Ohjelmistotekniikan menetelmät

Luento 6, 16.08.2017

Esimerkkijärjestelmä

Tehtävänhallinta

Esimerkkijärjestelmä

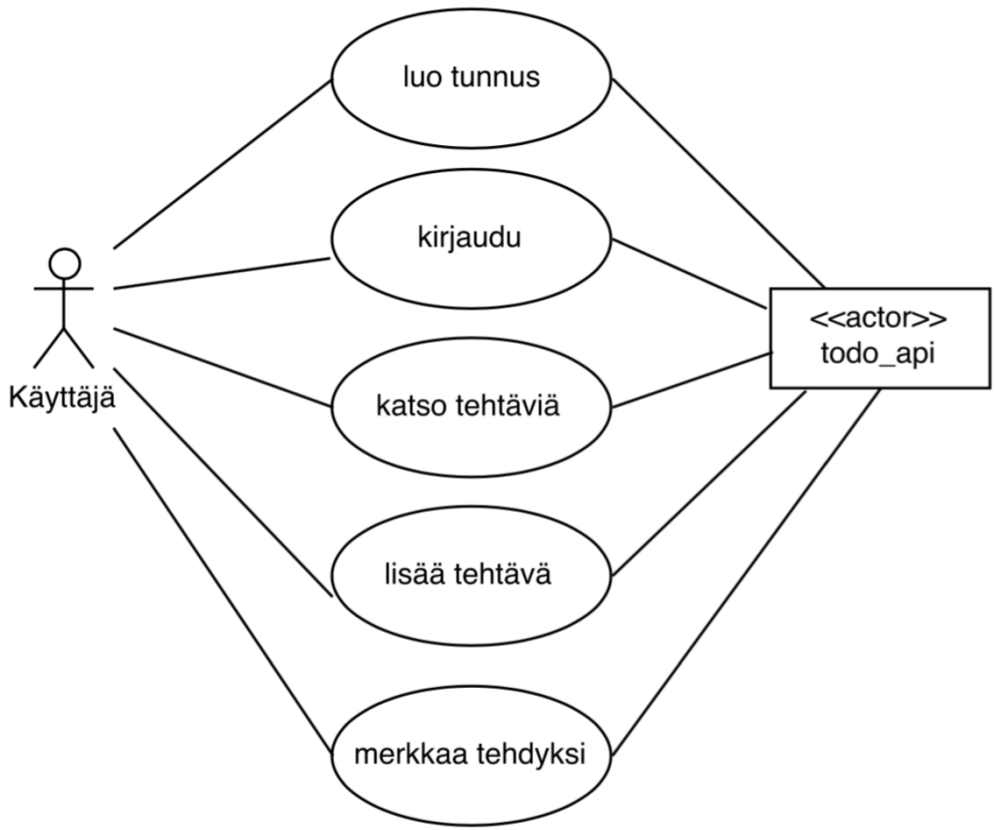
Tehtävänhallinta

- Tutustutaan ohjelmiston suunnitteluun tarkastelemalla yksinkertaista tehtävänhallintasovellusta eli Todo-applikaatiota
- Sovelluksen avulla käyttäjä voi asettaa itselleen tehtäviä, eli "todoja" ja merkitä tehtäviä tehdyksi
- Ohjelma on tarkoitus toteuttaa Javalla gaafisen käyttöliittymän omaavana sovelluksena
- Käyttäjien tehtävälistat ovat talletettuna verkkoon, joten käyttäjä näkee oman tehtävälistansa kaikilta koneilta, jonne hän on asentanut sovelluksen
  - tehtävälistat tallettava *todo-api* sijaitsee osoitteessa https://otm-todo-api.herokuapp.com ja tarjoaa tehtävälistojen tallentamiseen ns. REST-rajapinnan
    - api on jo valmiina, eli sen toteuttaminen ei kuulu nyt suunniteltavan sovelluksen kehittäjille
  - kehitettävä sovellus kommunikoi todo-api:n kanssa HTTP-protokollaa käyttäen

Esimerkkijärjestelmä

Tehtävänhallinta

- Sovelluksella hallittavat tehtävälistat ovat henkilökohtaisia
  - järjestelmän käyttäjien on luotava käyttäjätunnus
  - ja kirjauduttava sisään päästäkseen käsittelemään omaa tehtävälistaansa
- Sovelluksen käyttötapauskaavio seuraavassa



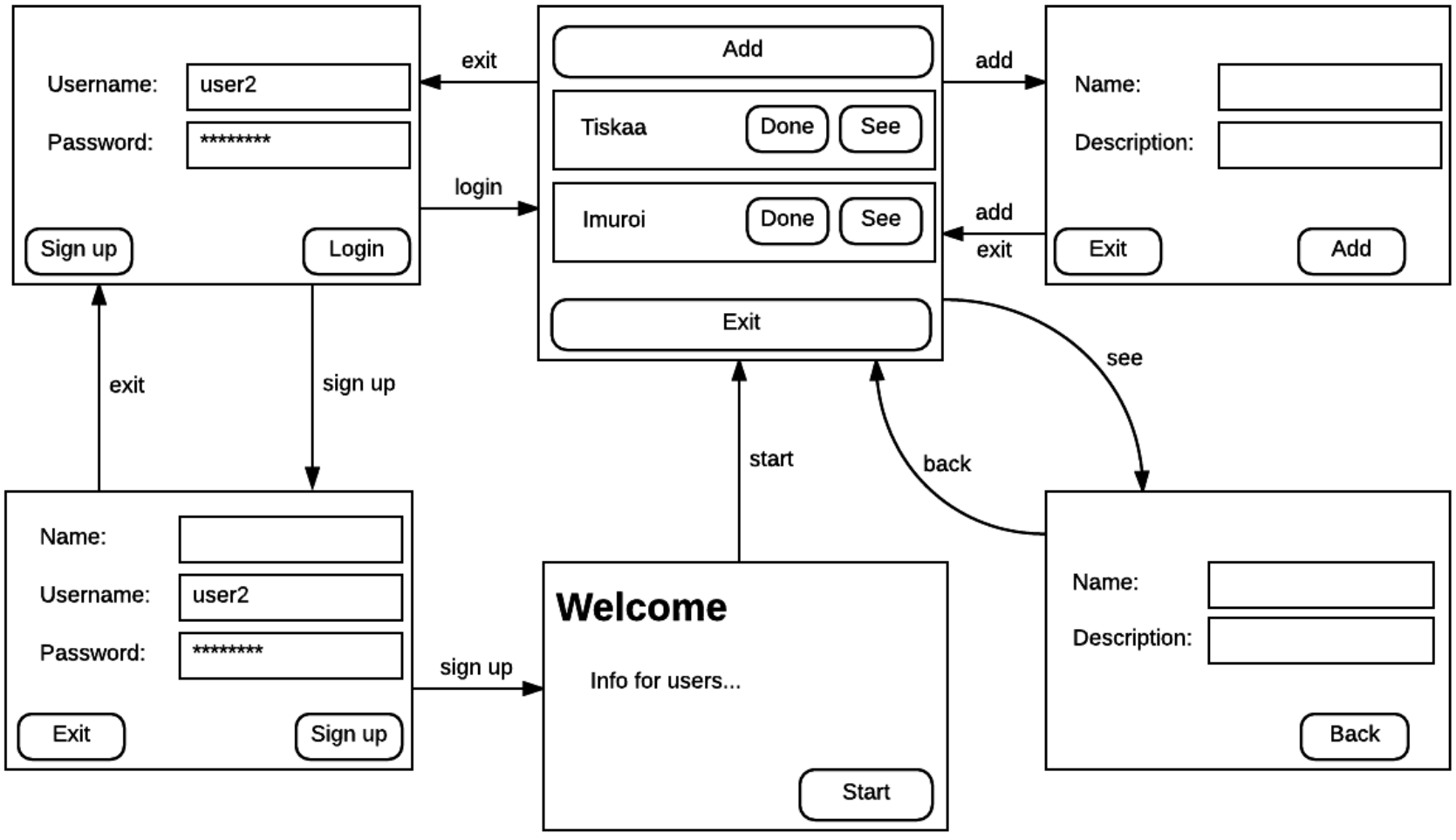
Esimerkkijärjestelmä

Käyttöliittymäluonnos

- Käyttötapausten tekstuaalisten kuvausten sijaan päätetään tehdä sovelluksen haluttua toimintaa kuvaava *käyttöliittymäluonnos*
- Käyttöliittymäluonnos koostuu joukosta näkymiä, ja näkymien välisistä siirtymistä
- Käyttäjän suorittamat toiminnot, esim. sovelluksen painikkeiden klikkaukset saavat aikaan sen, että järjestelmä siirtyy näkymästä toiseen
- Voidaankin ajatella, että sovelluksella on useita eri *tiloja*, ja käyttöliittymäkuvaus vastaa UML:n tilakaaviota
  - Kun sovellus käynnistetään, on sovellukseen kirjauduttava, jotta päästään *tilaan*, jossa tehtävälistan tarkastelu on mahdollista
- Käyttöliittymäluonnos seuraavalla sivulla

Esimerkkijärjestelmä

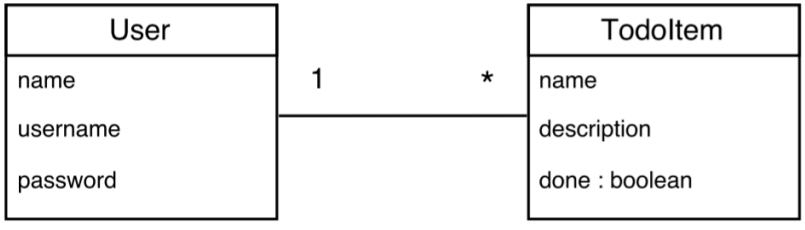
Käyttöliittymäluonnos



Esimerkkijärjestelmä

Määrittelyvaiheen luokkakaavio

- Sovelluksen määrittelyvaiheessa tai viimeistään suunnittelun alussa on hyödyllistä tunnistaa sovellusalueen käsitteistö ja muodostaa *määrittelyvaiheen luokkakaavio*
- Ohjelman halutun toiminnallisuuden kuvauksesta tunnistetaan seuraavat käsitteet
  - käyttäjä
  - tehtävä
  - tehtävälista
  - todo-api
- Tehtävälista tarkoittaa joukkoa tietyn käyttäjän tehtäviä, joten karsitaan se, samoin kuin todo-api, joka on ulkoinen järjestelmä
- Päätetään antaa luokille englanninkieliset nimet



Ohjelmiston suunnittelu

Ohjelmiston suunnittelu

- Ohjelmiston suunnittelu jakautuu karkeasti ottaen kahteen vaiheeseen:
  1. Arkkitehtuurisuunnittelu
  2. Oliosuunnittelu
- Ensimmäinen vaihe on **arkkitehtuurisuunnittelu**, jonka aikana hahmotellaan järjestelmän rakenne karkeammalla tasolla
- Tämän jälkeen suoritetaan **oliosuunnittelu**, eli suunnitellaan oliot, jotka ottavat vastuulleen järjestelmältä vaaditun toiminnallisuuden toteuttamisen
  - Yksittäiset oliot eivät yleensä pysty toteuttamaan kovin paljoa järjestelmän toiminnallisuudesta
  - Erityisesti oliosuunnitteluvaiheessa tärkeäksi seikaksi nouseekin *olioiden välinen yhteistyö*, eli se vuorovaikutus, jolla oliot saavat aikaan halutun toiminnallisuuden
- Oliosuunnittelu tapahtuu useimmiten ainakin osittain ohjelmoinnin yhteydessä

Ohjelmiston suunnittelu

Ohjelmiston arkkitehtuuri

Ohjelmiston suunnittelu

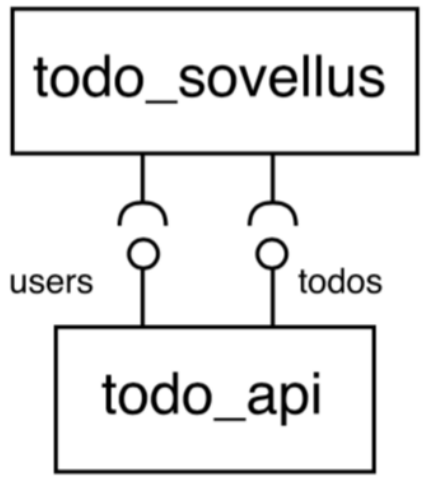
Ohjelmiston arkkitehtuuri

- *Ohjelmiston arkkitehtuurilla* (engl. software architecture) tarkoitetaan ohjelmiston korkean tason rakennetta
  - jakautumista erillisiin komponentteihin
  - komponenttien välisiä suhteita
  - mekanismeja, jolla komponentit kommunikoivat keskenään
- *Komponentilla* tarkoitetaan yleensä kokoelmaa toisiinsa liittyviä olioita/luokkia, jotka suorittavat ohjelmassa jotain tehtäväkokonaisuutta
  - esim. käyttöliittymän voitaisiin ajatella olevan yksi komponentti
  - vastaavasti tietokantayhteydestä huolehtiva koodi voisi muodostaa oman komponentin
- Komponentit voivat myös olla erillisiä "sovelluksia", jotka kommunikoivat internetin tai esim. yhteisen tietokannan välityksellä
- Iso komponentti voi muodostua useista alikomponenteista

