Ohjelmistotuotanto

Matti Luukkainen ja ohjaajat Jami Kousa, Tero Tapio, Mauri Karlin

syksy 2019

Luento 8

19.11.2019

Miniprojektien aloitustilaisuudet

- Aloitustilaisuudet (jokainen osallistuu yhteen tilaisuuteen)
  - maanantai 18.11. klo 14-16 C222
  - tiistai 19.11. klo 14-16 A128 Chemicum
  - keskiviikko 20.11. klo 12-14 C222
  - torstai 21.11. klo 14-16 C222
- Loppudemot (jokainen ryhmä osallistuu toiseen demoista)
  - maanantai 9.12. klo 14-17
  - tiistai 10.12. klo 14-17

- Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekeminen sisältää
  - vaatimusten analysoinnin ja määrittelyn
  - suunnittelun
  - toteuttamisen
  - testauksen ja
  - ohjelmiston ylläpidon
- Vaatimusmäärittelytä ja testausta käsitelty

- Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekeminen sisältää
  - vaatimusten analysoinnin ja määrittelyn
  - suunnittelun
  - toteuttamisen
  - testauksen ja
  - ohjelmiston ylläpidon
- Vaatimusmäärittelytä ja testausta käsitelty
- Siirrymme käsittelemään ohjelmiston suunnittelua ja toteuttamista
  - osa suunnittelusta tapahtuu vasta toteutusvaiheessa, joten suunnittelun ja toteuttamisen käsittelyä ei voi eriyttää

- Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekeminen sisältää
  - vaatimusten analysoinnin ja määrittelyn
  - suunnittelun
  - toteuttamisen
  - testauksen ja
  - ohjelmiston ylläpidon
- Vaatimusmäärittelytä ja testausta käsitelty
- Siirrymme käsittelemään ohjelmiston suunnittelua ja toteuttamista
  - osa suunnittelusta tapahtuu vasta toteutusvaiheessa, joten suunnittelun ja toteuttamisen käsittelyä ei voi eriyttää
- Suunnittelun tavoite *miten saadaan toteutettua vaatimusmäärittelyn mukaisella tavalla toimiva ohjelma*

Ohjelmiston suunnittelu

- Jakautuu kahteen vaiheeseen:
  - arkkitehtuurisuunnittelu
  - olio/komponenttisuunnittelu

Ohjelmiston suunnittelu

- Jakautuu kahteen vaiheeseen:
  - arkkitehtuurisuunnittelu
  - olio/komponenttisuunnittelu
- Ajoittuminen riippuu käytettävästä tuotantoprosessista:
  - vesiputousmallissa vaatimusmäärittelyn jälkeen, ennen toteutuksen aloittamista, tarkasti dokumentoitu
  - ketterissä menetelmissä suunnittelua tehdään tarvittava määrä jokaisessa iteraatiossa, ei suunnitteludokumenttia

Ohjelmiston suunnittelu

- Jakautuu kahteen vaiheeseen:
  - arkkitehtuurisuunnittelu
  - olio/komponenttisuunnittelu
- Ajoittuminen riippuu käytettävästä tuotantoprosessista:
  - vesiputousmallissa vaatimusmäärittelyn jälkeen, ennen toteutuksen aloittamista, tarkasti dokumentoitu
  - ketterissä menetelmissä suunnittelua tehdään tarvittava määrä jokaisessa iteraatiossa, ei suunnitteludokumenttia
- Vesiputousmallin suunnitteluprosessi tuskin on enää käytössä
  - "jäykimmissäkin" prosesseissa ainakin vaatimusmäärittely ja arkkitehtuurisuunnittelu limittyvät
- Tarkkaa ennen ohjelmointia tapahtuvaa suunnittelua toki edelleen tapahtuu ja joihinkin tilanteisiin se sopiikin

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)

