Ohjelmistotekniikan menetelmät

Matti Luukkainen ja 7 ohjaajaa

9.3.2018

Ohjelmistotekniikan menetelmät

- Kurssilla tutustutaan ohjelmistokehityksen periaatteisiin sekä menetelmiin ja sovelletaan niitä toteuttamalla pieni harjoitustyö
- Kurssi nykyään *aineopintoja*
- Pakollisina *esitietoina*
  - Ohjelmoinnin jatkokurssi
  - Tietokantojen perusteet
- Hyödyllinen esitieto. Tietokone työvälineenä
- Kurssimateriaali https://github.com/mluukkai/otm-2018
- Ikävä kyllä kurssilla sama nimi kuin “vanhalla” Ohjelmistotekniikan menetelmillä. Sisätö ja suoritusmuoto nyt radikaalisti erilainen

Suoritusmuoto

- Kolmella ensimmäisellä viikolla ohjauksessa tai omatoimisesti tehtävät laskarit
  - palautetaan “internettiin”
- Viikolla 2 aloitetaan itsenäisesti tehtävä harjoitustyö
- Työtä edistetään pala palalta viikoittaisten tavoitteiden ohjaamana
- Kurssilla ei ole koetta
- Harjoitustyö tulee tehdä kurssin aikataulujen puitteissa
- Kesken jäänyttä harjoitustyötä ei voi jatkaa seuraavalla kurssilla (keväällä 2019)
- Muista siis varata riittävästi aikaa (10-15h viikossa) koko periodin ajaksi!

Luento, deadlinet ja ohjaus

- Ennakkotiedoista poiketen kurssilla on vain yksi luento **nyt** eli ma 12.3. klo 14-16 A111
- Laskareiden ja harjoitustyön välitavoitteiden viikoittaiset deadlinen *tiistaina klo 23:59*
- Paja salissa B221

alku	ma	ti	ke	to	pe
10			x		
12	x	x	x		
14	x	x	x		
16		x			x
18		x			

Arvosteluperusteet

- Jaossa 60 pistettä jotka jakautuvat seruaavasti
  - Viikkodeadlinet 17p
  - Koodikatselmointi 3p
  - Dokumentaatio 10p
  - Testaus 7p
  - Lopullinen ohjelma 23p
    - Laajuus, ominaisuudet ja koodin laatu
- Arvosanaan 1 riittää 30 pistettä, arvosanaan 5 tarvitaan noin 55 pistettä.
- Läpipääsyyn vaatimuksena on lisäksi vähintään 10 pistettä lopullisesta ohjelmasta.

TEORIA

Ohjelmistotuotanto

- Kun ollaan tekemässä suurempaa ohjelmistoa ulkoiselle asiakkaalle, tarvitaan systemaattinen työskentelymenetelmä
  - muuten riskinä mm. että lopputulos ei vastaa asiakkaan tarvetta
- Ohjelmiston systemaattinen kehittäminen, eli ohjelmistotuotanto (engl. Software engineering) sisältää useita erilaisia aktiviteettejä/vaiheita
  - *vaatimusmäärittely* selvitetään kuinka ohjelmiston halutaan toimivan
  - *suunnittelu* mietitään, miten halutunkaltainen ohjelmisto tulisi rakentaa
  - *toteutusvaiheessa* määritelty ja suunniteltu ohjelmisto koodataan
  - *testauksen* tehtävä on varmistaa ohjelmiston laatu, että se ei ole liian buginen ja että se toimii kuten vaatimusmäärittely sanoo
  - *ylläpitovaiheessa* ohjelmisto on jo käytössä ja siihen tehdään bugikorjauksia ja mahdollisia laajennuksia

Vaatimusmäärittely

- Kartoitetaan ohjelman tulevien käyttäjien tai tilaajan kanssa se, mitä toiminnallisuutta ohjelmaan halutaan
- Ohjelman toiminnalle siis asetetaan asiakkaan haluamat vaatimukset.
- Tämän lisäksi kartoitetaan ohjelman toimintaympäristön ja toteutusteknologian järjestelmälle asettamia rajoitteita
- Tuloksena on useimmiten jonkinlainen dokumentti, johon vaatimukset kirjataan.
- Dokumentin muoto vaihtelee suuresti, se voi olla paksu mapillinen papereita tai vaikkapa joukko postit-lappuja

Vaatimusten kirjaaminen

- On olemassa lähes lukemattomia tapoja dokumentoida vaatimuksen
- Kurssin aiemmissa versioissa käyttäjien vaatimukset dokumentointiin *käyttötapauksina* (engl. use case)
  - tapa on jo vanhahtava ja hylkäämme sen
- Kurssilla ohjelmistotuotanto tutustumme nykyään yleisesti käytössä oleviin *käyttäjätarinoihin* (engl. user story)
- Käytämme tällä kurssilla hieman kevyempää tapaa, ja kirjaamme järjestelmältä toivotun toiminnallisuuden vapaamuotoisena ranskalaisista viivoista koostuvana feature-listana

Kurssin esimerkkisovellus: TodoApp

- *todoapp* eli sovellus, jonka avulla käyttäjien on mahdollista pitää kirjaa omista tekemättömistä töistä, eli *todoista*
- Vaatimusmäärittely aloitetaan tunnistamalla järjestelmän erityyppiset *käyttäjäroolit*
- Todo-sovelluksesta tunnistetaan kaksi käyttäjäroolia
  - normaaleja käyttäjät
  - laajemmilla oikeuksilla varustetut ylläpitäjät
- kun sovelluksen käyttäjäroolit ovat selvillä, mietitään mitä toiminnallisuuksia kunkin käyttäjäroolin halutaan pystyvän tekemään sovelluksen avulla

TodoApp:in vaatimusmäärittely

- Todo-sovelluksen normaalien käyttäjien toiminnallisuuksia ovat esim. seuraavat
  - käyttäjä voi luoda järjestelmään käyttäjätunnuksen
  - käyttäjä voi kirjautua järjestelmään
  - kirjautumisen jälkeen käyttäjä näkee omat tekemättömät työt eli todot
  - käyttäjä voi luoda uuden todon
  - käyttäjä voi merkitä todon tehdyksi, jolloin todo häviää listalta
- Ylläpitäjän toiminnallisuuksia esim. seuraavat
  - ylläpitäjä näkee tilastoja sovelluksen käytöstä
  - ylläpitäjä voi poistaa normaalin käyttäjätunnuksen

Vaatimusmäärittely: toimintaympäristön rajoitteet, käyttöliittymä

- Ohjelmiston vaatimuksiin kuuluvat myös *toimintaympäristön rajoitteet*
- Todo-sovellusta koskevat seuraavat rajoitteet
  - ohjelmiston tulee toimia Linux- ja OSX-käyttöjärjestelmillä varustetuissa koneissa
  - käyttäjien ja töiden tiedot talletetaan paikallisen koneen levylle
- Vaatimusmäärittelyn aikana hahmotellaan yleensä myös sovelluksen käyttöliittymä

Vatimusten kirjaamisesta voi ottaa tarkemmin mallia sovelluksen GitHub-repositoriosta https://github.com/mluukkai/OtmTodoApp