Ohjelmiston suunnittelu

Ohjelmiston arkkitehtuuri

- Voimme ajatella, että esimerkkiohjelmistomme koostuu kahdesta "pääkomponentista"
  - internetissä olevasta todo-api:sta
  - omalle koneelle asennettavasta, nyt suunniteltavasta todo-sovelluksesta
- Ohjelmistomme karkean tason arkkitehtuuri *UML-komponenttikaaviona*



Ohjelmiston suunnittelu

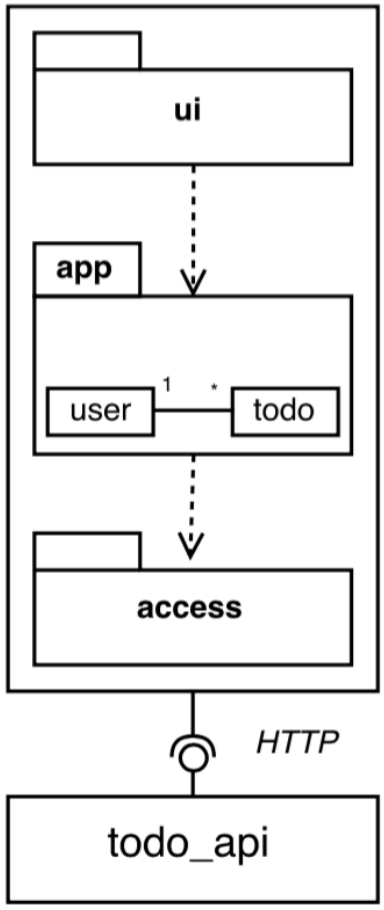
Komponenttikaavio

- Komponenttikaavio kuvaa ohjelmiston komponentteja ja niiden välisiä rajapintoja
- Kannattaa huomata, että rajapinnan käsite ei nyt tarkoita samaa kuin Javan interface, vaan yleisemmin jotain tapaa, joka mahdollistaa sovellusten kommunikoinnin
- Komponentit merkitään laatikkoina
  - nyt suunniteltava sovellus on komponentti *todo-sovellus* ja toinen komponentti on verkossa toimiva *todo-api*
  - Kaavioon on merkitty, että todo-api tarjoaa kaksi *rajapintaa*, toinen huolehtii käyttäjistä (users) ja toinen tehtävistä (todos)
  - todo-sovellus käyttää molempia todo-api:n tarjoamia rajapintoja
- Komponentin tarjoama rajapinta kuvataan symbolilla ⚲
- Jos komponentti hyödyntää rajapintaa, liitetään se rajapintaan symbolilla ⅄
- Emme tutustu kurssilla tämän tarkemmin komponenttikaavioihin

Sovelluksen tarkempi kuvaus

Komponenttikaavio

- Ohessa karkea hahmotelma Todo-sovelluksen arkkitehtuurista *pakkauskaaviona*
- Järjestelmä on jakautunut kolmeen komponenttiin
  - Käyttöliittymä (ui), Sovelluslogiikka (app) ja Tallennuspalvelut (access)
- Jokainen komponentti on kuvattu omana *pakkauksena*, eli isona laatikkona, jonka vasempaan ylälaitaan liittyy pieni laatikko
- Pakkausten välillä on *riippuvuuksia*
  - Käyttöliittymä riippuu sovelluslogiikasta
  - Sovelluslogiikka riippuu tallennuspalveluista
- Järjestelmä perustuu **kerrosarkkitehtuuriin**



Sovelluksen tarkempi kuvaus

Pakkauskaavio

- Pakkauskaaviossa yksi komponentti kuvataan pakkaussymbolilla
  - Pakkauksen nimi on joko keskellä symbolia tai ylänurkan laatikossa
- Pakkausten välillä olevat riippuvuudet ilmaistaan katkoviivanuolena, joka suuntautuu pakkaukseen, johon riippuvuus kohdistuu
- Riippuvuus tarkoittaa käytännössä sitä, että *käyttöliittymän oliot kutsuvat sovelluslogiikan olioiden metodeja*
- Pakkauksen sisältö on mahdollista piirtää pakkaussymbolin sisään
  - Pakkauksen sisällä voi olla alipakkauksia tai luokkia
- Riippuvuudet voivat olla myös alipakkausten välisiä
- Huom: edellisen sivun kaaviomme on oikeastaan yhdistelmä pakkaus- ja komponenttikaaviota, missä sovelluskomponentin sisäinen rakenne on kuvattu pakkauskaavion avulla

Kerrosarkkitehtuuri

Kerrosarkkitehtuuri

- *Kerrosarkkitehtuuri* (engl. layered architecture) yksi hyvin tunnettu *arkkitehtuurimalli* (engl. architecture pattern), eli periaate, jonka mukaan tietynlaisia ohjelmia kannattaa pyrkiä rakentamaan
- Kerros on kokoelma toisiinsa liittyviä olioita tai alikomponentteja, jotka muodostavat esim. toiminnallisuuden suhteen loogisen kokonaisuuden ohjelmistosta

Kerrosarkkitehtuurissa on pyrkimyksenä järjestellä komponentit siten, että ylempänä oleva kerros käyttää ainoastaan alempana olevien kerroksien tarjoamia palveluita

Kerrosarkkitehtuuri

Kerrosarkkitehtuurin etuja

- Kerroksittaisuus helpottaa ylläpitoa
  - Kerroksen sisältöä voi muuttaa vapaasti jos sen palvelurajapinta eli muille kerroksille näkyvät osat säilyvät muuttumattomina
  - Sama pätee tietysti mihin tahansa komponenttiin
- Jos kerroksen palvelurajapintaan tehdään muutoksia, aiheuttavat muutokset ylläpitotoimenpiteitä ainoastaan ylemmän kerroksen riippuvuuksia omaavin osiin
- Sovelluslogiikan riippumattomuus käyttöliittymästä helpottaa ohjelman siirtämistä uusille alustoille
- Alimpien kerroksien palveluja, kuten esim. tallennuspalvelukerrosta voidaan ehkä uusiokäyttää myös muissa sovelluksissa
- Ylemmät kerrokset voivat toimia korkeammalla abstraktiotasolla
  - Kaikkien ohjelmoijien ei tarvitse ymmärtää kaikkia detaljeja, osa voi keskittyä tietoliikenneyhteyksiin, osa käyttöliittymiin, osa sovelluslogiikkaan

Kerrosarkkitehtuuri

Ei pelkkiä kerroksia...

- Myös kerrosten sisällä ohjelman loogisesti toisiinsa liittyvät komponentit kannattaa ryhmitellä omiksi kokonaisuuksiksi, joka voidaan UML:ssa kuvata pakkauksena
- Yksittäisistä komponenteista kannattaa tehdä mahdollisimman *yhtenäisiä* toiminnallisuudeltaan
  - eli sellaisia, joiden osat kytkeytyvät tiiviisti toisiinsa ja palvelevat ainoastaan yhtä selkeästi eroteltua tehtäväkokonaisuutta
  - komponentilla tulee siis olla vain yksi *vastuu*, eli tehtäväalue, jota se hoitaa
- Samalla pyrkimyksenä on, että erilliset komponentit ovat mahdollisimman *löyhästi kytkettyjä* (engl. loosely coupled)
  - komponenttien välisiä riippuvuuksia pyritään minimoimaan
- Selkeä jakautuminen komponentteihin myös helpottaa työn jakamista suunnittelu- ja ohjelmointivaiheessa sekä testausta, laajennusta että ylläpitoa!

Kerrosarkkitehtuuri

Käyttöliittymän ja sovelluslogiikan eriyttäminen

- Kerrosarkkitehtuurin ylimpänä kerroksena on yleensä käyttöliittymä
- Pidetään järkevänä, että **ohjelman sovelluslogiikka on täysin erotettu käyttöliittymästä**
  - Sovelluslogiikan erottaminen lisää koodin määrää, joten jos kyseessä "kertakäyttösovellus", ei ylimääräinen vaiva kannata
- Käytännössä tämä tarkoittaa kahta asiaa:
  - **Sovelluksen palveluja toteuttavilla olioilla ei ole suoraa yhteyttä käyttöliittymän olioihin** (esim. Swing componentit tai HTML-koodia generoivat luokat)
  - **Käyttöliittymän toteuttavat oliot eivät sisällä ollenkaan ohjelman sovelluslogiikkaa**
  - Käyttöliittymäoliot ainoastaan piirtävät käyttöliittymäkomponentit ruudulle, välittävät käyttäjän komennot eteenpäin sovelluslogiikalle ja heijastavat sovellusolioiden tilaa käyttäjille

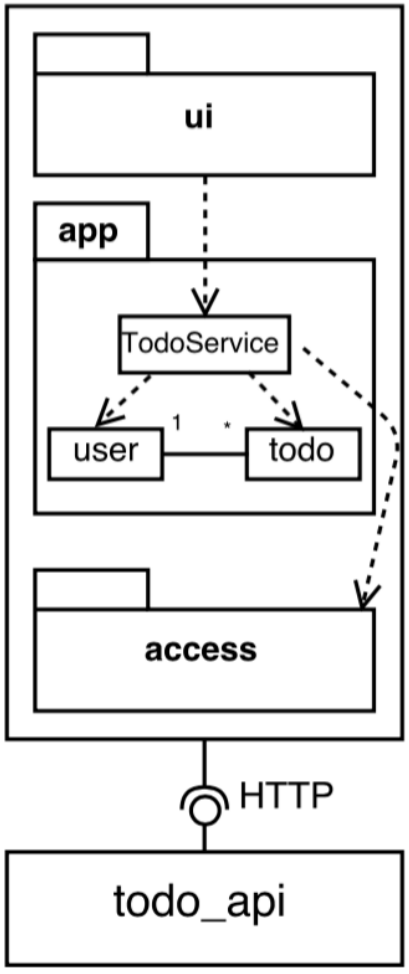
Kerrosarkkitehtuuri

Käyttöliittymän ja sovelluslogiikan eriyttäminen

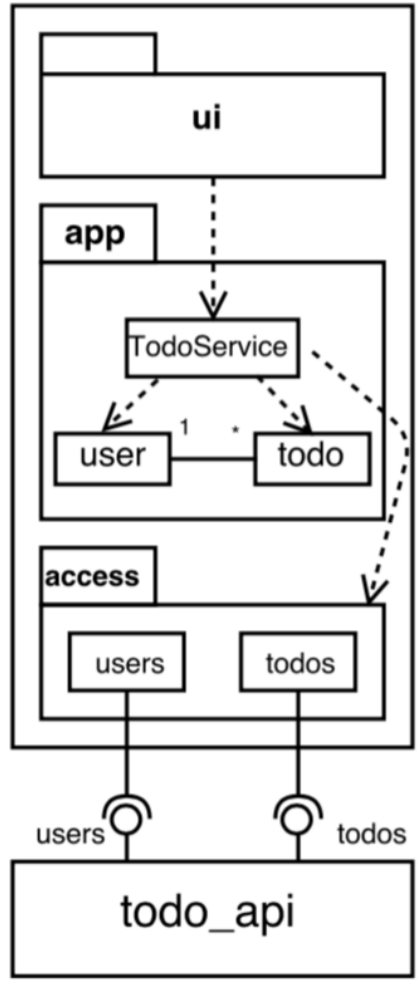
- Käytännössä erottelu tehdään liittämällä käyttöliittymän ja sovellusalueen olioiden väliin erillisiä komponentteja, jotka koordinoivat käyttäjän komentojen aiheuttamien toimenpiteiden suoritusta sovelluslogiikassa
  - Erottelun pohjana on Ivar Jacosonin kehittämä idea oliotyyppien jaoittelusta kolmeen osaan, rajapintaolioihin (boundary), ohjausolioihin (control) ja sisältöolioihin (entity)
- Käyttöliittymän (eli rajapintaolioiden) ja sovelluslogiikan (eli sisältöolioiden) yhdistävät **ohjausoliot**
  - Joissain yhteyksissä, esim. Java Spring -sovelluskehyksen yhteydessä samasta asiasta käytetään nimitystä *palvelu* (engl. service)
- Käyttöliittymä ei siis itse tee mitään sovelluslogiikan kannalta oleellisia toimintoja, vaan ainoastaan välittää käyttäjien komentoja ohjausolioille, jotka huolehtivat sovelluslogiikan olioiden manipuloimisesta