Ohjelmiston arkkitehtuuri

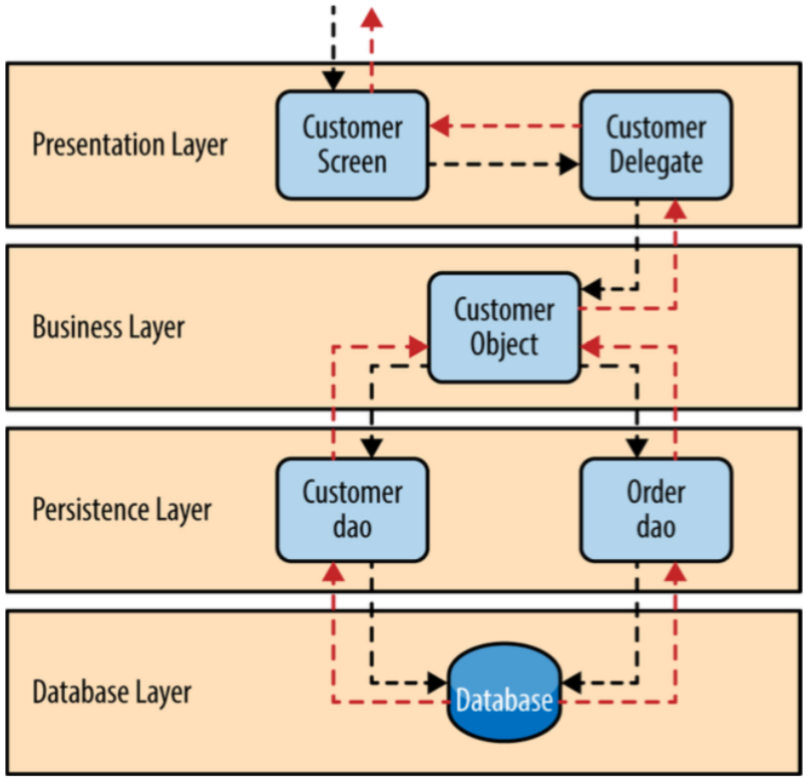
- *Ohjelmiston arkkitehtuuri on järjestelmän perusorganisaatio, joka sisältää järjestelmän osat, osien keskinäiset suhteet, osien suhteet ympäristöön sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua ja evoluutiota* (IEEE)
- Järjestelmälle asetetuilla ei-toiminnallisilla laatuvaatimuksilla (engl.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

Arkkitehtuurityyli

- Ohjelmiston arkkitehtuuri perustuu yleensä yhteen tai useampaan *arkkitehtuurityyliin* (architectural style)
  - hyväksi havaittua tapaa strukturoida tietyntyyppisiä sovelluksia
- Tyylejä suuri määrä
  - Kerrosarkkitehtuuri
  - MVC
  - Pipes-and-filters
  - Repository
  - Client-server
  - Publish-subscribe
  - Event driven
  - REST
  - Microservice

Kerrosarkkitehtuuri

- *Kerros* on kokoelma toisiinsa liittyviä olioita komponentteja, jotka muodostavat toiminnallisuuden suhteen loogisen kokonaisuuden
  - Kerros käyttää ainoastaan alempana olevan kerroksen palveluita



- Kerrokset omalla abstraktiotasollaan
  - Ylimmät kerrokset ovat lähellä käyttäjää: UI ja sovelluslogiikka
  - Alimmat kerrokset taas keskittyvät koneläheisiin asioihin: esim. tiedon tallennus