Suunnittelu

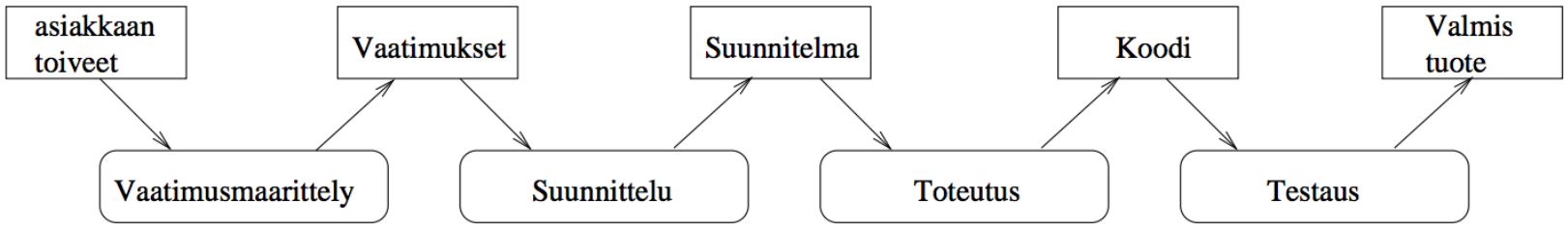
- Ohjelmiston suunnittelu jakautuu yleensä kahteen erilliseen vaiheeseen
- *Arkkitehtuurisuunnittelussa* määritellään ohjelman rakenne karkealla tasolla
  - mistä suuremmista rakennekomponenteista ohjelma koostuu
  - miten komponentit yhdistetään, eli minkälaisia komponenttien väliset rajapinnat ovat
  - mitä riippuvuuksia ohjelmalla on esim. tietokantoihin ja ulkoisiin rajapintoihin
- Arkkitehtuurisuunnittelua tarkentaa *oliosuunnittelu*, missä mietitään ohjelmiston yksittäisten komponenttien rakennetta
  - eli minkälaisisista luokista komponentit koostuvat
  - miten luokat kutsuvat toistensa metodeja sekä mitä apukirjastoja luokat käyttävät.
- Myös ohjelmiston suunnittelu, erityisesti sen arkkitehtuuri dokumentoidaan usein jollain tavalla

Testaus

- Toteutuksen yhteydessä ja sen jälkeen järjestelmää testataan.
- Testausta on monentasoista.
- **Yksikkötestauksessa** tutkitaan yksittäisten metodien ja luokkien toimintaa.
  - Yksikkötestauksen tekee usein testattavan komponentin ohjelmoija.
- Kun erikseen ohjelmoidut komponentit (eli luokat) yhdistetään, suoritetaan **integraatiotestaus**
  - varmistetaan erillisten komponenttien yhteentoimivuus
  - integraatiotestaus tapahtuu useimmiten ohjelmoijien toimesta
- **Järjestelmätestauksessa** testataan ohjelmistoa kokonaisuutena ja verrataan, että se toimii vaatimusdokumentissa sovitun määritelmän mukaisesti
  - järjestelmätestaus suoritetaan ohjelman todellisen käyttöliittymän kautta
  - järjestelmätestauksen saattaa tapahtua erillisen laadunhallintatiimin toimesta

Vesiputousmalli

- Ohjelmistoja on perinteisesti tehty vaihe vaiheelta etenevän *vesiputousmallin* (engl. waterfall model) mukaan.
- Vesiputousmallissa edellä esitellyt ohjelmistotuotannon vaiheet suoritetaan peräkkäin



- usein eri vaiheet erillisten tiimien tekemiä
- edellyttää raskasta dokumentaatiota

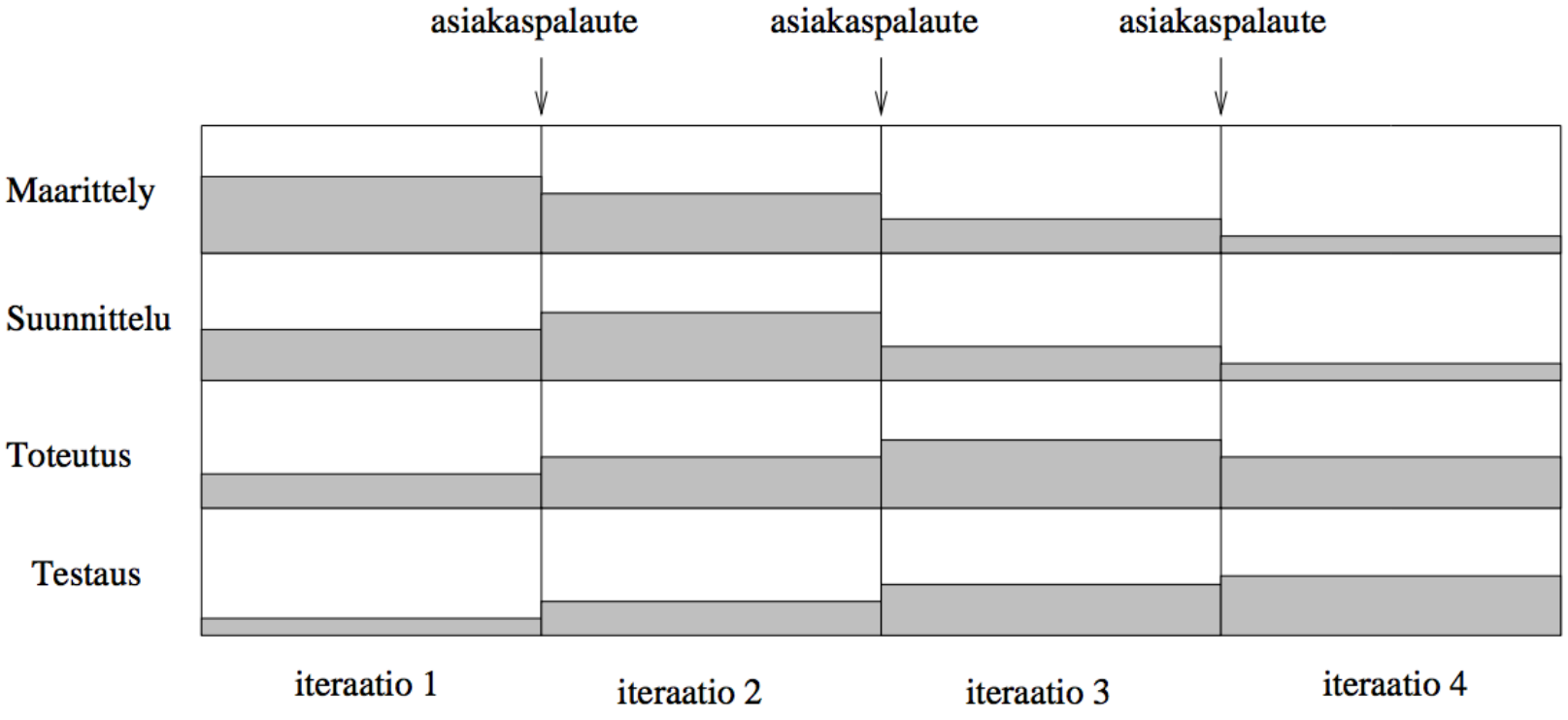
Vesiputousmallin ongelmat

- Mallin toimivuus perustuu siihen oletukseen, että ohjelman vaatimukset pystytään määrittelemään täydellisesti ennen kuin suunnittelu ja ohjelmointi alkaa
  - Näin ei useinkaan ole. On lähes mahdotonta, että asiakkaat pystyisivät tyhjentävästi ilmaisemaan kaikki ohjelmalle asettamansa vaatimukset
  - Vasta käyttäessään valmista ohjelmaa asiakkaat alkavat ymmärtää, mitä he olisivat ohjelmalta halunneet
  - Vaikka ohjelman vaatimukset olisivat kunnossa vaatimusten laatimishetkellä, muuttuu toimintaympäristö ohjelman kehitysaikana niin ratkaisevasti, että valmistuessaan ohjelma on vanhentunut
- Toinen suuri ongelma on se, että ohjelmistoa aletaan testata verrattain myöhäisessä vaiheessa
  - Erityisesti integraatiotestauksessa on tyypillistä että ohjelmasta löydetään pahoja ongelmia, joiden korjaaminen hidastaa ohjelmiston valmistumista paljon ja käy kalliiksi

