

Ohjelmistotuotanto

Matti Luukkainen ja ohjaajat Valtteri Kantanen, Hannah
Leinson, Riku Rauhala, Ville Saastamoinen

syksy 2023

Luento 8

21.11.2023

Kurssipalaute

- ▶ Kurssipalaute
 - ▶ Kurssilla lopussa kerättävän palautteen lisäksi ns. jatkuva palaute <https://norppa.helsinki.fi>

Ohjelmiston elinkaaren vaiheet

- ▶ Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekemiseen kuuluu
 - ▶ vaatimusten analysointi ja määrittely
 - ▶ **suunnittelu**
 - ▶ **toteutus**
 - ▶ testaus/laadunhallinta
 - ▶ ohjelmiston ylläpito

Ohjelmiston elinkaaren vaiheet

- ▶ Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekemiseen kuuluu
 - ▶ vaatimusten analysointi ja määrittely
 - ▶ **suunnittelu**
 - ▶ **toteutus**
 - ▶ testaus/laadunhallinta
 - ▶ ohjelmiston ylläpito
- ▶ Jakautuu kahteen vaiheeseen:
 - ▶ arkkitehtuurisuunnittelu
 - ▶ olio/komponenttisuunnittelu

Ohjelmiston elinkaaren vaiheet

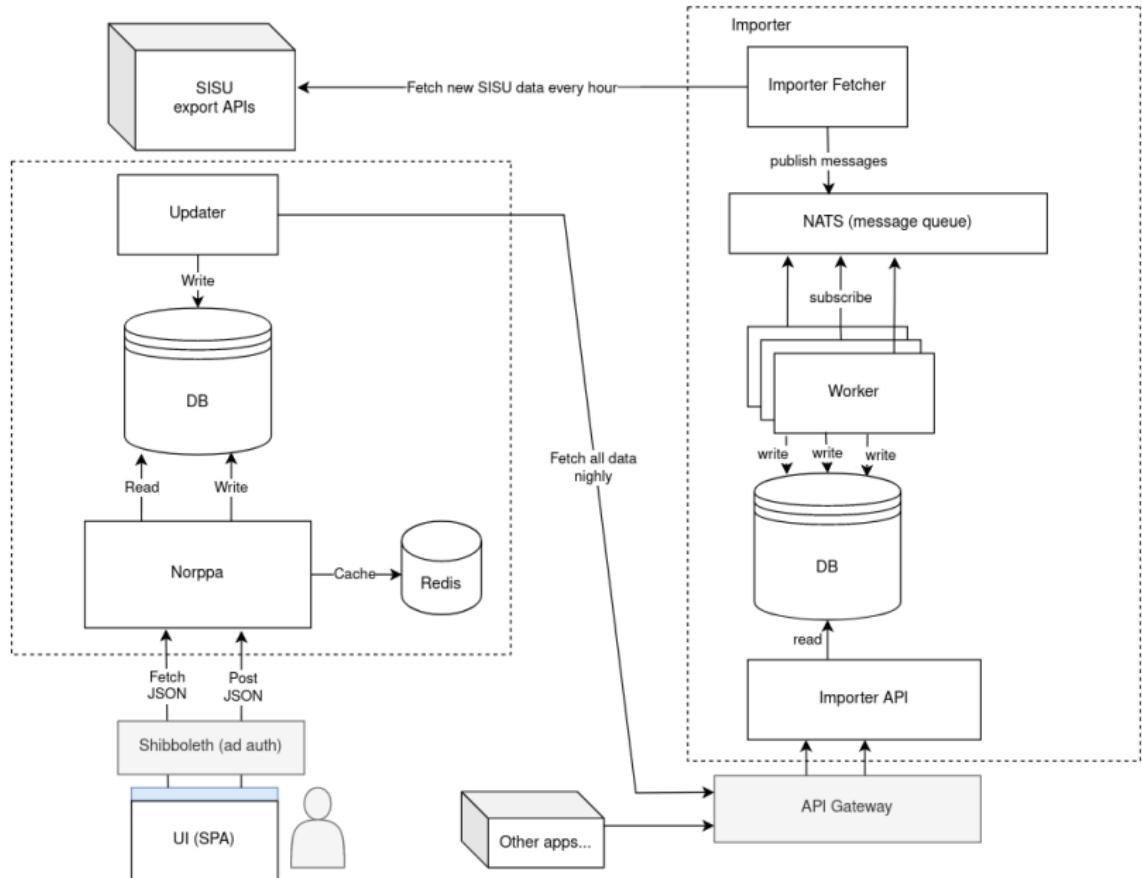
- ▶ Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekemiseen kuuluu
 - ▶ vaatimusten analysointi ja määrittely
 - ▶ **suunnittelu**
 - ▶ **toteutus**
 - ▶ testaus/laadunhallinta
 - ▶ ohjelmiston ylläpito
- ▶ Jakautuu kahteen vaiheeseen:
 - ▶ arkkitehtuurisuunnittelu
 - ▶ olio/komponenttisuunnittelu
- ▶ Näiden lisäksi UI/UX-suunnittelu

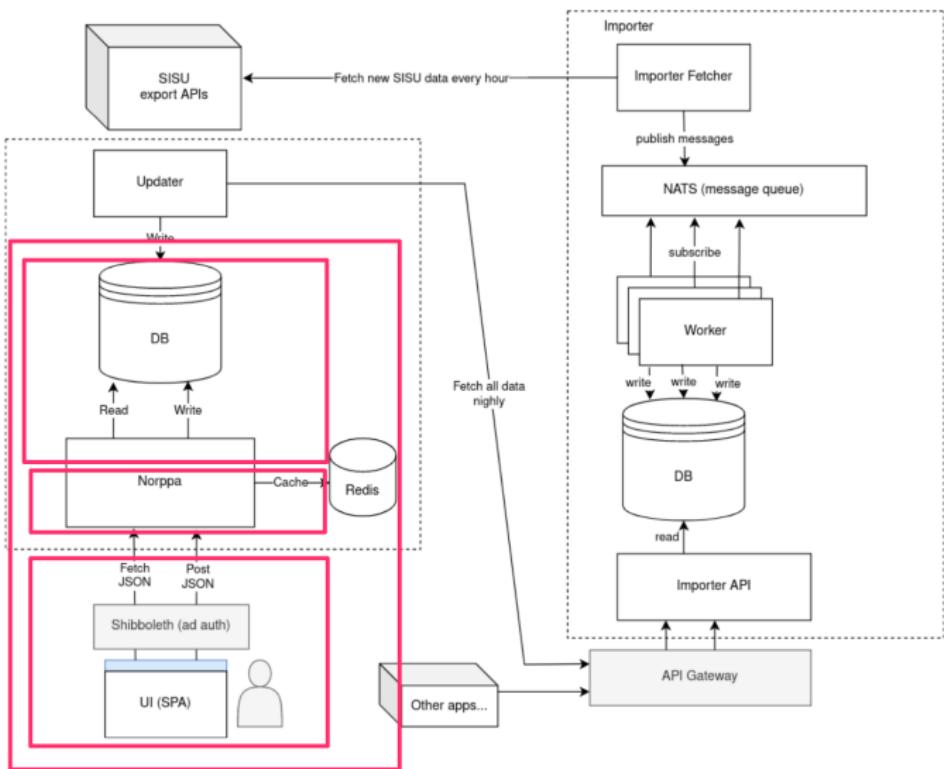
Arkkitehtuurityyli

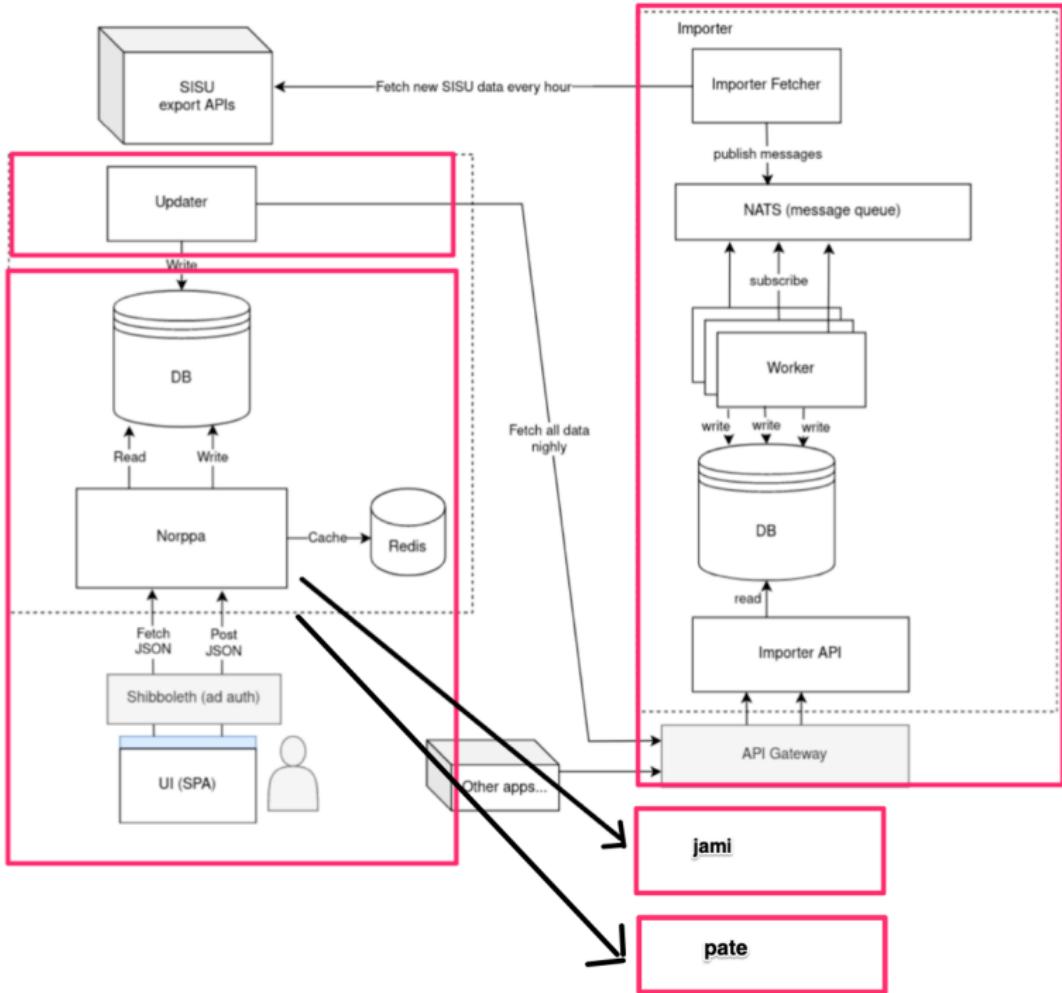
- ▶ Ohjelmiston arkkitehtuuri perustuu yleensä yhteen tai useampaan **arkkitehtuurityyliin** (architectural style)
 - ▶ hyväksi havaittua tapaa strukturoida tietyytyyppisiä sovelluksia
- ▶ Tyylejä suuri määrä
 - ▶ Kerrosarkkitehtuuri
 - ▶ Mikropalveluarkkitehtuuri
 - ▶ MVC
 - ▶ Pipes-and-filters
 - ▶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

