

# TDD – Tarina käyttöönotosta

Näkemys onnistuneeseen käyttöönottoon



- Novaloc Oy on vuoden 2007 alussa perustettu ohjelmistoalan yritys
- Liiketoiminta-alue IT-palveluliiketoiminnassa
  - Asiakaslähtöisten projektien tekeminen
  - Konsultointi
- Nykyisellään työllistää kymmenkunta henkilöä
- Pääasiallisesti käytetyt teknologiat Java, Spring Framework ja Apache Tomcat



- Niklas Collin
  - Vanhempi järjestelmäsuunnittelija
  - Novalocin perustajajäsen
  - ScrumMaster, koodari, hallituksen jäsen – riippuu tilanteesta
  - TTY:n kasvatti – vieläkin
    - Tutkintouudistus varmistanee valmistumisen kuitenkin

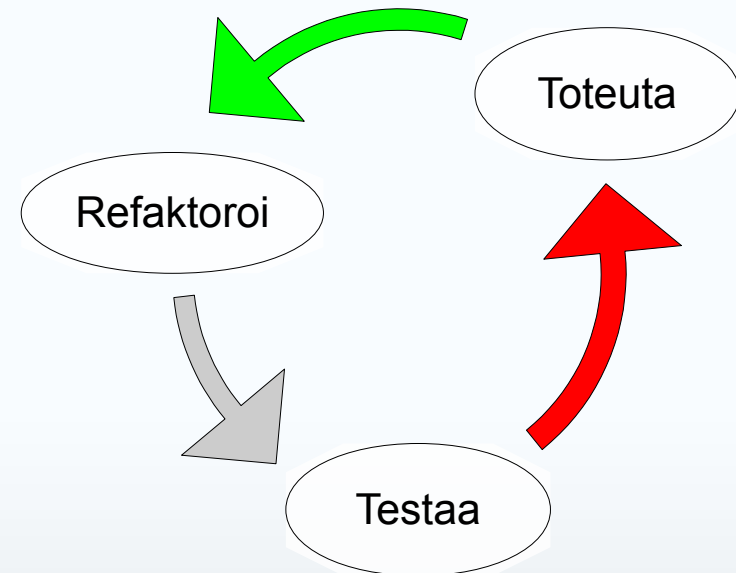


- Yksikkötestaus
  - Yksittäisen luokan testaaminen
  - Ulkoiset riippuvuudet tulisi minimoida
- Integraatiotestaus
  - Useamman luokan yhteistoiminnan testaus
  - Ulkoiset riippuvuudet sallitaan
  - Tietokantatestit jne. ovat integraatiotestejä
- Järjestelmätestaus
  - Koko paketin testaus kerralla
  - WWW-sovelluksissa käsittää selaimen läpi käskyttämisen

- Käytännössä yksikkötestausta voi tehdä kahdella tapaa:
  - "Mockist" ja "Classicist"
- Molemmissa tavoissa on etunsa, TDD:tä lienee kuitenkin helpompi lähteä opettelemaan käyttämällä mockist -tapaa
  - Perusteluna se, että mockist-tapa antaa koodarille mahdollisuuden miettiä myös toteutuskoodin sisäistä toimintaa testiä kirjoittaessa → pienempi hyppäys tavoissa
- Myöhemmin suosisin enemmän classicist -tapaa, kunhan peruskäytännöt on opittu
  - Perusteluna testien pienempi korjauksien tarve jos jotain muutetaan, koska mock-testit puuttuvat toteutuskoodin sisäiseen logiikkaan

- Mitä on TDD? Miksi Novaloc päätti siirtyä käyttämään sitä?
- Test-Driven Developmentilla (TDD, testiohjattu kehitys) on kahdenlainen tarkoitus:
  - Toimia matalan tason suunnittelun lähtökohtana
  - Varmistaa kattava testaus yksikkö- ja integraatiotestaus tasolla
- Pääasiallisena korkeamman tason motivaationa siis laadun varmistaminen
- Kattavan testipatteriston avulla minimoidaan regressio ja täten muutosten tekeminen on helpompaa