Ketterä ohjelmistokehitys

- Vesiputousmallin heikkoudet ovat johtaneet viime vuosina yleistyneiden *ketterien (engl. agile) ohjelmistokehitysmenetelmien* käyttöönottoon
- Ketterä ohjelmistokehitys etenee yleensä siten, että ensin kartoitetaan pääpiirteissään ohjelman vaatimuksia ja ehkä hahmotellaan järjestelmän alustava arkkitehtuuri.
- Tämän jälkeen suoritetaan useita *iteraatioita* (joista käytetään yleisesti myös nimitystä sprintti), joiden aikana ohjelmaa rakennetaan pala palalta eteenpäin
- Kussakin iteraatiossa suunnitellaan ja toteutetaan valmiiksi pieni osa ohjelman vaatimuksista
- Asiakas pääsee kokeilemaan järjestelmää jokaisen iteraation jälkeen. Tällöin voidaan jo aikaisessa vaiheessa todeta, onko kehitystyö etenemässä oikeaan suuntaan ja vaatimuksia voidaan tarvittaessa tarkentaa ja muuttaa

Ketterä ohjelmistokehitys



Teemme kurssin harjoitustyötä ketterässä hengessä viikon mittaisilla iteraatioilla

TYÖKALUJA

Työkaluja

- Tarvitsemme ohejelmisokehityksessä suuren joukon käytännön työkaluja.
- Komentorivi ja Versionhallinta
  - Olet jo ehkä käyttänyt muilla kursseilla komentoriviä ja git-versionohallintaa, molemmat ovat tärkeässä roolissa ohjelmistokehityksessä ja niiden harjoittelu on aiheena viikon 1 laskareissa
- Maven
  - Olet todennäköisesti ohjelmoinut Javaa NetBeansilla ja tottunut painamaan “vihreää nappia” tai “mustaa silmää”
  - tutkimme kurssilla hieman miten Javalla tehdyn ohjelmiston hallinnointi (esim. koodin kääntäminen, koodin sekä testin suorittaminen ja koodin paketoiminen NetBeansin ulkopuolella suoritettavissa olevaksi jar-paketiksi) tapahtuu NetBeansin “ulkopuolella”
  - Java-projektien hallinnointiin on olemassa muutama vaihtoehto. Käytämme *mavenia*, joka lienee jo useimmille osittain tuttu esim. Tietokantojen perusteista

JUnit

- Ohjelmistojen testaus tapahtuu nykyään ainakin yksikkö- ja integraatiotestien osalta automatisoitujen testityökalujen toimesta
- Java-maailmassa testausta dominoi lähes yksinvaltiaan tavoin JUnit. Tulet kurssin ja myöhempienkin opintojesi aikana kirjoittamaan paljon JUnit-testejä
- Viiko 2 laskareissa harjoitellaan JUnitin perusteita

Checkstyle

- Automaattisten testien lisäksi koodille voidaan määritellä erilaisia automaattisesti tarkastettavia tyylillisiä sääntöjä
- Näiden avulla on mahdollista ylläpitää koodin luettavuutta ja varmistaa että joka puolella koodia noudatetaan samoja tyylillisiä konventioita
- Käytämme kurssilla tarkoitukseen Checkstyle-nimistä työkalua
- Ohjelmoinnin perusteet ja jatkokurssi käyttivät Checkstyleä valvomaan ohjelman sisennystä
- Kurssilla kontrolloimme mm. muuttujien nimiä, sulkumerkkien sijoittelua, välilyönnin käytön systemaattisuutta

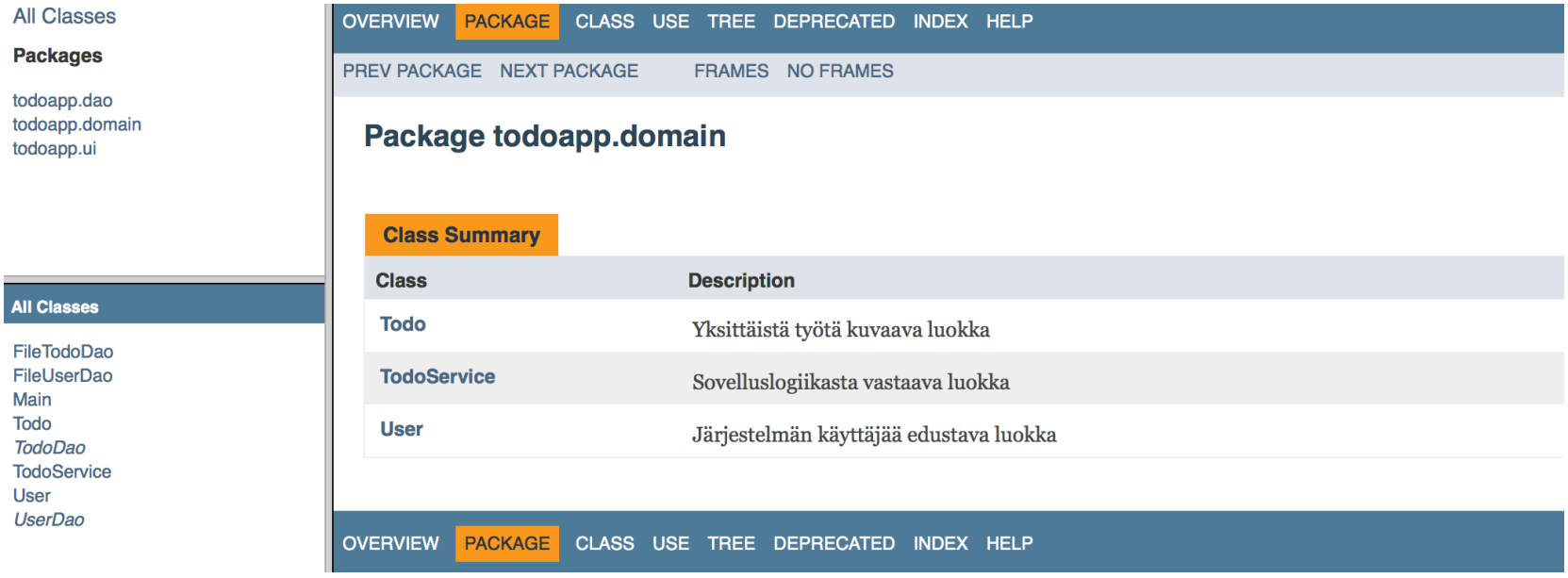
JavaDoc

- Osa ohjelmiston dokumentointia on lähdekoodinluokkien julkisten metodien kuvaus
- Javassa lähdekoodi dokumentoidaan käyttäen JavaDoc-työkalua
- Dokumentointi tapahtuu kirjoittamalla koodin yhteyteen sopivasti muotoiltuja kommentteja

```
5   /**
6   * Uuden todon lisääminen kirjautuneena olevalle käyttäjälle
7   *
8   * @param   content   luotavan todon sisältö
9   *
0   * @return true jos todon luominen onnistuu, muuten false
1   */
2
3   public boolean createTodo(String content) {
4       Todo todo = new Todo(content, loggedIn);
```

