TT00AA21-2001 Ohjelmistotuotanto (TT08S1E)

Refaktorointi

Metropolia

Refaktorointi

1. Mitä on hyvä koodi?
2. Koodin evoluutio ohjelmiston elinkaaren aikana.
3. Mitä refaktorointi on?
4. Miksi refaktoroidaan?
5. Milloin refaktoroidaan?
6. Refaktoroinnin ongelmia
7. Miten refaktoroidaan?
8. Code Smells

Mitä on hyvä koodi?

- "Any fool can write code that a computer can understand. Good Programmers write code that humans can understand" -Martin Fowler

- Ohjelmakoodin lukemiseen käytetään vähintään 10 kertaa enemmän aikaa kuin sen kirjoittamiseen!
  - Ero kasvaa ajan kuluessa suuremmaksi.
  - Ohjelmoijat lukevat koodia jatkuvasti tehdessään siihen muutoksia
  - Koodin luettavuuden parantaminen nopeuttaa koodin kirjoittamista!
    - Luettavuuden parantaminen voidaan siis tehdä kirjoittamisen kustannuksella.

Mitä on hyvä koodi?

- Hyvä ohjelmakoodi?
  - Koodia pystyy lukemaan ymmärrettävinä lauseina
  - Koodi tekee sen, mitä olettaa sen tekevän, eikä mitään muuta.
    - Ei sisällä yllätyksiä
  - Koodi läpäisee kaikki testit, mukaan lukien kääntäjän ja katselmoinnin.
  - Koodi on yhtenäinen muun koodin kanssa.

- Extramateriaali:
  Clean Code – Chapter 1: Clean Code

Koodin evoluutio ohjelmiston elinkaaren aikana

Myytti:

- Hyvin hallinnoidulla ohjelmistoprojektilla saadaan aikaiseksi stabiili vaatimusmäärittely, jonka pohjalta ohjelmisto suunnitella niin, että koodausvaihe voidaan tehdä alusta loppuun asti, testata ja unohtaa.
- Myytin mukaan mahdollisia muutoksia koodin tulee vasta ylläpitovaiheessa.

- Todellisuus?

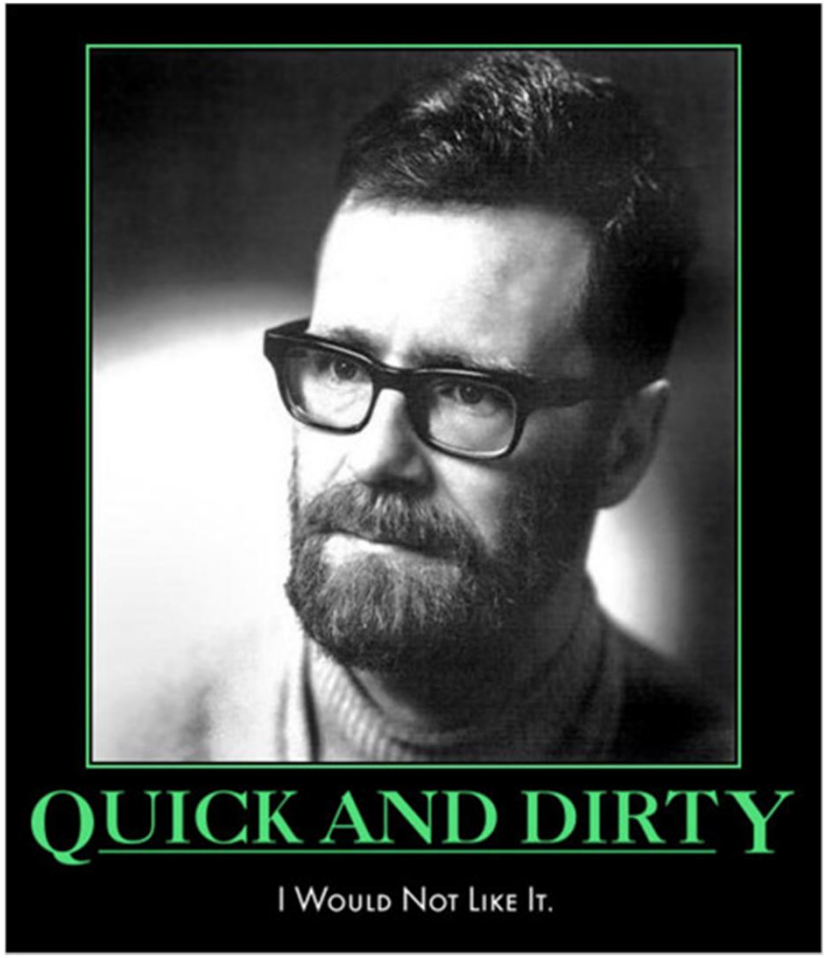
Koodin evoluutio ohjelmiston elinkaaren aikana

