

# Ohjelmistotuotanto

Luento 8

18.4.2012

Arkkitehtuuri ketterissä menetelmissä

# Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen (agile manifestin periaattita):
  - Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
  - Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Ketterät menetelmät suosivat yksinkertaisuutta suunnitteluratkaisuissa
  - Simplicity--the art of maximizing the amount of work not done--is essential.
  - YAGNI eli you are not going to need it -periaate
- Arkkitehtuuriin suunnittelu ja dokumentointi on perinteisesti ollut melko pitkäkestoinen vaihe joka on edeltänyt ohjelmoinnin aloittamista
  - BUFD eli Big Up Front Design
- Ketterät menetelmät ja ”arkkitehtuurivetoinen” ohjelmistotuotanto ovat siis jossain määrin keskenään ristiriidassa

# Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien yhteydessä puhutaan inkrementaalisisesta tai evolutiivisesta suunnittelusta ja arkkitehtuurista
- Arkkitehtuuri mietitään riittävällä tasolla projektin alussa
- Jotkut projektit alkavat ns. nollasprintillä ja alustava arkkitehtuuri määritellään tällöin
  - Scrumin varhaisissa artikkeleissa puhuttiin ”pre game”-vaiheesta jolloin mm. alustava arkkitehtuuri luodaan
  - Sittemmin koko käsite on hävinnyt Scrumista ja Ken Schwaber jopa eksplisiittisesti kieltää ja tyrmää koko ”nollasprintin” olemassaolon:  
<http://www.scrum.org/assessmentdiscussion/post/1317787>
- Ohjelmiston ”lopullinen” arkkitehtuuri muodostuu iteraatio iteraatiolta samalla kun ohjelmaan toteutetaan uutta toiminnallisuutta
  - Esim. kerrosarkkitehtuurin mukaista sovellusta ei rakenneta ”kerros kerrallaan”
  - jokaisessa iteraatiossa tehdään pieni pala jokaista kerrosta, sen verran kuin iteraation toiminnallisuuksien toteuttaminen edellyttää
  - <http://msdn.microsoft.com/en-us/architecture/ff476940>

# Arkkitehtuuri ketterissä menetelmissä

- Perinteisesti arkkitehtuurista on vastannut ohjelmistoarkkitehti ja ohjelmoijat ovat olleet velvoitettuja noudattamaan arkkitehtuuria
- Ketterissä menetelmissä ei suosita erillistä arkkitehdin roolia, esim. Scrum käyttää kaikista ryhmän jäsenistä nimikettä developer
- Ketterien menetelmien ideaali on, että kehitystiimi luo arkkitehtuurin yhdessä, tämä on myös yksi agile manifestin periaatteista:
  - The best architectures, requirements, and designs emerge from self-organizing teams.
- Arkkitehtuuri on siis koodin tapaan tiimin yhteisomistama, tästä on muutamia etuja
  - Kehittäjät sitoutuvat paremmin arkkitehtuurin noudattamiseen kuin ”norsunluutornissa” olevan tiimin ulkopuolisen arkkitehdin määrittelemään arkkitehtuuriin
  - Arkkitehtuurin dokumentointi voi olla kevyt ja informaali (esim. valkotaululle piirretty) sillä tiimi tuntee joka tapauksessa arkkitehtuurin hengen ja pystyy sitä noudattamaan

# Inkrementaalinen arkkitehtuuri

- Ketterissä menetelmissä oletuksena on, että parasta mahdollista arkkitehtuuria ei pystytä suunnittelemaan projektin alussa, kun vaatimuksia, toimintaympäristöä ja toteutusteknologioita ei vielä tunneta
  - Jo tehtyjä arkkitehtoonisia ratkaisuja muutetaan tarvittaessa
- Eli kuten vaatimusmäärittelyn suhteen, myös arkkitehtuurin suunnittelussa ketterät menetelmät pyrkii välttämään liian aikaisin tehtävää ja myöhemmin todennäköisesti turhaksi osoittautuvaa työtä
- Inkrementaalinen lähestymistapa arkkitehtuurin muodostamiseen edellyttää koodilta hyvää laatua ja toteuttajilta kurinalaisuutta
- Martin Fowler <http://martinfowler.com/articles/designDead.html>:
  - Essentially evolutionary design means that the design of the system grows as the system is implemented. Design is part of the programming processes and as the program evolves the design changes.
  - In its common usage, evolutionary design is a disaster. The design ends up being the aggregation of a bunch of ad-hoc tactical decisions, each of which makes the code harder to alter
- Seuraavaksi siirrymme käsittelemään oliosuunnittelua