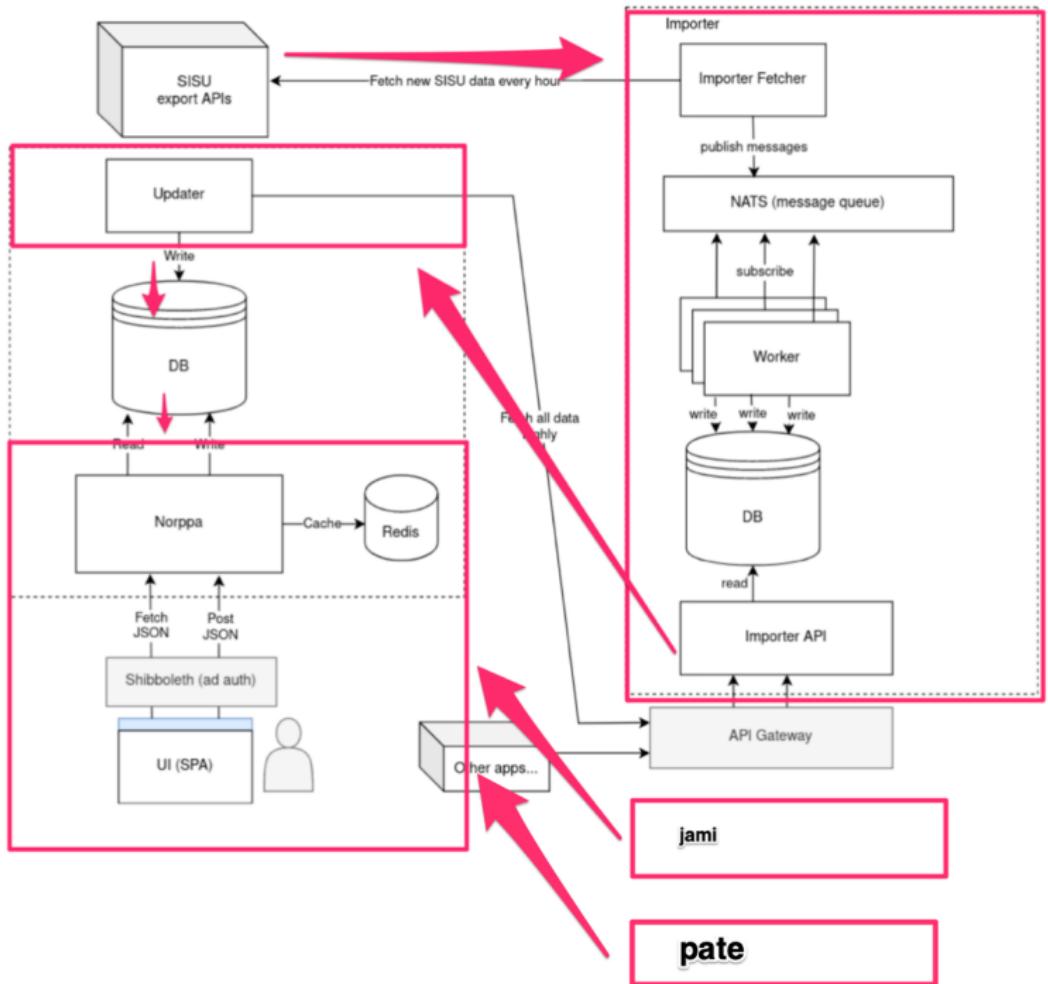
Kurssipalautejärjestelmä Norppa

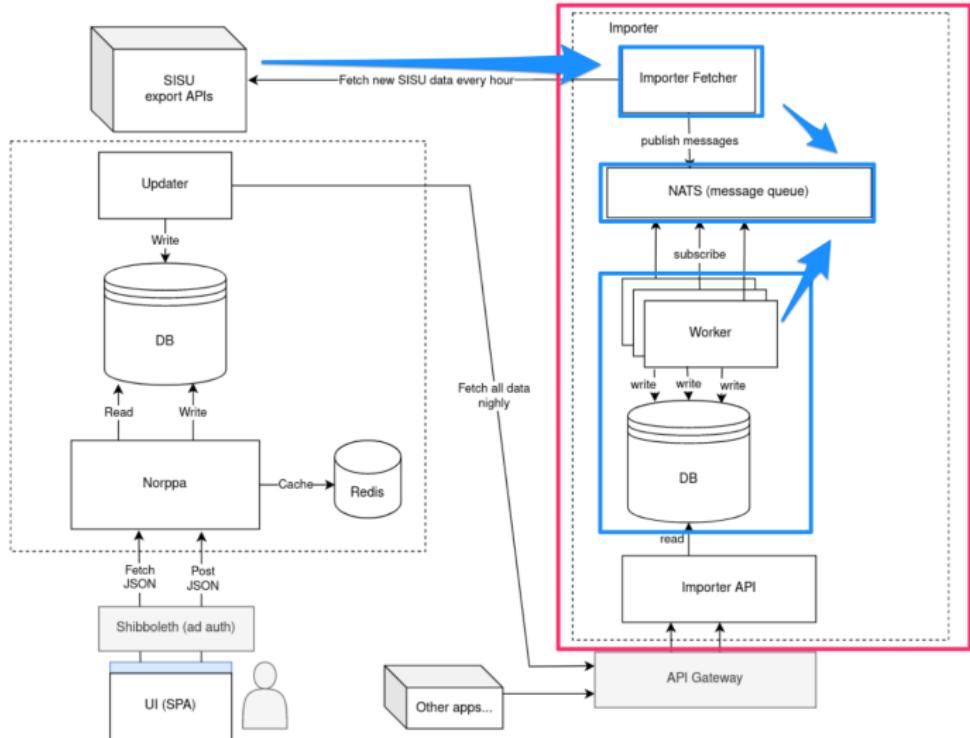
- ▶ Kerrosarkkitehtuuri
- ▶ Mikropalvelu
- ▶ Publish subscribe / Event driven











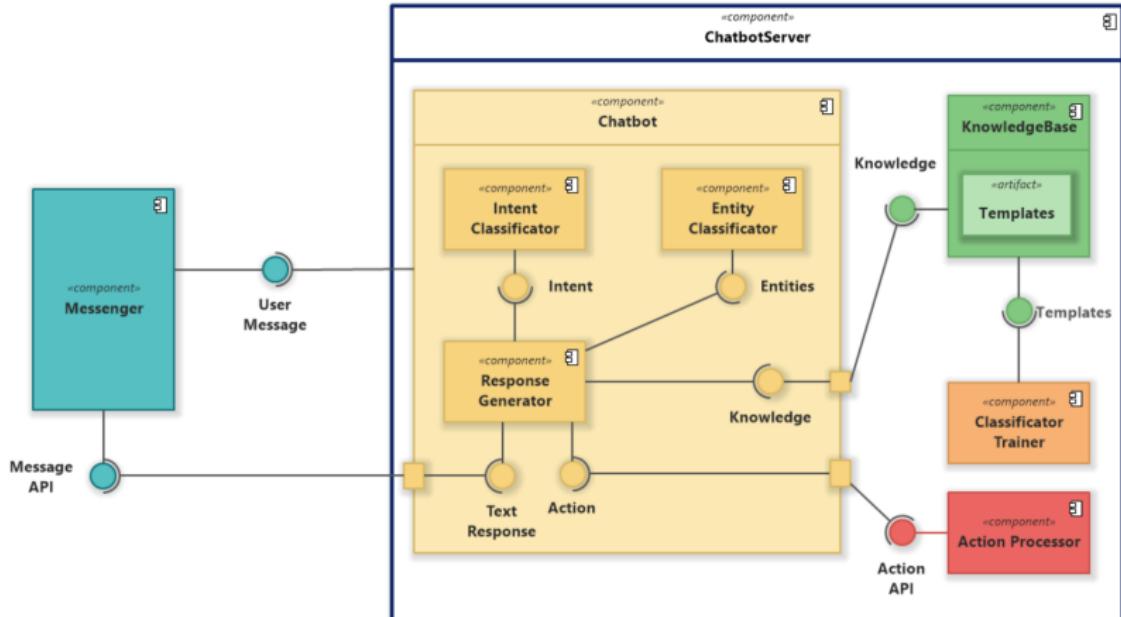
Arkkitehtuurin kuvaamisesta

- ▶ On tilanteita, missä sovelluksen arkkitehtuuri on dokumentoitava jollain tavalla