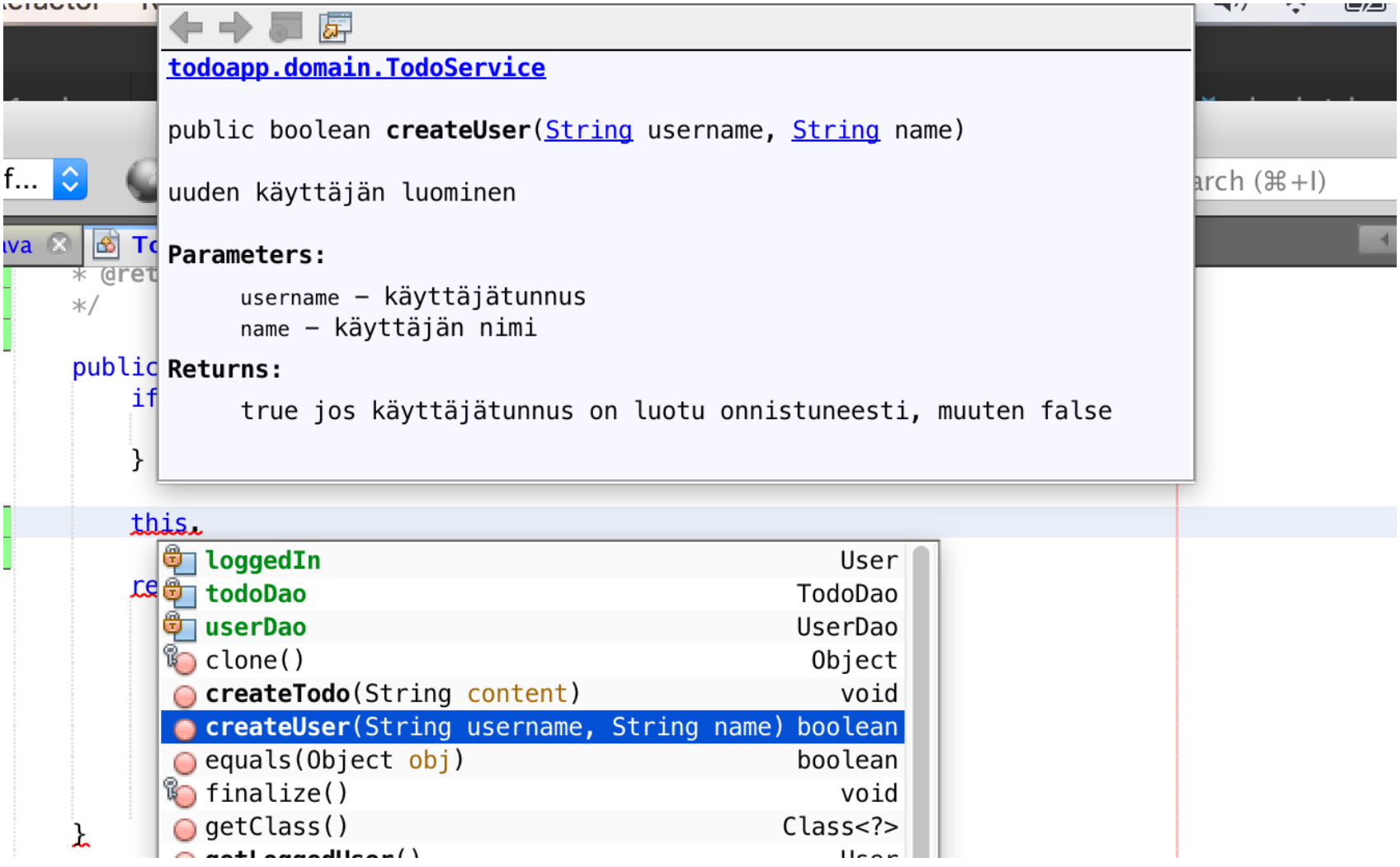
JavaDoc

Sovelluksen JavaDocia voi tarkastella selaimen avulla



JavaDoc

NetBeans osaa näyttää ohjelmoidessa koodiin määritellyn javadocin



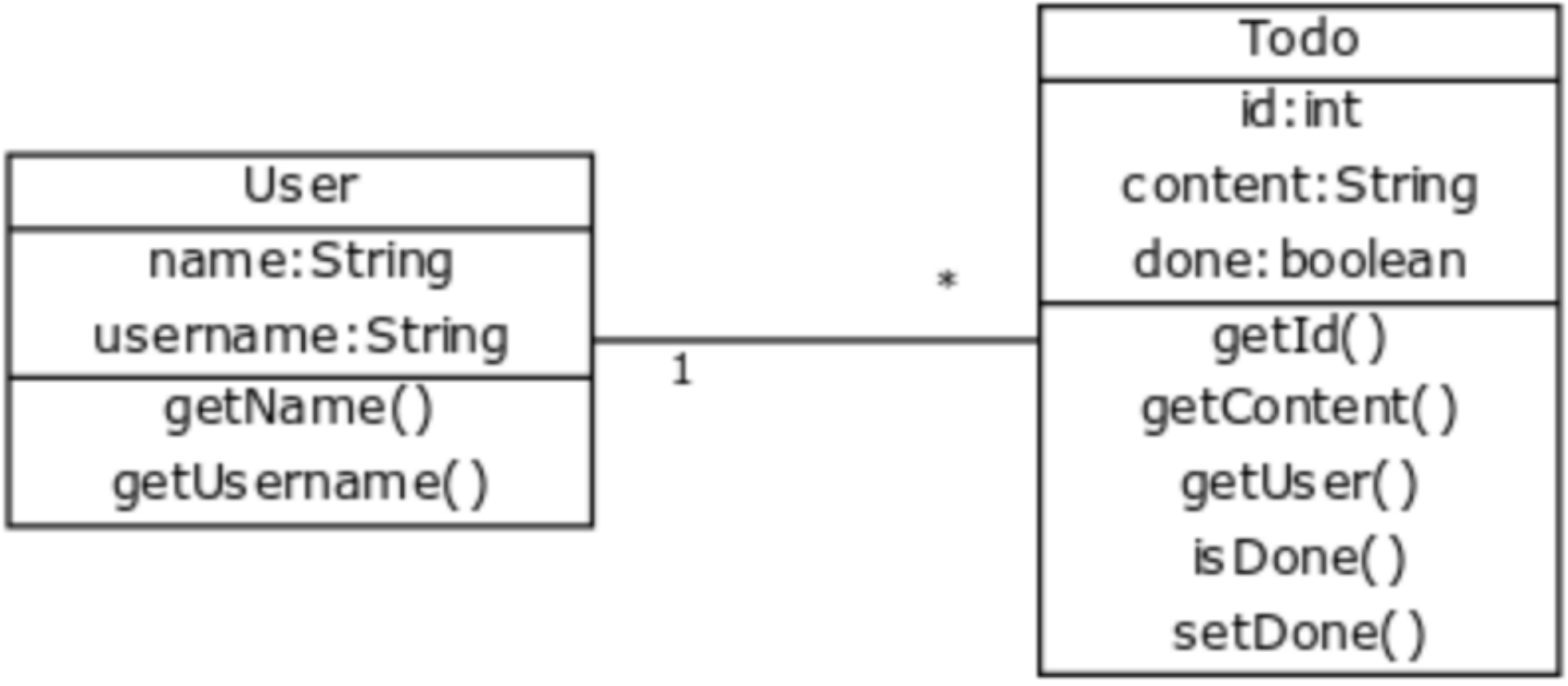
UML

Luokkakaaviot

- Luokkakaavioiden käyttötarkoitus on ohjelman luokkien ja niiden välisten suhteiden kuvailu
- Todo-sovelluksen oleellista tietosisältöä kuvaavat luokat