- Ohjelmakoodi tulee muuttumaan useita kertoja ohjelman elinkaaren aikana!
  - Muuttuvat määrittelyt, vaatimukset ja suunnitelmat
  - Virheidenkorjaus
  - Muuttumatonta koodia ei ole olemassa!
- Ohjelmistoprojektin ajasta 30-65% käytetään ohjelmointiin
  - Koodin kirjoittamiseen, debuggaamiseen ja testaukseen.
- Paljonko aikaa käytetään koodin lukemiseen?
  - Entä ymmärtämiseen?

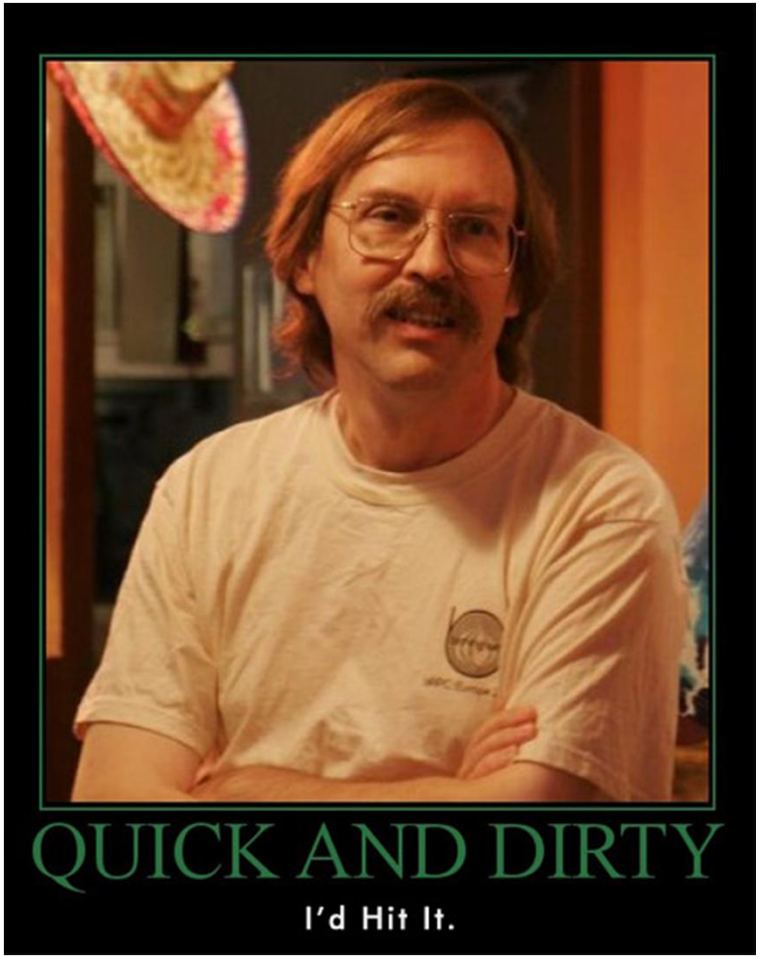
Ongelma: Miten koodimuutoksia ja korjauksia tehdään?

Koodin evoluutio ohjelmiston elinkaaren aikana

Koodin evoluutio ohjelmiston elinkaaren aikana



VS



Edsger Dijkstra
(Structured programming)

Larry Wall
(Perl)

Mitä refaktorointi on?

- Järjestelmän sisäisen rakenteen parantamista siten, että se on paremmin ymmärrettävissä ja siihen on helpompi tehdä muutoksia, ilman että järjestelmän toiminnallisuus muuttuu.
- Ohjelmakoodin suunnittelun parantamista, ohjelmakoodin kirjoittamisen jälkeen.
  - Vastakohta etukäteen tehtävälle suunnittelulle.
- Refaktorointi tehdään pienillä järjestelmällisillä ohjelmakoodin parannuksilla, joilla kumulatiivin vaikutus koko järjestelmän paranemiseen.

Mitä refaktorointi ei ole?

- Refaktorointi ei ole:
  - Virheiden korjaamista
  - Uusien toiminnallisuuksien lisäämistä.
  - Koodin kuntoon hakkerointia
    - Refaktorointi tehdään järjestelmällisesti!
  - Uudelleenkirjoittamista
    - Refaktoroinnissa parannetaan jo olemassa olevaa koodia
  - Optimointia
    - Refaktorointi on optimoinnin vastakohta!