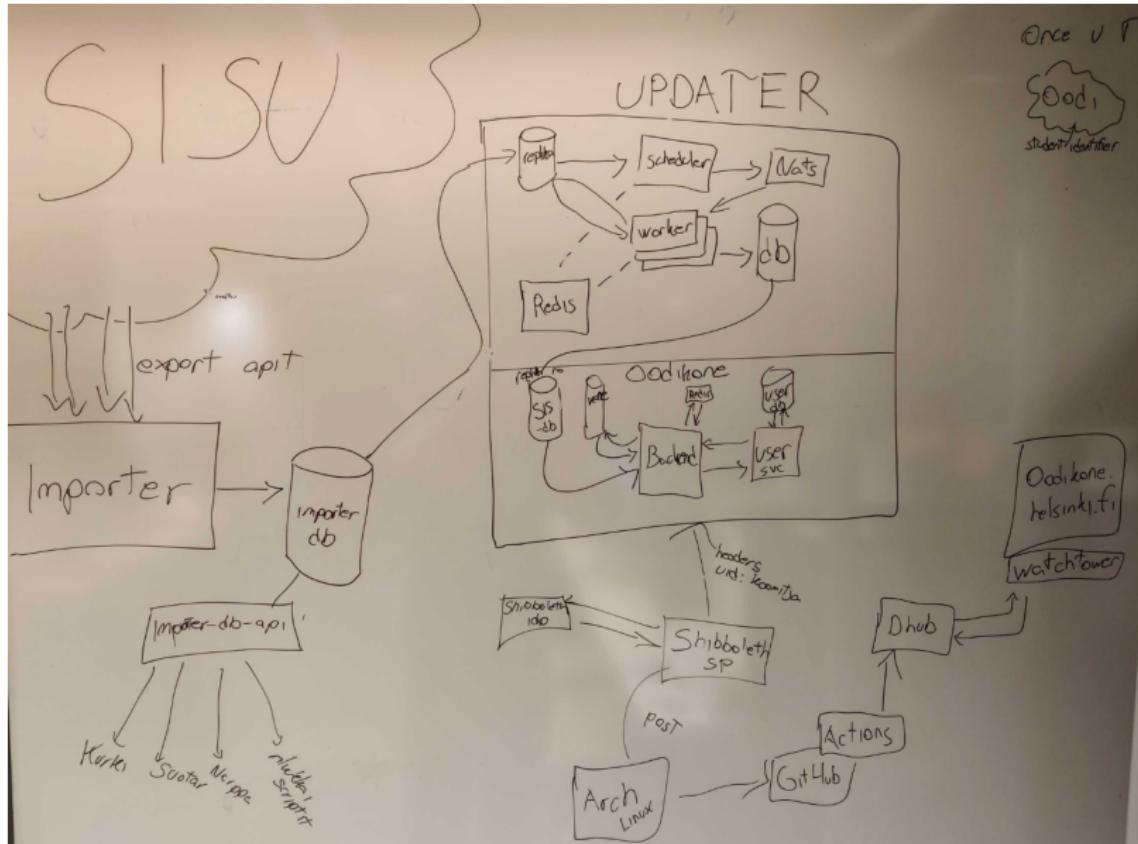
Arkkitehtuurin kuvaamisesta

- ▶ On tilanteita, missä sovelluksen arkkitehtuuri on dokumentoitava jollain tavalla
- ▶ Arkkitehtuurien kuvaamiselle ei olemassa vakiintunutta formaattia
 - ▶ UML:n luokka- ja pakauskaaviot sekä komponentti- ja sijoittelu kaaviot joskus käytökelpoisia
 - ▶ Useimmiten käytetään epäformaaleja laatikko/nuoli-kaavioita

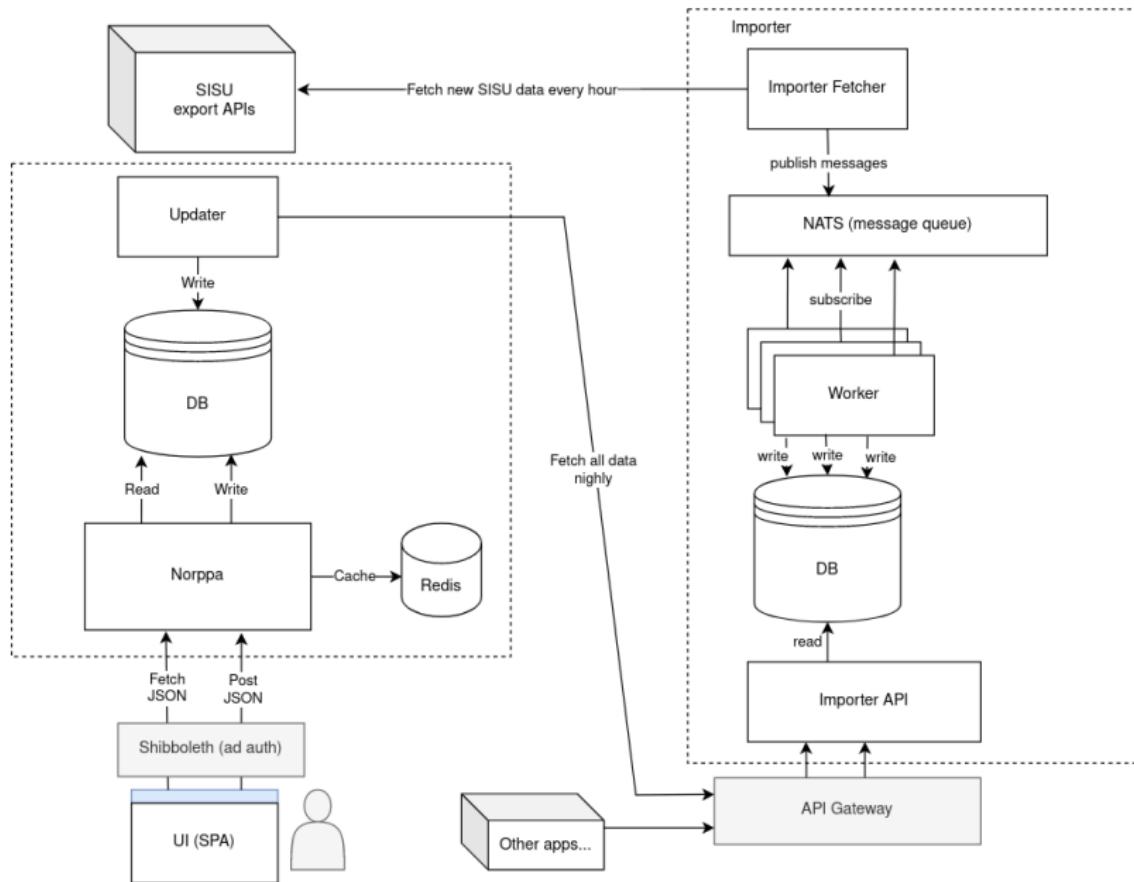
UML komponenttikaavio



Laatikko ja nuoli -kaavio



Hienompi laatikko ja nuoli -kaavio



Arkkitehtuurin kuvaamisesta

- ▶ Arkkitehtuurikuvaus kannattaa tehdä useasta eri tarpeita palvelevasta *näkökulmasta*
 - ▶ korkean tason kuvaksen voi olla hyödyksi esim. vaatimusmäärittelyssä
 - ▶ tarkemmat kuvaukset toimivat ohjeena tarkemmassa suunnittelussa ja ylläpitovaiheen aikaisessa laajentamisessa

Arkkitehtuurin kuvaamisesta

- ▶ Arkkitehtuurikuvaus kannattaa tehdä useasta eri tarpeita palvelevasta *näkökulmasta*
 - ▶ korkean tason kuvaksen 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

Arkkitehtuuri ketterissä menetelmissä

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

Arkkitehtuuri ketterissä menetelmissä

- ▶ Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen
- ▶ Periaatteita
 - ▶ *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*
 - ▶ *Deliver working software frequently...*

Arkkitehtuuri ketterissä menetelmissä

- ▶ Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen
- ▶ Periaatteita
 - ▶ *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*
 - ▶ *Deliver working software frequently...*
- ▶ Ketterät menetelmät suosivat yksinkertaisuutta
 - ▶ *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
- ▶ Periaatteita
 - ▶ *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*
 - ▶ *Deliver working software frequently...*
- ▶ Ketterät menetelmät suosivat yksinkertaisuutta
 - ▶ *Simplicity, the art of maximizing the amount of work not done, is essential*
- ▶ Arkkitehtuuriin suunnittelu ja dokumentointi on perinteisesti pitkäkestoinen, ohjelmoinnin aloittamista edeltävä vaihe

Arkkitehtuuri ketterissä menetelmissä

- ▶ Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen
- ▶ Periaatteita
 - ▶ *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*
 - ▶ *Deliver working software frequently...*
- ▶ Ketterät menetelmät suosivat yksinkertaisuutta
 - ▶ *Simplicity, the art of maximizing the amount of work not done, is essential*
- ▶ Arkkitehtuurin suunnittelu ja dokumentointi on perinteisesti pitkäkestoinen, ohjelmoinnin aloittamista edeltävä vaihe
- ▶ Ketterät menetelmät ja “arkkitehtuurivetoinen” ohjelmistotuotanto siis jossain määrin ristiriidassa

Arkkitehtuuri ketterissä menetelmissä

- ▶ Ketterien menetelmien yhteydessä puhutaan *inkrementaalisesta suunnittelusta ja arkkitehtuurista*

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 uutta toiminnallisuutta toteutetaan

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 uutta toiminnallisuutta toteutetaan
- ▶ Esim. kerrosarkkitehtuurin mukaista sovellusta ei rakenneta “kerros kerrallaan”
 - ▶ Jokaisessa iteraatiossa tehdään pieni pala jokaista kerrostaa, sen verran kuin iteraation tavoitteiden toteuttaminen edellyttää

Ankrementaalinen arkkitehtuuri

- ▶ Alussa ns. *walking skeleton*
 - ▶ sisältää tynkäversiot ohjelmiston komponenttirakenteesta



Ankrementaalinen arkkitehtuuri

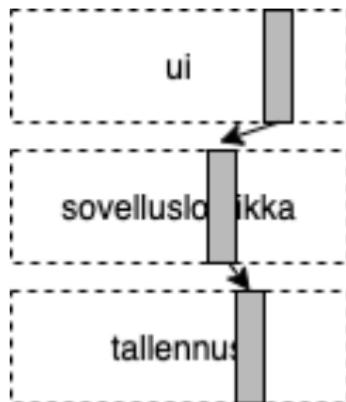
- ▶ Alussa ns. *walking skeleton*
 - ▶ sisältää tynkäversiot ohjelmiston komponenttirakenteesta



- ▶ Rakennetaan skeletonin varaan tuotetta story storyltä

Ominaisuuksiin perustuva integraatio

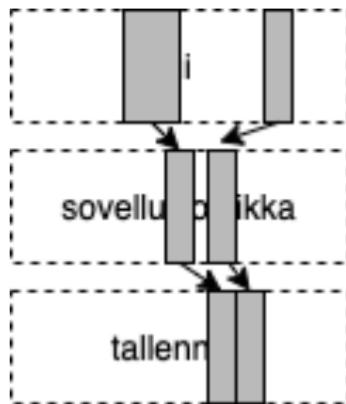
- ▶ Alussa ns. *walking skeleton*
 - ▶ sisältää tynkäversiot ohjelmiston komponenttirakenteesta



- ▶ Rakennetaan skeletonin varaan tuotetta story storyltä

Ominaisuuksiin perustuva integraatio

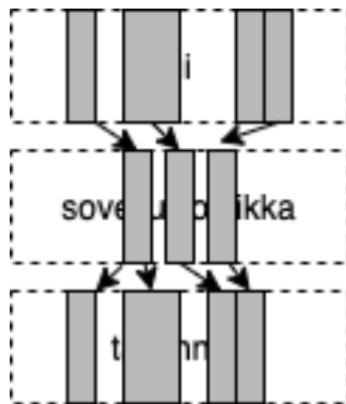
- ▶ Alussa ns. *walking skeleton*
 - ▶ sisältää tynkäversiot ohjelmiston komponenttirakenteesta