Sovelluksen tarkempi kuvaus

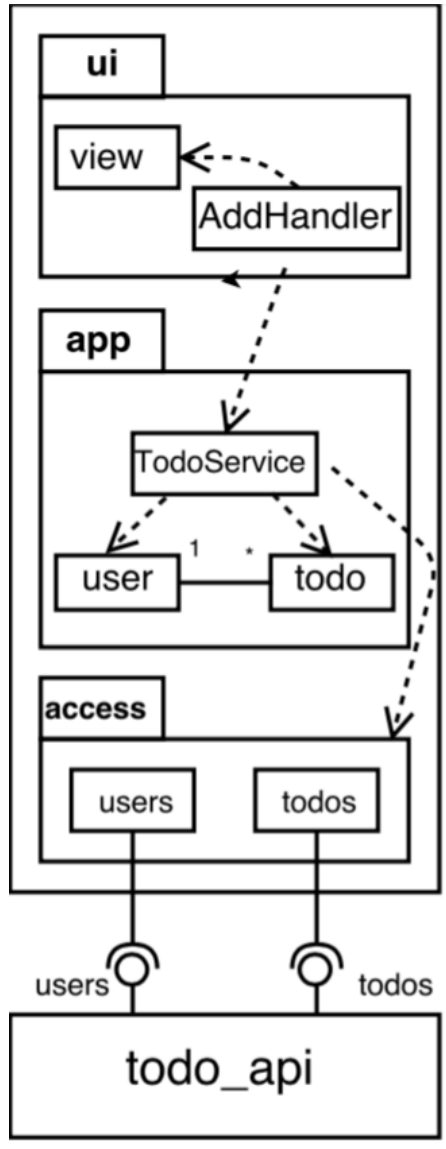
- Palataan jälleen sovelluksemme pariin
- Päätetään lisätä sovellukseen logiikasta huolehtiva ohjausolio, kutsutaan sitä nimellä *TodoService*
- Olion vastuulla sovelluslogiikan lisäksi on myös tallennuspalveluiden käyttö
- Pakkauskaavio näyttää nyt seuraavalta



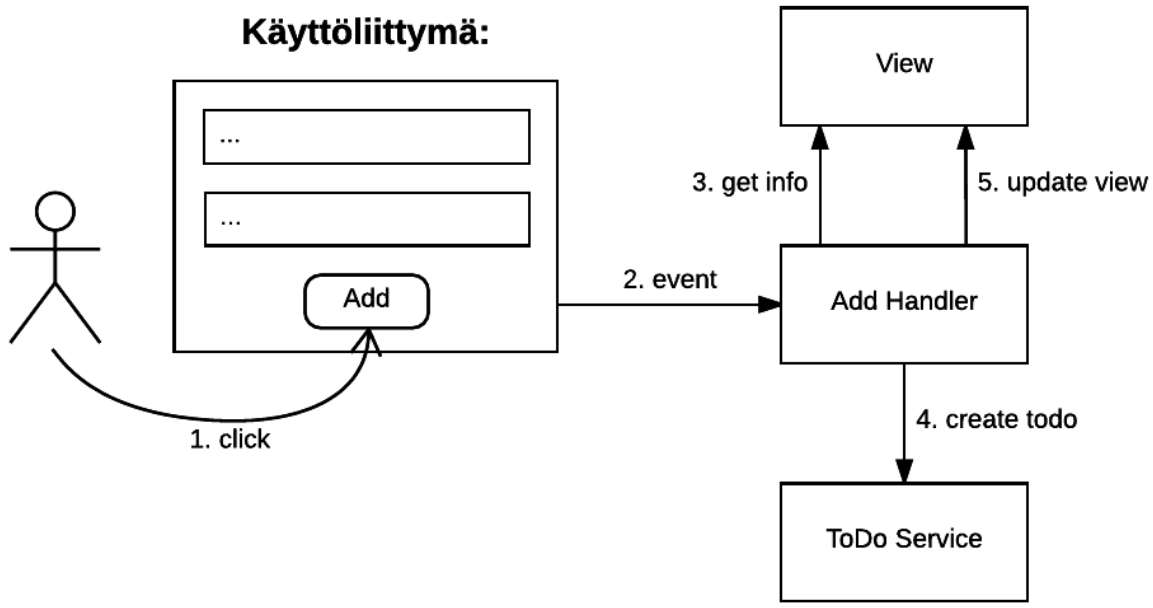
- Pakkauskaaviossa oleva tallennuspalvelu on siis komponentti, joka on vastuussa todo-apin kanssa kommunikoimisesta
  - todo-api tallettaa tiedot käyttäjistä, ja käyttäjien tehtävälistoista
- Tallennuspalvelun vastuulla siis sekä käyttäjien hallinta että tehtävälistat
- Päätetään toteuttaa tallennuspalvelu siten, että nämä kaksi vastuuta on eriytetty omiin luokkiinsa *UserAccess* ja *TodoAccess*



- Käyttöliittymäkoodissa on syytä tehdä selkeä vastuujako näkymän muodostavan koodin ja käyttäjän toimintoihin, esim. näppäinten painalluksiin reagoivan koodin kesken
- Javan Swing:illä toteutetuissa käyttöliittymissä *tapahtumankäsittelijät* toteuttavat käyttäjän toimintoihin reagoinnin
- Tapahtumankäsittelijöiden vastuulle jää käyttäjän toimiin reagoiminen
  - tapahtumankäsittelijät kutsuvat tarvittavia todo-servicen tarjoamia sovelluslogiikkaa suorittavia metodeja
- Näkymän koodi taas huolehtii ainoastaan siitä, että ruudulla on piirtyneenä oikeat asiat

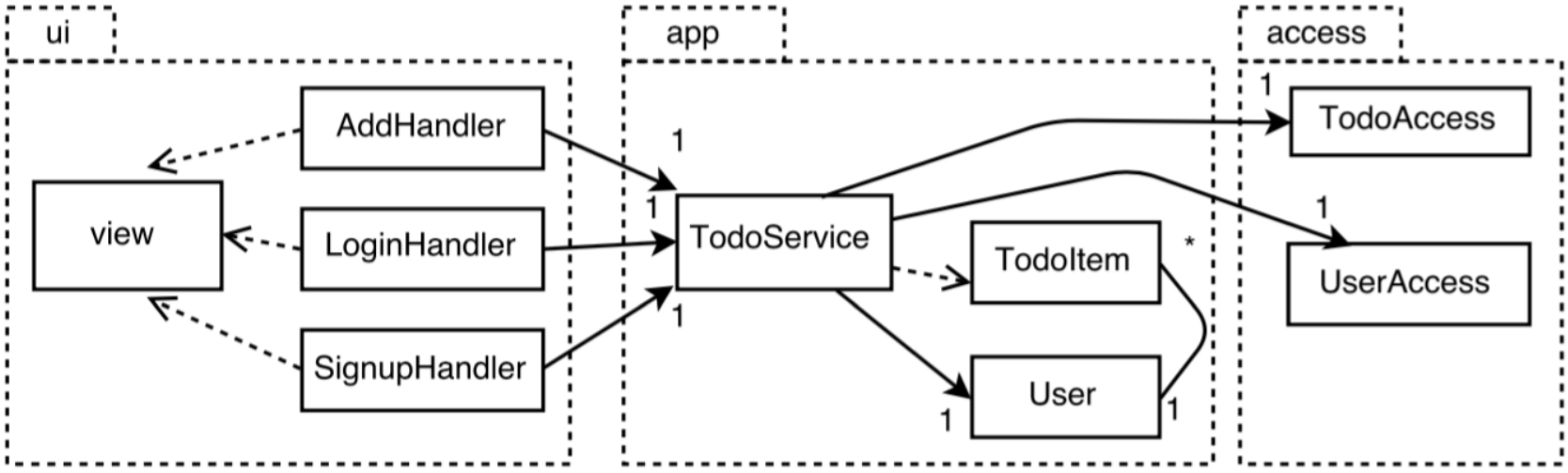


- Toiminnan logiikka *kommunikaatiokaaviona*, esimerkkinä uuden tehtävän lisäys
  1. Syötettyään kenttiin tapahtuman tiedot, käyttäjä painaa nappia
  2. Javan runtime kutsuu napille rekisteröityä tapahtumankäsittelijää
  3. Tapahtumankäsittelijä hakee näytön kentistä käyttäjän kirjoittaman syötteen
  4. Tapahtumankäsittelijä kutsuu todoServicen uuden tehtävän luomisesta huolehtivaa metodia
  5. Tapahtumankäsittelijä pyytää näyttöä päivittämään itsensä



Alustava luokkarakenne

- Päädymme siis alla kuvattuun luokkarakenteeseen
  - jotta luokkien yhteydet erottuisivat paremmin, on pakkausrakenne merkitty kuvaan katkoviivalla, kyse ei ole virallisesta UML-käytännöstä
- Näkymä on nyt merkitty vain yhtenä luokkana *view*, todellisuudessahan näkymä koostetaan useista käyttöliittymäolioista: labeleista, tekstikentistä, painikkeista, paneleista ...



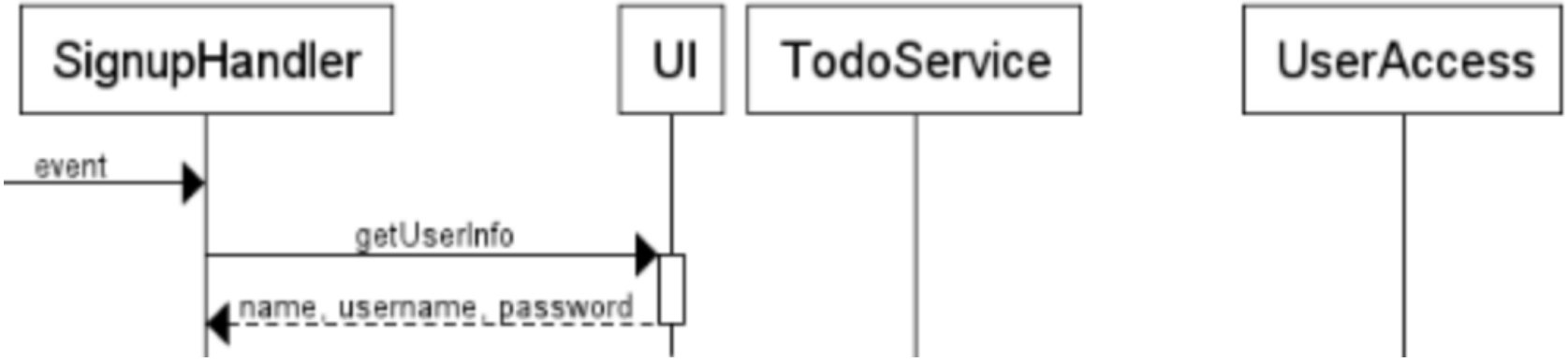
Käyttötapaukset toteuttava toiminnallisuus

- Kun alustava luokkarakenne on valmis ja luokkien välinen vastuujako selvillä, on aika siirtyä tarkempaan *oliosuunnitteluun*
- Suunnittelu voi edetä usealla vaihtoehtoisella tavalla
  - kerroksittain
  - käyttötapaus kerrallaan
  - jotain näiden väliltä
- Suunnittelussa painopisteeksi nousee se miten ja missä järjestyksessä oliot kutsuvat toistensa metodeja
- Todellisuudessa lienee järkevintä suunnitella ainoastaan "riittävällä" tasolla, ja toteuttaa osa suunnitelluista osista koodina, ja laajentaa jälleen suunnitelmaa
- Ohjelman suunnittelu on sinä määrin orgaaninen prosessi, että sitä ei voi toistaa kalvoilla
- Kuvataan seuraavassa *sekvenssikaavioina* suunnittelun lopputulosta *käyttötapauksittain*

