kulmakivi on periaate **Favour composition over inheritance** eli **suosi yhteistoiminnassa toimivia oliota perinnän sijaan**
  - **http://www.artima.com/lejava/articles/designprinciples4.html**
  - Perinnällä on paikkansa, mutta sitä tulee käyttää harkiten
  - Itseopiskelumateriaalissa on esimerkki ongelmallisesta perinnästä

Lisää koodin laatuattribuutteja: DRY

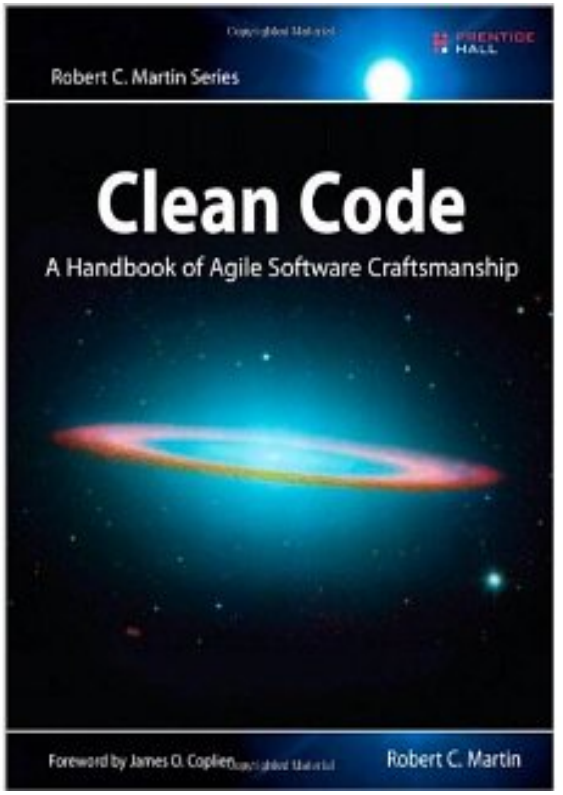
- Käsittelimme koodin laatuattribuuteista kapselointia, koheesiota ja riippuvuuksien vähäisyyttä, seuraavana vuorossa **redundanssi** eli **toisteisuus**
- Aloittelevaa ohjelmoijaa pelotellaan toisteisuuden vaaroista uran ensiaskelista alkaen: **älä copypastaa koodia!**
- Alan piireissä toisteisuudesta varoittava periaate kulkee nimellä **DRY, don't repeat yourself**
  - *”Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”*
  - http://c2.com/cgi/wiki?DontRepeatYourself
  - DRY-periaate menee oikeastaan vielä paljon pelkkää koodissa olevaa toistoa eliminointia pidemmälle
- Ilmeisin toiston muoto koodissa on juuri copypaste ja se onkin helppo eliminoida esim. metodien avulla
- Kaikki toisteisuus ei ole yhtä ilmeistä ja monissa suunnittelumalleissa on kyse juuri hienovaraisempien toisteisuuden muotojen eliminoinnista

Lisää laatuattribuutteja

- **Testattavuus**
  - Hyvä koodi on helppo testata kattavasti iyksikkö- ja integraatiotestein
  - Helppo testattavuus seuraa yleensä siitä, että koodi koostuu löyhästi kytketyistä, selkeän vastuun omaavista komponenteista ja ei sisällä toisteisuutta
  - Kääntäen, jos koodin kattava testaaminen on vaikeaa, on se usein seurausta siitä, että olioiden vastuut eivät ole selkeät, olioilla on liikaa riippuvuuksia ja toisteisuutta on paljon
  - Olemme pyrkineet jo ensimmäiseltä viikolta asti koodin hyvään testattavuuteen esim. purkamalla riippuvuuksia rajapintojen ja dependency injectionin avulla
- **Koodin selkeys ja luettavuus**
  - Suuri osa ”ohjelmointiin” kuluvasta ajasta kuluu olemassaolevan koodin (joko kehittäjän itsensä tai jonkun muun kirjoittaman) lukemiseen

Koodin luettavuus

- Perinteisesti ohjelmakoodin on ajateltu olevan väkisinkin kryptistä ja vaikeasti luettavaa
  - Esim. c-kielessä on tapana ollut kirjoittaa todella tiivistä koodia, jossa yhdellä rivillä on ollut tarkoitus tehdä mahdollisimman monta asiaa
  - Metodikutsuja on vältetty tehokkuussyistä
  - Muistinkäyttöä on optimoitu uusiokäyttämällä muuttujia ja ”koodaamalla” dataa bittitasolla
  - ...