- ▶ Rakennetaan skeletonin varaan tuotetta story storyltä

Ominaisuuksiin perustuva integraatio

- ▶ Alussa ns. *walking skeleton*
 - ▶ sisältää tynkäversiot ohjelmiston komponenttirakenteesta



- ▶ Rakennetaan skeletonin varaan tuotetta story storyltä

Arkkitehtuuri ketterissä menetelmissä

- ▶ Perinteisesti arkkitehtuurin luonut *ohjelmistoarkkitehti*
 - ▶ ohjelmoijat velvoitettuja noudattamaan arkkitehtuuria

Arkkitehtuuri ketterissä menetelmissä

- ▶ Perinteisesti arkkitehtuurin luonut *ohjelmistoarkkitehti*
 - ▶ ohjelmoijat velvoitettuja noudattamaan arkkitehtuuria
- ▶ Ketterissä menetelmissä ei suosita erillistä arkkitehdin roolia
 - ▶ Scrumissa kaikista tiimiläisistä käytetään nimikettä developer

Arkkitehtuuri ketterissä menetelmissä

- ▶ Perinteisesti arkkitehtuurin luonut *ohjelmistoarkkitehti*
 - ▶ ohjelmoijat velvoitettuja noudattamaan arkkitehtuuria
- ▶ Ketterissä menetelmissä ei suosita erillistä arkkitehdin roolia
 - ▶ Scrumissa kaikista tiimiläisistä käytetään nimikettä developer
- ▶ Ketterä idealli: kehitystiimi luo arkkitehtuurin yhdessä
 - ▶ *The best architectures, requirements, and designs emerge from self-organizing teams*

Arkkitehtuuri ketterissä menetelmissä

- ▶ Perinteisesti arkkitehtuurin luonut *ohjelmistoarkkitehti*
 - ▶ ohjelmoijat velvoitettuja noudattamaan arkkitehtuuria
- ▶ Ketterissä menetelmissä ei suosita erillistä arkkitehdin roolia
 - ▶ Scrumissa kaikista tiimiläisistä käytetään nimikettä developer
- ▶ Ketterä idealli: kehitystiimi luo arkkitehtuurin yhdessä
 - ▶ *The best architectures, requirements, and designs emerge from self-organizing teams*
- ▶ **Arkkitehtuuri koodin tapaan tiimin yhteisomistama**

Arkkitehtuuri ketterissä menetelmissä

- ▶ Perinteisesti arkkitehtuurin luonut *ohjelmistoarkkitehti*
 - ▶ ohjelmoijat velvoitettuja noudattamaan arkkitehtuuria
- ▶ Ketterissä menetelmissä ei suosita erillistä arkkitehdin roolia
 - ▶ Scrumissa kaikista tiimiläisistä käytetään nimikettä developer
- ▶ Ketterä idealli: kehitystiimi luo arkkitehtuurin yhdessä
 - ▶ *The best architectures, requirements, and designs emerge from self-organizing teams*
- ▶ **Arkkitehtuuri koodin tapaan tiimin yhteisomistama**
- ▶ Etuja:
 - ▶ kehittäjät sitoutuvat paremmin arkkitehtuurin noudattamiseen kuin "norsunluutornissa" olevan arkkitehdin määrittelemään
 - ▶ dokumentaatio voi olla kevyt, tiimi tuntee arkkitehtuurin hengen ja pystyy sitä noudattamaan

Inkrementaalinen arkkitehtuuri: edut ja riskit

- ▶ Oletus: optimaalista arkkitehtuuria ei pystytä suunnittelemaan projektin alussa, kun vaatimuksia, toimintaympäristöä ja toteutusteknologioita ei tunneta
 - ▶ Jo tehtyjä arkkitehtuuriratkaisuja muutetaan tarvittaessa

Inkrementaalinen arkkitehtuuri: edut ja riskit

- ▶ Oletus: optimaalista arkkitehtuuria ei pystytä suunnittelemaan projektin alussa, kun vaatimuksia, toimintaympäristöä ja toteutusteknologioita ei tunneta
 - ▶ Jo tehtyjä arkkitehtuuriratkaisuja muutetaan tarvittaessa
- ▶ Kuten vaatimusmäärittelyssä, myös arkkitehtuurin suunnittelussa ketterä pyrkii välttämään *liian aikaisin tehtävää, myöhemmin ehkä turhaksi osoittautuvaa työtä*

Inkrementaalinen arkkitehtuuri: edut ja riskit

- ▶ Oletus: optimaalista arkkitehtuuria ei pystytä suunnittelemaan projektin alussa, kun vaatimuksia, toimintaympäristöä ja toteutusteknologioita ei tunneta
 - ▶ Jo tehtyjä arkkitehtuuriratkaisuja muutetaan tarvittaessa
- ▶ Kuten vaatimusmäärittelyssä, myös arkkitehtuurin suunnittelussa ketterä pyrkii välittämään *liian aikaisin tehtävää, myöhemmin ehkä turhaksi osoittautuvaa työtä*
- ▶ Inkrementaalinen arkkitehtuuri edellyttää koodilta hyvää sisäistä laatua ja kehittäjiltä kurinalaisuutta
 - ▶ muuten seurauksena on kaaos

TAUKO 10 min

Olio/komponenttisuunnittelu

Olio/komponenttisuunnittelu

- ▶ Sovelluksen arkkitehtuuri antaa raamit, jotka ohjaavat sovelluksen tarkempaa suunnittelua ja toteuttamista

Olio/komponenttisuunnittelu

- ▶ Sovelluksen arkkitehtuuri antaa raamatit, jotka ohjaavat sovelluksen tarkempaa suunnittelua ja toteuttamista
- ▶ *Olio- tai komponenttisuunnittelu*
 - ▶ tarkentaa arkkitehtuuristen komponenttien väliset rajapinnat sekä hahmottelee ohjelman luokka- tai moduulirakenteen

Olio/komponenttisuunnittelu

- ▶ Sovelluksen arkkitehtuuri antaa raamatit, jotka ohjaavat sovelluksen tarkempaa suunnittelua ja toteuttamista
- ▶ *Olio- tai komponenttisuunnittelu*
 - ▶ tarkentaa arkkitehtuuristen komponenttien väliset rajapinnat sekä hahmottelee ohjelman luokka- tai moduulirakenteen
- ▶ Vesiputousmallissa komponenttisuunnittelu tehty ennen ohjelmointia ja dokumentoituu tarkkaan esim. UML:lä

Olio/komponenttisuunnittelu

- ▶ Sovelluksen arkkitehtuuri antaa raamatit, jotka ohjaavat sovelluksen tarkempaa suunnittelua ja toteuttamista
- ▶ *Olio- tai komponenttisuunnittelu*
 - ▶ tarkentaa arkkitehtuuristen komponenttien väliset rajapinnat sekä hahmottelee ohjelman luokka- tai moduulirakenteen
- ▶ Vesiputousmallissa komponenttisuunnittelu tehty ennen ohjelointia ja dokumentoitua tarkkaan esim. UML:lä
- ▶ Ketterässä tarkka suunnittelu tehdään vasta ohjelmoitaessa

Olio/komponenttisuunnittelu

- ▶ Sovelluksen arkkitehtuuri antaa raamatit, jotka ohjaavat sovelluksen tarkempaa suunnittelua ja toteuttamista
- ▶ *Olio- tai komponenttisuunnittelu*
 - ▶ tarkentaa arkkitehtuuristen komponenttien väliset rajapinnat sekä hahmottelee ohjelman luokka- tai moduulirakenteen
- ▶ Vesiputoosmallissa komponenttisuunnittelu tehty ennen ohjelmointia ja dokumentoitu tarkkaan esim. UML:lä
- ▶ Ketterässä tarkka suunnittelu tehdään vasta ohjelmoitaessa
- ▶ Suunnittelussa pyritään maksimoimaan *koodin sisäinen laatu*
 - ▶ helppo ylläpidettävyys ja laajennettavuus

Olio/komponenttisuunnittelu

- ▶ Sovelluksen arkkitehtuuri antaa raamatit, jotka ohjaavat sovelluksen tarkempaa suunnittelua ja toteuttamista
- ▶ *Olio- tai komponenttisuunnittelu*
 - ▶ tarkentaa arkkitehtuuristen komponenttien väliset rajapinnat sekä hahmottelee ohjelman luokka- tai moduulirakenteen
- ▶ Vesiputoosmallissa komponenttisuunnittelu tehty ennen ohjelmointia ja dokumentoitu tarkkaan esim. UML:lä
- ▶ Ketterässä tarkka suunnittelu tehdään vasta ohjelmoitaessa
- ▶ Suunnittelussa pyritään maksimoimaan *koodin sisäinen laatu*
 - ▶ helppo ylläpidettävyys ja laajennettavuus
- ▶ Ohjelmistosuunnittelu on “enemmän taidetta kuin tiedettä”, kokemus ja hyvien käytänteiden tuntemus auttaa
 - ▶ kehitetty monia suunnittelumenetelmiä, mikään niistä ei ole vakiintunut

Laadukas koodi

- ▶ Tavoitteena siis **sisäiseltä laadultaan** hyvä koodi

Laadukas koodi

- ▶ Tavoitteena siis **sisäiseltä laadultaan** hyvä koodi
- ▶ *Sisäinen laatu* (internal quality)
 - ▶ onko virheiden jäljitys ja korjaaminen helppoa
 - ▶ onko koodia helppo laajentaa ja jatkokehittää
 - ▶ pystytäänkö koodin toiminnallisuuden oikeellisuus varmistamaan muutoksia tehtäessä

Laadukas koodi

- ▶ Tavoitteena siis **sisäiseltä laadultaan** hyvä koodi
- ▶ *Sisäinen laatu* (internal quality)
 - ▶ onko virheiden jäljitys ja korjaaminen helppoa
 - ▶ onko koodia helppo laajentaa ja jatkokehittää
 - ▶ pystytäänkö koodin toiminnallisuuden oikeellisuus varmistamaan muutoksia tehtäessä
- ▶ Jos sisäinen laatu rapistuu
 - ▶ alkaa vaikuttamaan myös ulkoiseen eli käyttäjän kokemaan laatuun
 - ▶ kehitystiimin velositeetti alkaa tippua

Laadukkaan koodin tuntomerkejä

Laadukkaan koodin tuntomerkejä

- ▶ Laadukkaalla koodilla joukko yhteneviä ominaisuuksia, tai *laatuattribuutteja*, esim. seuraavat:
 - ▶ kapselointi
 - ▶ korkea koheesion aste
 - ▶ riippuvuuksien vähäisyys
 - ▶ toisteettomuus
 - ▶ testattavuus
 - ▶ selkeys