Käyttötapaukset toteuttava toiminnallisuus

Käyttäjätunnuksen luonti

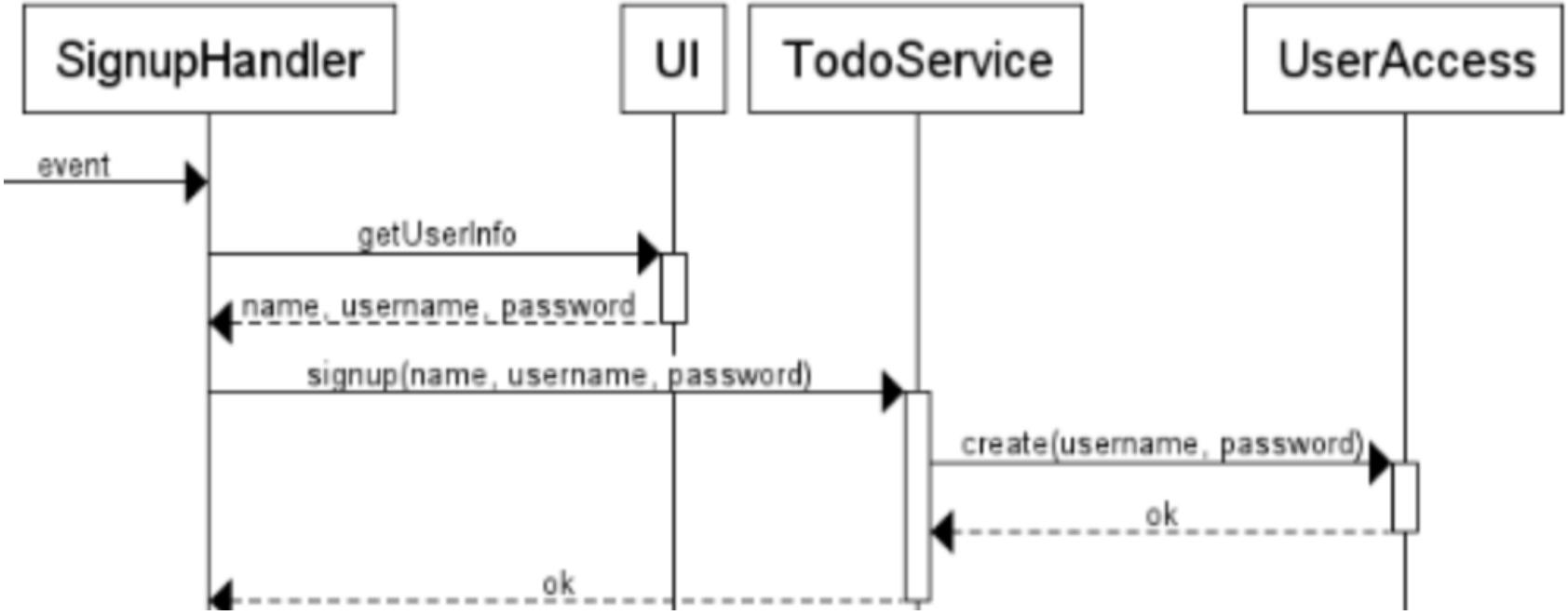
- Käyttäjän täyttää tietonsa (nimi, käyttäjätunnus, salasana) uuden käyttäjän lomakkeelle ja klikkaa "rekisteröidy"
- Tapahtumakäsittelijä *SignupHandler* reagoi klikkaukseen ja hakee (metodilla *getUserInfo*) näkymän kentistä käyttäjän tiedot



Käyttötapaukset toteuttava toiminnallisuus

Käyttäjätunnuksen luonti

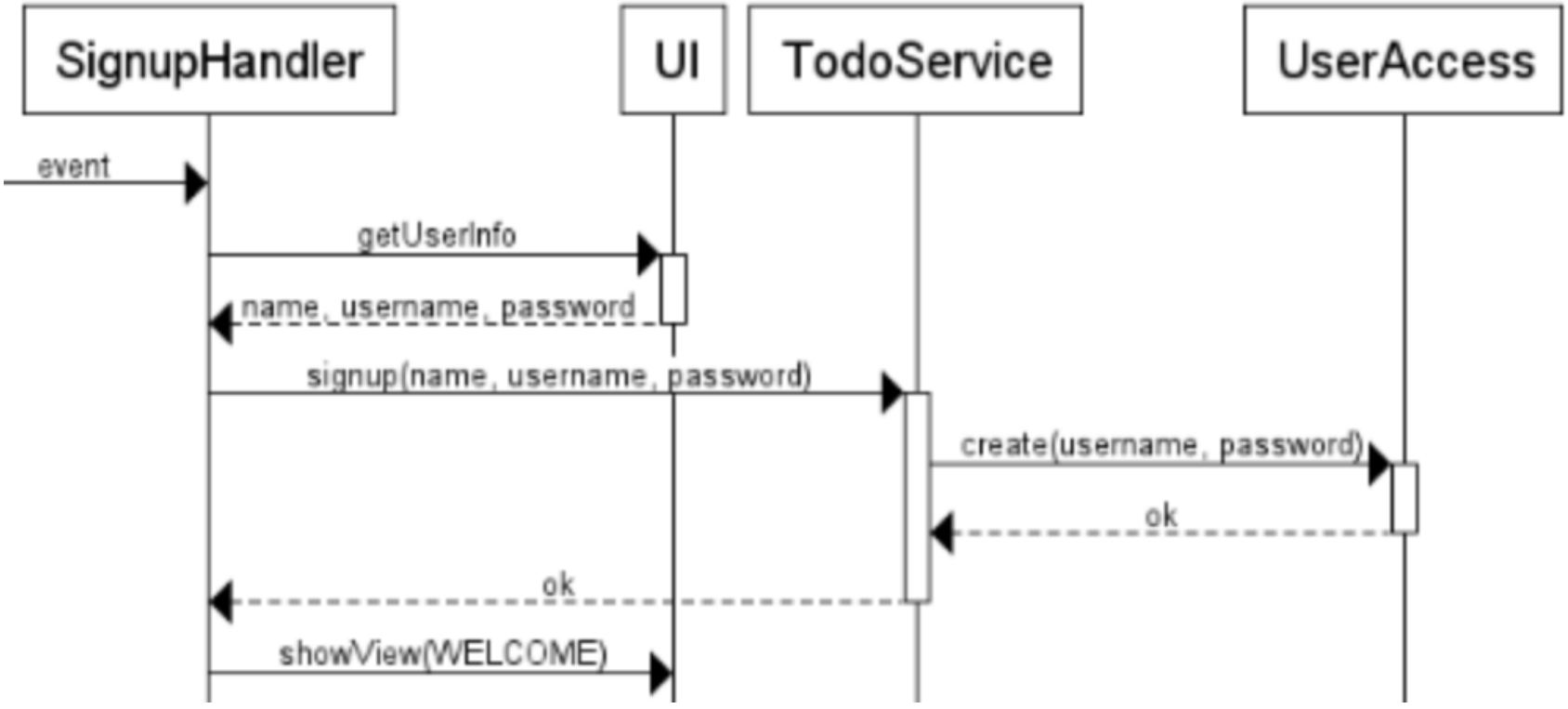
- Tapahtumankäsittelijä kutsuu sovelluslogiikan *TodoService* metodia *signUp* ja antaa käyttäjän tiedot parametrina
- sovelluslogiikka kutsuu *UserAccess*:n käyttäjän luovaa metodia
- UserAccess luo käyttäjän ottamalla HTTP-protokollaa käyttäen yhteyttä *todo-apiin*
  - tätä vaihetta ei kaavioon ole merkitty



Käyttötapaukset toteuttava toiminnallisuus

Käyttäjätunnuksen luonti

- Lopulta tapahtumankäsittelijä vaihtaa näytettävää näkymää kutsumalla metodia *showView(WELLCOME)*
  - näkymässä tervehditään uutta käyttäjää ja annetaan ohjeita sovelluksen käyttöön



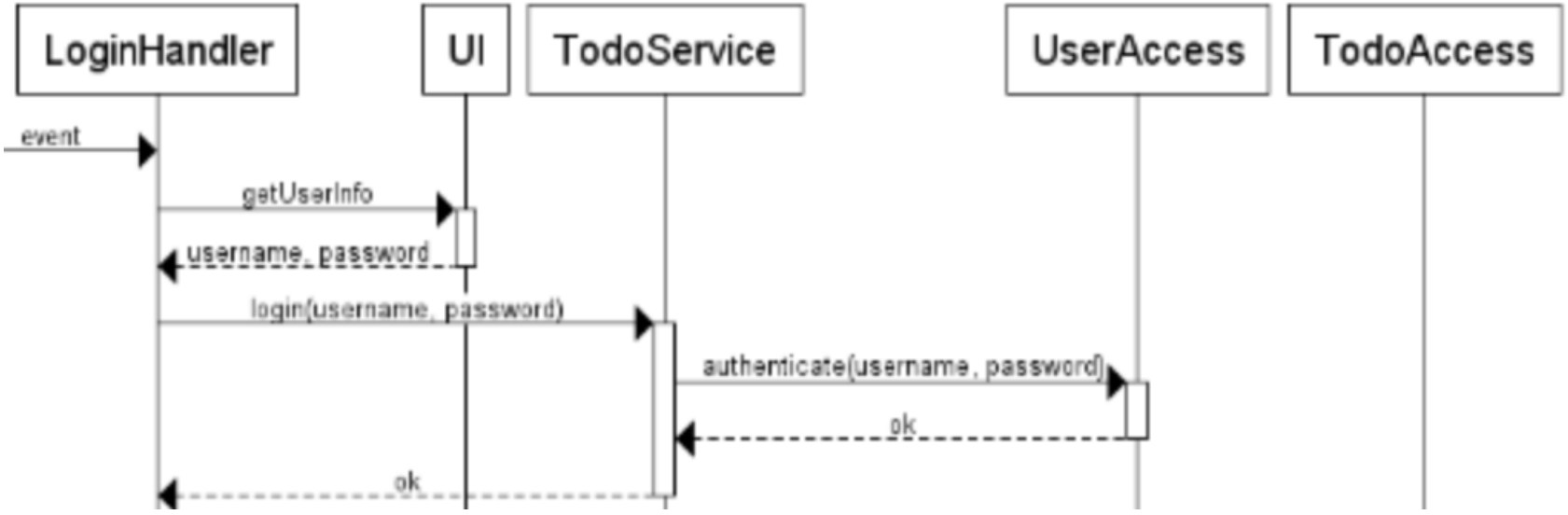
Käyttöliittymän toteutuksesta

- Käyttöliittymä on toteutettu käyttäen *CardLayoutia*
- Sovelluksen jokaisesta näkymästä on tehty oma *JPanel*
- Näkymät rekisteröidään CardLayoutille
  - jokaiselle näkymälle annetaan rekisteröinnin yhteydessä nimi
- CardLayoutin näkymistä on aina vain yksi näkyvillä, muut ovat piilossa
- Metodilla *showView* vaihdetaan näkyvillä olevaa näkymää
- Parametrina metodilla on näkyville tuotavan näkymän nimi

Käyttötapaukset toteuttava toiminnallisuus

Käyttäjä kirjautuu

- Tapahtumakäsittelijä *LoginHandler* reagoi klikkaukseen ja hakee metodilla *getUserInfo* näkymästä käyttäjän tiedot
- Tapahtumankäsittelijä pyytää sovelluslogiikkaa kirjaamaan käyttäjän järjestelmään kutsumalla metodia *login*
- Sovelluslogiikka delegoi kirjautumisen *UserAccess*-oliolle
  - UserAccess autentikoi käyttäjän ottamalla HTTP-protokollalla yhteyttä *todo-apiin*