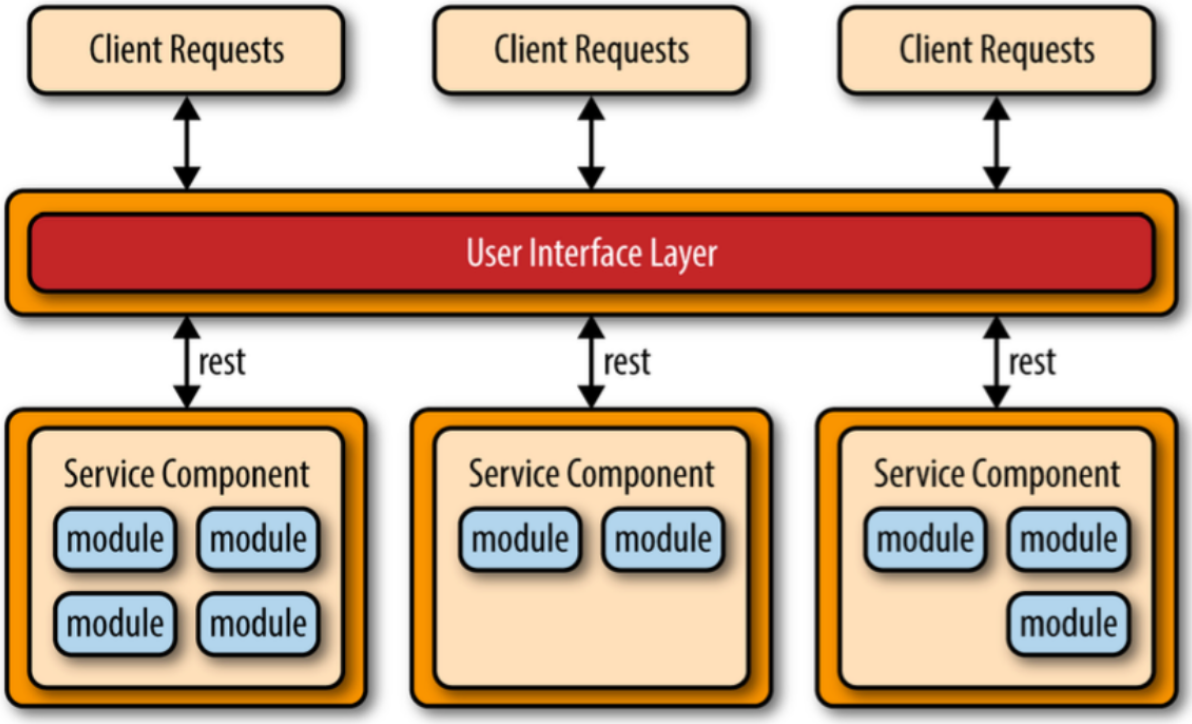
- Kerrokset omalla abstraktiotasollaan
  - Ylimmät kerrokset ovat lähellä käyttäjää: UI ja sovelluslogiikka
  - Alimmat kerrokset taas keskittyvät koneläheisiin asioihin: esim. tiedon tallennus
- Kerroksittaisuus helpottaa ylläpitoa
  - yhden kerroksen muutokset vaikuttavat korkeintaan yläpuolella olevaan kerrokseen

- Kerrokset omalla abstraktiotasollaan
  - Ylimmät kerrokset ovat lähellä käyttäjää: UI ja sovelluslogiikka
  - Alimmat kerrokset taas keskittyvät koneläheisiin asioihin: esim. tiedon tallennus
- Kerroksittaisuus helpottaa ylläpitoa
  - yhden kerroksen muutokset vaikuttavat korkeintaan yläpuolella olevaan kerrokseen
- Sovelluslogiikan riippumattomuus käyttöliittymästä helpottaa ohjelman siirtämistä uusille alustoille
- Alimpien kerroksien palveluja, voidaan osin uusiokäyttää myös muissa sovelluksissa

- Kerrokset omalla abstraktiotasollaan
  - Ylimmät kerrokset ovat lähellä käyttäjää: UI ja sovelluslogiikka
  - Alimmat kerrokset taas keskittyvät koneläheisiin asioihin: esim. tiedon tallennus
- Kerroksittaisuus helpottaa ylläpitoa
  - yhden kerroksen muutokset vaikuttavat korkeintaan yläpuolella olevaan kerrokseen
- Sovelluslogiikan riippumattomuus käyttöliittymästä helpottaa ohjelman siirtämistä uusille alustoille
- Alimpien kerroksien palveluja, voidaan osin uusiokäyttää myös muissa sovelluksissa
- Saattaa johtaa massiivisiin monoliittisiin sovelluksiin, joita on vaikea laajentaa ja skaalaata suurille käyttäjämäärille