Laadukkaan koodin tuntomerkkejä

- ▶ Laadukkaalla koodilla joukko yhteneviä ominaisuuksia, tai *laatuattribuutteja*, esim. seuraavat:
 - ▶ kapselointi
 - ▶ korkea koheesion aste
 - ▶ riippuvuuksien vähäisyys
 - ▶ toisteettomuus
 - ▶ testattavuus
 - ▶ selkeys
- ▶ *Suunnittelumallit* auttavat luomaan koodia, joissa sisäinen laatu kunnossa
 - ▶ kurssin aikana nähty jo *dependency injection, repository*
 - ▶ lisää kurssimateriaalissa ja laskareissa

Koodin laatuattribuutti: kapselointi

Koodin laatuattribuutti: kapselointi

- ▶ *Kapselointi ohjelmoinnin peruskursseilla:*
 - ▶ *oliomuuttujat tulee määritellä piilotetuksi ja niille tulee tehdä tarvittaessa aksessorimetodit*

Koodin laatuattribuutti: kapselointi

- ▶ *Kapselointi ohjelmoinnin peruskursseilla:*
 - ▶ *oliomuuttujat tulee määritellä piilotetuksi ja niille tulee tehdä tarvittaessa aksessorimetodit*
- ▶ *Olion sisäisen tilan lisäksi kapseloinnin kohde voi olla mm. käytettävän olion tyyppi, käytetty algoritmi, olioiden luomisen tapa, käytettävän komponentin rakenne*

Koodin laatuatribuutti: kapselointi

- ▶ *Kapselointi ohjelmoinnin peruskursseilla:*
 - ▶ *oliomuuttujat tulee määritellä piilotetuksi ja niille tulee tehdä tarvittaessa aksessorimetodit*
- ▶ Olion sisäisen tilan lisäksi kapseloinnin kohde voi olla mm.
käytettävä olion tyyppi, käytetty algoritmi, olioiden luomisen tapa, käytettävä komponentin rakenne
- ▶ Näkyy myös arkkitehtuurin tasolla
 - ▶ kerrosarkkitehtuuri: ylempi kerros käyttää ainoastaan alemman kerroksen ulospäin tarjoamaa rajapintaa, muu kapseloitu
 - ▶ mikropalvelut: yksittäinen palvelu kapseloi sisäisen logiikan, tiedon säilytystavan ja tarjoaa ainoastaan verkon välityksellä käytettävän rajapinnan

Koodin laatuattribuutti: koheesio

Koodin laatuatribuutti: koheesio

- ▶ *Koheesio:*

- ▶ kuinka pitkälle metodin, luokan tai komponentin koodi keskittyy tietyn yksittäisen toiminnallisuuden toteuttamiseen
- ▶ hyvänä pidetään mahdollisimman korkeaa koheesion astetta

Koodin laatuatribuutti: koheesio

- ▶ *Koheesio:*
 - ▶ kuinka pitkälle metodin, luokan tai komponentin koodi keskittyy tietyn yksittäisen toiminnallisuuden toteuttamiseen
 - ▶ hyvänä pidetään mahdollisimman korkeaa koheesion astetta
- ▶ Luokkataslon koheesio
 - ▶ luokan *vastuulla* vain yksi asia, tunnetaan myös nimellä *single responsibility principle*

Koodin laatuatribuutti: koheesio

- ▶ *Koheesio:*
 - ▶ kuinka pitkälle metodin, luokan tai komponentin koodi keskittyy tietyn yksittäisen toiminnallisuuden toteuttamiseen
 - ▶ hyvänä pidetään mahdollisimman korkeaa koheesion astetta
- ▶ Luokkataslon koheesio
 - ▶ luokan *vastuulla* vain yksi asia, tunnetaan myös nimellä *single responsibility principle*
- ▶ Arkkitehtuurin tasolla
 - ▶ kerrosarkkitehtuurin kerrokset samalla abstraktiotasolla, esim. käyttöliittymä tai tietokanttarajapinta
 - ▶ mikropalvelu toteuttaa tiettyyn liiketoiminnan tason toiminnallisuuden, esim. suosittelualgoritmin tai käyttäjien hallinnan

Metoditason koheesio

```
def populate(self):
    connection = sqlite3.connect(DATABASE_FILE_PATH)
    connection.row_factory = sqlite3.Row

    cursor = connection.cursor()
    cursor.execute(SQL_SELECT_PARTS)
    rows = cursor.fetchall()

    parts = []

    for row in rows:
        parts.append(Part(row["name"], row["brand"], row["retail_price"]))

    connection.close()

    return parts
```

Metoditason koheesio

```
def populate(self):
    connection = sqlite3.connect(DATABASE_FILE_PATH)
    connection.row_factory = sqlite3.Row

    cursor = connection.cursor()
    cursor.execute(SQL_SELECT_PARTS)
    rows = cursor.fetchall()

    parts = []

    for row in rows:
        parts.append(Part(row["name"], row["brand"], row["retail_price"]))

    connection.close()

    return parts
```

- ▶ metodi tekee kolmea eri asiaa, jotka eri abstraktiotasolla

Metoditason koheesio

```
def populate(self):
    connection = self.get_database_connection()
    rows = self.get_rows(connection)
    parts = self.get_parts_by_rows(rows)
    connection.close()
    return parts

def get_database_connection(self):
    connection = sqlite3.connect(DATABASE_FILE_PATH)
    connection.row_factory = sqlite3.Row
    return connection

def get_rows(self, connection):
    cursor = connection.cursor()
    cursor.execute(SQL_SELECT_PARTS)
    return cursor.fetchall()

def get_parts_by_rows(self, rows):
    parts = []
    for row in rows:
        parts.append(Part(row["name"], row["brand"], row["retail_price"]))
    return parts
```

Luokkataslon koheesi

- ▶ Single responsibility -periaate: *luokalla yksi syy muuttua*

Luokkataslon koheesio

- ▶ Single responsibility -periaate: *luokalla yksi syy muuttua*

```
class Laskin:  
    def __init__(self):  
        self.lue = input  
        self.kirjoita = print  
  
    def suorita(self):  
        while True:  
            luku1 = int(self.lue("Luku 1:"))  
  
            if luku1 == -9999:  
                return  
  
            luku2 = int(self.lue("Luku 2:"))  
  
            if luku2 == -9999:  
                return  
  
            vastaus = self.laske_summa(luku1, luku2)  
  
            self.kirjoita(f"Summa: {vastaus}")  
  
    def laske_summa(self, luku1, luku2):  
        return luku1 + luku2
```

Luokkataslon koheesio

```
class Laskin:
    def __init__(self, io):
        self.io = io

    def suorita(self):
        while True:
            luku1 = int(self.io.lue("Luku 1:"))

            if luku1 == -9999:
                return

            luku2 = int(self.io.lue("Luku 2:"))

            if luku2 == -9999:
                return

            vastaus = self.laske_summa(luku1, luku2)

            self.io.kirjoita(f"Summa: {vastaus}")

    def laske_summa(self, luku1, luku2):
        return luku1 + luku2
```

Luokkataslon koheesio

```
class Laskin:
    def __init__(self, io):
        self.io = io

    def suorita(self):
        while True:
            luku1 = int(self.io.lue("Luku 1:"))

            if luku1 == -9999:
                return

            luku2 = int(self.io.lue("Luku 2:"))

            if luku2 == -9999:
                return

            vastaus = self.laske_summa(luku1, luku2)

            self.io.kirjoita(f"Summa: {vastaus}")

    def laske_summa(self, luku1, luku2):
        return luku1 + luku2
```

- delegoidaan osa vastuista eri luokalle

Koodin laatuattribuutti: riippuvuuksien vähäisyys

Koodin laatuattribuutti: riippuvuuksien vähäisyys

- ▶ Pyrkimys korkeaan koheesioon johtaa ohjelmiin, joissa suuri määrä olioita/komponentteja

Koodin laatuattribuutti: riippuvuuksien vähäisyys

- ▶ Pyrkimys korkeaan koheesioon johtaa ohjelmiin, joissa suuri määrä olioita/komponentteja
- ▶ Olioiden oltava keskenään vuorovaikutuksessa toteuttaakseen ohjelman toiminnallisuuden: *paljon keskinäisiä riippuvuuksia*

Koodin laatuatribuutti: riippuvuuksien vähäisyys

- ▶ Pyrkimys korkeaan koheesioon johtaa ohjelmiin, joissa suuri määrä olioita/komponentteja
- ▶ Olioiden oltava keskenään vuorovaikutuksessa toteuttaakseen ohjelman toiminnallisuuden: *paljon keskinäisiä riippuvuuksia*
- ▶ *Riippuvuuksien vähäisyyden* periaate
 - ▶ eliminoidaan *tarpeettomat* riippuvuudet
 - ▶ sekä riippuvuudet konkreettisiin asioihin

Koodin laatuatribuutti: riippuvuuksien vähäisyys

- ▶ Pyrkimys korkeaan koheesioon johtaa ohjelmiin, joissa suuri määrä olioita/komponentteja
- ▶ Olioiden oltava keskenään vuorovaikutuksessa toteuttaakseen ohjelman toiminnallisuuden: *paljon keskinäisiä riippuvuuksia*
- ▶ *Riippuvuuksien vähäisyyden periaate*
 - ▶ eliminoidaan *tarpeettomat* riippuvuudet
 - ▶ sekä riippuvuudet konkreettisiin asioihin
- ▶ Hyödynnetään *dependence injection* -suunnittelumallia

Koodin laatuatribuutti: riippuvuuksien vähäisyys

```
def main():
    stats = StatisticsService()

class StatisticsService:
    def __init__(self):
        reader = PlayerReader()
```

VS

```
def main():
    reader = PlayerReader("https://studies.cs.helsinki.fi/nhlstats/2021-22/players.txt")
    stats = StatisticsService(reader)

class StatisticsService:
    def __init__(self, reader):
        self._players = reader.get_players()
```

Koodin laatuattribuutti: toisteettomuus

Koodin laatuattribuutti: toisteettomuus

- ▶ Aloittelevaa ohjelmoijaa pelotellaan toisteisuuden vaaroista uran ensiaskelista alkaen: älä copypastaa koodia!