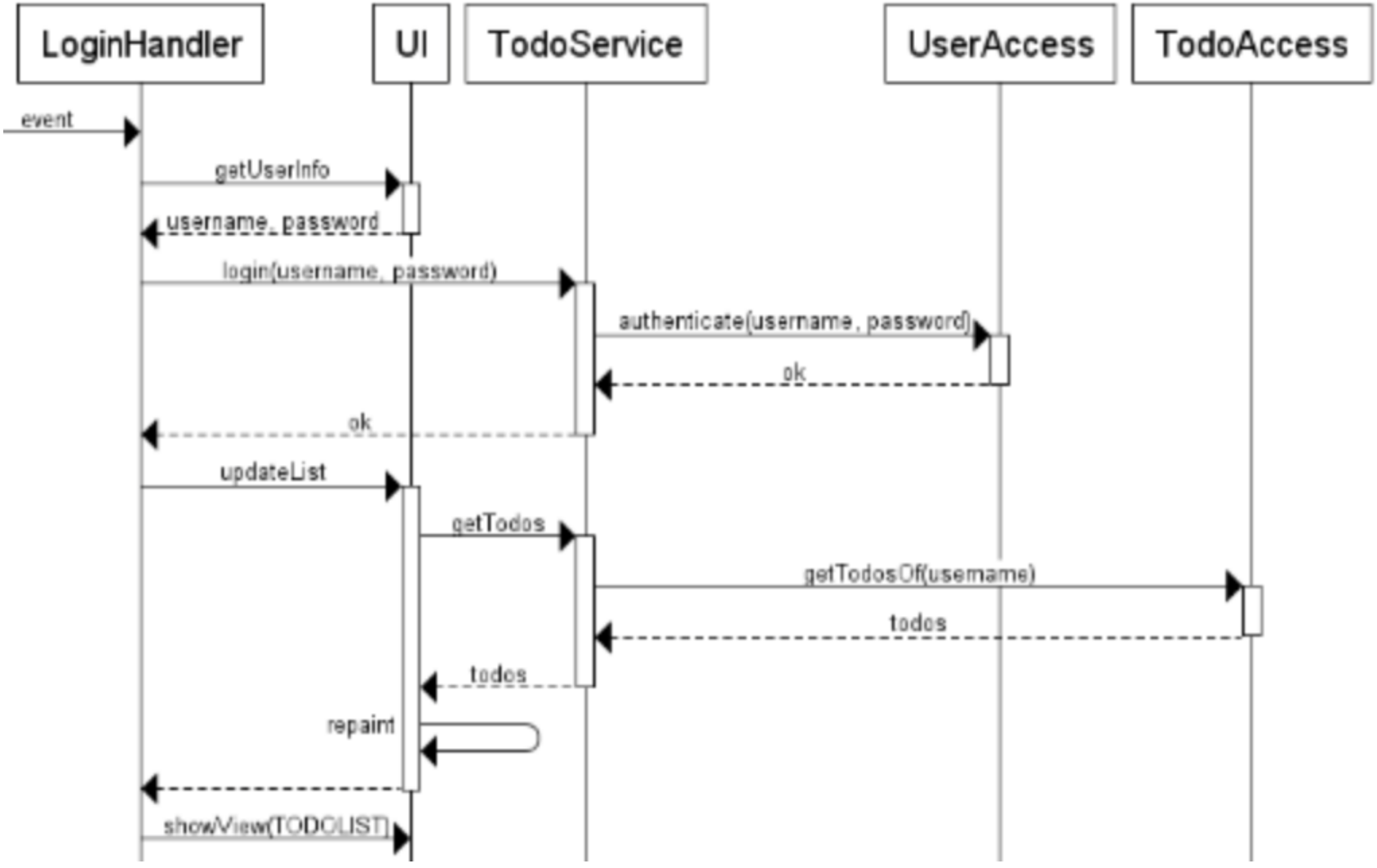
Käyttötapaukset toteuttava toiminnallisuus

Käyttäjä kirjautuu

- Kun käyttäjä on autentikoitu, kutsuu tapahtumankäsittelijä käyttöliittymän metodia *updateList*, joka hakee sovelluslogiikalta kaikki käyttäjän tehtävät ja lisää ne tehtävien näkymään
- Sovelluslogiikka pyytää *TodoAccess*:ia hakemaan tehtävälistan *todo-apilta*
- Lopulta tapahtumankäsittelijä pyytää näyttöä vaihtamaan näkymää

Käyttötapaukset toteuttava toiminnallisuus

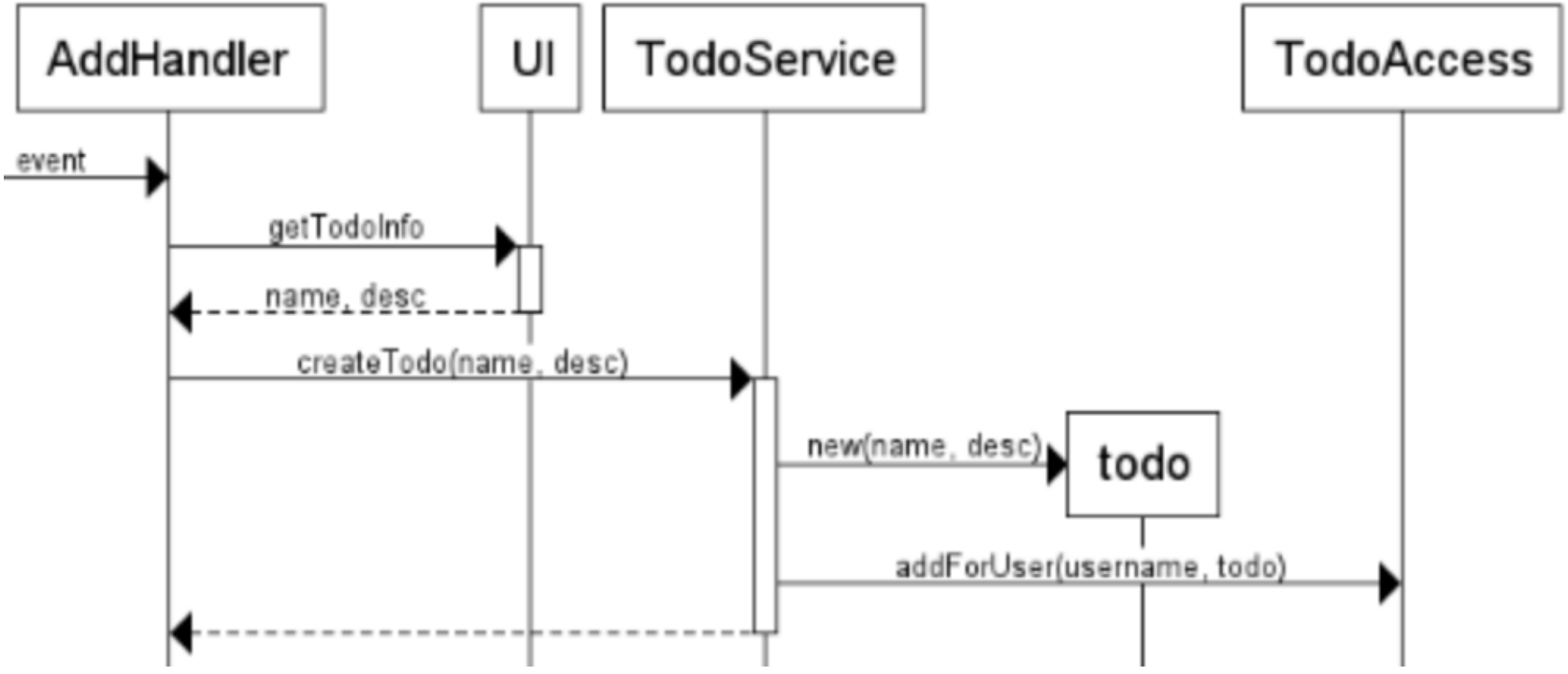
Käyttäjä kirjautuu



Käyttötapaukset toteuttava toiminnallisuus

uuden tehtävän luonti

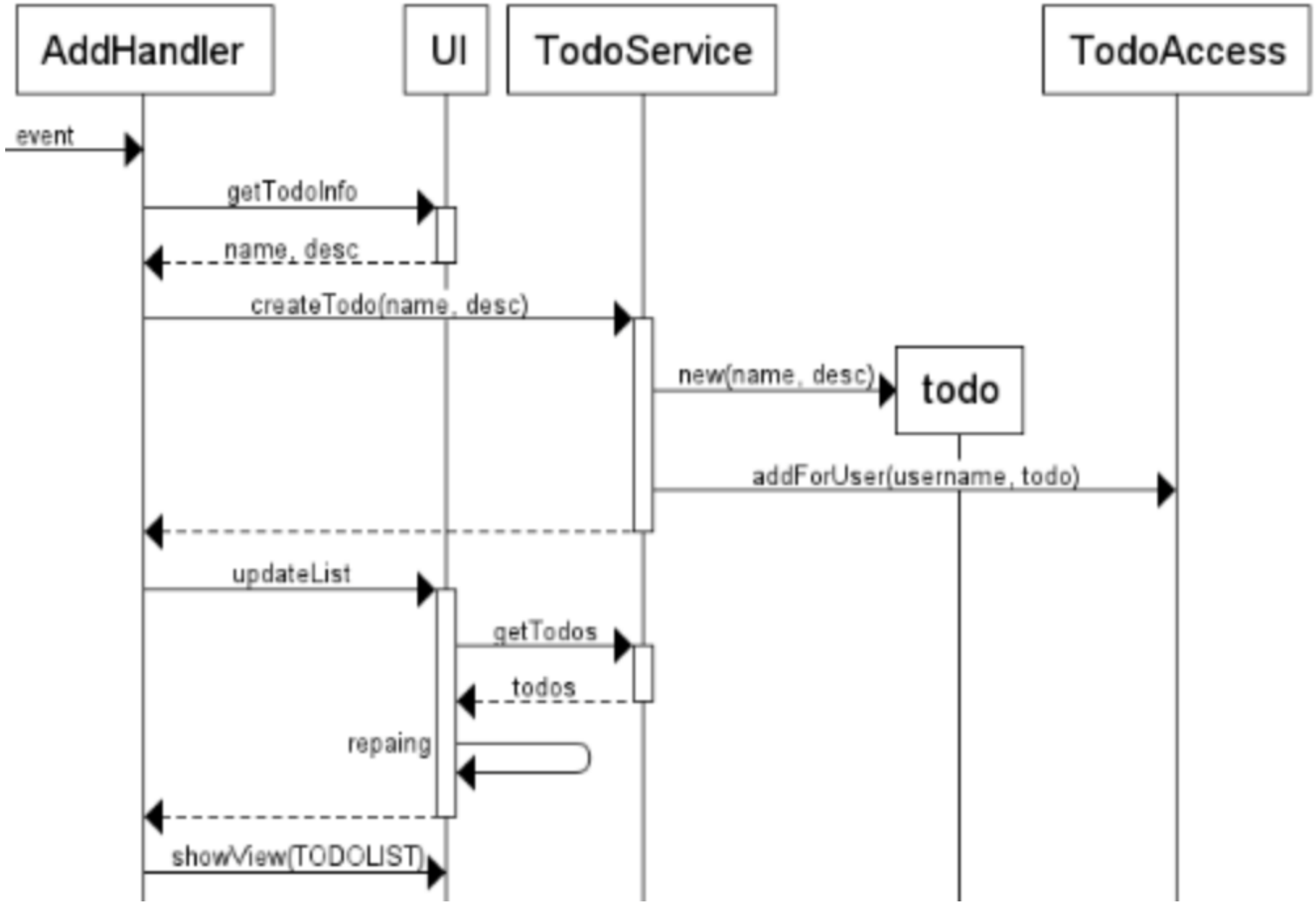
- Uuden tehtävän luonti alkaa jälleen sillä että tapahtumakäsittelijä hakee tehtävän tiedot ja kutsuu sovelluslogiikkaa
- Sovelluslogiikka luo uuden tehtävän ja pyytää *TodoAccess*:ia tallettamaan tehtävän *todo-apiin*



Käyttötapaukset toteuttava toiminnallisuus

uuden tehtävän luonti

- Lopuksi päivitetään näytöllä oleva tehtävälista ja palataan tehtävälistanäyttöön



- Ohjelman muun toiminnallisuuden käyttötapaukset noudattavat samaa logiikka kuin nyt kuvaamamme, joten sivuutamme niiden tarkemman tarkastelun
- Edellä tuli ohjelman suunnittelusta kuvattua lähinnä valmis lopputulos
- Todellisuudessa suunnittelu- ja ohjelmointiprosessi ei ollut yhtä suoraviivainen, vaan eteni seuraavasti:
  1. todo-apin ja sovelluksen yhteys (tuloksena luokat UserAccess ja TodoAccess)
     - Ennen aloittamista loin testausta varten todo-apiin käyttäjiä ja käyttäjille tehtäviä
  2. Käyttöliittymän navigaatiorakenne, eli CardLayoutin valinta ja sen toiminnallisuuden testailu
  3. Käyttötapauksien sisältämä toiminnallisuus, suunilleen seuraavassa järjestyksessä
     - kirjautuminen
     - kirjautuneen käyttäjän tehtävälistan näyttäminen
     - uuden tehtävän luominen
     - tehtävän merkkaaminen tehdyksi
     - uuden käyttäjän luominen
  4. Uusien toimintojen toteutus johti moneen kertaan ohjelman olemassaolevan rakenteen pieneen muokkaamiseen