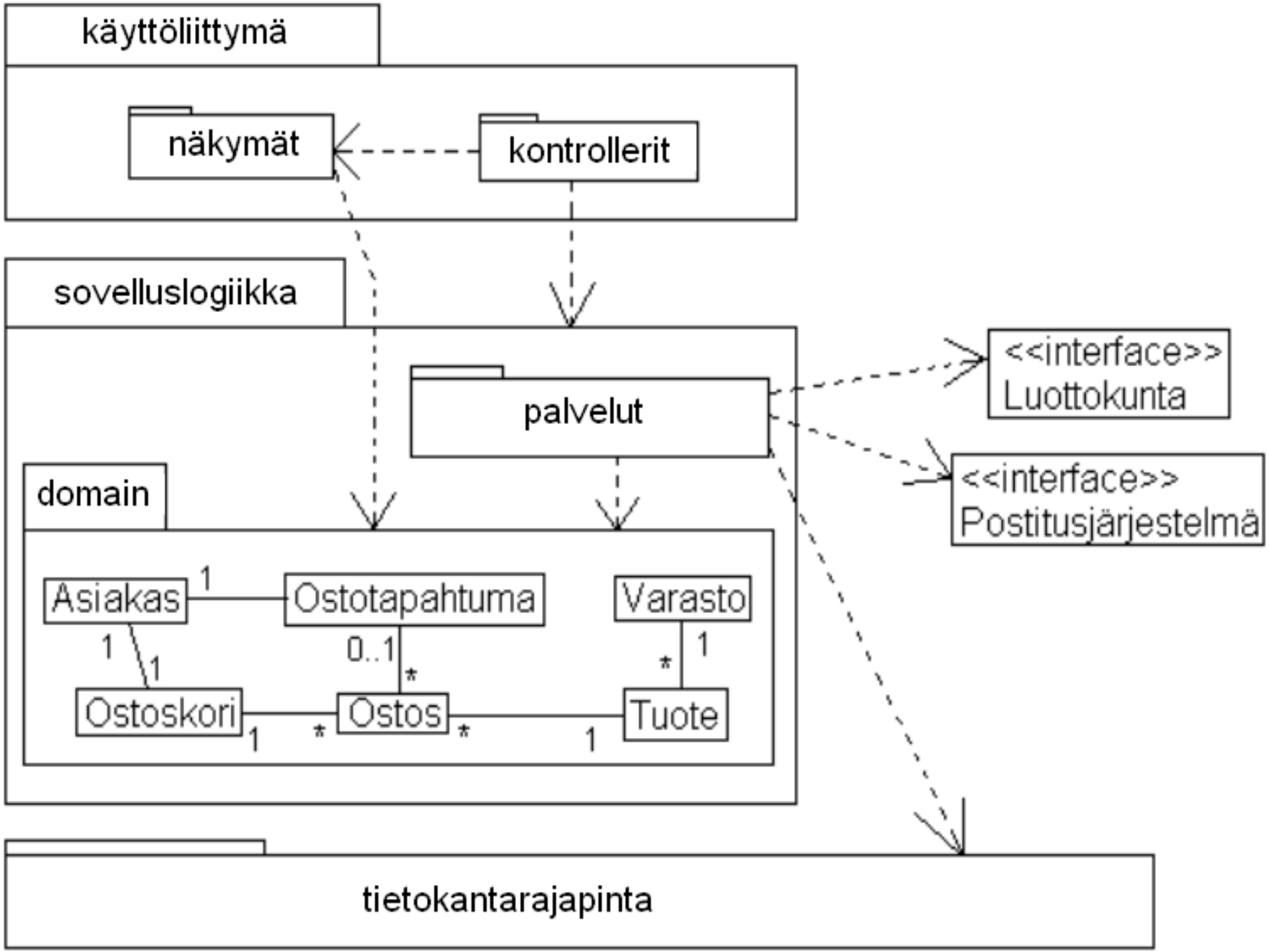
- Ajat ovat muuttuneet ja nykytrendin mukaista on pyrkiä kirjoittamaan koodia, joka nimennällään ja muodollaan ilmaisee mahdollisimman hyvin sen mitä koodi tekee
- Selkeän nimennän lisäksi muita luettavan eli ”puhtaan” koodin (**clean code**) tunnusmerkkejä ovat jo monet meille entuudestaan tutut asiat
  - www.planetgeek.ch/wp-content/uploads/2011/02/Clean-Code-Cheat-Sheet-V1.3.pdf

