Käytännön formaalit menetelmät

TIEA202 Johdatus ohjelmistotekniikkaan, syksy 2006

Antti-Juhani Kaijanaho

Jyväskylän yliopisto
Tietotekniikan laitos

21. marraskuuta 2006

Luennoija

- Antti-Juhani Kaijanaho <antkaij@mit.jyu.fi>
- ohjelmistotekniikan assistentti
- pro gradu v. 2002 eräästä formaalista menetelmästä
- tutkimusaiheena ohjelmointikielet
  - (mutta myös formaalit menetelmät kiinnostavat edelleen)
- luennoinut Formaalien menetelmien kurssia v. 2003–2005
- luennoi keväällä syventävän opintojakson TIES542 Ohjelmointikielten perisstteet
- avustaa keväällä prof. Tuomo Rossia TIES543 Formaalit menetelmät -kurssilla

Sisältö

Kirjallisuus

- Antti-Juhani Kaijanaho, Tommi Kärkkäinen: *Formaalit menetelmät*. Luentomoniste 12, Jyväskylän yliopisto, Tietotekniikan laitos, 2005.
- Muuta yleisluontoista kirjallisuutta on nihkeästi. Useista yksittäisistä menetelmistä ja notaatioista on erinomaisia kirjoja.
- Ohjelmistotekniikan oppikirjoissa (Pressman ym.) on yleensä luku formaaleista menetelmistä. Ne vain raapaisevat pintaa.
- Paljon hajanaisia tieteellisiä julkaisufoorumeita, ks. esim. `http://vl.fmnet.info/pubs/`.

Laatu?

> Equipped with the basic concepts of class, object and genericity, you can now write software modules that implement possibly parametrized types of data structures. Congratulations. This is a significant step in the quest for better software architectures.
>
> But the techniques seen so far are not sufficient to implement the comprehensive view of quality introduced at the beginning of this book. The quality factors on which we have concentrated — **reusability, extensibility, compatibility – must not be attained at the expense of reliability** (*correctness* and *robustness*).

– Bertrand Meyer: *Object-oriented Software Construction*, Second Edition, Prentice-Hall 1997, luku 11. Lihavointi AJK:n.

Laatu?

> *Program testing can be used to show the presence of bugs, but never to show their absence!*

– Edsger W. Dijkstra: *Notes On Structured Programming*, EWD249, 1970.

Formaali?

- suomeksi *muodollinen*
- matematiikassa erityismerkitys: 'mekaanisesti (tietokoneella) tarkistettavissa oleva'
- matematiikassa myös: 'erityisen täsmällisesti tehty'.
- ohjelmistotekniikassa:
  - yleensä yleiskielen merkitys
  - formaalien menetelmien yhteydessä matemaatikka-merkitys
- Huomaa sekaantumisen vaara ja varo!

Formaalit menetelmät

- hyvin laaja ala
- ei kaikenkattavaa määritelmää
- yhteistä matemaattisen täsmällisyyden korostaminen
- tavallista tietojenkäsittelylogiikan teorioiden soveltaminen
- ohjelmistotekniikan "omaa matematiikkaa"
  - sovellusalueen matikka ei kuulu tähän

Formaalit menetelmät 2

- Formaalit menetelmät -nimistä tutkimusalaa ei oikeastaan ole.
- kattaa sekalaisen joukon välineitä
- yhdistää löyhästi matemaattisen täsmällisyyden vaaliminen
- miksiköhän koko käsite on olemassa?
  - Ghetto-ilmiö?
  - Tekoäly-ilmiö?

Formaaliuden muotoja

- ohjelman vaatimusten täsmällistäminen aikaisessa vaiheessa (*formal specification*)
- virhetyyppien poissulkeminen staattisin tarkastuksin (*static checking*)
- kriittisen komponentin oikeellisuuden varmistaminen (*correctness proofs*)
- ohjelman johtaminen tehtäväkuvauksesta (*program derivation*, *contract refinement*)
- automaattinen testaus (*QuickCheck*, *assertions*)

Hyötyjä?

- analyyttinen ote:
  - Tiedetään mitä tehdään.
  - Virheiden määrä vähenee.
  - selkeärakenteinen speksi ja koodi
- abstrakti ote:
  - Kokonaisuus hallussa käsiä heiluttelematta?
  - Uudelleenkäyttö jäsentyy.

Esteitä

- litteä maa -efekti
- tekoäly-efekti
- todellinen ongelma: matemaattisen taidon puute

Formaaliuden tasot

Tasolla 0 ei käytetä lainkaan formaaleja menetelmiä eikä muuta matematiikkaa kuin sovellusalan omaa.