Oliosuunnittelun periaatteita

Oliosuunnittelun periaatteita

- Ohjelmointikielet tarjoavat paljon erilaisia mekanismeja, mm. ohjelmoinnin jatkokurssillakin edellisinä viikoilla tarkastelun alla olleet perinnän ja rajapinnat
- Aloittelevalle ohjelmoijalle ei kuitenkaan ole ollenkaan selvää miten kielen mekanismeja olisi järkevä käyttää, eli minkälaista on "hyvä" ja toisaalta "huono" koodi
- Lähtökohtana on tietysti se, että koodi toteuttaa ohjelmalle asetetut vaatimukset, eli...
  1. ohjelmalla on ne ominaisuudet, joita asiakas haluaa
  2. ohjelma on riittävässä määrin virheetön
  3. ohjelma on riittävän tehokas asiakkaan tarpeisiin
- Ohjelman *sisäinen laatu*, eli se mitä toteutus- ja suunnitteluratkaisuja koodia kirjoitettaessa on käytetty, on myös tärkeää
- Jos ohjelma on sisäiseltä laadultaan huonoa, ohjelman ylläpito- ja laajennuskustannukset voivat nousta niin suuriksi että ohjelmisto muuttuu jossain vaiheessa käyttökelvottomaksi

Oliosuunnittelun periaatteita

- Aikojen saatossa on huomattu, että sisäiseltä laadultaan hyvissä ohjelmissa on tiettyjä samankaltaisia piirteitä, ja näitä tutkimalla on päädytty joukkoon hyvän oliosuunnittelun periaatteita
- **Periaatteita on useita, tarkastellaan tänään seuraavia neljää:**
  - Single responsibility principle
  - Program to an interface, not to an Implementation
  - Riippuvuuksien minimointi
  - Favour composition over inheritance

Oliosuunnittelun periaatteita

Single responsibility principle

- **Single responsibility** tarkoittaa karkeasti ottaen, että **oliolla tulee olla vain yksi vastuu** eli yksi asiakokonaisuus, mihin liittyvästä toiminnasta luokan oliot itse huolehtivat
- Robert C. Martin:
  "A class should have only one reason to change."
- Todo-sovelluksen suunnittelussa periaatetta on noudatettu suhteellisen hyvin
  - käyttölittymästä on eristetty sovelluslogiikka kokonaan
  - käyttäjän interaktioon reagoiminen on eriytetty tapahtumankäsittelijöille
  - sovelluslogiikan suorittamista koordinoi oma luokka
  - käyttäjä ja tehtävät on talletettu omiin luokkiinsa
  - todo-apin kanssa kommunikoinnin hoitavat omat luokkansa jotka vielä jaettu kahteen vastuualueeseen

Oliosuunnittelun periaatteita

Program to an interface, not to an Implementation

- "**Program to an interface, not to an Implementation**", eli ...ohjelmoi käyttämällä rajapintoja äläkä konkreettisia implementaatioita
- Laajennettavuuden kannalta ei ole hyvä idea olla riippuvainen konkreettisista luokista, sillä ne saattavat muuttua
- Parempi on tuntea vain rajapintoja (tai abstrakteja luokkia) ja olla tietämätön siitä mitä rajapinnan takana on
- Tämä mahdollistaa myös rajapinnan takana olevan luokan korvaamisen kokonaan uudella luokalla

Oliosuunnittelun periaatteita

Riippuvuuksien minimointi

- **Minimoi riippuvuudet**, eli älä tee *spagettikoodia*, jossa kaikki oliot tuntevat toisensa
- Pyri eliminoimaan riippuvuudet siten, että luokat tuntevat mahdollisimman vähän muita luokkia, ja mielellään nekin vain rajapintojen kautta

- Kerrosarkkitehtuuri tähtää osaltaan riippuvuuksien eliminointiin
- Katsomme ensi viikolla konkreettisesti sitä, miten kerrosten riippuvuuksien hallinta hoidetaan sovelluksessamme siten, että **turhien riippuvuuksien** eliminoimisen ansiosta saamme sovelluksesta helposti testattavan

Oliosuunnittelun periaatteita

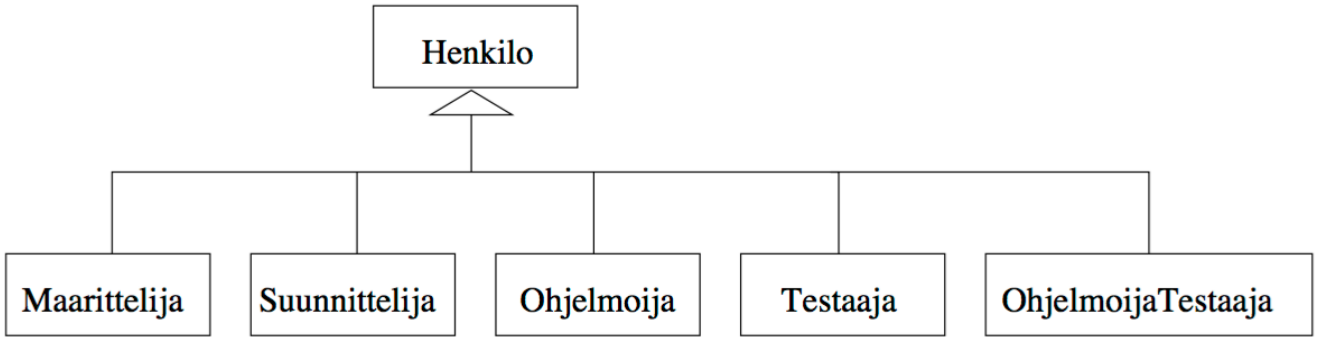
Favour composition over inheritance

- **Favour composition over inheritance** eli suosi yhteistoiminnassa toimivia olioita perinnän sijaan
- Perinnällä on paikkansa, mutta sitä tulee käyttää harkiten!
- Toissa viikolla näimme esimerkin eräästä ongelmallisesta tavasta perinnän soveltamiselle ja siitä miten ongelma katoaa kun käytetään perinnän sijaan yhteistoiminnassa olevia olioita
- Ohjelmistoyrityksen henkilöstöhallintajärjestelmä
  - henkilö voi toimia määrittelijänä, suunnittelijana, ohjelmoijana tai testaajana
  - tai samaan aikaan useassa eri tehtävässä
  - tehtävät voivat myös muuttua

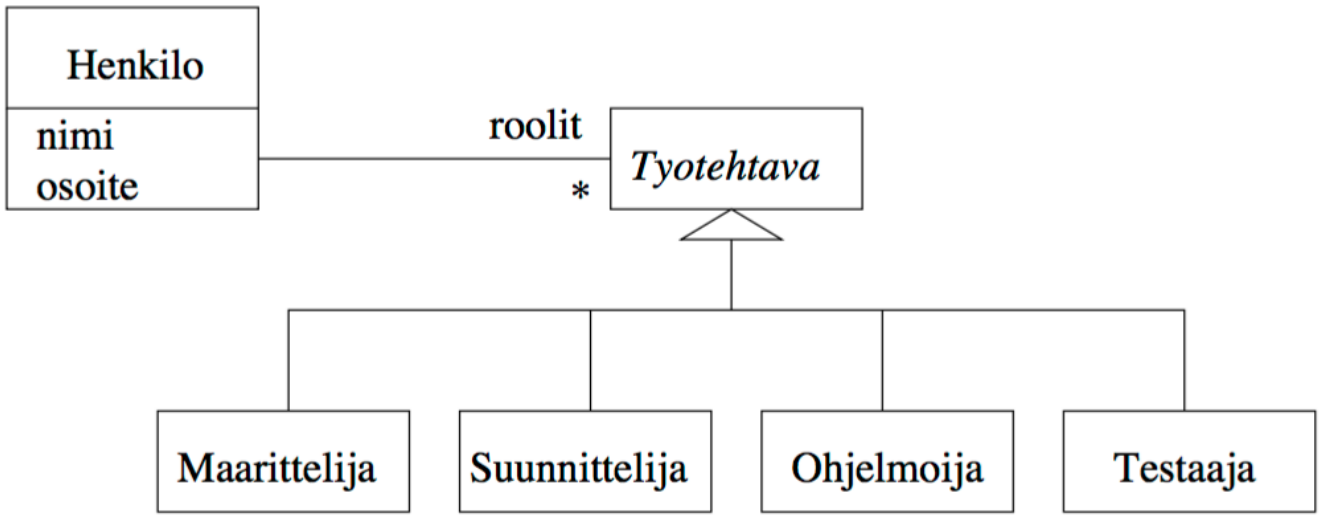
Oliosuunnittelun periaatteita

Favour composition over inheritance: roolin mallintaminen omana luokkana

- perintä johtaa moniin ongelmiin . . .



- henkilön *työroolin mallintaminen omana luokkana* ratkaisee ongelmat



Oliosuunnittelun periaatteita

Kannattaako periaatteita noudattaa?

- Onko näissä periaatteissa järkeä? Kyllä, sillä niiden noudattaminen lisää ohjelmien ylläpidettävyyttä
- Kannattaako periaatteita noudattaa: useimmiten
  - joskus kuitenkin voi olla jonkun muun periaatteen nojalla viisasta rikkoa jotain toista periaatetta...
  - Jos kyseessä "kertakäyttökoodi", ei luonnollisesti kannata panostaa ylläpidettävyyteen
- "ikiaikaisia periaatteita", motivaationa ohjelman muokattavuuden, uusiokäytettävyyden ja testattavuuden parantaminen
- Huonoa oliosuunnittelua on verrattu velan (engl. *technical debt*) ottamiseen
- Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain, mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin myöhemmin jos ohjelmaa on tarkoitus laajentaa tai muuttaa
- Joissain tilanteissa tosin velan ottaminen voi kannattaa

Koodihajut

Koodi haisee: merkki huonosta suunnittelusta

Koodihajut

Koodi haisee: merkki huonosta suunnittelusta

- Seuraavassa alan ehdoton asiantuntija Martin Fowler selittää mistä on kysymys **koodin hajuissa** (engl. code smell):
  - **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 that **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.

Koodihajut

Erilaisia koodihajuja

- **Koodihajuja on hyvin monenlaisia ja monentasoisia**
- Aloittelijankin on hyvä oppia tunnistamaan ja välttämään tavanomaisimpia
- Muutamia esimerkkejä hajuista:
  - Duplicated code (Copy&Paste -koodia)
  - Methods too big
  - Classes with too many instance variables
  - Classes with too much code
  - Uncommunicative name
  - Comments
- Internetistä löytyy paljon hajulistoja, esim:
  - `https://sourcemaking.com/refactoring/smells`
  - `http://c2.com/xp/CodeSmell.html`
  - `https://blog.codinghorror.com/code-smells/`

Koodin refaktorointi

Lääke koodihajuihin

Koodin refaktorointi

Lääke koodihajuihin

- Lääke koodihajuun on **refaktorointi** eli muutos koodin rakenteeseen, joka kuitenkin pitää koodin toiminnan ennallaan
- Erilaisia koodin rakennetta parantavia refaktorointeja on lukuisia
  - Katso esim. `http://sourcemaking.com/refactoring`
- Muutama hyvin käyttökelpoinen ja nykyaikaisessa kehitysympäristössä (esim NetBeans, Eclipse, IntelliJ) automatisoitu refaktorointi:
  - **Rename method** (rename variable, rename class, CTRL+R)
    - Eli uudelleennimetään huonosti nimetty asia
  - **Extract method**
    - Jaetaan liian pitkä metodi erottamalla siitä omia apumetodejaan
  - **Extract interface**
    - Luodaan automaattisesti rajapinta perustuen jonkun luokan metodeihin ja korvataan suora riippuvuus luokkaan riippuvuudella luotuun rajapintaan

Koodin refaktorointi

Miten refaktorointi kannattaa tehdä

- Refaktoroinnin melkein ehdoton edellytys on kattavien yksikkötestien olemassaolo
  - Refaktoroinninhan on tarkoitus ainoastaan parantaa luokan tai komponentin sisäistä rakennetta, ulospäin näkyvän toiminnallisuuden pitäisi pysyä muuttumattomana
- Kannattaa ehdottomasti edetä pienin askelin, eli 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

Test Driven Development

Test Driven Development

- Kirjoittamalla testejä ainoastaan valmiille koodille jää huomattava osa yksikkötestien hyödyistä saavuttamatta
  - Esim. refaktorointi edellyttäisi testejä
- JUnit ei ole alunperin tarkoitettu jälkikäteen tehtävien testien kirjoittamiseen, JUnitin kehittäjällä Kent Beckillä oli alusta asti mielessä jotain paljon järkevämpää ja mielenkiintoisempaa...