- Test – Code – Refactor
- Toteutus etenee lyhyissä sykleissä
  - Syklin pituus muutamasta kymmenestä sekunnista viiteen minuuttiin
- Ensin kirjoitetaan testi, joka testaa yksinkertaista osuutta toiminnasta
- Tätä testiä vasten tehdään haluttu toiminnallisuus
- Lopuksi refaktoroidaan jos sille on tarvetta



- Oleellinen osa testiohjattua kehitystä on aieperustainen ohjelmointi (Programming by Intention)
- Tarkoituksena on lähestyä ohjelmakoodin kehitystä black-box -ajattelulla
  - Rajapintoja tehdessä ei mietitä toiminnallisuutta, vaan sitä miten kyseistä rajapintaa haluttaisiin käyttää
- Pyritään jakamaan ohjelmakoodi useisiin metodeihin, joiden nimet ovat mahdollisimman kuvaavia
  - Ohjelmakoodista tulee luettavampaa
  - Esimerkki avaa ideaa

# Aieperustainen ohjelmointi - esimerkki



```
public class CurrencyConverter {  
  
    public static void main(String[] args) {  
        BigDecimal sum = new BigDecimal(args[0]);  
        String from = args[1];  
        String to = args[2];  
  
        Properties currencies = new Properties();  
        try {  
            currencies.load(ClassLoader.getResourceAsStream("currencies.properties"));  
        } catch (Exception e) {  
            e.printStackTrace();  
            System.exit(1);  
        }  
  
        Object fromCurrencyObj = currencies.get(from);  
        Object toCurrencyObj = currencies.get(to);  
  
        if (fromCurrencyObj == null) {  
            System.err.println("Invalid currency type '" + from + "'");  
            System.exit(1);  
        } else if (toCurrencyObj == null) {  
            System.err.println("Invalid currency type '" + to + "'");  
            System.exit(1);  
        }  
  
        BigDecimal fromCurrencyRate = new BigDecimal(fromCurrencyObj.toString());  
        BigDecimal toCurrencyRate = new BigDecimal(toCurrencyObj.toString());  
  
        BigDecimal result = sum.multiply(fromCurrencyRate.divide(toCurrencyRate));  
  
        System.out.println(sum + " " + from + " = " + result + " " + to);  
    }  
}
```

# Aieperustainen ohjelmointi - esimerkki



```
public class ImprovedCurrencyConverter {  
  
    public static void main(String[] args) {  
        BigDecimal sum = new BigDecimal(args[0]);  
        String from = args[1];  
        String to = args[2];  
  
        Properties currencies = loadCurrencyFile (); ←  
  
        BigDecimal fromCurrencyRate = getCurrencyRate (from, currencies); ←  
        BigDecimal toCurrencyRate = getCurrencyRate (to, currencies); ←  
  
        BigDecimal result = calculateResult (sum, fromCurrencyRate, toCurrencyRate); ←  
  
        System.out.println(sum + " " + from + " = " + result + " " + to);  
    }  
  
    ...  
}
```

- Jaottelu metodeihin on perustason ohjelmointiosaamista
- Aieperustaisessa ohjelmoinnissa jaottelua ei kuitenkaan tehdä jälkikäteen, vaan yllä merkityt metodikutsut tehdään **ennen** varsinaisen toteutuskoodin kirjoittamista
- Tarkoituksena on esittää ohjelmoijan **aie** tulevaisuuden toiminnallisuudesta

- Miten aieperustainen ohjelmointi sitten liittyy TDD:hen?
- Ohjelmien teko TDD:nä on käytännössä aieperustaista ohjelmointia testien suunnasta katsottuna
  - Kirjoitat aieperustaisesti testin, joka kutsuu ohjelmakoodia, jota ei ole vielä olemassakaan
- Yhdessä TDD ja aieperustainen ohjelmointi muodostavat vahvan työskentelykäytännön, joka varmistaa niin rajapintametodien järkevän rakenteen kuin myös selkeän ja luettavan koodin rajapinnan takana