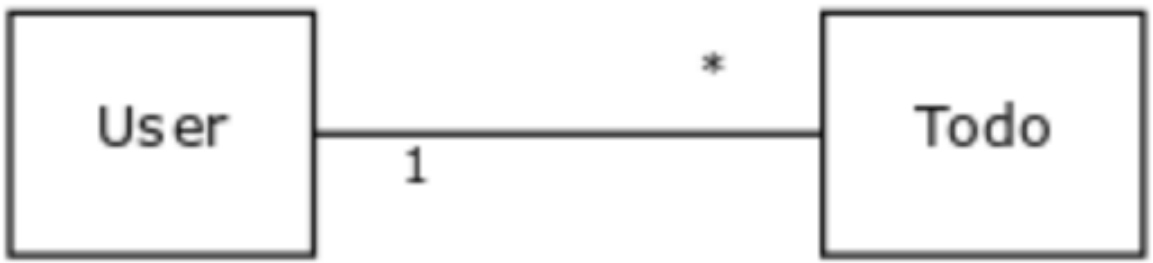
```
public class User {
    private String name;
    private String username;
    // ...
}

public class Todo {
    private int id;
    private String content;
    private boolean done;
    private User user;
    // ...
}
```

Todo-sovelluksen tietosisällön luokkakaavio



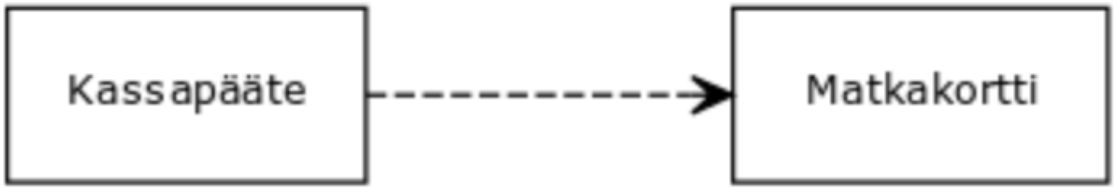
Todo-sovelluksen tietosisällön luokkakaavio

Yleensä ei ole mielekästä kuvata luokkia tällä tarkkuudella, eli luokkakaavioihin riittää merkitä luokan nimi



Riippuvuus

- UML-kaavioissa olevat “viivat” kuvaavat luokkien olioiden välistä *pysyvää yhteyttä*
- Joissain tilanteissa on mielekästä merkata kaavioihin myös ei-pysyvää suhdetta kuvaava katkoviiva, eli *riippiipuvuus*.
- Kassapääte *käyttää* hetkellisesti *maksukorttia* lounaiden maksuun
- Kasspäätteen ja maksukortin välillä ei kuitenkaan ole pidempiaikaista suhdetta

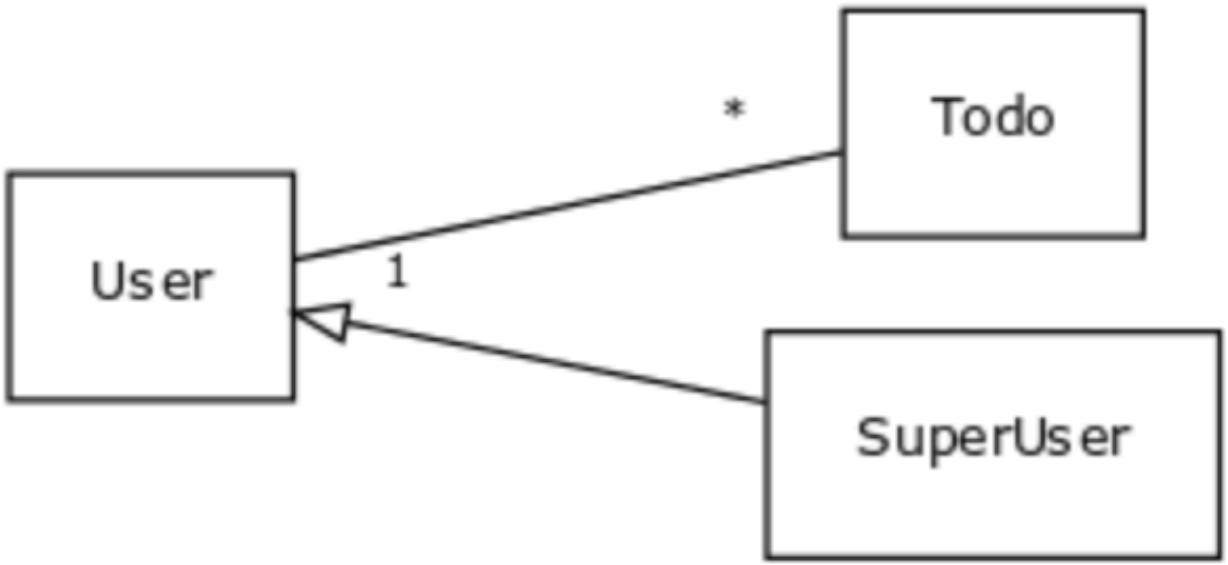


Maksukortti ja kassapääte

```
public class Maksukortti {
    private double saldo;
    // ...
}

public class Kassapaate {
    private int edulliset;
    private int maukkaat;

    public boolean syoEdullisesti(Maksukortti kortti) {
        // ...
        kortti.otaRahaa(EDULLISEN_HINTA);
    }
}
```

Rajapinnan toteutus ja perintä luokkakaaviossa

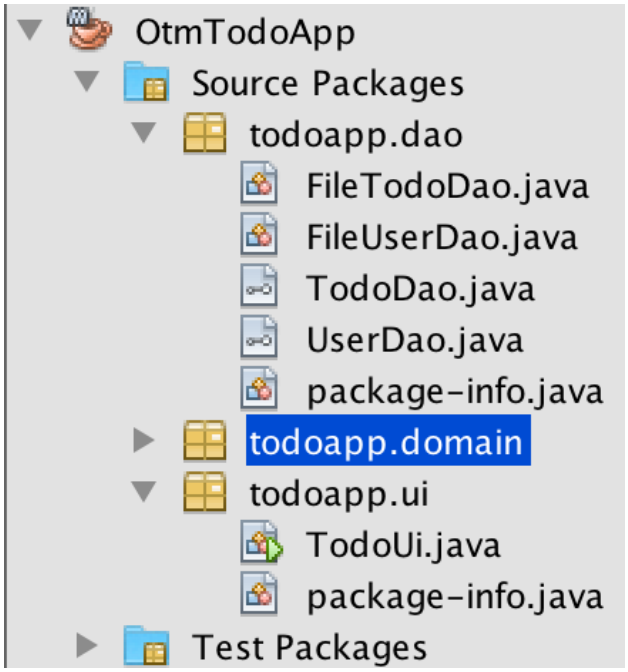
- Jos Todo-sovelluksessa olisi normaalin käyttäjän eli luokan *User* perivä ylläpitäjää kuvaava luokka *SuperUser*, merkattaisiin se luokkakaavioon seuraavasti