# Oliosunnittelu

# Oliosunnittelu

- Käytettäessä ohjelmiston toteutukseen olio-ohjelmointikieltä, on suunnitteluvaiheen tarkoituksena löytää sellaiset oliot, jotka pystyvät yhteistoiminnallaan toteuttamaan järjestelmän vaatimuksen
- Oliosunnittelua ohjaa ohjelmistolle suunniteltu arkkitehtuuri
- Ohjelman ylläpidettävyyden kannalta on suunnittelussa hyvä noudattaa ”ikiaikaisia” hyvän suunnittelun käytänteitä
  - Ketterissä menetelmissä tämä on erityisen tärkeää sillä jos ohjelman rakenne pääsee rapistumaan, on ohjelmaa vaikea laajentaa jokaisen sprintin aikana
- Ohjelmiston suunnitteluun on olemassa useita erilaisia menetelmiä, mikään niistä ei kuitenkaan ole vakiintunut
  - Kurssilla Ohjelmistojen mallintaminen tutustuttiin ohimennen ns. vastuupohjaiseen oliosunnitteluun (Responsibility driven object design)
- Ohjelmistosuunnittelu onkin ”enemmän taidetta kuin tiedettä”, kokemus ja hyvien käytänteiden opiskelu toki auttaa
- Erityisesti ketterissä menetelmissä tarkka oliosunnittelu tapahtuu yleensä ohjelmoinnin yhteydessä



# Oliosuunnittelu

- Emme keskity tällä kurssilla mihinkään yksittäiseen oliosuunnittelumenetelmään
- Sensijaan tutustumme muutamiin tärkeisiin ”menetelmäriippumattomiin” teemoihin:
- Laajennettavuuden ja ylläpidettävyyden suhteen laadukkaan koodin/oliosuunnittelun tunnusmerkkeihin ja laatuattribuutteihin
  - kapselointi, koheesio, riippuvuuksien vähäisyys, toisteettomuus, selkeys, testattavuus
- ja niitä tukeviin ”ikiaikaisiin” hyvän suunnittelun periaatteisiin
- *Koodinhajuihin* eli merkkeihin siitä että suunnittelussa ei kaikki ole kunnossa
- *Refaktorointiin* eli koodin rajapinnan ennalleen jättävään rakenteen parantamiseen
- Erilaisissa tilanteissa toimiviksi todettuihin geneerisiä suunnitteluratkaisuja dokumentoiviin *suunnittelumalleihin*
  - Olemme jo nähdeen muutamia suunnittelumalleja, ainakin seuraavat: dependency injection, singleton, data access object

# Helposti ylläpidettävän koodin tunnusmerkit

- Ylläpidettävyyden ja laajennettavuuden kannalta tärkeitä seikkoja
  - Koodin tulee olla luettavuudeltaan selkeää, eli koodin tulee kertoa esim. nimennällään mahdollisimman selkeästi mitä koodi tekee, eli tuoda esiin koodin alla oleva ”design”
  - Yhtä paikkaa pitää pystyä muuttamaan siten ettei muutoksesta aiheudu sivuvaikutuksia sellaisiin kohtiin koodia jota muuttaja ei pysty ennakoimaan
  - Jos ohjelmaan tulee tehdä laajennus tai bugikorjaus, tulee olla selvää mihin kohtaan koodia muutos tulee tehdä
  - Jos ohjelmasta muutetaan ”yhtä asiaa”, tulee kaikkien muutosten tapahtua vain yhteen kohtaan (metodiin tai luokkaan)
  - Muutosten ja laajennusten jälkeen tulee olla helposti tarkastettavissa ettei muutos aiheuta sivuvaikutuksia muualle järjestelmään
- Näin määritelty koodin *sisäinen laatu* on erityisen tärkeää ketterissä menetelmissä, joissa koodia laajennetaan iteraatio iteraatiolta
- Jos koodin sisäiseen laatuun ei kiinnitetä huomiota, on väistämätöntä että pidemmässä projektissa kehitystiimin velositeetti alkaa tippua ja eteneminen alkaa vaikeutua iteraatio iteraatiolta
  - Koodin sisäinen laatu on siis usein myös asiakkaan etu