Mikropalveluarkkitehtuuri

- Mikropalveluarkkitehtuuri (microservice) pyrkii vastaamaan näihin haasteisiin
  - sovellus koostataan useista (jopa sadoista) pienistä verkossa toimivista autonomisista palveluista
  - jotka keskenään verkon yli kommunikoiden toteuttavat järjestelmän toiminnallisuuden



Mikropalveluarkkitehtuuri

- Yksittäisistä palveluista pyritään tekemään mahdollisimman riippumattomia
  - palvelut eivät kutsu toistensa metodeja, kommunikointi aina verkon välityksellä
  - eivät käytä yhteistä tietokantaa
  - eivät jaa koodia

Mikropalveluarkkitehtuuri

- Yksittäisistä palveluista pyritään tekemään mahdollisimman riippumattomia
  - palvelut eivät kutsu toistensa metodeja, kommunikointi aina verkon välityksellä
  - eivät käytä yhteistä tietokantaa
  - eivät jaa koodia
- Mikropalveluiden ovat pieniä ja huolehtia vain “yhdestä asiasta”
- Verkkokaupan mikropalveluita voisivat olla
  - käyttäjien hallinta
  - tuotteiden suosittelu
  - tuotteiden hakutoiminnot
  - ostoskorin toiminnallisuus
  - ostosten maksusta huolehtiva toiminnallisuus

Etuja

- Kun järjestelmään lisätään toiminnallisuutta, se yleensä tarkoittaa uusien palveluiden toteuttamista tai ainoastaan joidenkin palveluiden laajentamista
  - Sovelluksen laajentaminen voi olla helpompaa kuin kerrosarkkitehtuurissa

Etuja

- 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
- Mikropalveluiden käyttö mahdollistaa sen, että sovellus voidaan helposti koodata “monella kielellä”, toisin kuin monoliittisissa projekteissa

Haasteita

- Sovelluksen jakaminen järkeviin mikropalveluihin on vaikeaa
- Testaaminen ja debuggaus voi olla vaikeaa koska asioita tapahtuu niin monessa paikassa

Haasteita

- Sovelluksen jakaminen järkeviin mikropalveluihin on vaikeaa
- Testaaminen ja debuggaus voi olla vaikeaa koska asioita tapahtuu niin monessa paikassa
- Kymmenistä tai jopa sadoista mikropalveluista koostuvan ohjelmiston operoiminen tuotantopalvelimilla on haastavaa ja vaatii pitkälle menevää automatisointia
- Sama koskee sovelluskehitysympäristöä ja jatkuvaa integraatiota

Haasteita

- Sovelluksen jakaminen järkeviin mikropalveluihin on vaikeaa
- Testaaminen ja debuggaus voi olla vaikeaa koska asioita tapahtuu niin monessa paikassa
- Kymmenistä tai jopa sadoista mikropalveluista koostuvan ohjelmiston operoiminen 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

