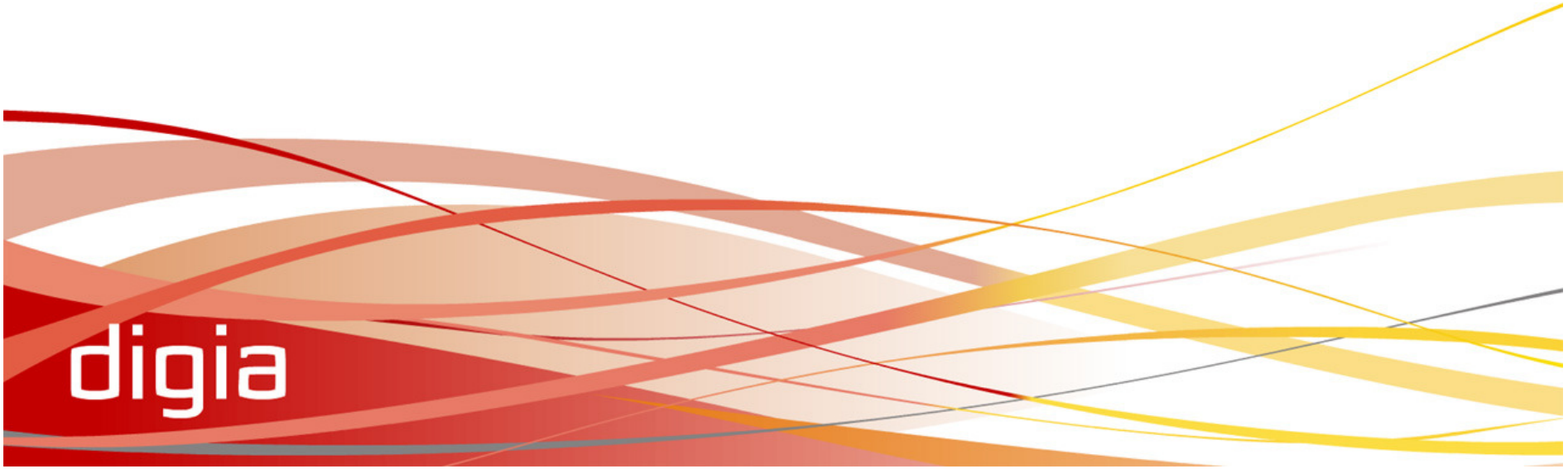
Code Quality

Koodia luetaan useammin kuin sitä kirjoitetaan

Panu Oksala



Yhdeksän teesiä

1. Voita kompleksivisuus (yksinkertaista!)
2. Valitse prosessisi, ole tietoinen laadusta koko ajan
3. Kirjoita ohjelmia ensiksi ihmisille, sitten vasta koneelle (luettavuus, ylläpidettävyys, laajennettavuus)
4. Mieti mitä ja miksi, sitten vasta miten
5. Merkintätavat, auttavat hallitsemaan kompleksivisuutta, saat keskittymiskykysi parempaan käyttöön
6. Ohjelmoi toimialueen termeillä
7. Varo suden kuoppia, käytä maalaisjärkeä ja pysy valppaana
8. Iteroi, toista vaatimuksia, suunnittelua, arvioita ja koodia
9. Ole valmis oppimaan ja kokeilemaan, vältä kangistumista vanhoihin tapoihin

digia

DRY

- **Don't Repeat Yourself**
  - Sisältää koodin, dokumentaation, kommentit, tietokannat…
  - DRY prinsiipin rikkominen vaatii muutostilanteessa aina muutoksia useampaan kohtaan
- ✓ **WET = Write Everything Twice**

http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

digia

Hyvän luokan keskeisimmät ominaisuudet

- Abstraktio
  - Luokka kuvastaa vain ja ainoastaan yhtä asiaa
  - Luokalla on yhtenäinen rajapinta ulospäin (Save & Load, Insert & Delete…)
- Kun luokalla on vain yksi tehtävä, luokalla on vain yksi syy muuttua. Ks. *Single responsibility principle*
- Esimerkkejä
  - Customer – Kuvaa yhtä asiakasta. Ei asiakkaan tapahtumia, osoitteita tai tilauksia
  - ✓ Janitor/Manager luokan nimessä kuvastaa, että luokalla ei ole selkää tehtävää
  - ✓ CustomerRelation – Kuvaa asiakkaan suhdetta… mihin? Toiseen asiakkaaseen?