Rajapinnan toteutus merkitään samalla tavalla valkoisella nuolenpäällä

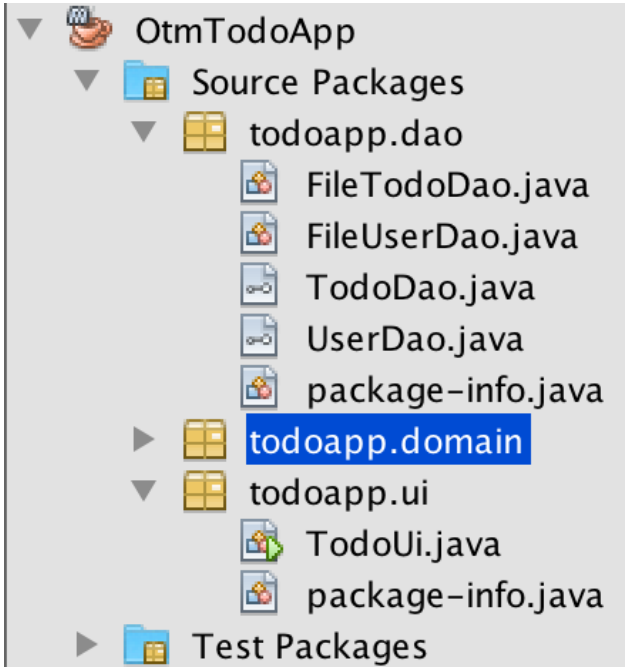
Pakkauskaavio

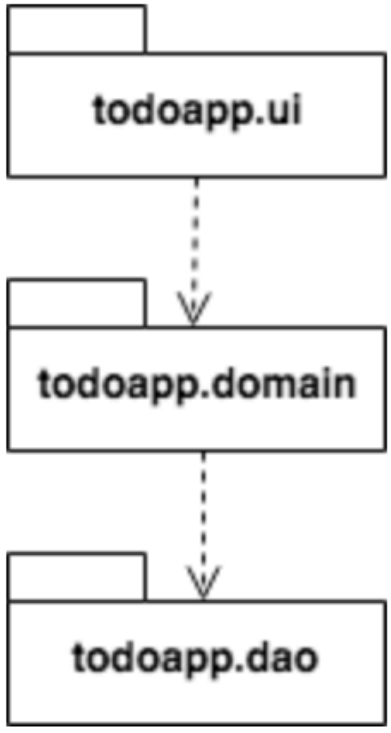
Todo-sovelluksen koodi on sijoitettu *pakkauksiin* seuraavasti:



Pakkauskaavio

Pakkausrakenne voidaan kuvata UML:ssä pakkauskaaviolla

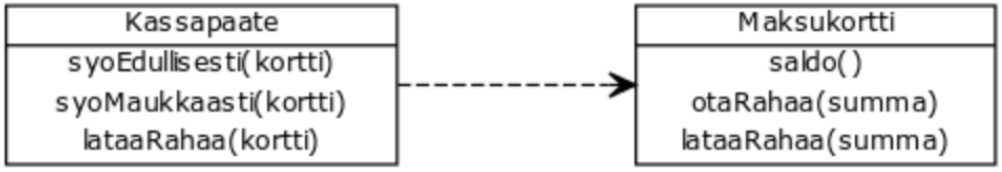




Pakkausten välille on merkitty riippuvuus jos pakkauksen luokat käyttävät toisen pakkauksen luokkia

Toiminnan kuvaaminen

- Luokka- ja pakkauskaaviot kuvaavat ohjelman rakennetta
- Ohjelman toimita ei kuitenkaan tule niistä ilmi millään tavalla.
- Esim. ohpen Unicafe-tehtävä



- Vaikka kaavioon on merkitty metodien nimet, ei ohjelman toimintalogiikka
- esim. mitä tapahtuu kun kortilla ostataan edullinen lounas, selviä kaaviosta millään tavalla

Sekvenssikaavio

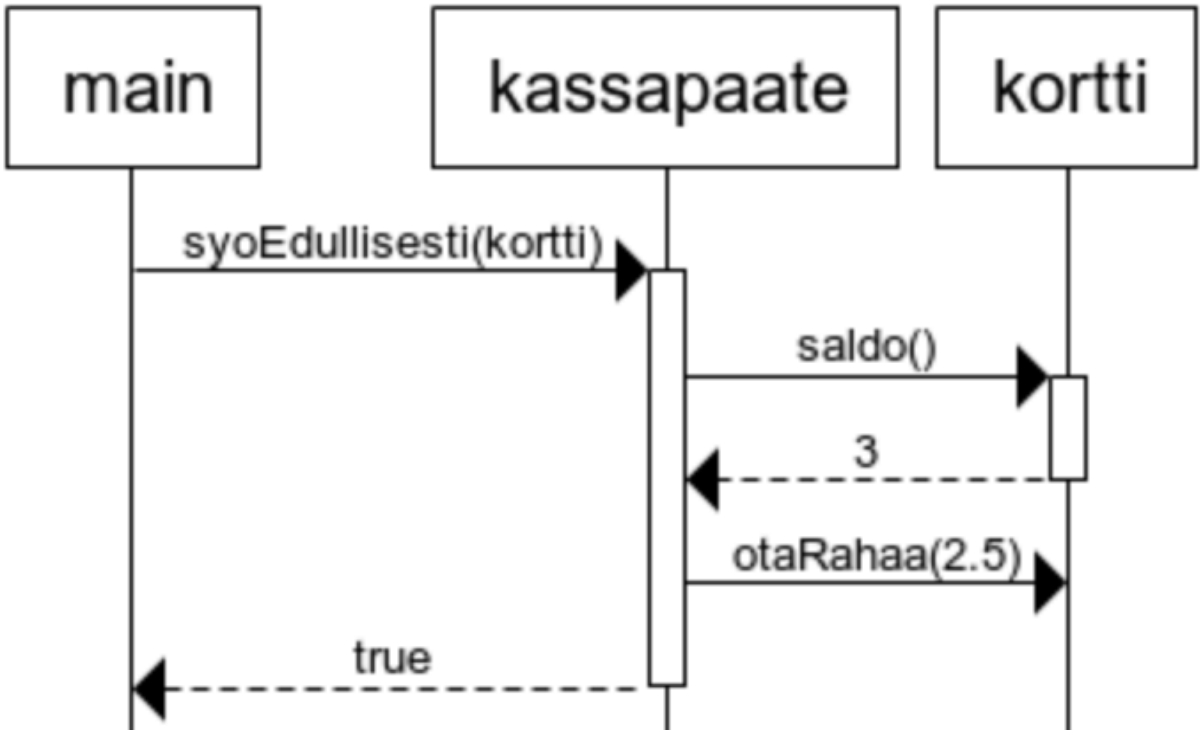
- Tietokantojen perusteiden viikolla 4 on lyhyt maininta sekvenssikaavioista.
- Sekvenssikaaviot on alunperin kehitetty kuvaamaan verkossa olevien ohjelmien keskinäisen kommunikoinnin etenemistä
- Sekvenssikaaviot sopivat kohtuullisen hyvin kuvaamaan myös sitä, miten ohjelman oliot kutsuvat toistensa metodeja suorituksen aikana

Sekvenssikaavio

Koodia katsomalla näemme, että lounaan maksaminen tapahtuu siten, että ensin kassapääte kysyy kortin saldoa ja jos se on riittävä, vähentää kassapääte lounaan hinnan kortilta ja palauttaa *true*:

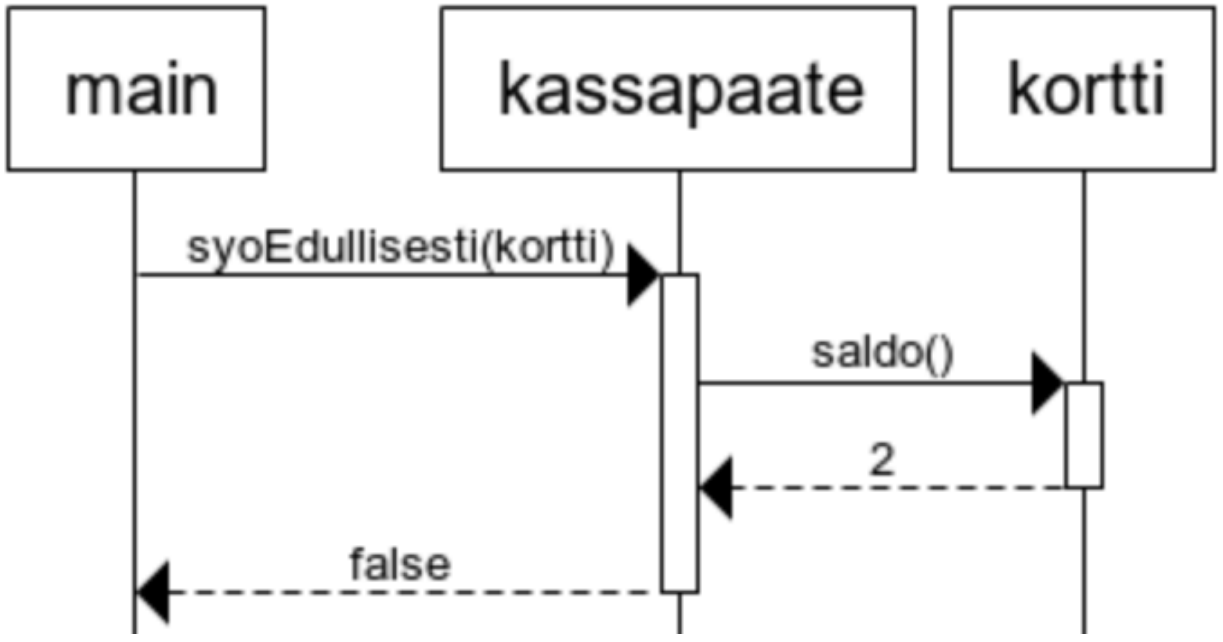
```
public boolean syoEdullisesti(Maksukortti kortti) {
    if (kortti.saldo() < EDULLISEN_HINTA) {
        return false;
    }

    kortti.otaRahaa(EDULLISEN_HINTA);
    this.edulliset++;
    return true;
}
```

Onnistunut ostos sekvenssikaaviona



- Sekvenssikaaviossa oliot laatikoita, joista lähtee alaspäin olion "elämänlanka"
- Aika etenee ylhäältä alas
- Metodikutsut nuolia joka yhdistävää kutsujan ja kutsutun olion elämänlangat
- Paluuarvo merkitään katkoviivalla.

eääonnistunut ostos sekvenssikaaviona



- Sekvenssikaaviot kuvaavat siis yksittäistä tapahtumasarjaa
- Toiminnallisuuden kuvaamiseen tarvitaankin yleensä useampi sekvenssikaavio

HARJOITUSTYÖ

Yleistä

- Kurssin pääpainon muodostaa viikolla 2 aloitettava harjoitustyö
- Harjoitustyössä toteutetaan itsenäisesti ohjelmisto omavalintaisesta aiheesta
- Tavoitteena on soveltaa ja syventää ohjelmoinnin perus- ja jatkokursseilla opittuja taitoja ja harjoitella tiedon omatoimista etsimistä
- Harjoitustyötä tehdään itsenäisesti, mutta tarjolla on runsaasti pajaohjausta

Työn eteneminen

- Edettävä viikottaisten tavoitteiden mukaan
- Työ on saatava valmiiksi kurssin aikana ja sitä on toteutettava tasaisesti, muuten kurssi katsotaan keskeytetyksi
- Samaa ohjelmaa ei voi jatkaa seuraavalla kurssilla (eli keväällä 2019), vaan työ on aloitettava uudella aiheella alusta
- Koko kurssin arvostelu perustuu pääasiassa harjoitustyöstä saataviin pisteisiin
- Osa pisteistä kertyy viikoittaisten välitavoitteiden kautta, osa taas perustuu työn lopulliseen palautukseen

Kieli

- Harjoitustyön ohjelmointikieli on pääsääntöisesti Java
- Ohjelmakoodin muuttujat, luokat ja metodit kirjoitetaan englanniksi
- Dokumentaatio voidaan kirjoittaa joko suomeksi tai englanniksi
- Ohjaajat saattavat suostua muihinkin kieliin, asia on syytä sopia kurssin ensimmäisen viikon aikana
  - Javascript, Ruby ja C++ ainakin käyvät
- Web-sovelluksia kurssilla ei sallita
  - Sovelluksissa sallitaan toki webissä olevat komponentit, mutta sovelluksen käyttöliittymän tulee olla ns desktop-sovellus
  - Javascriptillä hyvä valinta desktop-sovelluksiin on Electron
- Muuta kuin Javaa käytettäessä ohjaus saattaa olla "huonompaa"
- Muita kieliä valitessa on myös syytä pitää mielessä, että kieleltä on syytä löytyä Javan Mavenia, Junitia, jacocoa, checkstyleä ja JavaDocia vastaavat asiat

Ohjelman toteutus

- Toteutus etenee “iteratiivisesti ja inkrementaalisesti”
  - heti toteutetaan pieni osa toiminnallisuudesta
  - ohjelman ydin pidetään koko ajan toimivana, uutta toiminnallisuutta lisäten, kunnes tavoiteltu ohjelman laajuus on saavutettu
- Ohjelman rakenteeseen kannattaa kysyä vinkkejä pajasta, sekä ottaa mallia ohjelmoinnin jatkokurssista sekä kurssisivuilta käytyvistä vihjeistä
- Iteratiiviseen tapaan tehdä ohjelma liittyy kiinteästi automatisoitu testaus
- Aina uutta toiminnallisuutta lisättäessä ja vanhaa muokatessa täytyy varmistua, että kaikki vanhat ominaisuudet toimivat edelleen
- Jotta ohjelmaa pystyisi testaamaan, on tärkeää että sovelluslogiikkaa ei kirjoiteta käyttöliittymän sekaan
- Graafiseen käyttöliittymään suositellaan JavaFX:ää
- Tiedon talletus joko tiedostoon tai tietokantaan

Ohjelman toteutus

- tavoitteena on tuottaa ohjelma, joka voitaisiin antaa toiselle opiskelijalle ylläpidettäväksi ja täydennettäväksi
- Lopullisessa palautuksessa on oltava lähdekoodin lisäksi dokumentaatio ja automaattiset testit sekä jar-tiedosto, joka mahdollistaa ohjelman suorittamisen NetBeansin ulkopuolella.
- Toivottava dokumentaation taso käy ilmi kurssin referenssisovelluksesta https://github.com/mluukkai/OtmTodoApp

Hyvän aiheen ominaisuudet

- **Itseäsi kiinnostava aihe**
- Riittävän mutta ei liian laaja
  - Vältä eeppisiä aiheita, aloita riittävän pienestä
  - Valitse aihe, jonka perustoiminnallisuuden saa toteutettua nopeasti, mutta jota saa myös laajennettua helposti
  - Hyvässä aiheessa on muutamia logiikkaluokkia, tiedoston tai tietokannankäsittelyä ja logiikasta eriytetty käyttöliittymä
- Kurssilla pääpaino on Ohpessa ja Ohjassa
  - Toimivuus ja varautuminen virhetilanteisiin
  - Luokkien vastuut
  - Ohjelman selkeä rakenne
  - Laajennettavuus ja ylläpidettävyys
- **Tällä kurssilla ei ole tärkeää:**
  - Tekoäly
  - Grafiikka
  - Tietoturva
  - Tehokkuus