- TDD terminä on nykyisellään monikäsitteinen
- Alunperin TDD on lähtöisin Extreme Programming -metodologiasta, nykyisin TDD on käytössä myös XP:n harjoittajien ulkopuolella
- Nykyisellään TDD:n voi jakaa neljään erilaiseen versioon:
  - 1) Testiohjattu kehitys**
  - 2) Testisuuntautunut kehitys**
  - 3) Testiohjattu suunnittelu (*alkuperäinen XP-tapa*)**
  - 4) Testiohjattu kehitys ja suunnittelu**
- Novaloc päätyi vaihtoehtoon 4.

- TDD on saanut osakseen paljon hehkutusta ja suoranaista hypetystä, mutta myös kritisoijia riittää
- Tunnetuimpia vastustajia lienee James "Cope" Coplien
  - Cope vastustaa erityisesti alkuperäistä XP:n tapaa
- Myös VTT:llä ovat erityisesti Sinisalo ja Abrahamsson tehneet tutkimuksia, jotka ovat tuoneet huolestuttavia tuloksia koskien TDD:n toimivuutta
  - Tutkimukset eivät kuitenkaan ole olleet riittävän kattavia, jotta niistä voisi vetää lopullisia johtopäätöksiä
  - TDD on suhteellisen vähän tutkittu aihe, viestit kentältä kuitenkin ilmaisevat sen toimivan

# Miten Novaloc toimi aluksi?

- Ensin yksi henkilö opetteli perusteellisesti sen, että mistä on oikein kyse
- Tämän jälkeen tehtiin yksi hyvin pieni (n. 1000 riviä) asiakasprojekti yksin tämän henkilön toimesta
  - Ei-kriittinen projekti, epäonnistuminen ei olisi haitannut merkittävästi
- Projektista opittiin ja oppeja hyödynnettiin kun TDD otettiin suuremmassa mittakaavassa käyttöön



- Seuraavaksi tiimi kasvatettiin neljään ja työn alle otettiin isompi projekti, jonka tarkoituksena oli kehittää monikäyttöinen alusta tuleville projekteille
- Aikaisempi yksinäinen henkilö astui ScrumMasterin rooliin ja alkoi poistamaan esteitä tiimin tieltä
  - Tällainen este oli mm. TDD:n opettaminen tiimille
- Tiimin työskentelyä seurattiin ja kannustettiin
  - Mukaan otettiin myös CI-palvelin, joksi valittiin Hudson
  - Buildauksesta huolehti Maven



- Alustan saavuttaessa pisteen, jossa sen päälle alettiin tekemään ensimmäistä asiakasprojektiä, oli koodirivejä vajaa 30 000
- Testikattavuudet 80-90% riippuen kattavuustyypistä ja moduulista
  - Aivan kaikkea ei testattu, model-objektien getterit ja setterit generoitiin jne.
- Sprinttien päätteeksi syntynyt koodi käytännössä bugitonta
- Yleisesti ottaen laatu pysyi selvästi korkeampana kuin aikaisemmin

- Novalocin siis voidaan sanoa onnistuneen TDD:n käyttöönotossa, mitä tehtiin oikein?
- ScrumMasterin rooli osoittautui kriittiseksi: ilman aktiivista kannustajaa, esteiden poistajaa ja kehitystyökalujen parantamista olisi TDD voinut osoittautua liian työlääksi hyppäykseksi kehittäjille
- Kehitystiimi tunsu toisensa erinomaisesti ja oli tehnyt paljon töitä keskenään
  - Helpotti asioiden opettamista
  - Kouluttajan jatkuva läsnäolo edesautti asioiden sisäistämistä

