# Ohjelmistojen mallintaminen

Luento 11, 7.12.

# Viime viikolla...

- Oliosuunnittelun yleiset periaatteet
  - **Single responsibility** eli luokilla vain yksi vastuu
  - **Program to an interface, not to concrete implementation**, eli suosi rajapintoja
  - **Favor composition over inheritance**, eli älä väärinkäytä perintää
- Merkki huonosta suunnittelusta: **koodihaju (engl. code smell)**
- Lääke huonoon suunnitteluun/koodihajuun: **refaktorointi**
  - Muutetaan koodin rakennetta parempaan suuntaan muuttamatta toiminnallisuutta
- Mietittiin Kirjasto-esimerikin puitteissa miten ohjelman määrittelyä ja suunnittelua lähestytään ketterässä hengessä:
  - Vaatimukset käyttötapauksia
  - Määrittelyvaiheen luokkamalli
  - Arkkitehtuuri
  - Oliosuunnittelu
- Oliosuunnittelussa käytössä **vastuupohjainen, responsibility driven** -tekniikka
  - Noudattaa kaikkia ym. yleisiä oliosuunnitteluperiaatteita

# Tänään hieman testauksesta ja ohjelmoinnista

- Ohjelmiston elinkaareen kuuluvat vaiheet ovat siis
  - Määrittely
  - Suunnittelu
  - Toteutus
  - Testaus
  - Ylläpito
- Kuten on todettu, ketterissä menetelmissä vaiheet etenevät rinnakkain toistuen lyhyissä iteraatioissa, joiden aikana toteutetaan ohjelmaan uusia ominaisuuksia
- Testauskin jakautuu vaiheisiin:
  - **Yksikkötestaus**
    - Toimivatko yksittäiset metodit ja luokat kuten halutaan?
  - **Integrointitestaus**
    - Varmistetaan komponenttien yhteentoimivuus
  - **Järjestelmä/hyväksymistestaus**
    - Toimiiko kokonaisuus niin kuin vaatimusdokumentissa sanotaan?
- **Regressiotestauksella** tarkoitetaan järjestelmälle muutosten ja bugikorjausten jälkeen ajettavia testejä jotka varmistavat että muutokset eivät riko mitään ehjää

# Testauksen automatisointi

- Yksittäistä testiä sanotaan testitapaukseksi (test case)
  - Yksikkötesteissä on yleensä jokaista testattavan luokan metodia kohti *vähintään* yksi testitapaus, yleensä useampia
- Yksikkötestauksessa pyritään mahdollisimman hyvään *kattavuuteen*
  - Kattavuutta voidaan mitata esimerkiksi sillä kuinka suurta osaa koodiriveistä testit tutkivat
- Manuaalisesti tapahtuva testaus on toivottoman työlästä
  - Erityisesti sen takia, että ei riitä että ohjelma testataan kerran, jokaisen muutoksen jälkeen on tehtävä regressiotestaus joka varmistaa että muutos ei riko mitään
- Testit kannattaa siis tehdä koodiksi joka voidaan ajaa siten, että testikoodi varmistaa automaattisesti testattavan koodin toiminnan
  - Testikoodin ajamisen täytyy olla helppoa, ”nappia painamalla” tapahtuvaa
- xUnit-testauskehys on automatisoidun yksikkö- ja integrointitestauksen defacto-standard
  - JUnit on Javalle tarkoitettu xUnitin versio

# JUnit

- JUnitin peruskäyttö on todella helppoa erityisesti modernien kehitysympäristöjen (Netbeans, Eclipse, IntelliJ) yhteydessä
- ks. pieni JUnit-tutoriaali osoitteesta
  - https://wiki.helsinki.fi/display/ohpeJaOhja2010/JUnit
- Ohjelmoinnin harjoitustyössä ja muutenkin kaikessa laitoksella jatkossa tehtävässä ohjelmoinnissa vaaditaan JUnit-testien käyttöä
  - Puhumattakaan työelämässä tapahtuvasta ohjelmoinnista
- **Testien tekeminen jälkikäteen on toivottoman työlästä...**
- JUnit ei ole alunperin tarkoitettu noin typerään toimintaan, JUnitin kehittäjällä Kent Beckillä oli alusta asti mielessä jotain paljon järkevämpää ja mielenkiintoisempaa


**Test Drive it!**

# TDD eli Test Driven Development

- Ohjelmoija (eikä siis erillinen testaaja) kirjoittaa testikoodin
- Testit laaditaan ennen koodattavan luokan toteutusta, yleensä jo ennen lopullista suunnittelua
- Sovelluskoodi kirjoitetaan täyttämään testien sisältämät vaatimukset
  - Testit määrittelevät miten ohjelmoitavan luokan tulisi toimia
  - Testit toimivatkin osin koodin dokumentaationa, sillä testit myös näyttävät miten testattavaa koodia käytetään
- Testaus ohjaa kehitystyötä eikä ole erillinen toteutuksen jälkeinen laadunvarmistusvaihe
  - Oikeastaan TDD ei ole testausmenetelmä vaan ohjelmiston kehitysmenetelmä, joka tuottaa sivutuotteenaan automaattisesti ajettavat testit
- Testien on ennen toteutuksen valmistumista epäonnistuttava
  - Näin pyritään varmistamaan, että testit todella testaavat haluttua asiaa
- Toiminnallisuus katsotaan toteutetuksi, kun testit menevät läpi