Test Driven Development

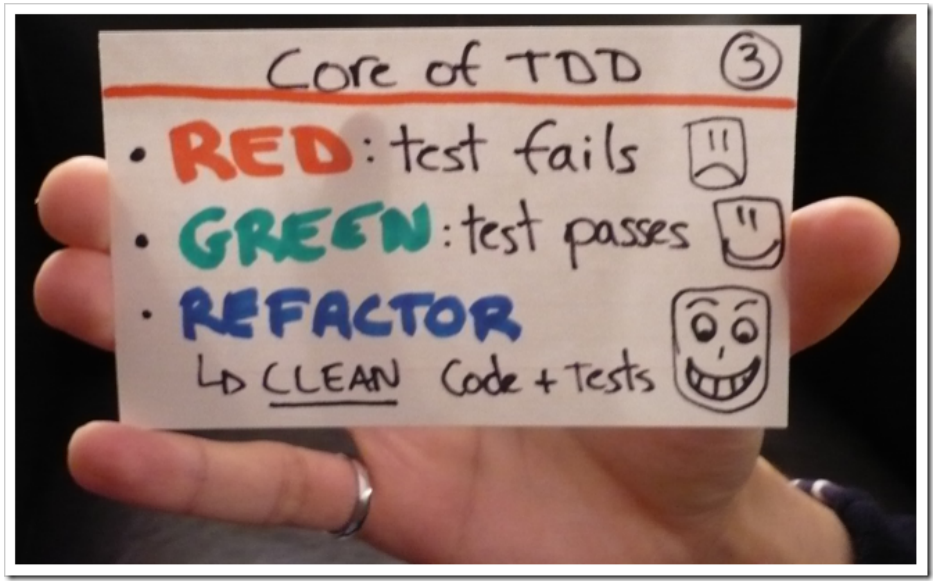
- TDD:ssä ohjelmoija (eikä siis erillinen testaaja) kirjoittaa testikoodin
- Testit laaditaan ennen koodattavan luokan toteutusta, yleensä jo ennen lopullista suunnittelua
- Sovelluskoodi kirjoitetaan täyttämään testien asettamat vaatimukset
  - Testit määrittelevät miten ohjelmoitavan luokan tulisi toimia
  - Testit toimivatkin osin koodin dokumentaationa, sillä testit myös näyttävät miten testattavaa koodia käytetään
- Testien on ennen toteutuksen valmistumista epäonnistuttava!
  - Näin pyritään varmistamaan, että testit todella testaavat haluttua asiaa

Oikeastaan TDD ei ole testausmenetelmä vaan ohjelmiston kehitysmenetelmä, joka tuottaa sivutuotteenaan automaattisesti ajettavat testit

Test Driven Development

TDD-sykli

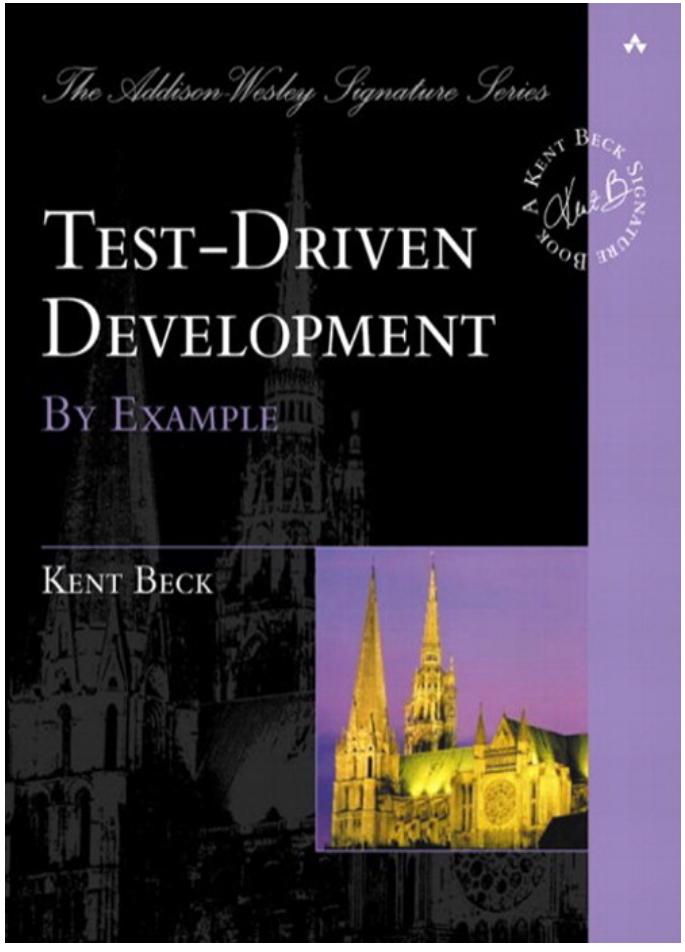
1. Tehdään **yksi** testitapaus
   - testitapaus testaa ainoastaan yhden "pienen"asian
2. Tehdään koodi joka läpäisee testitapauksen
3. Refaktoroidaan koodia, eli parannellaan koodin laatua ja struktuuria
   - Testit varmistavat koko ajan ettei mitään mene rikki
4. Kun koodin rakenne on kunnossa, palataan vaiheeseen (1)



Test Driven Development

TDD-sykli

- Automaattinen testaus ja TDD ovat usein osana ketterää ohjelmistokehitystä
- Mahdollistaa turvallisen refaktoroinnin
  - Koodi ei rupea haisemaan
  - Ohjelman rakenne säilyy laajennukset mahdollistavana
- Tämän viikon laskareiden paikanpäällä tehtävässä tehtävässä pääsemme itse kokeilemaan TDD:tä



Kertausta ja uutta asiaa testauksesta

Ohjelmistotuotantoprosessin vaiheita

- Kurssilla tutustuttu **ohjelmistotuotantoon**
  - yhteisnimitys niille työnteon ja työnjohdon menetelmille, joita käytetään, kun tuotetaan tietokoneohjelmia sekä monista tietokoneohjelmista koostuvia tietokoneohjelmistoja.
- *Ohjelmistotuotantoprosessiin* liittyy erilaisia vaiheita:
  1. Vaatimusmäärittely
  2. Suunnittelu
  3. Toteutus
  4. Testaus
  5. Ylläpito
- Perinteisesti vaiheet tehdään peräkkäin (*vesiputousmalli*)
- Vaihtoehtoinen tapa on työskennellä *iteratiivisesti*, eli toistaa vaiheita useaan kertaan peräkkäin ja limittäin (*Ketterä ohjelmistotuotanto*)
- Ennen kun jatkamme kurssin tärkeimpien teemojen kertaamista, puhutaan hieman testaamisesta

Kertausta testauksesta

- Tarkoitus on varmistaa, että ohjelmisto on riittävän laadukas käytettäväksi
- Jakautuu vaiheisiin, joita kutsutaan myös *testaustasoiksi*
  - **Yksikkötestaus**
    - Toimivatko yksittäiset metodit ja luokat kuten halutaan?
  - **Integraatiotestaus**
    - Toimivatko yksittäiset *moduulit* yhdessä halutulla tavalla?
  - **Järjestelmä/hyväksymistestaus**
    - Toimiiko kokonaisuus niin kuin vaatimusdokumentissa sanotaan?
  - **Regressiotestaus**
    - järjestelmälle muutosten ja bugikorjausten jälkeen ajettavia testejä jotka varmistavat että muutokset eivät riko mitään
    - Regressiotestit koostuvat yleensä yksikkö-, integraatio- ja järjestelmä/hyväksymätesteistä

Lähes kaikki kurssilla tähän mennessä kirjoitetut testit ovat olleet yksikkötestejä, jotka on automatisoitu JUnitin avulla

Kertausta testauksesta

Testien laatu

- Normaalien syötteiden lisäksi on testeissä erityisen tärkeää tutkia testattavien metodien toimintaa *virheellisillä* syötteillä ja erilaisilla *raja-arvoilla*
- Testien laatua on kurssin aikana mitattu kolmella tavalla:
  - **Rivikattavuus** mittaa miten montaa prosenttia koodiriveistä testit suorittavat
  - **Haarautumakattavuus** taas mittaa miten monta prosenttia koodin haarautumakohdista (if:in ja toistolauseiden ehdot) on suoritettu
  - **Mutaatiotestauksella** tarkasteltiin huomaavatko testit koodiin asetettuja bugeja
    - Mutaatiotestauskirjasto tekee koodiin pieniä muutoksia, eli mutantteja, esim. muuttaa ehdossa <:n <=:ksi
    - Jos testit eivät mene mutanttien takia rikki, on epäilys että testit eivät huomaisi koodissa olevaa virhettä ja testejä tulee parantaa
- Kun käytimme Maven-käännösympäristöä, oli testien laatumittareiden käyttö helppoa...

Kertausta testauksesta

Testaus ja riippuvuuksien eliminointi

- Testaus muuttuu joskus tarpeettoman hankalaksi, jos ohjelmissa on luokkien välillä tarpeettomia riippuvuuksia
  - Eräs viime viikon laskareissa kokeillun Test Driven Development (TDD) -menetelmän hyviä puolia on se, että koodista tulee "automaattisesti" hyvin testattavissa olevaa ja turhia riippuvuuksia ei koodissa yleensä esiinny
- Seuraavalla kalvolla Ohjelmoinnin perusteiden viikon 3 tehtävän *Lottoarvonta* ratkaisu
  - Lottorivi-olion konstruktori luo Random-olion, jonka avulla lottonumerot arvotaan
- Luokan testaaminen on nyt erittäin vaikeaa, testeissähän ei pystytä kontrolloimaan Random-olion arpomia lukuja
- Lottorivistä tulisi testata ainakin seuraavat asiat
  - lottorivi ei voi sisältää samaa arvoa useampaan kertaan
  - lottorivi voi saada arvoja väliltä 1-39

Kertausta testauksesta

Lottorivi ja riippuvuuden injektointi

```
public class Lottorivi {
    private ArrayList<integer> numerot;
    private Random random;

    public Lottorivi() {
        random = new Random();
        arvoNumerot();
    }

    public ArrayList<integer> numerot() {
        return numerot;
    }

    public void arvoNumerot() {
        numerot = new ArrayList<>();
        while (numerot.size() < 7) {
            int numero = random.nextInt(39) + 1;
            if (!numerot.contains(numero)) {
                numerot.add(numero);
            }
        }
    }
}
```

Kertausta testauksesta

Lottorivi ja riippuvuuden injektointi

- Ongelmana Lottorivin testaamisessa ei sinänsä ole riippuvuus Random-olioon, vaan se, että lottorivin ulkopuolelta ei ole tapaa päästä käsiksi Lottorivin luomaan Random-olioon
- Muutetaan lottorivin konstruktoria seuraavasti:

```
public Lottorivi(Random random)
```

- Eli lottorivi muodostetaan nyt luomalla Random-olio, joka annetaan lottoriville konstruktorin parametrina

```
Random random = new Random();
Lottorivi lotto = new Lottorivi( random );
```

- Tekniikasta käytetään nimitystä **riippuvuuksien injektointi** (engl. *dependency injection*)
  - riippuvuutena olevaa olioa ei luoda konstruktorissa tai luokan sisällä
  - konstruktorille annetaan valmiiksi luotu riippuvuus, konstruktori laittaa riippuvuutena olevan olion talteen oliomuuttujaan

Kertausta testauksesta

Lottorivi ja riippuvuuden injektointi

```
public class Lottorivi {
        private ArrayList<Integer> numerot;
        private Random random;

        public Lottorivi(Random random) {
                this.random = random;
                arvoNumerot();
        }

        public Lottorivi() {  // olio mahdollista luoda myös vanhaan tapaan
                this(new Random());
        }

        public ArrayList<Integer> numerot() {
                return numerot;
        }

        public void arvoNumerot() {
                numerot = new ArrayList<>();
                while (numerot.size() < 7) {
                        int numero = random.nextInt(39) + 1;
                        if (!numerot.contains(numero)) {
                                numerot.add(numero);
                        }
                }
        }
}
```