Huonon aiheen ominaisuuksia

- Kannattaa yrittää välttää aiheita, joissa pääpaino on tiedon säilömisessä tai monimutkaisessa käyttöliittymässä
- Paljon tietoa säilövät, esim. yli 3 tietokantataulua tarvitsevat sovellukset sopivat yleensä paremmin Tietokantasovellus-kurssille
- Käyttöliittymäkeskeisissä aiheissa (esim. tekstieditori) voi olla vaikea keksiä sovelluslogiikkaa, joka on enemmän tämän kurssin painopiste

Esimerkkejä aiheista

- Hyötyohjelmat
  - Aritmetiikan harjoittelua
  - Tehtävägeneraattori, joka antaa käyttäjälle tehtävän sekä mallivastauksen (esim. matematiikkaa, fysiikkaa, kemiaa, ...)
  - telegram- tai Slack-botti
  - Code Snippet Manageri
  - Laskin, funktiolaskin, graafinen laskin
  - Budjetointi-sovellus
  - Opintojen seurantasovellus
  - HTML WYSIWYG-editor (What you see is what you get)

Esimerkkejä aiheista

- Reaaliaikaiset pelit
  - Tetris
  - Pong
  - Pacman
  - Tower Defence
  - Asteroids
  - Space Invaders
  - Yksinkertainen tasohyppypeli, esimerkiksi The Impossible Game

Esimerkkejä aiheista

- Vuoropohjaiset pelit
  - Tammi
  - Yatzy
  - Miinaharava
  - Laivanupotus
  - Yksinkertainen roolipeli tai luolastoseikkailu
  - Sudoku
  - Muistipeli
  - Ristinolla (mielivaltaisen kokoisella ruudukolla?)

Esimerkkejä aiheista

- Korttipelit
  - En Garde
  - Pasianssi
  - UNO
  - Texas Hold'em
- Omaan tieteenalaan, sivuaineeseen tai harrastukseen liittyvät hyötyohjelmat
  - Yksinkertainen fysiikkasimulaattori
  - DNA-ketjujen tutkija
  - Keräilykorttien hallintajärjestelmä
  - Fraktaaligeneraattori

Harjoitustyön vaikutus kurssipisteisiin

Loppupalautuksen pisteet (yht 23) jakautuvat seuraavasti

- käyttöliittymä 4p
  - 0p yksinkertainen tekstikäyttöliittymä
  - 1-2p monimutkainen tekstikäyttöliittymä
  - 2-3p yksinkertainen graafinen käyttöliittymä
  - 4p laaja graafinen käyttöliittymä
- tiedon pysyväistalletus 4p
  - ei pysyväistallennusta
  - 1-2p tiedosto
  - 3-4p tietokanta
  - 3-4p internet
- sovelluslogiikan kompleksisuus 3p
- ohjelman lajuus 5p
- suorituskelpoinen jar-tiedosto 1p
- koodin laatu 6p

Koodin laatuvaatimukset

- Kurssin tavoitteena on, että tuotoksesi voisi ottaa kuka tahansa kaverisi tai muu opiskelija ylläpidettäväksi ja laajennettavaksi
- Lopullisessa palautuksessa yavoitteena on *Clean code* - selkeää, ylläpidettävää ja toimivaksi testattua koodia
- **nimentä**
  - Käytä mahdollisimman kuvaavia nimiä kaikkialla
  - Luokkien nimet aina isolla alkukirjaimella
  - Metodit, attribuutit, parametrit ja muuttujat aina *camelCase*
  - Muuttujat, joilla on iso käyttöalue, tulee olla erittäin selkeästi (vaikka pitkästi) nimettyjä.
  - Lyhyen metodin sisäisille muuttujille riittää yleensä lyhyt nimi.
  - Jos metodia käytetään vähän, tulee nimen olla mahdollisimman kuvaava.
  - Jos metodia käytetään useassa kohdassa koodia, voi sen nimi olla lyhyt ja ytimekäs

Koodin laatuvaatimukset

- **Ei pitkiä metodeja**
  - Sovelluslogiikan metodin pituuden tulee ilman erittäin hyvää syytä olla korkeintaan 10 riviä, mieluiten 5.
  - Pitkät metodit tulee jakaa useampiin metodeihin.
  - Yksi metodi - yksi pieni tehtävä. (Single Responsibility)
    - Helpottaa myös testaamista
  - Jos metodin voi jakaa useampaan metodiin, niin silloinhan se jo tekee useamman metodin hommat.
- **Ei copy-pastea**
  - Toistuvan koodin saa lähes aina hävitettyä
  - Tapauksesta riippuen luo metodi tai yliluokka, joka sisältää toistuvan koodin

Koodin laatuvaatimukset

- **Luokkien Single Responsibility**
  - Luokkien tulisi hoitaa vain yhtä asiaa
  - Eriytä käyttöliittymä ja sovelluslogiikka
    - Kaikki tulostaminen tulisi tapahtua käyttöliittymässä
    - Sovelluslogiikkaan liittyviä laskuja ja tapahtumia ei tehdä käyttöliittymässä
  - Toisaalta tiettyä asiaa ei pidä hoitaa useissa eri luokissa
  - Esimerkiksi tiedoston lukemista tai -kirjoittamista EI tulisi löytyä useasta luokasta
    - Tee oma luokka tiedostojen käsittelylle

Koodin laatuvaatimukset

- **Pakkaukset**
  - << Default package >> EI saa olla käytössä
  - Luokat tulee jakaa loogisesti pakkauksiin
    - Pakkausten nimet aina pienellä (*lowercase*)
  - Kaikkien pakkausten tulee olla yhden juuripakkauksen alla, esim. fi.omanimi
    - Sovelluslogiikkapakkaus olisi näin tehtynä siis fi.omanimi.solvelluslogiikka, käyttöliittymä fi.omanimi.gui
  - Yhdessä pakkauksessa yksi kokonaisuus
    - Esim. yhdessä pakkauksessa käyttäjätileihin liittyvät luokat
    - Toisessa muu logiikka
    - Kolmannessa käyttöliittymän luokat
  - Myös testipakkausten nimentä tulee olla oikea

Yleiset laatuvaatimukset

- Lisäksi lopulliseen arvosteluun palautetun ohjelman tulee toimia oikein.
  - Ohjelma ei saa missään tilanteessa kaatua
  - Ohjelma ei saa printata Exceptioneita (Stack tracea) komentoriville, vaikka virhe ei kaataisi ohjelmaa
- Varaudu siihen, että käyttäjä yrittää antaa vääriä syötearvoja
  - Esim. ohjelmasi haluaa numeron, tyhmä käyttäjä syöttää tekstiä
- Pelien sääntöjen tulisi toimia oikein
- Esim. muistipelissä ei saa kääntää jo käännettyä palaa
- Ristinollassa ei saa asettaa merkkiä ruutuun, jossa on jo merkki
- Jos ohjelmassasi tapahtuu vakava virhe, ohjelmasi voi esimerkiksi
- näyttää käyttäjäystävällisen virheilmoituksen
- ja sulkea ohjelman
- Ohjelmaan jäävät tunnetut ongelmat dokumentoidaan testausdokumenttiin