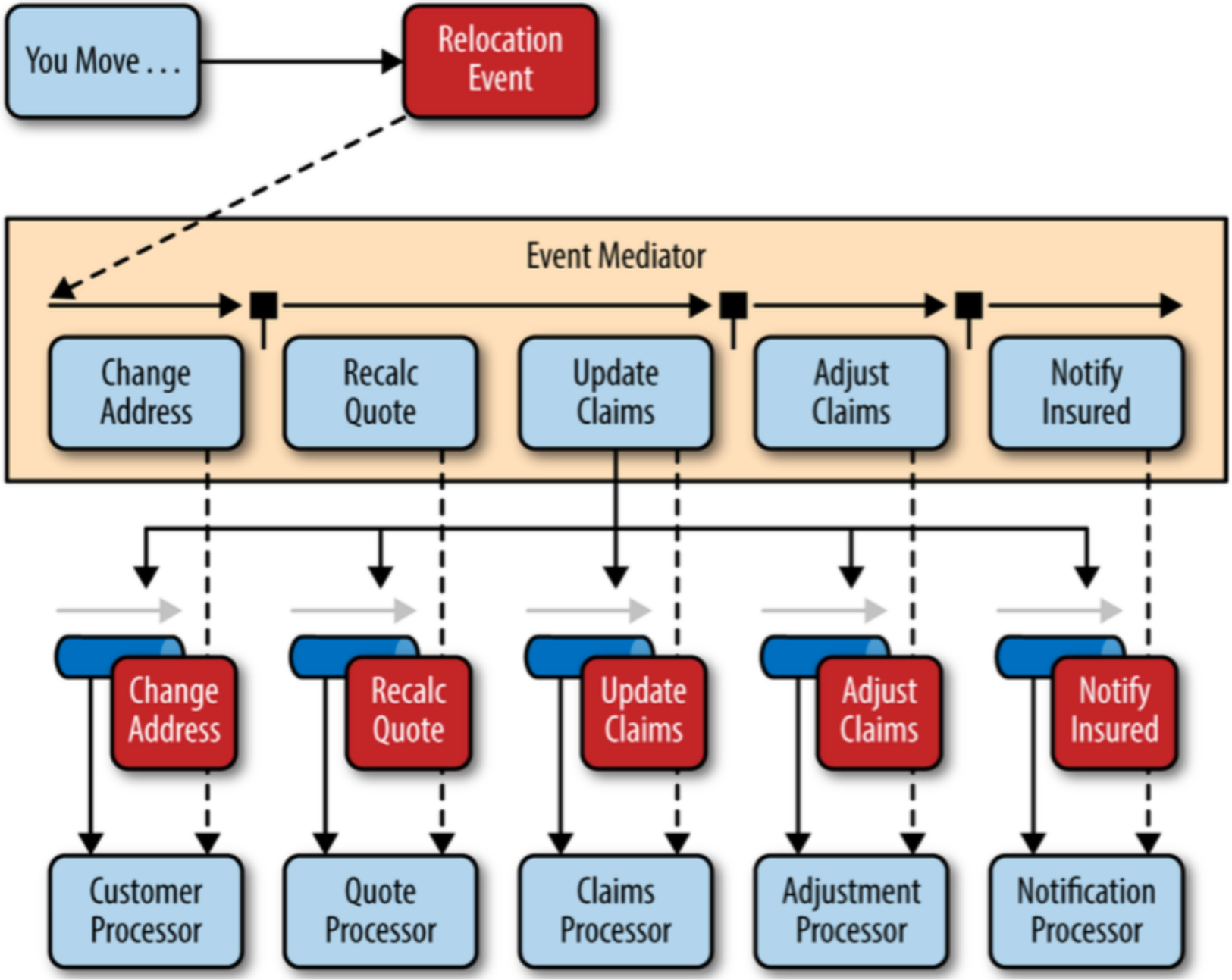
Mikropalveluarkkitehtuuri

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

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 }

Mikropalveluiden kommunikointi 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

Arkkitehtuurin kuvaamisesta

- On tilanteita, missä sovelluksen arkkitehtuuri täytyy dokumentoitava jollain tavalla
- Arkkitehtuurien kuvaamiselle ei olemassa vakiintunutta formaattia
  - Useimmiten käytetään epäformaaleja laatikko/nuoli-kaavioita
  - UML:n luokka- ja pakkauskaaviot sekä komponentti- ja sijoittelukaaviot joskus käyttökelpoisia

Arkkitehtuurin kuvaamisesta

- On tilanteita, missä sovelluksen arkkitehtuuri täytyy dokumentoitava jollain tavalla
- Arkkitehtuurien kuvaamiselle ei olemassa vakiintunutta formaattia
  - Useimmiten käytetään epäformaaleja laatikko/nuoli-kaavioita
  - UML:n luokka- ja pakkauskaaviot sekä komponentti- ja sijoittelukaaviot joskus käyttökelpoisia
- Arkkitehtuurikuvaus kannattaa tehdä useasta eri tarpeita palvelevasta *näkökulmasta*
  - korkean tason kuvauksen voi olla hyödyksi esim. vaatimusmäärittelyssä
  - tarkemmat kuvaukset toimivat ohjeena tarkemmassa suunnittelussa ja ylläpitovaiheen aikaisessa laajentamisessa
- Hyödyllinen arkkitehtuurikuvaus dokumentoi ja perustelee tehtyjä *arkkitehtuurisia valintoja*

Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen:
  - *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*

Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen:
  - *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*

Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen:
  - *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*
- Arkkitehtuuriin suunnittelu ja dokumentointi on perinteisesti pitkäkestoinen, ohjelmoinnin aloittamista edeltävä vaihe
- Ketterät menetelmät ja “arkkitehtuurivetoinen” ohjelmistotuotanto ovat siis jossain määrin keskenään ristiriidassa

Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien yhteydessä puhutaan inkrementaalisesta suunnittelusta ja arkkitehtuurista
- Arkkitehtuuri mietitään riittävällä tasolla projektin alussa
  - Jotkut projektit alkavat ns. nollasprintillä ja alustava arkkitehtuuri määritellään tällöin

Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien yhteydessä puhutaan inkrementaalisesta suunnittelusta ja arkkitehtuurista
- Arkkitehtuuri mietitään riittävällä tasolla projektin alussa
  - Jotkut projektit alkavat ns. nollasprintillä ja alustava arkkitehtuuri määritellään tällöin
- 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*

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 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 informaalisillä tiimi tuntee joka tapauksessa arkkitehtuurin hengen ja pystyy

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

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 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 muuten seurauksena on kaaos

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

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

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/osuunnittelu.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

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

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

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

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

Code smell

- Martin Fowlerin mukaan
  - *koodihaju* (code smell) on helposti huomattava merkki siitä että koodissa on jotain pielessä
  - jopa aloitteleva ohjelmoija saattaa pystyä havaitsemaan koodihajun, sen takana oleva todellinen syy voi olla jossain syvemmällä

Code smell

- Martin Fowlerin mukaan
  - *koodihaju* (code smell) on helposti huomattava merkki siitä että koodissa on jotain pielessä
  - jopa aloitteleva ohjelmoija saattaa pystyä havaitsemaan koodihajun, sen takana oleva todellinen syy voi olla jossain syvemmällä
- Koodihaju siis kertoo, että syystä tai toisesta *koodin sisäinen laatu* ei ole parhaalla mahdollisella tasolla.

Koodihajuja

- Koodihajuja on hyvin monenlaisia ja monentasoisia, esimerkkejä helposti tunnistettavista hajuista:
  - toisteinen koodi
  - liian pitkät metodit
  - luokat joissa on liikaa oliomuuttujia
  - luokat joissa on liikaa koodia
  - metodien liian pitkät parametrilistat
  - epäselkeät muuttujien, metodien tai luokkien nimet
  - kommentit

Koodihajuja

- Koodihajuja on hyvin monenlaisia ja monentasoisia, esimerkkejä helposti tunnistettavista hajuista:
  - toisteinen koodi
  - liian pitkät metodit
  - luokat joissa on liikaa oliomuuttujia
  - luokat joissa on liikaa koodia
  - metodien liian pitkät parametrilistat
  - epäselkeät muuttujien, metodien tai luokkien nimet
  - kommentit
- Pari monimutkaisempaa
  - Primitive obsession
  - Shotgun surgery

Refaktorointi

- Lääke koodin sisäisen laadun ongelmiin on *refaktorointi*
  - muutos koodin rakenteeseen, joka pitää sen toiminnallisuuden ennallaan

Refaktorointi

- Lääke koodin sisäisen laadun ongelmiin on *refaktorointi*
  - muutos koodin rakenteeseen, joka pitää sen toiminnallisuuden ennallaan
- Erilaisia koodin rakennetta parantavia refaktorointeja on lukuisia
  - *rename variable/method/class*
  - *extract method*
  - *move field/method*
  - *extract interface*
  - *extract superclass*
- Osa pystytään tekemään sovelluskehitysympäristön avustamana