Koodin laatuattribuutti: toisteettomuus

- ▶ 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*

Koodin laatuattribuutti: toisteettomuus

- ▶ 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*
- ▶ Koodin lisäksi periaate ulottuu koskemaan järjestelmän muitakin osia
 - ▶ tietokantaskeemaa, testejä, build-skriptejä

Koodin laatuattribuutti: toisteettomuus

- ▶ 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*
- ▶ Koodin lisäksi periaate ulottuu koskemaan järjestelmän muitakin osia
 - ▶ tietokantaskeemaa, testejä, build-skriptejä
- ▶ Suoraviivainen copypaste helppo eliminoida metodien avulla
 - ▶ kaikki toisteisuus ei ole yhtä ilmeistä, monissa suunnittelumalleissa kyse hienovaraisempien toisteisuuden muotojen eliminoinnista

Koodin laatuatribuutti: toisteettomuus

- ▶ 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*
- ▶ Koodin lisäksi periaate ulottuu koskemaan järjestelmän muitakin osia
 - ▶ tietokantaskeemaa, testejä, build-skriptejä
- ▶ Suoraviivainen copypaste helppo eliminoida metodien avulla
 - ▶ kaikki toisteisuus ei ole yhtä ilmeistä, monissa suunnittelumalleissa kyse hienovaraisempien toisteisuuden muotojen eliminoinnista
- ▶ Hyvä vs. paha copypaste
 - ▶ *three strikes and you refactor*

Koodin laatuattribuutti: testattavuus

Koodin laatuatribuutti: testattavuus

- ▶ Laadukas koodi on helppo testata kattavasti yksikkö- ja integraatiotestein
 - ▶ seuraa yleensä siitä, että koodi koostuu löyhästi kytketyistä, selkeän vastuun omaavista komponenteista

Koodin laatuattribuutti: testattavuus

- ▶ Laadukas koodi on helppo testata kattavasti yksikkö- ja integraatiotestein
 - ▶ seuraa yleensä siitä, että koodi koostuu löyhästi kytketyistä, selkeän vastuun omaavista komponenteista
- ▶ Hyvää testattavuutta auttaa turhien riippuvuuksien eliminointi dependency injection -periaatteen avulla

Koodin laatuatribuutti: testattavuus

- ▶ Laadukas koodi on helppo testata kattavasti yksikkö- ja integraatiotestein
 - ▶ seuraa yleensä siitä, että koodi koostuu löyhästi kytketyistä, selkeän vastuun omaavista komponenteista
- ▶ Hyvää testattavuutta auttaa turhien riippuvuuksien eliminointi dependency injection -periaatteen avulla
- ▶ Test driven development tuottaa varmuudella hyvin testattavissa olevaa koodia

Koodin laatuatribuutti: selkeys ja luettavuus

- ▶ Perinteisesti ajateltu että koodi kryptistä ja vaikeasti luettavaa
 - ▶ yleistä C-kielessä, pyritty esim. optimoimaan tehokkuutta ja muistinkäyttöä

Koodin laatuatribuutti: selkeys ja luettavuus

- ▶ Perinteisesti ajateltu että koodi kryptistä ja vaikeasti luettavaa
 - ▶ yleistä C-kielessä, pyritty esim. optimoimaan tehokkuutta ja muistinkäyttöä
- ▶ Nykytrendi: tehdän koodia, joka nimeämisen sekä rakenteen kautta ilmaisee hyvin sen, mitä koodi tekee

Koodin laatuatribuutti: selkeys ja luettavuus

- ▶ Perinteisesti ajateltu että koodi kryptistä ja vaikeasti luettavaa
 - ▶ yleistä C-kielessä, pyritty esim. optimoimaan tehokkuutta ja muistinkäyttöä
- ▶ Nykytrendi: tehdän koodia, joka nimeämisen sekä rakenteen kautta ilmaisee hyvin sen, mitä koodi tekee
- ▶ Miksi selkeää koodi on tärkeää?
 - ▶ joidenkin arvioiden mukaan jopa 90% "ohjelointiin" kuluvasta ajasta menee olemassa olevan koodin lukemiseen
 - ▶ oma aikoinaan niin selkeää koodi, ei enää olekaan yhtä selkeää parin kuukauden kuluttua

Code smell

- ▶ Koodi ei ole aina hyvää...

Code smell

- ▶ Koodi ei ole aina hyvää...
- ▶ 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

- ▶ Koodi ei ole aina hyvää...
- ▶ 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:
 - ▶ toisteenen koodi
 - ▶ liian pitkät metodit
 - ▶ luokat joissa on liikaa oliomuuttujia
 - ▶ luokat joissa on liikaa koodia
 - ▶ metodien liian pitkät parametristat
 - ▶ epäselkeät muuttujien, metodien tai luokkien nimet
 - ▶ kommentit

Koodihajuja

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

Refaktoriointi

Refaktoriointi

- ▶ Lääke koodin sisäisen laadun ongelmiin on *refaktoriointi*
 - ▶ koodin toiminnalisuuden ennallaan pitävä sisäisen rakenteen muutos

Refaktoriointi

- ▶ Lääke koodin sisäisen laadun ongelmiin on *refaktoriointi*
 - ▶ koodin toiminnalisuuden ennallaan pitävä sisäisen rakenteen muutos
- ▶ Koodin rakennetta parantavia refaktorointeja on lukuisia, mm.
 - ▶ *rename variable/method/class*
 - ▶ *extract method*
 - ▶ *move field/method*
 - ▶ *extract superclass*

Refaktoriointi

- ▶ Lääke koodin sisäisen laadun ongelmiin on *refaktoriointi*
 - ▶ koodin toiminnalisuuden ennallaan pitävä sisäisen rakenteen muutos
- ▶ Koodin rakennetta parantavia refaktorointeja on lukuisia, mm.
 - ▶ *rename variable/method/class*
 - ▶ *extract method*
 - ▶ *move field/method*
 - ▶ *extract superclass*
- ▶ Osa pystytään tekemään sovelluskehitysympäristön avustamana
 - ▶ helpompaa staattisesti tyypitetyillä kielillä kuten Java

Miten refaktoriointi kannattaa tehdä

- ▶ Refaktorioidun edellytys on kattavien testien olemassaolo

Miten refaktoriointi kannattaa tehdä

- ▶ Refaktorioidin edellytys on kattavien testien olemassaolo
- ▶ Kannattaa ehdottomasti edetä pienin askelin
 - ▶ yksi hallittu muutos kerrallaan
 - ▶ testit suoritettava mahdollisimman usein

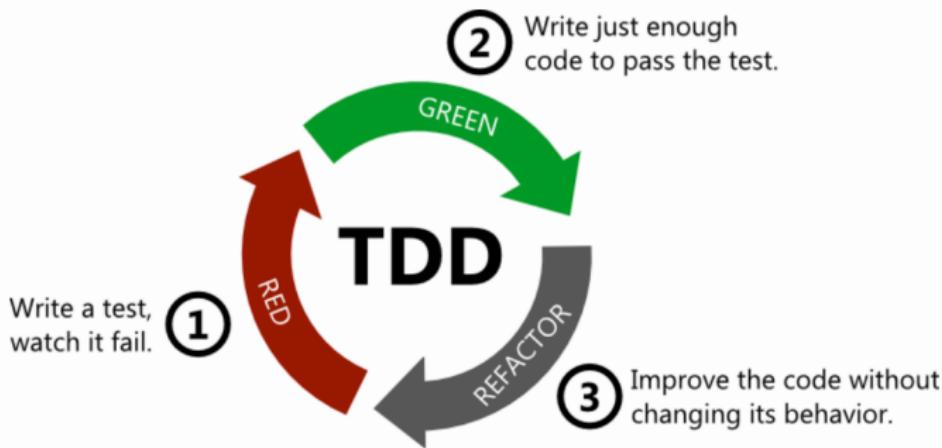
Miten refaktoriointi kannattaa tehdä

- ▶ Refaktorioiden edellytys on kattavien testien olemassaolo
- ▶ Kannattaa ehdottomasti edetä pienin askelin
 - ▶ yksi hallittu muutos kerrallaan
 - ▶ testit suoritettava mahdollisimman usein
- ▶ Refaktoriointia kannattaa suorittaa lähes jatkuvasti
 - ▶ pitää koodin rakenteen selkeänä ja helpottaa sekä nopeuttaa koodin laajentamista

Miten refaktoriointi kannattaa tehdä

- ▶ Refaktorioidin edellytys on kattavien testien olemassaolo
- ▶ Kannattaa ehdottomasti edetä pienin askelin
 - ▶ yksi hallittu muutos kerrallaan
 - ▶ testit suoritettava mahdollisimman usein
- ▶ Refaktoriointia kannattaa suorittaa lähes jatkuvasti
 - ▶ pitää koodin rakenteen selkeänä ja helpottaa sekä nopeuttaa koodin laajentamista
- ▶ Osa refaktorioidista on helppoa ja suoraviivaista, aina ei näin ole
 - ▶ joskus tarve tehdä isoja, jopa viikkojen kestoisia refaktointeja joissa ohjelman rakenne muuttuu paljon

Refaktoriointi tärkeä osa Test driven development -menetelmää



1. Kirjoitetaan sen verran testiä että testi ei mene läpi
2. Kirjoitetaan koodia sen verran, että testi menee läpi
3. **Jos huomataan koodin rakenteen menneen huonoksi refaktoroidaan koodin rakenne paremmaksi**
4. Jatketaan askeleesta 1

15.1.2024-

Avoin yliopisto: Test-Driven Development 4 + 1 cr

- ▶ Esko Luontola Nitor (Suomen johtava TDD-asiantuntija)

Tekninen velka

Tekninen velka

- ▶ Koodi ei ole aina laadultaan optimaalista

Tekninen velka

- ▶ Koodi ei ole aina laadultaan optimaalista
- ▶ Huonoa suunnittelua tai/ja ohjelointia kuvaavat käsitykset *tekninen velka* (technical debt)

Tekninen velka

- ▶ Koodi ei ole aina laadultaan optimaalista
- ▶ Huonoa suunnittelua tai/ja ohjelointia kuvaa käsite *tekninen velka* (technical debt)
- ▶ Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain
 - ▶ hätäinen ratkaisu tullaan maksamaan korkoineen takaisin *jos* ohjelmaa on tarkoitus laajentaa

Tekninen velka

- ▶ Koodi ei ole aina laadultaan optimaalista
- ▶ Huonoa suunnittelua tai/ja ohjelointia kuvaa käsite *tekninen velka* (technical debt)
- ▶ Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain
 - ▶ 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
- ▶ Huonoa suunnittelua tai/ja ohjelointia kuvaa käsite *tekninen velka* (technical debt)
- ▶ Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain
 - ▶ 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 voi olla järkevää tai jopa välttämätöntä
 - ▶ voidaan saada tuote nopeammin markkinoille tekemällä tietoisesti huonoa designia, joka korjataan myöhemmin