Tasolla 1 käytetään epämuodollista matematiikkaa lähinnä dokumentaation tai ideoinnin apuvälineen.

- Käytetyn matematiikan syntaksi ja semantiikka horjuu eikä siten sovellu automaattisesti käsiteltäväksi.

Tasolla 2 käytetty matemaattinen notaatio on kiinnitetty sekä syntaksinsa että semantiikkansa osalta. Työskentelyä tukemassa käytetään joitakin automaattisia työkaluja.

Tasolla 3 menetelmän syntaksi ja semantiikka on täysin määritelty ja täysin automatisoitu. Tuotosten sisäinen konsistenttius ja tuotosten keskinäinen oikeellisuus voidaan mekaanisesti todentaa.

Seitsemän myyttiä

1. Formaalit menetelmät kykenevät takaamaan, että ohjelmisto on täydellinen.
2. Formaaleissa menetelmissä on kyse pelkästään ohjelmien oikeellisuustodistuksista.
3. Formaalit menetelmät auttavat vain kriittisissä (safety-critical) projekteissa.
4. Formaalit menetelmät vaativat korkeasti koulutettuja matemaatikkoja.
5. Formaalit menetelmät kasvattavat kehityskustannuksia.
6. Formaalit menetelmät eivät kelpaa käyttäjille.
7. Formaaleja menetelmiä ei käytetä todellisten, laajojen ohjelmistojen kehityksessä.

– Anthony Hall: *Seven Myths of Formal Methods*, IEEE Software, vol 7, no 5, September 1990.

Seitsemän muuta myyttiä

8. Formaalit menetelmät hidastavat kehitysprosessia.
9. Formaaleilla menetelmillä ei ole työkalutukea.
10. Formaalit menetelmät korvaavat perinteisen ohjelmistotuotantoprosessin.
11. Formaalit menetelmät sopivat vain softan kehitykseen.
12. Formaalit menetelmät ovat tarpeettomia.
13. Formaaleja menetelmiä ei tueta.
14. Formaalien menetelmien ihmiset käyttävät aina formaaleja menetelmiä.

– Jonathan P. Bowen, Michael G. Hinchey: *Seven More Myths of Formal Methods*, IEEE Software, vol 12, no 4, July 1995.

Kymmenen käskyä

1. Valitse tilanteeseen sopiva notaatio.
2. Formalisoi mutta älä yliformalisoi.
3. Arvioi kulut.
4. Hanki formaalien menetelmien guru saapuville.
5. Älä hylkää perinteisiä menetelmiäsi.
6. Dokumentoi tarpeeksi.
7. Älä luovu laatuvaatimuksistasi.
8. Älä ole dogmaattinen.
9. Testaa, testaa ja vielä kerran testaa.
10. Käytä uudelleen.

– Jonathan P. Bowen, Michael G. Hinchey: *Ten Commandments of Formal Methods*, Computer, vol 28, no 4, April 1995.

While-ohjelmat 1

- Unohdetaan (toistaiseksi) luokat, oliot, aliohjelmat jne
- While-ohjelmassa voi olla:
  - sijoituslauseita: $x \leftarrow 4$
  - if-lauseita: **if** $x > 4$ **then** $\cdots$ **else** $\cdots$ **fi**
  - while-lauseita: **while** $x > 4$ **do** $\cdots$ **od**
  - väitelauseita: $\{\, x > 0 \,\}$
- On vain yksi tyyppi: kokonaisluvut (ei ylivuotoa)
- Lausekkeissa saa käyttää normaaleja yhteen-, vähennys- ja kertolaskutoimituksia
- Jakolasku on $\lceil a/b \rceil$ (pyöristetään ylöspäin lähimpään kokonaislukuun) ja $\lfloor a/b \rfloor$ (pyöristetään alaspäin lähimpään kokonaislukuun)
- Jakojäännös a **mod** b

While-ohjelmat 2

- Perusehtolausekkeet: $a = b$, $a < b$ jne
- Ja-ehtolauseke: $p \wedge q$
- Tai-ehtolauseke: $p \vee q$
- Ei-ehtolauseke: $\neg p$
- Ehtolausekkeet eivät ole lausekkeita eikä niillä ole siten tyyppiä. Ehtolauseketta voidaan käyttää vain if-lauseen ja while-lauseen ehtona sekä väitelauseessa.
- Väitelauseen ehtolausekkeen tulee olla tosi aina suorituksen osuessa väitelauseen kohdalle.
- Jos näin ei ole, ohjelman suoritus ei ole sallittu.