Suunnittelumalleja

- Suunnittelumallit siis tarjoavat hyviä kooditason ratkaisuja siitä, miten koodi kannattaa muotoilla, jotta siitä saadaan sisäiseltä laadultaan hyvää, eli kapseloitua, hyvän koheesion omaavaa ja eksplisiittiset turhat riippuvuudet välttävää
- Kurssin itseopiskelumateriaalissa tutustutaan seuraaviin suunnittelumalleihin
  - Factory
  - Strategy
  - Command
  - Template method
  - Komposiitti
  - Proxy
  - Model view controller
  - Observer
- Sunnittelumallien soveltamista harjoitellaan viikon 5-7 laskareissa
- Itseopiskelumateriaali löytyy osoitteesta https://github.com/mluukkai/ohjelmistotuotanto2018/blob/master/web/oliosuunnittelu.md

Kapselointi ja koheesio ja riippuvuuksien minimointi arkkitehtuuritasolla

- Arkkitehtuurin yhteydessä mainitsimme kerrosarkkitehtuurin, josta esimerkkinä oli Kumpula biershopin arkkitehtuuri
- Kerroksittaisuudessa periaate on sama kuin useiden suunnittelumallien ja hyvän oliosuunnittelussa yleensäkin **kapseloidaan monimutkaisuutta ja detaljeja rajapintojen taakse**
- Tarkoituksena ylläpidettävyyden parantaminen ja kompleksisuuden hallinnan helpottaminen
  - Kerroksen N käyttäjää on turha vaivata kerroksen sisäisellä rakenteella
  - Eikä sitä edes kannata paljastaa, koska näin muodostuisi eksplisiittinen riippuvuus käyttäjän ja N:n välille
- Pyrkimys siihen että *kerrokset ovat mahdollisimman korkean koheesion omaavia*, eli ”yhteen asiaan” keskittyvä
  - Käyttöliittymä
  - Tietokantayhteydet
  - Liiketoimintalogiikka
- Kerrokset taas ovat keskenään mahdollisimman *löyhästi kytkettyjä*


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

Domain Driven Design

- Viimeaikaisena voimakkaasti nousevana trendinä on käyttää sovelluksen koodin tasolla nimentää, joka vastaa liiketoiminta-alueen eli ”bisnesdomainin” terminologiaa
  - Yleisnimike tälle tyylille on Domain Driven Design, DDD
  - ks esim. http://www.infoq.com/articles/ddd-evolving-architecture
- Ohjelmiston arkkitehtuurissa on DDD:tä sovellettaessa (ja muutenkin kerrosarkkitehtuuria sovellettaessa) on kerros joka kuvaa *domainin*, eli sisältää *liiketoimintaoliot*
- Esim. Kumpula Biershopin domain-oliot:
  - Tuote
  - Varasto
  - Ostos
  - Ostoskori
  - Asiakas
  - Ostostapahtuma