Hyvän luokan keskeisimmät ominaisuudet

- Tiedon piilotus (Encapsulation)
  - Luokan sisäiset toiminnot on piilotettu (Private)
  - Julkisten muuttujien ja metodien määrä on pieni
    - Piilota aluksi, julkaise tarvittaessa. Säilytä kuitenkin eheä rajapinta (Add/Remove, Start/Stop…)

- Esimerkiksi
  - Customer
    +Insert()
    +Update()
    +Delete()
  - ✓ Customer
    +Insert()
    +Update()
    - Delete()

Hyvän luokan keskeisimmät ominaisuudet

- Riippuvuudet
  - Riippuvuus toisen luokan toteutuksesta. Metodikutsua tehtäessä, tutkitaan sisäistä toteutusta ja luodaan sen pohjalta kutsu -> luodaan riippuvuus kutsuttavan metodin toteutukseen, ei itse metodiin
- Esimerkki
  - ✓ Kutsutaan Customer luokan Insert() metodia puutteellisella datalla, koska tiedetään että Customer luokka täyttää itse puuttuva tiedot => luodaan riippuvuus Insert() metodin toteutukseen
  - Kutsutaan Customer luokan Insert() metodia oikein täytetyllä datalla tai eriytetään puutteellisten tietojen täyttö omaan metodiin, jota hyödynnetään

Hyvän luokan keskeisimmät ominaisuudet

- Riippuvuudet
  - Law of Demeter: Luokka A saa kutsua mitä tahansa omaa metodia. Jos luokka A instantoi luokan B, se saa kutsua mitä tahansa luokan B julkista metodia. Luokka A ei kuitenkaan saa kutsua luokan C metodia luokan B kautta.
  - Yksinkertaistettuna: Käytä vain yhtä pistettä lauseissa (a.Method(), ei a.b.Method())
- Esimerkkejä
  - Customer luokka instantoi Address luokan ja julkaisee **Address luokan propertyja, omina propertyinaan**. Order luokka käyttää Customer luokan propertyja
  - ✓ Customer luokka julkaisee propertyna luokan Address, jota käytetään luokasta Order

```
public void SetCustomerCityToLohja()
{
    OrderCustomer.CustomerAddress.City = "LOHJA";
}
```

Luokka Customer, instantoi Addressin, mutta luokka Order käyttää sitä luokan Customer kautta

Hyvän luokan keskeisimmät ominaisuudet

- Luokan muuttujien lukumäärä: 7±2
  - Ihmisen muistin rajojen mukaan
- Pidä metodien lukumäärä mahdollisimman pienenä, muistaen kuitenkin metodien omat rajoitteet => mieti tarvittaessa luokan pilkkomista pienempiin kokonaisuuksiin
- Mitä vähemmän luokka käyttää muita luokkia, sitä vähemmän siinä on keskimäärin virheitä (Basili, Briand ja Melo 1996)
- http://en.wikipedia.org/wiki/Anti-pattern#Object-oriented_design_anti-pattenrs

Metodit

Metodin nimeäminen

- Hyvä metodin nimi kuvaa kaiken mitä metodi tekee
  - Customer luokan GetAllCustomerOrders palauttaa kaikki asiakkaan tilaukset
- Vältä merkityksettömiä verbejä: Process, Deal, Perform…

```
private void HandleCustomerAddressData()
{
    CustomerAddress.StreetAddress.Trim();
    CustomerAddress.ZipCode.Trim();
    CustomerAddress.City.Trim();
}
```

- Älä erota metodeja numeroilla
  - ✓ GetCustomerOrderById1… GetCustomerOrderById2
- Käytä ennemmin kuvaavaa pitkää nimeä kuin typistettyä ja ei kuvaavaa
  - DeleteAllOpenOrders vs. DeleteOrders