# Koodin laatuattribuutteja

- Edellä lueteltuihin hyvän koodin tunnusmerkkeihin päästään kiinnittämällä huomio seuraaviin laatuattribuutteihin
  - Kapselointi
  - Koheesio
  - Riippuvuuksien vähäisyys
  - Toisteettomuus
  - Testattavuus
  - Selkeys
- Tutkitaan nyt näitä laatuattribuutteja sekä periaatteita miten on mahdollista kirjoittaa koodia joka on näiden mittarien mukaan laadukasta

# Kapselointi

- Ohjelmointikursseilla on määritelty kapselointi seuraavasti
  - ”Tapaa ohjelmoida olion toteutuksen yksityiskohdat luokkamäärittelyn sisään – piiloon olion käyttäjältä – kutsutaan kapseloinniksi. Olion käyttäjän ei tarvitse tietää mitään olioiden sisäisestä toiminnasta. Eikä hän itse asiassa edes saa siitä mitään tietää vaikka kuinka haluaisi!”
- Aloitteleva ohjelmoija assosioi kapseloinnin yleensä seuraavaan periaatteeseen:
  - Oliomuuttujat tulee määritellä privaateiksi ja niille tulee tehdä tarvittaessa setterit ja getterit
- Tämä on kuitenkin hieman kapea näkökulma kapselointiin
- Jatkossa näemme esimerkkejä monista kapseloinnin muista muodoista. Kapseloinnin kohde voi olla mm.
  - Käytettävän olion tyyppi, algoritmi, olioiden luominen, käytettävän komponentin rakenne
- Monissa suunnittelumalleissa on kyse juuri eritasoisten asioiden kapseloinnista

# Koheesio

- Koheesiolla tarkoitetaan sitä, kuinka pitkälle metodissa, luokassa tai komponentissa oleva ohjelmakoodi on keskittynyt tietyn toiminnallisuuden toteuttamiseen
- Hyvänä asiana pidetään mahdollisimman korkeata koheesion astetta
- Koheesioon tulee siis pyrkiä kaikilla ohjelman tasoilla, metodeissa, luokissa, komponenteissa ja jopa muuttujissa
- **Metoditason koheesiossa** pyrkimyksenä että metodi tekee itse vain yhden asian
- Metoditason koheesiota ilmentävä Kent Beckin ”Composed method”-suunnittelumalli ohjeistaa seuraavasti
- *The composed method pattern* defines three key statements:
  - Divide your programs into methods that perform one identifiable task.
  - Keep all the operations in a method at the same level of abstraction.
  - This will naturally result in programs with many small methods, each a few lines long.
  - <http://www.ibm.com/developerworks/java/library/j-eaed4/index.html>

# Koheesio ja Single responsibility -periaate

- Metoditason koheesioon päästään jakamalla ”koheesioton” metodi useisiin metodeihin joita alkuperäinen metodi kutsuu
  - Alkuperäinen metodi alkaa toimia korkeammalla abstraktiotasolla, koodinoiden kutsumiensa yhteen asiaan keskittyvien metodien suoritusta
  - esimerkki <https://wiki.helsinki.fi/display/ohtu2012/luento8> kohdassa ”koheesio metoditasolla”
- **Luokkatason koheesiossa** pyrkimyksenä on, että luokan **vastuulla** on vain yksi asia
- Ohjelmistojen mallintamisesta tuttu **Single Responsibility (SRP)** -periaate tarkoittaa oikeastaan täysin samaa
  - [www.objectmentor.com/resources/articles/srp.pdf](http://www.objectmentor.com/resources/articles/srp.pdf)
  - Uncle Bob tarkoittaa yhden vastuun määritelmää siten että luokalla on yksi vastuu jos sillä on vain yksi syy muuttua
- esimerkejä <https://wiki.helsinki.fi/display/ohtu2012/luento8> kohdassa ”single responsibility -periaate eli koheesio luokkatasolla”
- Vastakohta SRP:tä noudattavalle luokalle on jumalaluokka/olio
  - <http://blog.decayingcode.com/post/anti-pattern-god-object.aspx>