Domain Driven Design

- Domain-oliot tai osa niistä yleensä mäpätään tietokantaan
  - Mäppäyksessä käytetään usein DAO-suunnittelumallia, johon tutustuimme ohimennen laskareissa 3
    - DAO-suunnittelumallia käsitellään nykyään jossain määrin myös kurssilla Tietokantojen perusteet
  - DAO:n lisäksi on muitakin mäppäystapoja, kuten Ruby on Railsin käyttämä Active Record
- Domain-oliot tietokantaan mäppäävät komponentit muodostavat oman kerroksen kerrosarkkitehtuurissa
- Joissain suunnittelutyyleissä Domain-olioiden ja sovelluksen käyttöliittymän välissä on vielä erillinen palveluiden kerros
  - http://martinfowler.com/eaaCatalog/serviceLayer.html
  - Wepaa käyville Service Layer on tuttu Java Spring:illä tehdyistä web-sovelluksista
- Palvelut koordinoivat domain-olioille suoritettavaa toiminnallisuutta, esim. *ostoksen laitto ostoskoriin* tai *ostosten maksaminen*
- Ideana on eristää palveluiden avulla sovelluslogiikka käyttöliittymältä

Tekninen velka

- Edellisten kalvojen aikana tutustuimme moniin ohjelman sisäistä laatua kuvaaviin attribuutteihin:
  - kapselointi, koheesio, riippuvuuksien vähäisyys, testattavuus, luettavuus
- Tutustuimme myös yleisiin periaatteisiin, joiden noudattaminen auttaa päätymään laadukkaaseen koodiin
  - single responsibility principle, program to interfaces, favor composition over inheritance, don't repeat yourself
- Itseopiskelumateriaalissa esitellään suunnittelumalleja (design patterns), jotka tarjoavat tiettyihin sovellustilanteisiin sopivia yleisiä ratkaisumalleja
- Koodi ja oliosuunnittelu ei ole aina hyvää, ja joskus on jopa asiakkaan kannalta tarkoituksenmukaista tehdä ”huonoa” koodia
- Huonoa oliosuunnittelua ja huonon koodin kirjoittamista on verrattu ***velan*** **(engl. design debt tai technical debt)** ottamiseen
  - http://www.infoq.com/articles/technical-debt-levison

Tekninen velka

- Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain, mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin myöhemmin *jos* ohjelmaa on tarkoitus laajentaa
  - Käytännössä käy niin, että tiimin velositeetti laskee, koska teknistä velkaa on maksettava takaisin, jotta järjestelmään saadaan toteutettua uusia ominaisuuksia
- Jos korkojen maksun aikaa ei koskaan tule, ohjelma on esim. pelkkä prototyyppi tai sitä ei oteta koskaan käyttöön, voi ”huono koodi” olla asiakkaan kannalta kannattava ratkaisu
  - Esim. uuden ominaisuuden käyttökelpoisuuden validointiin toteutettava minimal viable product (MVP) on luonteeltaan sellainen, että sitä tehdessä otetaan tietoisesti teknistä velkaa
- Vastaavasti joskus voi lyhytaikaisen teknisen velan ottaminen olla järkevää tai jopa välttämätöntä
  - Esim. voidaan saada tuote nopeammin markkinoille tekemällä tietoisesti huonoa designia, joka korjataan myöhemmin

Tekninen velka

- Tekniselle velalle on yritetty jopa arvioida hintaa:
  - http://www.infoq.com/news/2012/02/tech-debt-361
- Kaikki tekninen velka ei ole samanlaista, Martin Fowler jaottelee teknisen velan neljään eri luokkaan:
  - Reckless and deliberate: ”*we do not have time for design”*
  - Reckless and inadverent*: ”what is layering”?*
  - Prudent and inadverent: ”*now we know how we should have done it*”
  - Prudent and deliberate: *”we must ship now and will deal with consequences”*
  - http://martinfowler.com/bliki/TechnicalDebtQuadrant.html
- Teknisen velan takana voi siis olla monenlaisia syitä, esim. holtittomuus, osaamattomuus, tietämättömyys tai tarkoituksella tehty päätös

Koodi haisee: merkki huonosta suunnittelusta

- Seuraavassa alan ehdoton asiantuntija Martin Fowler selittää mistä on kysymys **koodin hajuista**:
  - **A code smell is a surface indication that usually corresponds to a deeper problem in the system.** The term was first coined by Kent Beck while helping me with my Refactoring book.
  - The quick definition above contains a couple of subtle points. Firstly **a smell is by definition something that's quick to spot** - or sniffable as I've recently put it. *A long method is a good example of this - just looking at the code and my nose twitches if I see more than a dozen lines of java.*
  - The second is that smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there - smells aren't inherently bad on their own - they **are often an indicator of a problem rather than the problem themselves.**
  - One of the nice things about smells is tha**t it's easy for inexperienced people to spot them**, even if they don't know enough to evaluate if there's a real problem or to correct them. I've heard of lead developers who will pick a "smell of the week" and ask people to look for the smell and bring it up with the senior members of the team. Doing it one smell at a time is a good way of gradually teaching people on the team to be better programmers.