Metodit

- Metodin koheesio (=keskinäinen vetovoima):
- Metodi tekee vain ja ainoastaan yhden asian
  - Customer luokan IsCompanyCustomer, ei tallenna asiakkaan osoitetta ja tarkista sen jälkeen onko asiakas yritysasiakas
  - Pitkä metodin nimi paljastaa yleensä heikon koheesion. Esimerkiksi ValidateAndFixCustomerName
    - Metodi tekee vähintäänkin 2 asiaa
  - Vahvasti samaa dataa käyttävät asiat, voidaan niputtaa samaan metodiin. Esimerkiksi StartUp metodi lukisi config tiedoston ja alustaisi sen pohjalta kaikki muuttujat.
  - Suuremman koheesion metodit on yleensä helpompi yksikkötestata

Metodit

- CQS prinsiippi – Metodi suorittaa joko toiminnon (Command) tai palauttaa dataa (Query), ei molempia.
  - ”Asking a question should not change the answer”
  - GetStatus = Query
  - CreateOrder = Command
  - Lisätietoja: http://en.wikipedia.org/wiki/Command-query_separation
  - Maalaisjärkeä

Metodit

- Metodien pituudet
  - Maksimi teoreettisesti noin 200 riviä. Poikkeuksia toki löytyy!
  - Minimiä ei ole (yhden rivin metodi on varsin hyvä)
  - Metodia lukiessa, muistettavien asioiden määrä kasvaa pituuden kanssa
- Metodin rakenne
  - ✓ Vältä syviä sisäkkäisiä logiikoita (koodi muistuttaa nuolenpäätä)

```
If(xxx)
   If(yyy)
      If(zzz)…
```

  - Käytä tarvittaessa returnia yksinkertaistuksessa. Returnit keskellä kuitenkin vaikuttaa ylläpidettävyyteen heikentävästi

```
If(xxx = false)
   return;
If(yyy)…
```

Muuttujat

- Muuttujien nimeäminen
  - Samalla tavoin kuin metoditkin, kuvaava ja tarvittavan pitkä
    - Andy Lester on nimennyt huonoimmaksi muuttujan nimeksi ”Data”. Toiseksi huonoin nimi on ”Data2”
  - Muistetaan notaatio (luokan muuttujat _xx jne.)
  - Vältä samankaltaisia nimiä lohkoissa. Esimerkiksi input ja inputValue. Nimet on helppo sekoittaa keskenään
- Muuttujien esittely
  - Mahdollisimman lähellä käyttökohtaa => ei kannata esitellä kaikkia muuttujia metodin alussa
  - Pidä muuttujan elinikä mahdollisimman lyhyenä
    - Mahdollisimman vähän globaaleja muuttujia
    - Esittelyt lohkojen sisälle
  - Tyypitä mikäli mahdollista (!object), hyödynnä tarvittaessa genericsejä

Vakiot

- Käytä vakioita ja enumeraatioita esittämään taikanumeroita.
  - If (realOrder.InsertType == 0)
    realOrder.OrderInsertType = 3;
  - If (realOrder.InsertType == InsertTypes.UNKNOWN)
    realOrder.InsertType = InsertTypes.CUSTOMERS_INITIAVE;
- Vakiot helpottavat koodin luettavuutta ja helpottavat ylläpitoa
- Muutokset on luotettavampi toteuttaa kun numerot eivät sekoitu keskenään (*”Onkohan tämäkin 3 InserType vertailua?”*)

Boolean muuttujat

- Käytä tarvittaessa boolean muuttujia yksinkertaistamaan vertailuja
  - ✓
```
if (CustomerId == string.Empty &&
   (AddressFlagStatus == AddressFlags.Inserted || AddressFlagStatus == AddressFlags.Updated)
                        …
```
  -
```
bool isCustomerIdEmpty = string.IsNullOrEmpty(CustomerId);
bool hasAddressChanged = (AddressFlagStatus = AddressFlags.Inserted ||  AddressFlagStatus =
      AddressFlags.Updated);

if (isCustomerIdEmpty && addressChanged )
                        …
```
- Muutokset jälkimmäiseen esimerkkiin on huomattavasti mukavampi toteuttaa