# TDD-sykli

(1) tehdään **yksi** testitapaus

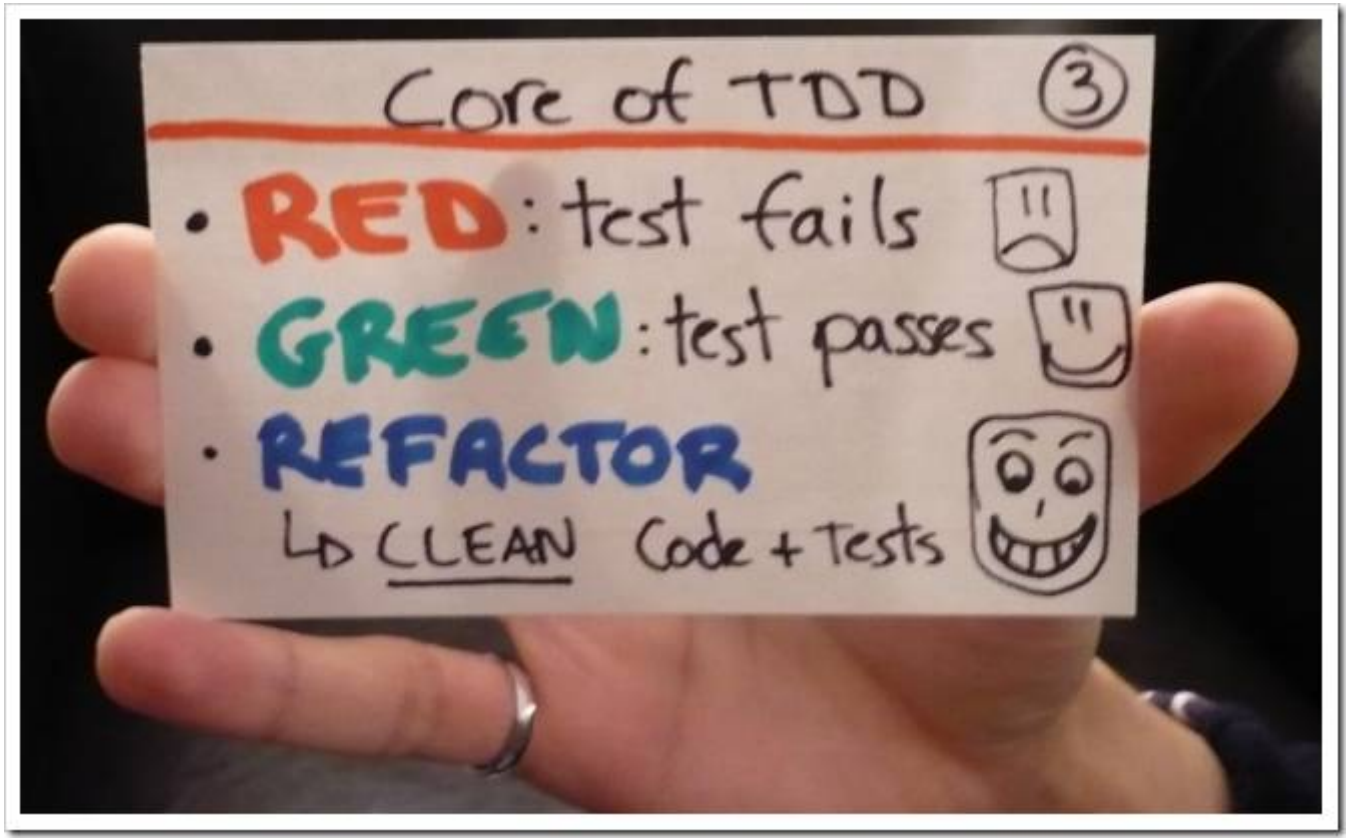
- testitapaus testaa ainoastaan yhden ”pienen” asian

(2) Tehdään koodi joka läpäisee testitapauksen

(3) **refaktoroidaan** koodia, eli parannellaan koodin laatua ja struktuuria

- Testit varmistavat koko ajan ettei mitään mene rikki

Kun koodin rakenne on kunnossa, palataan vaiheeseen (1)



# TDD

- Automaattinen testaus ja TDD ovat usein osana ketterää ohjelmistokehitystä
- Mahdollistaa turvallisen refaktoroinnin
  - Koodi ei rupea haisemaan
  - Ohjelman rakenne säilyy laajennukset mahdollistavana
- Luennolla demotaan TDD:tä ja toista tärkeää ketterän ohjelmistokehityksen käytäntöä **pariohjelmointia**