- ▶ Kaikki tekninen velka ei samanlaista, taustalla voi olla
 - ▶ holtittomuus, osaamattomuus, tietämättömyys tai tarkoituksella tehty päätös

- ▶ Kaikki tekninen velka ei samanlaista, taustalla voi olla
 - ▶ holtittomuus, osaamattomuus, tietämättömyys tai tarkoituksella tehty päätös
- ▶ Martin Fowler jaottelee teknisen velan neljään eri luokkaan:
 - ▶ Reckless and deliberate: “we do not have time for design”
 - ▶ Reckless and inadvertent: “what is layering”?
 - ▶ Prudent and inadvertent: “now we know how we should have done it”
 - ▶ **Prudent and deliberate: “we must ship now and will deal with consequences”**

- ▶ Kaikki tekninen velka ei samanlaista, taustalla voi olla
 - ▶ holtittomuus, osaamattomuus, tietämättömyys tai tarkoituksella tehty päätös
- ▶ Martin Fowler jaottelee teknisen velan neljään eri luokkaan:
 - ▶ Reckless and deliberate: “we do not have time for design”
 - ▶ Reckless and inadvertent: “what is layering”?
 - ▶ Prudent and inadvertent: “now we know how we should have done it”
 - ▶ **Prudent and deliberate: “we must ship now and will deal with consequences”**
- ▶ Joskus tekninen velka pakottaa koodaamaan koko järjestelmän uudelleen

Ohjelmiston elinkaaren vaiheet

Ohjelmiston elinkaaren vaiheet

- ▶ Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekemiseen kuuluu
 - ▶ vaatimusten analysointi ja määrittely
 - ▶ suunnittelu
 - ▶ toteutus
 - ▶ testaus/laadunhallinta
 - ▶ **ohjelmiston ylläpito**

Ohjelmiston elinkaaren vaiheet

- ▶ Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekemiseen kuuluu
 - ▶ vaatimusten analysointi ja määrittely
 - ▶ suunnittelu
 - ▶ toteutus
 - ▶ testaus/laadunhallinta
 - ▶ **ohjelmiston ylläpito**
- ▶ Ohjelmistot ovat suurimman osan elinajastaan ylläpitovaiheessa

Ohjelmiston elinkaaren vaiheet

- ▶ Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekemiseen kuuluu
 - ▶ vaatimusten analysointi ja määrittely
 - ▶ suunnittelu
 - ▶ toteutus
 - ▶ testaus/laadunhallinta
 - ▶ **ohjelmiston ylläpito**
- ▶ Ohjelmistot ovat suurimman osan elinajastaan ylläpitovaiheessa
- ▶ Jos ensimmäinen versio julkaistaan nopeasti, ovat ketterät ohjelmistoprojektit “jatkuvassa” ylläpitovaiheessa

Muutama ylläpitovaiheen kannalta oleellinen asia

- ▶ Backupit

Muutama ylläpitovaiheen kannalta oleellinen asia

- ▶ Backupit
- ▶ Sovelluksen lokit
- ▶ Analytiikka
- ▶ Virheiden monitorointi

Sovelluksen lokit

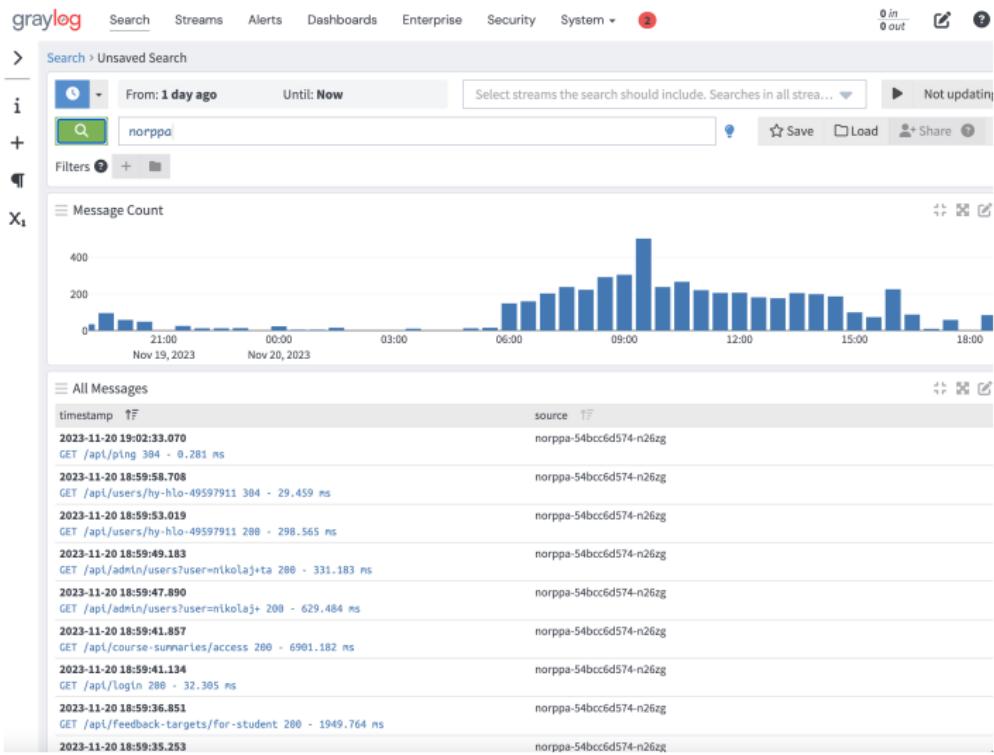
- Sovellusten tulee tulostaa lokiviestejä erilaisista mielenkiintoisista tilanteista

```
80      const processingTime = (Date.now() - processingStart).toFixed(0)
81      const totalTime = (Date.now() - loopStart).toFixed(0)
82      loopStart = Date.now()
83
84      logger.debug(`[UPDATERLOOP]`, {
85        url,
86        offset,
87        items: currentData.length,
88        requestTime,
89        processingTime,
90        totalTime,
91      })
92
93      count += currentData.length
94      offset += limit
95
96    } catch (e) {
97      if (!e.isLoggedIn) {
98        logError('Unknown updaterloop error:', e)
99      }
100    }
101  }
102
103  const duration = Date.now() - start
104  logger.info(
105    `[UPDATER] Updated ${count} items at ${((duration / count).toFixed(
106      4,
107    ))ms/item}, total time ${((duration / 1000).toFixed(2))s}`,
108  )
109
110 module.exports = mangleData
```

func mangleData

Sovelluksen lokit: Graylog

- Sovellusten lokit tulee kerätä paikkaan, mistä niitä helppo tarkastella



Analytiikka: Grafana

► Lokeja voidaan hyödyntää erilaiseen analytiikkaan



Virheiden monitorointi: Sentry

► Virhetilanteista voidaan muodostaa hälytyksiä

norppa-sentry

+ Add a bookmark

Sentry APP 3:47 PM Today

Error

Request failed with status code 504

PALAUTE-6 via Send a notification for new issues | Today at 3:47 PM

Resolve... Ignore Select Assignee...

Sentry APP 4:38 PM

TypeError

(n=g.slice()).splice is not a function. (In '(n=g.slice()).splice(r,1);' '(n=g.slice()).splice' is undefined)

PALAUTE-G via Send a notification for new issues | Today at 4:38 PM

Resolve... Ignore Select Assignee...

Sentry APP 6:20 PM

N+1 Query

```
db - INSERT INTO "feedback_target_logs"  
("id","data","feedback_target_id","user_id","created_at","updated_at") VALUES  
(DEFAULT,$1,$2,$3,$4,$5) RETURNING  
"id","data","feedback_target_id","user_id","created_at","updated_at","user_id","feedback_tar  
get_id";
```

PALAUTE-19 via Send a notification for new issues | Today at 6:19 PM

Resolve... Ignore Select Assignee...

Sentry APP 6:42 PM

ApplicationError

Virheiden monitorointi: Sentry

Issues > PALAUTTE-G

TypeError handleChange(@mui/material/ToggleButtonGroup/ToggleButtonGroup)

● (n=g.slice()).splice is not a function. (In 'n=g.slice().splice(r,1);', 'n=g.slice()' is undefined)

Details Activity User Feedback Attachments Tags All Events Merged Issues Replays

Event ID: d7980f80 Nov 20, 2:38 PM

Tags: @helsinki.fi, Mobile Safari Version: 17.1.1, iOS Version: 17.1.1, iPhone Model: iPhone

browser: Mobile Safari 17.1.1, browser.name: Mobile Safari, device: iPhone, device.family: iPhone, environment: production, handled: yes, level: error, mechanism: instrument, os: iOS 17.1.1, os.name: iOS, release: e3906c94a555, url: https://norppa.helsinki.fi/course-summary, user: id:hy-hlo-1456049

Stack Trace

TypeError
(n=g.slice()).splice is not a function. (In 'n=g.slice().splice(r,1)', 'n=g.slice()' is undefined)

mechanism: instrument, handled: true, function: addEventListener, handler: bound Ht, target: EventTarget

JS
./node_modules/@mui/material/ToggleButtonGroup/ToggleButtonGroup.js in handleChange at line 117:16

```
112     }
113     const index = value && value.indexOf(buttonValue);
114     let newValue;
115     if (value && index >= 0) {
116       newValue = value.slice();
117       newValue.splice(index, 1);
118     } else {
119       newValue = value ? value.concat(buttonValue) : [buttonValue];
120     }
121     onChange(event, newValue);
122   };

```

./node_modules/@mui/material/ToggleButton/ToggleButton.js in <object>.onClick at line 123:15

Loppupäätelmiä testauksesta

Loppupäätelmiä testauksesta

- ▶ Ketterissä menetelmissä kantavana teemana on *arvon tuottaminen asiakkaalle*
 - ▶ Sopii ohjeeksi myös arvioitaessa testauksen laajuutta
 - ▶ Testauksella ei ole itseisarvoista merkitystä
 - ▶ Testaamattomuus alkaa pian heikentää tuotteen laatua liikaa

Loppupäätelmiä testauksesta

- ▶ Ketterissä menetelmissä kantavana teemana on *arvon tuottaminen asiakkaalle*
 - ▶ Sopii ohjeeksi myös arvioitaessa testauksen laajuutta
 - ▶ Testauksella ei ole itseisarvoista merkitystä
 - ▶ Testaamattomuus alkaa pian heikentää tuotteen laatua liikaa
- ▶ Testausta ja laadunhallintaa on tehtävä paljon ja toistuvasti
 - ▶ automatisointi on yleensä pidemmällä tähtäimellä kannattavaa

Loppupäätelmiä testauksesta

- ▶ Ketterissä menetelmissä kantavana teemana on *arvon tuottaminen asiakkaalle*
 - ▶ Sopii ohjeeksi myös arvioitaessa testauksen laajuutta
 - ▶ Testauksella ei ole itseisarvoista merkitystä
 - ▶ Testaamattomuus alkaa pian heikentää tuotteen laatua liikaa
- ▶ Testausta ja laadunhallintaa on tehtävä paljon ja toistuvasti
 - ▶ automatisointi on yleensä pidemmällä tähtäimellä kannattavaa
- ▶ Automatisointi ei ole halpaa eikä helppoa
 - ▶ Väärin, väärään aikaan tai väärälle tasolle tehdyt automatisoidut testit voivat tuottaa enemmän harmia ja kustannuksia kuin hyötyä

- ▶ Jos ohjelmistossa komponentteja, jotka tullaan poistamaan tai korvaamaan, ei niiden testejä kannata automatisoida
 - ▶ esim. jos kyseessä *minimal viable product*

- ▶ Jos ohjelmistossa komponentteja, jotka tullaan poistamaan tai korvaamaan, ei niiden testejä kannata automatisoida
 - ▶ esim. jos kyseessä *minimal viable product*
- ▶ Väliaikaiseksi tarkoitettu komponentti voi jäädä järjestelmään vuosiksi...