# Riippuvuuksien vähäisyys

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

# Riippuvuuksien vähäisyys

- Konkreettisen riippuvuuden eliminointi voidaan tehdä rajapintojen avulla
  - Olemme tehneet näin kurssilla usein, mm. Verkkokaupan riippuvuus Varastoon, Pankkiin ja Viitegeneraattoriin korvattiin rajapinnoilla
  - Dependency Injection -suunnittelumalli toimi usein apuvälineenä konkreettisen riippumisen eliminoinnissa
- Osa luokkien välisistä riippuvuuksista on tarpeettomia ja ne kannattaa eliminoida muuttamalla luokan vastuita
- Perintä muodostaa riippuvuuden perivän ja perittävän luokan välille, tämä voi jossain tapauksissa olla ongelmallista
- Yksi oliosuunnittelun kulmakivi on periaate **Favour composition over inheritance** eli **suosi yhteistominnessa toimivia oliota perinnän sijaan**
  - <http://www.artima.com/lejava/articles/designprinciples4.html>
  - Perinnällä on paikkansa, mutta sitä tulee käyttää harkiten
- ks. esimerkki <https://wiki.helsinki.fi/display/ohtu2012/luento8> kohdasta ”Favour composition over inheritance eli milloin ei kannata periä”



# Suunnittelumalli: Static factory

- Tili-esimerkissä käytetty **Static factory method** on yksinkertaisin erilaisista tehdas-suunnittelumallin varianteista
- luokkaan tehdään staattinen tehdasmetodi tai metodeita jotka käyttävät konstruktoria ja luovat luokan ilmentymät
  - Konstruktorin suora käyttö usein estetään määrittelemällä konstruktori privateksi
- Tehdasmethodin avulla voidaan piilottaa olion luomiseen liittyviä yksityiskohtia
  - Esimerkissä Korke-rajapinnan toteuttavien olioiden luominen ja jopa olemassaolo oli tehdasmethodin avulla piilotettu tilin käyttäjältä
- Tehdasmethodin avulla voidaan myös piilottaa käyttäjältä luodun olion todellinen luokka
  - Esimerkissä näin tehtiin määräaikaistilin suhteen
- Staattinen tehdasmetodi ei ole testauksen kannalta erityisen hyvä ratkaisu, esimerkissämme olisi vaikea luoda tili jolle annetaan Korke-rajapinnan toteuttama mock-olio
  - nyt se tosin onnistuu kun konstruktoria ei ole täysin piilotettu

# Suunnittelumalli: Strategy

- Esimerkissä tilien koron laskenta hoidettiin Strategy-suunnittelumallin avulla
  - [http://sourcemaking.com/design\\_patterns/strategy](http://sourcemaking.com/design_patterns/strategy)
  - <http://www.oodesign.com/strategy-pattern.html>
  - <http://www.netobjectives.com/PatternRepository/index.php?title=TheStrategyPa>
- Usein strategyn avulla hoidetaan tilanne missä käyttäytyminen on muuten sama mutta tietyissä kohdissa on käytössä eri "algoritmi"
  - Esimerkissämme tämä "algoritmi" oli korkoprosentin määrittäminen
- Sama tilanne voidaan hoitaa usein myös perinnän avulla käyttämättä erillisiä olioita, strategy kuitenkin mahdollistaa huomattavasti dynaamisemmän ratkaisun sillä strategia-olioa voi vaihtaa ajoaikana
- Strategyn käyttö ilmentää hienosti "favour composition over inheritance"-periaatetta
- Usein strategiaa käytetään korvaamaan ohjelmassa oleva ikävä if- tai switch-hässäkkä
- ks. esimerkki <https://wiki.helsinki.fi/display/ohtu2012/luento8> kohdasta "laskin ilman iffejä"