Järjestyksellä on väliä

- Metodien kutsut saattavat olla riippuvaisia toisista kutsuista. Nimeät metodit siten että riippuvuudet tulee ilmi, Initialize, Create, jne.

```
public Order CreateOrder() { … }
public void InitializeOrder(Order ord) { … }
```

- Järjestele koodi niin että sitä luetaan ylhäältä alaspäin ja luettaessa ei tarvitse hyppiä pitkin koodia.

```
Order ord = new order();
ord.CustomerId = 123;
ord.CurrentStatus = OrderStatus.DELIVERED;

if( orderedCustomer.IsCompanyCustomer)
{
    orderedCustomer.FirstName = orderedCustomer.FirstName.ToUpper();
    orderedCustomer.LastName = orderedCustomer.LastName.ToUpper();
}

Address adr = new Address();
adr.City = "JYVÄSKYLÄ";
adr.StreetAddress = "KATUTIE 1";

if (!orderedCustomer.HasAddress)
{
    Debug.WriteLine("Customer doesn't have address. Cannot deliver order");
}
 ord.Priority = Order.PRIORITY_LOW;
```

Järjestyksellä on väliä

- Vertailujen järjestely
  - Todennäköisimmin toteutuva vaihtoehto mielellään if haaraan, eikä elseksi. switch casessa ylimmäksi case kohdaksi
  - Poikkeustapaukset viimeiseksi
  - Vältä tyhjiä if/else haaroja

```
if (xxx)
  // Do nothing here
else
  yyy();
```

```
if (!xxx)
  yyyy();
```

- Parametrien järjestys
  - 1. Sisään tulevat parametrit (input)
  - 2. Sisään tulevat ja ulos lähtevät (input and output)
  - 3. Ulos lähtevät (output)
  - Ryhmittele samankaltaiset parametrit
  - Käytä kaikkia parametreja!

Kommentointi

- Mitä pitäisi kommentoida ja miten?
  - Vältä turhaa ASCII hifistelyä kommenteissa. Ylläpitäminen on työlästä
  - Kommentoi poikkeavat ratkaisut
  - Kommentoi syötteiden rajoitukset (numeroavaruudet, kokorajoitukset jne.)
  - Elä kommentoi koodia itseään, kommentoi asioita joita koodi ei voi kertoa (abstraktioita, koodin käyttökohteita, käytettyjä yleisiä ratkaisuja, jne.)
    - Hyvä koodi kommentoi itse-itsensä. Jos joudut selittämään koodia kommenteilla, mieti koodin jäsentelyä ja muuttujien nimeämistä!

Päivien lukumäärä esimerkki

- Esimerkki huonosta toteutuksesta kuukauden päivien hakemiseksi

```
Private int GetMonthsDayAmount(int month)
{
      if (month == 1)
         return 31;
      else if (month == 2)
         return 28;
      else if (month == 3)
         return 31;
      else if (month == 4)
         return 30;
      else if (month == 5)
         return 31;
      …
      else
         return -1;
}
```

Päivien lukumäärä esimerkki

- Hieman parempi toteutus, hakutaulukolla

```
int _daysInMonth[] = {31, 28, 31, 30, 31…};

Private Function GetMonthsDayAmount(int month)
{
   if (month < _daysInMonth.Length)
      return _daysInMonth[month];
   else
      Return -1;
 }
```

Päivien lukumäärä esimerkki

- .NET kirjastoa hyödyntävä toteutus

```
Private Function GetMonthDayAmount(int month)
{
   return DateTime.DaysInMonth(DateTime.Now.Year, month);
}
```

- Sama asia voidaan toteuttaa monella eri tavalla. Pysähdy välillä miettimään, olisiko parempaa menetelmää olemassa
- Hyödynnä valmiita kirjastoja ja suunnittelumalleja

Kiitos!

panu.oksala@digia.com