Kertausta testauksesta

Lottorivi ja riippuvuuden injektointi

- Koska lottorivi saa käyttämänsä Random-olion konstruktorin parametrina, voimme helposti luoda *omia versioitamme Randomista* ja käyttää niitä apuna testaamisessa
- Seuraavassa luokka *RandomStub*, joka "arpoo" sille konstruktorin parametrina annetut luvut

```
public class RandomStub extends Random {
        List<Integer> numbers;
        public RandomStub(Integer ... luvut) {
            numbers = new ArrayList<>();
            numbers.addAll(Arrays.asList(luvut));
        }

        @Override // korvataan peritty toiminnallisuus
        public int nextInt(int bound) {
            return numbers.remove(0);
        }
}
```

- Koska RandomStub *perii* luokan Random, voidaan sitä käyttää kaikkialla randomin paikalla

Kertausta testauksesta

Lottorivi ja riippuvuuden injektointi

- Testaaminen muuttuu nyt helpoksi
- Seuraavassa testi, joka varmistaa, että sama luku ei esiinny useaan kertaan lottonumerossa, vaikka Random-olio palauttaakin saman luvun monta kertaa

```
public class LottoriviTest {
    @Test
    public void eiSamojaNumeroita() {
        Random randomStubi = new RandomStub(1,1,1,2,3,4,5,6,7);
        Lottorivi rivi = new Lottorivi(randomStubi);
        List<Integer> odotettu = Arrays.asList(2,3,4,5,6,7,8);
        assertSamatNumerot(odotettu, rivi.numerot());
    }

    private void assertSamatNumerot(List<Integer> odotettu, List<Integer> tulos) {
        assertEquals(odotettu.size(), tulos.size());
        for (Integer luku : tulos) {
            assertTrue(odotettu.contains(luku));
       }
    }
}
```

Integraatiotestaus

Integraatiotestaus

- Yksikkötestaus on käsitteenä suhteellisen hyvin ymmärretty
  - yksikkötestit kohdistuvat yhteen luokkaan
- Järjestelmä/hyväksymätestauksessa taas testataan ohjelmiston tarjoamaa toiminnallisuutta käyttöliittymän läpi samaan tapaan kuin loppukäyttäjä tulee järjestelmää käyttämään
- Kaikki näiden väliin jäävät testauksen muodot ovat *integraatiotestausta*
- Kurssilla on jo nähty muutamia esimerkkejä integraatiotesteistä
- Viikon 6 laskareissa olleen palkanlaskennan testit ovat oikeastaan integraatiotestejä https://github.com/mluukkai/OTM2016/tree/master/koodi/Palkanl
- Vaikka testit kohdistuvatkin luokan Palkanlaskenta metodeihin, ne kuitenkin oleellisesti testaavat palkanlaskennasta, henkilöistä ja muutamasta muusta luokasta muodostuvan kokonaisuuden toimintaa

Integraatiotestaus

Integraatiotestaus ja ”hankalat” riippuvuudet

- Testien kannalta on hieman ikävää, jos testattavan järjestelmän jokin komponentti keskustelee tietokannan tai jonkin verkossa olevan komponentin kanssa
  - Testien suoritus hidastuu
  - Testien tulos saattaa olla riippuvainen tietokannan sisällöstä
  - Verkon yhteysongelmat voivat vaikuttaa testien toimivuuteen
- Näissä tilanteissa on järkevää toteuttaa hankalista luokista testausta varten valekomponentti eli *stub* ja injektoida se testattavalle järjestelmälle hankalan riippuvuuden sijaan
  - Stubin tulee toimia testattavan järjestelmän kannalta kuten oikea komponentti
- Tämä tietysti edellyttää, että riippuvuudet pystytään injektoimaan testattaville luokille samoin kuin lottoesimerkissä
- Stub-olio on helppo luoda perimällä alkuperäinen riippuvuus ja *korvata* kokonaan siinä testien kannalta merkityksellisten metodien toiminnallisuus

Integraatiotestaus

Integraatiotestaus ja ”hankalat” riippuvuudet

- Palkkojen maksun yhteydessä suoritetaan tilisiirto luokan MaksupalveluRajapinta metodilla *suoritaTilisiirto*
- Konkreettisen tilisiirron tekeminen on tietenkin testien kannalta täysin tarpeetonta
- Palkanlaskennan mallivastauksen testeissä käytetäänkin todellisen maksupalvelurajapinnan sijaan stub-olioa:

```
public class MaksupalveluStub extends MaksupalveluRajapinta {
    @Override
    public void suoritaTilisiirto(Maksusuoritus maksu) {
        // stubin ei tarvitse tehdä mitään sillä Palkanlaskenta ainoastaan
    }
}

public class PalkanlaskentaTest {
    Palkanlaskenta p;
    @Before
    public void setUp() {
        Map<String, OutputBuilder> outputBuilders = new HashMap<>();
        outputBuilders.put("csv", new CsvBuilder());
        p = new Palkanlaskenta(new MaksupalveluStub(), outputBuilders);
    }
```

Järjestelmätestaus

Järjestelmätestaus

Järjestelmätestauksessa tulee varmistaa toimiiko ohjelmisto sille vaatimusmäärittelyssä asetettujen vaatimusten mukaan

- Testauksen tulee tapahtua saman rajapinnan läpi, miten loppukäyttäjä järjestelmää käyttää, eli sovelluksesta riippuen joko komentoriviltä, graafisesta käyttöliittymästä tai webselaimesta
- Järjestelmätestit kannattaa muodostaa ohjelmiston käyttötapausten mukaan
- Tarkastellaan Ohjelmoinnin jatkokurssin ensimmäisen viikon tehtävää *Laskin*
- Käyttäjän syötteiden simuloiminen on mahdollista korvaamalla ohjelman käyttämä syötevirta *System.in* omalla oliolla
- Vastaavasti voimme korvata tulostusvirran *System.out*

Järjestelmätestaus

Laskimen testi, joka korvaa syöte- ja tulostevirran

- Järjestelmätestauksen voi hoitaa myös JUnit-kirjaston avulla.
  - JUnit ei ole tarkoitukseen optimaalinen, mutta ”riittävän hyvä”, ja Javalle ei kauheasti parempiakaan vaihtoehtoja ole tarjolla

```
public class LaskinTest {
    private ByteArrayOutputStream tulosvirta;

    @Beforepublic void setUp() {
        tulosvirta = new ByteArrayOutputStream();
        System.setOut(new PrintStream(tulosvirta));
    }

    @Test
    public void summaJaLopetus() {
        String syote = "summa\n"+"1\n"+"2\n"+"lopetus\n";
        System.setIn(new ByteArrayInputStream(syote.getBytes()));
        Laskin laskin = new Laskin();laskin.kaynnista();
        String tulostus = tulosvirta.toString();

        assertTrue(tulostus.contains("summa 3"));
        assertTrue(tulostus.contains("laskuja laskettiin 1"));
    }
}
```

Järjestelmätestaus

Laskimen testi, joka korvaa syöte- ja tulostevirran

- Testin simuloitu syöte on merkkijono, jossa yksittäiset syöterivit päättyvät rivinvaihtoon
- Merkkijonosta luodaan bittivirta, joka asetetaan *ohjelman syötevirran* arvoksi
- setUp-metodissa testi asettaa uuden ByteArrayOutputStream-olion ohjelman tulosvirraksi metodin System.setOut avulla
- Laskimen suorituksen jälkeen tulosvirtaolio muutetaan toString-metodilla merkkijonoksi ja tarkastetaan, että ohjelman tulostus on odotetun kaltainen
- Testit olettavat, että ohjelmassa luodaan ainoastaan yksi Scanner-olio
- Vaihtoehtoinen ratkaisu syötevirran korvaamiselle olisi ollut muuttaa laskinta siten, että lukija-olio oltaisiin injektoitu konstruktorin parametrina

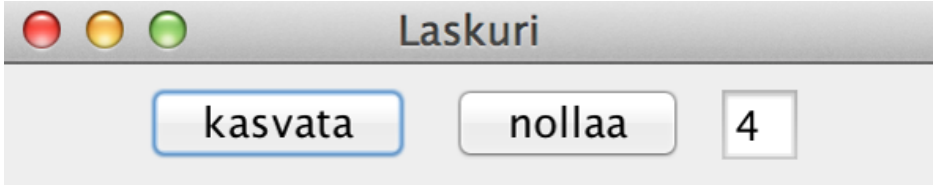
Graafisen käyttöliittymän testaus

Java-Swing käyttöliittymän testausta

Graafisen käyttöliittymän testaus

Java-Swing käyttöliittymän testausta

- Tarkastellaan yksinkertaista graafista ohjelmaa, jonka avulla laskurin arvoa voidaan kasvattaa tai laskuri voidaan nollata



- Javan Swing-kirjastolla tehtyjen graafisten sovellusten testaaminen ei loppujenlopuksi ole kovin vaikeaa
- Asetetaan komponenteille nimet:

```
private void luoKomponentit(Container container) {
    add = new Jbutton("kasvata");
    add.setName("kasvata");
    reset = new Jbutton("nollaa");
    reset.setName("nollaa");
    reset.setEnabled(false);
    show = new JTextField("  "+value);
    show.setName("tulos");
}
```

Graafisen käyttöliittymän testaus

Java-Swing käyttöliittymän testausta

- AssertJ [1] on johtava työkalu Swing-sovellusten testaamiseen:

```
public class LaskuriTest extends AssertJSwingJUnitTestCase {
    private FrameFixture window;

    @Test
    public void laskuriAlussaNolla() {
        window.textBox("tulos").requireText("0");
    }

    @Test
    public void laskuriKasvaa() {
        window.button("kasvata").click();
        window.textBox("tulos").requireText("1");
        window.button("kasvata").click();
        window.button("kasvata").click();
        window.textBox("tulos").requireText("3");
    }
}
```

- window-olio tarjoaa pääsyn eri komponentteihin. Normaalisti käytettävien assert-lauseiden sijaan testien odotettu tulos määritellään require-määreiden avulla

[1] http://joel-costigliola.github.io/assertj/

Kurssikoe

Tietoa kurssikokeesta

Kurssikoe

Tietoa kurssikokeesta

- **Koe tilaisuudet:**
  1. Ke 23.8.2017 klo 17-20 $\leftarrow$ suositus
  2. Ma 18.9.2017 klo 17-20
  3. Ke 18.10.2017 klo 17-20
- **Ilmoittautuminen viimeistään 10pv ennen koetilaisuutta**
- Nippelitason detaljien sijasta kokeessa arvostetaan enemmän sovellusosaamista
  - Tosin eräät detaljit ovat tärkeitä (seuraavat diat)
- Kokeeseen saa tuoda mukanaan 2-puoleisen, **itse tehdyn, käsin kirjoitetun** yhden A4-arkin lunttilapun.
- Lunttilapun teossa siis ei saa käyttää kopiokonetta, tietokonetta tai printteriä

Kurssikoe

Mikä on tärkeintä kurssilla/kokeessa?

- Kokonaiskuva: *Mitä? Miksi? Milloin? Missä? Miten?*
- **Ohjelmistotuotantoprosessin** vaiheet on syytä osata
- **Testaamisen rooli** ohjelmistotuotantoprosessissa ymmärrettävä
  - testejäkin saattaa joutua kirjoittamaan
  - kannattaa kerrata laskareiden testaukseen liittyvät asiat
- UML:sta ylivoimaisesti tärkeimpiä ovat
  - **Luokkakaaviot**
  - **Sekvenssikaaviot**
  - **Käyttötapauskaaviot**

Kokeessa on paljon tehtävää, joten käytä aika hyvin.

Kurssikoe

Mikä on tärkeintä kurssilla/kokeessa?

- **Käyttötapausmallinnus**
- **Käsiteanalyysi**, eli määrittelyvaiheen luokkamallin muodostaminen tekstistä
- **Takaisinmallinnus**, eli valmiista koodista luokka- ja sekvenssikaavioiden teko
- **Ohjelmiston suunnittelu**
  - jakautuminen *ohjelmiston arkkitehtuuriin*
  - ja oliosuunnitteluun ja sen periaatteisiin
  - Koodihajun käsite ja pari esimerkkiä
  - Refaktoroinnin käsite
  - Ymmärrys, miksi periaatteet ovat olemassa ja milloin niitä saa rikkoa ja mitä käy jos niitä rikkoo (tekninen velka)