# Suunnittelumalli: command eli komento

- Laskin-esimerkissä päädyttiin eristämään jokaiseen erilliseen laskuoperaatioon liittyvä toiminta omaksi oliokseen
- Kaikki operaation toteuttavat yksinkertaisen rajapinnan jolla on ainoastaan metodi **public void suorita()**
- Siirryttiin pelkän algoritmin kapseloivista strategiaolioista koko komennon suorituksen kapseloiviin olioihin eli **command-suunnittelumalliin**
  - <http://www.oodesign.com/command-pattern.html>
  - [http://sourcemaking.com/design\\_patterns/command](http://sourcemaking.com/design_patterns/command)
- Strategian tapaan komento-oliolla saadaan eliminoitua koodista if-ketjuja
  - Esimerkissä komennot luotiin tehdasmetodin tarjoavan olion avulla, if:in piilotettiin tehtaan sisälle
- Komento-olioiden suorita-metodi suoritettiin esimerkissä välittömästi, näin ei välttämättä ole, komentoja voitaisiin laittaa esim. jonoon
- Joskus komento-olioilla on myös undo-operaatio
  - Esim. editorien undo- ja redo-toiminnallisuus toteutetaan säilyttämällä komento-olioita jonossa

# Suunnittelumalli: template method

- Laskin-esimerkkiin oli piilotettu myös suunnittelumalli Template Method
  - <http://www.oodesign.com/template-method-pattern.html>
  - <http://www.netobjectives.com/PatternRepository/index.php?title=TheTem>
- Summa- ja Tulo-komentojen suoritus on oleellisesti samanlainen:
  - Lue luku1 käyttäjältä
  - Lue luku2 käyttäjältä
  - *Laske operaation tulos*
  - Tulosta operaation tulos
- Ainoastaan kolmas vaihe eli *operaation tuloksen laskeminen* eroaa
- Asia hoidettiin tekemällä abstrakti ylliluokka, joka sisältää metodin suorita() joka toteuttaa koko komennon suorituslogiikan
- Suorituslogiikan vaihtuva osa eli operaation laskun tulos on määritelty abstraktina metodina laske() jota suorita() kutsuu
- Konkreettiset toteutukset Summa ja Tulo ylikirjoittavat abstraktin metodin laske()

```
public abstract class KaksiparametrinenLaskuoperaatio implements Komento {
```

```
    // ...
```

```
    @Override
```

```
    public void suorita() {
```

```
        io.print("luku 1: ");
```

```
        int luku1 = io.nextInt();
```

```
        io.print("luku 2: ");
```

```
        int luku2 = io.nextInt();
```

```
        int tulos = laske();
```

```
        io.print("vastaus: "+tulos);
```

```
    }
```

```
    protected abstract int laske();
```

```
}
```

# Template method

```
public class Summa extends KaksiparametrinenLaskuoperaatio {  
    @Override  
    protected int laske() {  
        return luku1+luku2;  
    }  
}
```

- Abstraktin luokan määrittelemä suorita() on **template-metodi** joka määrittelee suorituksen siten, että osan suorituksen konkreettinen toteutus on abstraktissa metodissa jonka aliluokat ylikirjoittavat
- Template-metodin avulla siis saadaan määriteltyä ”geneerinen algoritmirunko”, jota voidaan aliluokissa erikoistaa sopivalla tavalla
- Strategy-suunnittelumalli on osittain samaa sukua Template-metodin kanssa, siinä kokonainen algoritmi tai algoritmin osa korvataan erillisessä luokassa toteutetulla toteutuksella
- Strategioita voidaan vaihtaa ajonaikana, template-methodissa olio toimii samalla tavalla koko elinaikansa

# Lisää koodin laatuattributteja: DRY

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

# Lisää laatuattributteja

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



# Koodin luettavuus

- Perinteisesti ohjelmakoodin on ajateltu olevan väkisinkin kryptistä ja vaikeasti luettavaa
  - Esim. c-kielessä on tapana ollut kirjoittaa todella tiivistä koodia jossa yhdellä rivillä on ollut tarkoitus tehdä mahdollisimman monta asiaa
  - Metodikutsuja on vältetty tehokkuussystä
  - ...
- Ajat ovat muuttuneet ja nykytrendin mukaista on pyrkiä kirjoittamaan koodia joka nimennällään ja muodollaan ilmaisee mahdollisimman hyvin sen mitä koodi tekee
- Selkeän nimennän lisäksi muita luettavan eli ”puhtaan” koodin (clean code) tunnusmerkkejä ovat jo monet meille entuudestaan tutut asiat
  - [www.planetgeek.ch/wp-content/uploads/2011/02/Clean-Code-Cheat-She](http://www.planetgeek.ch/wp-content/uploads/2011/02/Clean-Code-Cheat-She)