- Tiimi tajusi heti kättelyssä, että kirjaimellinen XP:n mukainen TDD:n seuraaminen ei voi toimia
  - Päädyttiin mukailevampaan ratkaisuun, jossa TDD:n rooli on koodipohjan kannalta kehittävä, ei suunnitteleva
- CI-palvelin ja -käytännöt olivat käytännön pakko
  - Erityisesti analyysityökalut autoivat paljastamaan "kurjat petturit"
- Testien suoritusnopeuteen kiinnitettiin erityistä huomiota
  - Mikäli testien ajaminen on liian hidasta, niin kehittäjät eivät enää aja niitä jatkuvasti ja TDD hajoaa käsiin
- Tiimin pitää saada keskittyä yksin projektiin, jossa käyttöönotto tehdään!

- Käyttöliittymäkerroksen kehittämisessä testiohjatusti ei ole mieltä
  - Automaattisilla testeillä ei kyetä testamaan kaikkea (esim. selainkohtaiset sivurakenteen hajoamiset)
- Kun toteutusratkaisu ei ole täysin selvä pitäisi käyttää ns. Spikeja asian selvittämiseen
  - Spiken toteuttaminen on kuitenkin vaikeaa ympäristössä, jossa opeteltavan asian vaativan ympäristön pystyttäminen veisi kohtuuttomasti aikaa
  - Tällaisissa tilanteissa toteutuskoodi syntyi opetteluun myötä ja testit kirjoitettiin jälkikäteen
  - Ei kuitenkaan ongelma kunhan muistetaan refaktoroida

- Ilmapiirin pitää olla refaktorointiin kannustava
  - Kenen tahansa koodeja saa ja pitää parantaa aina kun siihen näkee syyn (Common Code Ownership)
    - Kuitenkin pitää osata varoa aropupumaista asiasta toiseen pomppimista ja keskittyä oleelliseen
  - Ei painosteta jatkuvasti uusia ominaisuuksia tiimiltä, vaan luotetaan tiimin kykyyn toimittaa halutut ominaisuudet Sprintin päättyessä
- Jos ei refaktoroida myös testejä, niin TDD epäonnistuu
  - Tällöin seurauksena on korkeat testikattavuudet, mutta muutosten hallinta muuttuu todella vaikeaksi, koska testien itsensä korjaaminen on liian työlästä

- Testit ovat yhtä tärkeitä ja vaativat yhtä paljon huomiota kuin varsinainen toteutuskoodi
- Ilman refaktorointia TDD epäonnistuu ja tuloksena on vaikeasti ylläpidettävä sekamelska
- Käyttöönoton yhteydessä on tärkeää, että TDD:tä kokeileva tiimi tuntee toisensa hyvin ja on erittäin ammattitaitoinen
  - Myöhemmin nämä oppineet koodarit voivat levittää oppejaan muissa tiimeissä
- Tiimillä pitää olla "isällinen valvoja", joka auttaa minimoimaan uuden käytännön tuomat vaikeudet
- **TDD-koodarin pitää osata hyvien testien kirjoittaminen**

Kiitos kuuntelusta, toivottavasti esitys aiheutti ajatuksia, jotka johtavat parempiin käytäntöihin :)

Vastaan kysymyksiin parhaani mukaan

- Lisäluettavaa esityksen jälkeen:
  - <http://www.testdriven.com/>
  - Practical TDD and Acceptance TDD for Java Developers, Koskela 2007, ISBN 1-932394-85-0
  - Test driven Development: By Example, Beck 2004, ISBN 0321146530, 9780321146533
  - Using Continuous Integration? Better do the "Check In Dance", <http://codebetter.com/blogs/jeremy.miller/archive/2005/07/25/129797.aspx>
  - Red-Green-Refactor, <http://jamesshore.com/Blog/Red-Green-Refactor.html>
  - Mocks Aren't Stubs, <http://martinfowler.com/articles/mocksArentStubs.html>