Miten refaktorointi kannattaa tehdä

- Refaktoroinnin melkein ehdoton edellytys on kattavien testien olemassaolo
- Kannattaa ehdottomasti edetä pienin askelin
  - Yksi hallittu muutos kerrallaan
  - Testit suoritettava mahdollisimman usein

Miten refaktorointi kannattaa tehdä

- Refaktoroinnin melkein ehdoton edellytys on kattavien testien olemassaolo
- Kannattaa ehdottomasti edetä pienin askelin
  - Yksi hallittu muutos kerrallaan
  - Testit suoritettava mahdollisimman usein
- Refaktorointia kannattaa suorittaa lähes jatkuvasti
  - Koodin ei kannata antaa rapistua pitkiä aikoja
- Lähes jatkuva refaktorointi on helppoa
  - pitää koodin rakenteen selkeänä ja helpottaa sekä nopeuttaa koodin laajentamista

Miten refaktorointi kannattaa tehdä

- Refaktoroinnin melkein ehdoton edellytys on kattavien testien olemassaolo
- Kannattaa ehdottomasti edetä pienin askelin
  - Yksi hallittu muutos kerrallaan
  - Testit suoritettava mahdollisimman usein
- Refaktorointia kannattaa suorittaa lähes jatkuvasti
  - Koodin ei kannata antaa rapistua pitkiä aikoja
- Lähes jatkuva refaktorointi on helppoa
  - pitää koodin rakenteen selkeänä ja helpottaa sekä nopeuttaa koodin laajentamista
- Osa refaktoroinnista on helppoa ja suoraviivaista, aina ei näin ole
- Joskus tarve tehdä isoja, jopa viikkojen kestoisia refaktorointeja joissa ohjelman rakenne muuttuu paljon

Tekninen velka

- Koodi ei ole aina laadultaan optimaalista, joskus on jopa asiakkaan kannalta tarkoituksenmukaista tehdä vähemmän laadukasta koodia
- Huonoa suunnittelua tai/ja ohjelmointia kuvaa käsite *tekninen velka* (engl. technical debt).

Tekninen velka

- Koodi ei ole aina laadultaan optimaalista, joskus on jopa asiakkaan kannalta tarkoituksenmukaista tehdä vähemmän laadukasta koodia
- Huonoa suunnittelua tai/ja ohjelmointia kuvaa käsite *tekninen velka* (engl. technical debt).
- Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain
  - mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin *jos* ohjelmaa on tarkoitus laajentaa

Tekninen velka

- Koodi ei ole aina laadultaan optimaalista, joskus on jopa asiakkaan kannalta tarkoituksenmukaista tehdä vähemmän laadukasta koodia
- Huonoa suunnittelua tai/ja ohjelmointia kuvaa käsite *tekninen velka* (engl. technical debt).
- Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain
  - mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin *jos* ohjelmaa on tarkoitus laajentaa
- Jos korkojen maksun aikaa ei koskaan tule, voi “huono koodi” olla asiakkaan etu
  - Esim. minimal viable product (MVP)

Tekninen velka

- Koodi ei ole aina laadultaan optimaalista, joskus on jopa asiakkaan kannalta tarkoituksenmukaista tehdä vähemmän laadukasta koodia
- Huonoa suunnittelua tai/ja ohjelmointia kuvaa käsite *tekninen velka* (engl. technical debt).
- Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain
  - mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin *jos* ohjelmaa on tarkoitus laajentaa
- Jos korkojen maksun aikaa ei koskaan tule, voi "huono koodi" olla asiakkaan etu
  - Esim. minimal viable product (MVP)
- Lyhytaikainen tekninen velka voi olla järkevää tai jopa välttämätöntä
  - Esim. voidaan saada tuote nopeammin markkinoille tekemällä tietoisesti huonoa designia, joka korjataan myöhemmin

- Teknisen velan takana voi olla monia syitä: holtittomuus, osaamattomuus, tietämättömyys tai tarkoituksella tehty päätös

- Teknisen velan takana voi olla monia syitä: holtittomuus, osaamattomuus, tietämättömyys tai tarkoituksella tehty päätös
- 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”