Koodihajuja

- Koodihajuja on hyvin monenlaisia ja monentasoisia
- On hyvä oppia tunnistamaan ja välttämään tavanomaisimpia
- Internetistä löytyy paljon hajulistoja, esim:
  - http://sourcemaking.com/refactoring/bad-smells-in-code
  - http://c2.com/xp/CodeSmell.html
  - http://www.codinghorror.com/blog/2006/05/code-smells.html
- Muutamia esimerkkejä helposti tunnistettavista hajuista:
  - Duplicated code (eli koodissa copy pastea...)
  - Methods too big
  - Classes with too many instance variables
  - Classes with too much code
  - Long parameter list
  - Uncommunicative name
  - Comments (eikö kommentointi muka ole hyvä asia?)

Koodihajuja

- Seuraavassa pari ei ehkä niin ilmeistä tai helposti tunnistettavaa koodihajua
- **Primitive obsession**
  - Don't use a gaggle of primitive data type variables as a poor man's substitute for a class. If your data type is sufficiently complex, write a class to represent it.
  - http://sourcemaking.com/refactoring/primitive-obsession
- **Shotgun surgery**
  - If a change in one class requires cascading changes in several related classes, consider refactoring so that the changes are limited to a single class.
  - http://sourcemaking.com/refactoring/shotgun-surgery

Koodin refaktorointi

- Lääke koodihajuun on *refaktorointi* eli muutos koodin rakenteeseen joka kuitenkin pitää koodin toiminnan ennallaan
- Erilaisia koodin rakennetta parantavia refaktorointeja on lukuisia
  - ks esim. http://sourcemaking.com/refactoring
- Muutama käyttökelpoinen nykyaikaisessa kehitysympäristössä (esim NetBeans, Eclipse, IntelliJ) automatisoitu refaktorointi:
  - **Rename method** (rename variable, rename class)
    - Eli uudelleennimetään huonosti nimetty asia
  - **Extract method**
    - Jaetaan liian pitkä metodi erottamalla siitä omia apumetodejaan
  - **Extract interface**
    - Luodaan luokan julkisia metodeja vastaava rajapinta, jonka avulla voidaan purkaa olion käyttäjän ja olion väliltä konkreettinen riippuvuus
  - **Extract superclass**
    - Luodaan yliluokka, johon siirretään osa luokan toiminnallisuudesta

Miten refaktorointi kannattaa tehdä

- Refaktoroinnin melkein ehdoton edellytys on kattavien testien olemassaolo
  - Refaktoroinninhan on tarkoitus ainoastaan parantaa luokan tai komponentin sisäistä rakennetta, ulospäin näkyvän toiminnallisuuden pitäisi pysyä muuttumattomana
- Kannattaa ehdottomasti edetä pienin askelin
  - Yksi hallittu muutos kerrallaan
  - Testit on ajettava mahdollisimman usein ja varmistettava että mikään ei mennyt rikki
- Refaktorointia kannattaa suorittaa lähes jatkuvasti
  - Koodin ei kannata antaa ”rapistua” pitkiä aikoja, refaktorointi muuttuu vaikeammaksi
  - Lähes jatkuva refaktorointi on helppoa, pitää koodin rakenteen selkeänä ja helpottaa sekä nopeuttaa koodin laajentamista
- Osa refaktoroinneista, esim. metodien tai luokkien uudelleennimentä tai pitkien metodien jakaminen osametodeiksi on helppoa, aina ei näin ole
  - Joskus on tarve tehdä isoja refaktorointeja joissa ohjelman rakenne eli arkkitehtuuri muuttuu