Miksi refaktoroidaan?

- Parannetaan ohjelmiston suunnittelua!
  - Ohjelmakoodin laatu paranee!
  - Huonosti suunniteltua koodia on vaikea korjata ja muokata.
- Refaktorointi helpottaa ohjelmakoodin ymmärtämistä!
  - Koodin laatua parantamalla, koodi on helpompi ymmärtää.
  - Refaktorointia tehdessä myös oma ymmärrys koodista paranee:
    - Pienten muutosten teko auttaa ymmärtämään koodia isommassa mittakaavassa.
    - Pelkällä koodin lukemisella on vaikea saavuttaa tarkan tason ymmärtämistä.

Miksi refaktoroidaan?

- Auttaa löytämään virheitä
  - Tarkka ymmärrys koodista auttaa virheiden löytämisessä.
- Nopeuttaa ohjelmointia!
  - Mitä paremmin ymmärtää ohjelmakoodin, sen helpompi siihen on tehdä muutoksia.
  - Hyvin suunniteltuun ohjelmakoodin on helppo lisätä toiminnallisuutta.

Milloin refaktoroidaan?

- Partiolaisten sääntö:
  ”Jätä leirintäalue siistimmäksi kuin, mitä se oli sinne tultaessa.”
- Refaktorointia pitää tehdä jatkuvasti!
- Syy refaktoroida:
  - Refaktorointia tehdään, koska halutaan tehdä jotain muuta ja refaktorointi auttaa siinä.
- Jos refaktoroinnille varataan erikseen aikaa, riskinä on sen unohtaminen kiireessä.
  - Projektipäälliköt näkevät refaktoroinnin usein turhana työnä...

Milloin refaktoroidaan?

- Refaktoroi kun lisäät toiminnallisuutta:
  - Uusi toiminnallisuus on helpompi lisätä hyvin suunniteltuun koodin.
  - Myöhemmin lisättävät toiminnallisuudet hyötyvät aiemmasta refaktoroinnista.
- Refaktoroi kun korjaat virheitä:
  - Refaktorointi auttaa ymmärtämään ohjelmakoodia paremmin.
- Refaktoroi koodinkatselmoinnin yhteydessä.
- Joskus (harvoin) on parempi kirjoittaa koodi alusta lähtien uusiksi
  - Muista tällöin myös refaktoroida uusiksi kirjoitettu koodi!

Refaktoroinnin ongelmia

- Tietokantojen muuttaminen
  - Sovellukset ovat usein tiukasti sidoksissa tietokantaskeemaan
    - Muutoksilla tällöin kumulatiivisia vaikutuksia muuhun sovellukseen
  - Datan migraatio työlästä toteuttaa
  - Tietokannan kunnollinen erottaminen omalle tasolle auttaa ongelmaan.
- Julkiset rajapinnat
  - Koodia, jota ei omisteta ei voida muokata
  - Julkaistuun APIin tehtävät muutokset
    - Javan siirtyminen 1.4 -> 1.5 versioon
  - Säilytettävä usein tuki vanhemmille versioille.
    - Vanhat rajapinnat lisäävät kompleksisuutta.

Refaktoroinnin ongelmia

- Suuret refaktoroinnit vaikeasti muutettaviin rakenteisiin.
  - Esim. Arkkitehtuurin muuttaminen.
  - Yleensä ongelma on pienempi kuin kuvitellaan!
  - Tehdään pienillä refaktoroinneilla!
  - Suunnitellaan tehtävät toimenpiteet tarkemmin.

Refaktoroinnin ongelmia

- Refaktorointi ei välittömästi lisää sovelluksen arvoa
  - Sovelluksen toiminnallisuutta ei muuteta.
  - Tällöin refaktorointi nähdään usein turhana työnä.
- Mitä siis kerrot asiakkaalle / projektipäällikölle?
  - Jotkut pp:t ymmärtävät refaktoroinnin arvon ohjelmiston laadun kannalta.
  - Jotkut taas eivät…
- Refaktorointi nopeuttaa omaa kehitystyötäni, täten siis refaktoroin!
  - Älä kerro projektipäällikölle miten suoritit tehtävän!

Miten refaktoroidaan?

- Tehokas refaktorointi perustuu järjestelmälliseen muokkausprosessiin.
  - Tehdään pieniä muutoksia kerralla!!!
  - Varmistetaan muutoksien oikeellisuus.
  - Tee muutos, käännä ja testaa, tee seuraava muutos.
- Säilytä alkuperäinen koodi!
  - Jos jotain meni pieleen, voidaan palata alkuperäiseen versioon.
  - Toimiva versionhallinta hoitaa tämän ohjelmoijan puolesta.