- ▶ Jos ohjelmistossa komponentteja, jotka tullaan poistamaan tai korvaamaan, ei niiden testejä kannata automatisoida
 - ▶ esim. jos kyseessä *minimal viable product*
- ▶ Väliaikaiseksi tarkoitettu komponentti voi jäädä järjestelmään vuosiksi...
- ▶ Aluksi kannattaa ohjelman rakenteen ensin antaa stabiloitua, kattavammat testit vasta myöhemmin

- ▶ Jos ohjelmistossa komponentteja, jotka tullaan poistamaan tai korvaamaan, ei niiden testejä kannata automatisoida
 - ▶ esim. jos kyseessä *minimal viable product*
- ▶ Väliaikaiseksi tarkoitettu komponentti voi jäädä järjestelmään vuosiksi...
- ▶ Aluksi kannattaa ohjelman rakenteen ensin antaa stabiloitua, kattavammat testit vasta myöhemmin
- ▶ *Testattavuus* tulee pitää koko ajan mielessä

- ▶ Kattavien yksikkötestien tekeminen ei yleensä ole mielekästä ohjelman kaikille luokille

- ▶ Kattavien yksikkötestien tekeminen ei yleensä ole mielekästä ohjelman kaikille luokille
- ▶ Yksikkötestaus hyödyllisimmillään kompleksia logiikkaa sisältäviä luokkia testattaessa

- ▶ Kattavien yksikkötestien tekeminen ei yleensä ole mielekästä ohjelman kaikille luokille
- ▶ Yksikkötestaus hyödyllisimmillään kompleksia logiikkaa sisältäviä luokkia testattaessa
- ▶ Mielummin integraatiotason testejä ohjelman isompien komponenttien rajapintoja vasten
 - ▶ Pysyvät todennäköisemmin valideina komponenttien sisäisen rakenteen muuttuessa

- ▶ Kattavien yksikkötestien tekeminen ei yleensä ole mielekästä ohjelman kaikille luokille
- ▶ Yksikkötestaus hyödyllisimmillään kompleksia logiikkaa sisältäviä luokkia testattaessa
- ▶ Mielummin integraatiotason testejä ohjelman isompien komponenttien rajapintoja vasten
 - ▶ Pysyvät todennäköisemmin valideina komponenttien sisäisen rakenteen muuttuessa
- ▶ Käyttöliittymän läpi suoritettavat, käyttäjän interaktiota simuloivat testit usein hyödyllisimpia
 - ▶ Liian aikaisin tehtynä ne saattavat aiheuttaa kohtuuttoman paljon ylläpitovaivaa

- ▶ Testitapauksista kannattaa aina tehdä todellisia käyttöskenaarioita vastaavia
 - ▶ Pelkkiä testauskattavuutta kasvattavia testejä on turha tehdä

- ▶ Testitapauksista kannattaa aina tehdä todellisia käyttöskenaarioita vastaavia
 - ▶ Pelkkiä testauskattavuutta kasvattavia testejä on turha tehdä
- ▶ Erityisesti järjestelmätason testeissä kannattaa käyttää mahdollisimman oikeanlaista dataa
 - ▶ Koodissa hajoaa aina jotain kun käytetään oikeaa dataa riippumatta siitä miten hyvin testaus on suoritettu

- ▶ Testitapauksista kannattaa aina tehdä todellisia käyttöskenaarioita vastaavia
 - ▶ Pelkkiä testauskattavuutta kasvattavia testejä on turha tehdä
- ▶ Erityisesti järjestelmätason testeissä kannattaa käyttää mahdollisimman oikeanlaista dataa
 - ▶ Koodissa hajoaa aina jotain kun käytetään oikeaa dataa riippumatta siitä miten hyvin testaus on suoritettu
- ▶ Parasta on jos staging-ympäristössä on käytössä sama *data kuin* tuotantoymäristössä

- ▶ Ehdottomasti kaikkein tärkein laadunhallinnan kannalta on **mahdollisimman usein tapahtuva käyttöönotto**
 - ▶ edellyttää hyvin rakennettua deployment pipelineä, kohtuullista testauksen automatisointia

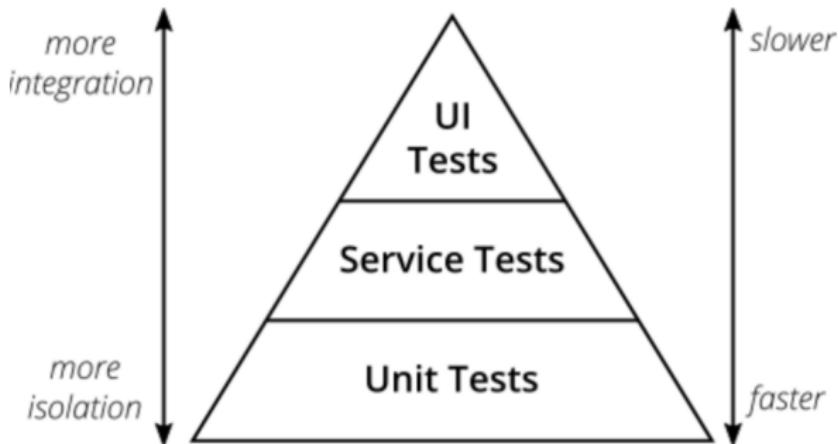
- ▶ Ehdottomasti kaikkein tärkein laadunhallinnan kannalta on **mahdollisimman usein tapahtuva käyttöönotto**
 - ▶ edellyttää hyvin rakennettua deployment pipelineä, kohtuullista testauksen automatisointia
- ▶ Trunk based development auttaa nopeaa käyttöönottoa feature brancheihin verrattuna

- ▶ Ehdottomasti kaikkein tärkein laadunhallinnan kannalta on **mahdollisimman usein tapahtuva käyttöönotto**
 - ▶ edellyttää hyvin rakennettua deployment pipelineä, kohtuullista testauksen automatisointia
- ▶ Trunk based development auttaa nopeaa käyttöönottoa feature brancheihin verrattuna
- ▶ Suosittelen että käyttöönotto tapahtuu niin usein kuin mahdollista, jopa useita kertoja päivässä
 - ▶ takaa sen, että pahoja integrointiongelmia ei synny
 - ▶ sovellukseen syntyvät regressiot havaitaan ja pystytään korjaamaan mahdollisimman nopeasti

- ▶ Ehdottomasti kaikkein tärkein laadunhallinnan kannalta on **mahdollisimman usein tapahtuva käyttöönotto**
 - ▶ edellyttää hyvin rakennettua deployment pipelineä, kohtuullista testauksen automatisointia
- ▶ Trunk based development auttaa nopeaa käyttöönottoa feature brancheihin verrattuna
- ▶ Suosittelen että käyttöönotto tapahtuu niin usein kuin mahdollista, jopa useita kertoja päivässä
 - ▶ takaa sen, että pahoja integrointiongelmia ei synny
 - ▶ sovellukseen syntyvät regressiot havaitaan ja pystytään korjaamaan mahdollisimman nopeasti
- ▶ Nopea käyttöönotto **pakottaa** laatuun

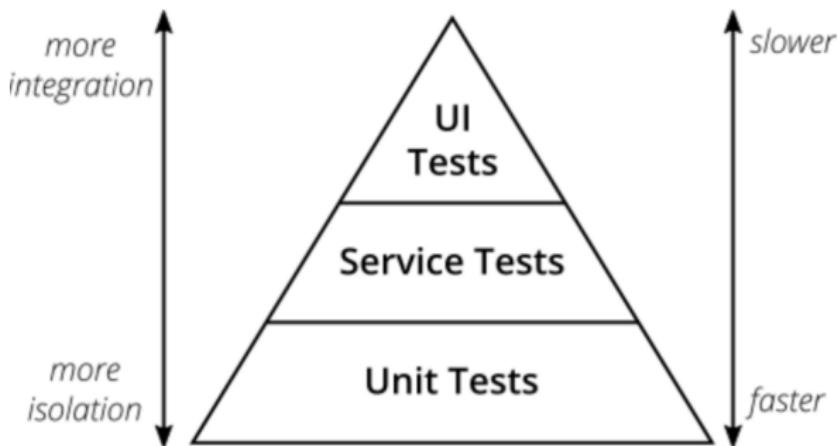
Testauspyramiidi

- ▶ Oma näkemykseni poikkeaa jossain määrin ns *testauspyramidista*



Testauspyramidi

- ▶ Oma näkemykseni poikkeaa jossain määrin ns *testauspyramidista*



- ▶ DISA: 570 yksikkötestiä, ja kaikki vihreällä. Softa ei edes käynnisty...

Guillermo Rauch



Guillermo Rauch
@rauchhg

Write tests. Not too many. Mostly integration.

4:43 PM · December 10th, 2016

24

473

1,360



Kent Dodds: testauspokaali

THE FOUR TYPES OF TESTS

End to End

A helper robot that behaves like a user to click around the app and verify that it functions correctly.

Sometimes called "functional testing" or e2e.

Integration

Verify that several units work together in harmony.

Unit

Verify that individual, isolated parts work as expected.

Static

Catch typos and type errors as you write the code.