Miten refaktoroidaan?

- Selvitä koodista ongelmakohtia hajujen perusteella
  - Refaktoroinnin edetessä löydetään usein muitakin korjattavia kohtia.
  - Kaikki ongelmakohdat on kirjataan ylös, jotta niihin voidaan palata myöhemmin.
- Tee aina yksi refaktorointi loppuun asti, ennen kuin aloitat seuraavan.
  - Älä yritä korjata kaikkea kerralla!
  - Varmista aina muutetun koodin toimivuus.

Miten refaktoroidaan?

- Testaa!
  - Lisää tarvittaessa uusia testitapauksia
  - Arvioi ja korjaa vanhoja testitapauksia.
- Arvioi jatkuvasti tehtyjä muutoksia.
- Nykyaikaiset IDEt sisältävät automaattisia refaktorointityökaluja
  - Opettele käyttämään!
  - Virheen mahdollisuus katoaa, kun tehtävä on automatisoitu.
  - Saatavilla on myös refaktorointiin keskittyviä työkaluja kuten ReSharper.

Refaktoroinnit

- http://www.refactoring.com/
- Refactoring: Improving the design of Existing code (Martin Fowler)
- Clean Code: A handbook of Agile Software Craftmanship (Robert C. Martin)
- Code Complete 2: A practical handbook of software construction
- ”suunnittelumalleja” koodin korjaamiseen:
  - Nimi
  - Miksi tehdään?
  - Miten tehdään? Askel askeleelta.
  - Esimerkki

Code Smells

- Refaktoroinnin tarpeita selvitetään koodin hajujen perusteella.
  - Yleisimpiä ongelmia koodissa
  - Miksi ne pitäisi korjata
  - Heuristinen menetelmä, ei ole siis eksaktia tiedettä.
- Hyvä ohjelmoija osaa usein tunnistaa huonon koodin, muttei osaa tehdä asialle mitään.
  - Hyvä refaktoroija osaa korjata huonon koodin!

- Extramateriaali:
  Clean Code – Chapter 17: Smells and Heuristics
- http://wiki.java.net/bin/view/People/SmellsToRefactorings
- http://www.codinghorror.com/blog/2006/05/code-smells.html

Nimeäminen

- Hyvät nimet muodostavat 90% ohjelmakoodin luettavuudesta!
  - Käytä nimien miettimiseen aikaa!
- Nimeä, uudelleennimeä ja vielä kerran uudelleennimeä!
  - Uudelleennimeäminen on helpoimpia refaktorointeja!
- Nimet:
  - Kuvaavia! Kertovat kaiken tarpeellisen!
  - Perustuvat sovittuun ohjelmointikäytäntöön!
  - Yksiselitteisiä!
  - Vältä enkoodauksia ja lyhennyksiä!
    - strcBACdata* _ptr_BACdata; ???
    - Nykyaikaiset kääntäjät osaavat selvittää tyypeistä kaiken tarpeellisen, tätä tietoa ei tarvitse kirjata itse ylös.
    - Nimien maksimipituus ei enää ole 8 merkkiä.

Duplikaattikoodi

- Sama koodi esiintyy useassa eri paikassa, jolloin siihen tehtävä muutokset joudutaan tekemään kaikkialla.
- Kertoo usein huonosta luokkasuunnittelusta.
  - Luokan sisällä kaksi metodia sisältää saman koodin.
  - Sisarluokat sisältävät saman metodin.
  - Toisistaan riippumattomat luokat käyttävät samaa koodia.
- Korjataan siirtämällä koodin toteutus vain yhteen paikkaan
- *Extract Methdod, Pull up Method, Form Template Method, Extract Class*

Ylipitkät metodit

- Liian pitkä metodi on toinen yleisimpiä ongelmia.
  - Metodilla pitkä parametrilista
  - Käyttää paljon tilapäisiä muuttujia.
  - Sisältää paljon kommentteja
  - Metodi yrittää tehdä liian monta asiaa (SRP!)
- Metodin pitää olla lyhyt, suositeltu pituus on 1 rivi.
- Korjataan jakamalla metodi pienempiin osiin.
- Tilapäisistä muuttujista voidaan muodostaa omia luokka.
- Parametrilista voidaan kapseloida omaan luokkaan, ja siirtää toiminnallisuutta siihen.
- *Extract Methdod, Replace Temp with Query, Replace Method with Method Object, Decompose conditional, Replace Parameter with Method, Introduce Parameter Object, Preserve Whole Object*

Suuret luokat

- Luokka yrittää tehdä liikaa asioita: (SRP!)
  - Luokalla on suuri määrä jäsenmuuttujia
  - Luokan rajapinta on suuri.
- Osa luokan jäsenmuuttujista voidaan kapseloida omaan luokkaan.
- Osa toiminnasta voidaan toteuttaa aliluokkana.
- Rajapintaa voidaan pienentää selvittämällä miten luokkaa käytetään ja erottamalla siitä useita rajapintoja.
- *Extract Class, Extract Subclass, Extract Interface, Replace Data with Value Object*

Poikkeuksellinen muutosten määrä

- Haulikkokirurgia:
  - Yhden muutoksen tekeminen vaatii useita muutoksia eri puolille koodia.
  - Kertoo usein duplikaattikoodista.
  - Luokkien väliset roolijaot eivät ole kunnossa.
- Muutoskeskittymät
  - Yhtä luokkaa joudutaan jatkuvasti muuttamaan eri syistä.
  - Luokka luultavasti tekee liian monta asiaa.
  - Korjataan luomalla uusi luokka, johon vain tietyt muutokset keskittyvät.
- *Move Method, Move Field, Inline Class, Extract Class*

Ominaisuuskateus

- Luokan metodi käyttää enemmän toisen luokan jäsentietoa kuin omaansa.
- Kertoo usein siitä, että metodi sijaitsee väärässä luokassa.
- Yleistä dataluokkien käytössä, jossa luokka ei sisällä muuta kuin jäsenmuuttujia.
- *Move Method, Move Field, Extract Method, Encapsulate Field, Encapsulate Collection*

Tietokasaumat

- Samat muuttujat esiintyvät jatkuvasti yhdessä ympäri ohjelmakoodia.
- Metodien parametrilistoissa välitetään aina samat muuttujat.
- Kertoo usein siitä, että tietojoukko kuuluu yhteen ja pitäisi kapseloida omaksi luokaksi.
- *Extract Class, Introduce Parameter Object, Preserve Whole Object*

Switch -lausekkeet

- Sama switch-lauseke esiintyy ohjelmakoodissa useampaan kertaan.
  - Kertoo huonosta oliopohjaisesta suunnittelusta
  - Ohjelmalogiikkaa ja dataa ei ole osattu erottaa oikein.
- Pystytään usein korjaamaan perinnällä ja polymorfismilla.
- switch == if
  - Pätee siis myös if-lausekkeisiin.
- http://www.antiifcampaign.com/
- *Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object*

Käyttämätön ohjelmakoodi

- Abstraktit luokat tai normaalit luokat jotka eivät tee mitään
- Metodit, joita ei koskaan kutsuta
- Käyttämättömät jäsenmuuttujat ja parametrit
- Pois-kommentoitu ohjelmakoodi!
- Kertoo usein ylisuunnittelusta: ohjelmakoodia on suunniteltu tulevaisuutta varten
  - Lisää pelkästään kompleksisuutta.
- Kaikki koodi, mitä ei käytetä, pitää poistaa!
  - Koodin saa takaisin versionhallinnasta, jos sille joskus tulee käyttöä.

Loogiset sidonnaisuudet

- SOLID periaatteet!
- Kaikki loogiset sidonnaisuudet ovat pahasta.
- Refaktoroi kaikki loogiset sidonnaisuudet siten, että niistä tulee fyysisiä.
- *Move Method, Move Field, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation, Hide Delegate*

Kommentit

- Kommentteja käytetään usein selittämään huonoa koodia!
  - Kertovat, että koodissa on jotain vikaa, koska sitä täytyy selitellä.
- Turhat kommentit vaikeuttavat koodin lukemista.
  - Itsestään selvät kommentit: // this is constructor
  - JavaDoc kommentit, joissa vain toistetaan metodin nimeä ja palautusarvoa
- Asiaankuulumaton tieto:
  - Kuuluu muualle kuin koodin sekaan
  - Versiohistoriaa, muutoshistoriaa, todo-listat...

Kommentit

- Vanhentuneet kommentit
  - Kommentit jotka eivät enää pidä paikkaansa
  - Vaikeuttavat koodin ymmärtämistä!
- Kommenttia kirjoittaessa:
  - Mieti miten korjaat koodin siten, että kommentointi on turhaa.
  - Jos kuitenkin tarvetta kommentoida, kirjoita kommentti huolellisesti!
    - Mitä kielioppivirheet yms... hätäiset kommentit kertovat kommentoidun koodin laadusta? Entä kirjoittajasta?
  - Hyödyllisin kommentti kertoo koodista ”miksi” jokin asia on tehty, ei miten.

Kysymyksiä?