Eukleideen algoritmi

$\{ a \geq 0 \wedge b \geq 0 \}$

$syt \leftarrow a$

$b' \leftarrow b$

while $b' \neq 0$ **do**

$t \leftarrow b'$

$b' \leftarrow syt$ **mod** b'

$syt \leftarrow t$

od

$\{ a$ **mod** $syt = 0 \wedge b$ **mod** $syt = 0 \}$

Hoaren tripla

{ esiehto }
koodi
{ jälkiehto }

- Esiehto on väite, joka sijaitsee koodin alussa.
- Esiehto ilmaisee, millaisin alkuarvoin ohjelman saa käynnistää.
- Jälkiehto on väite, joka sijaitsee koodin lopussa.
- Jälkiehto kertoo jotain ohjelman loppuarvoista *olettaen*, että esiehto pätee.

Väittämä

- engl. *assertion*
- suoritettava ohjelmalause
- tarkistaa, että ohjelman tila on ainakin osittain kunnossa
- sisältää testilausekkeen
  - aina tosi
  - jos epätosi, ohjelmassa on bugi
- käytä ohjelman sisäisen koherenssin tarkastamiseen
- Syötteen oikeellisuuden tarkistaminen väitteillä on niiden väärinkäyttöä.
- "It's not that I don't trust you but. . . I don't trust you"

Väittämät

- käännetään tuotantokoodissa tavallisesti pois päältä
- Hoare 1970-luvulla: väittämien kääntäminen pois päältä tuotantokäytössä muistuttaa pelastusliivien jättämistä maihin merille mentäessä.
- Hoare 2000-luvulla: väittämät ovat testioraakkeleita, jotka määrittelevät, milloin ohjelma (ainakin) toimii väärin.
- ensisijaisesti suoritettavaa dokumentaatiota
- toissijaisesti testivälineitä
- pieneltä osaltaan ohjelmien analyysin välineitä

Väittämät ohjelmointikielissä

- C:ssä ja C++:ssa `assert`-makro
- Javassa `assert`-lause
- tavallisesti väittämän laukeaminen kaataa ohjelman
  - usein parempi kaataa viallinen ohjelma kuin antaa sen jatkaa
- Nämä eivät riitä väittämien tehokäyttäjille.
- Harva ohjelmointikieli tarjoaa kunnollisen väittämätuen. Sellainen pitää siis rakentaa itse kirjastoina.
- Microsoftilla Hoaren mukaan:

```
SIMPLIFYING_ASSUMPTION
(strlen(input) < MAX_PATH, 'not yet checking for overflow')
COMPILE_TIME_CHECK (sizeof(x)==sizeof(y),
          'addition is undefined for arrays of different sizes')
switch (condition) { case 0: ... case 1: ...
default: UNREACHABLE('condition is really a boolean'); }
```

Väittämämakro C:ssä

Esimerkiksi C:ssä voi määritellä makroja eri väittämäkategorioille:

```
#define SIMPLIFYING_ASSUMPTION(test) \
  do { \
    if (!test) { \
      fprintf(stderr, "%s:%d(%s): Simplifying assertion %s failed.",\
              __FILE__, __LINE__, __func__, #test);\
      exit(EXIT_FAILURE);\
    }\
  } while(0)
```

Nyt sitten voidaan kehitystyön aikana sanoa `SIMPLIFYING_ASSUMPTION(n < 50)`, kun halutaan toteuttaa vain osa toiminnallisuudesta. Kehitystyön lopuksi kannattaa varmistaa koneellisella etsinnällä, että tuota makroa ei enää missään käytetä.

Sopimusohjelmointi

- engl. *programming by contract*
- väittämien sovellus
- Ohjelma koostuu luokista, joiden ajonaikaisia ilmentymiä sanotaan olioiksi.
- Kunkin luokan kullakin metodilla on kaksi väittämää: *esiehto* (*precondition*) ja *jälkiehto* (*postcondition*).
- Lisäksi kullakin luokalla on yksi luokkakohtainen väittämä, *luokkainvariantti* (*class invariant*).
- Esi- ja jälkiehdot muodostavat luokan ja sen olioiden käyttäjien (metodien kutsujat) välille *sopimuksen*.
  - luo oikeuksia ja velvollisuuksia
  - osapuolina luokan toteuttaja ja luokan käyttäjät
  - luo *luottamussuhteen* luokan ja sen käyttäjien välille.

Esiehto

- metodin kutsujalle velvollisuus ja metodille itselleen oikeus
- Kutsujan on huolehdittava, että esiehto pätee aina kun se kutsuu metodia.
- Metodi voi luottaa siihen, että metodiin tultaessa esiehto pätee
  - metodissa ei iffitellä esiehtoa!
- Jos esiehto ei päde metodiin tultaesssa, se on kutsujan bugi.
  - ilmenee esiehtoväittämän laukeamisena, jos väittämät ovat päällä
  - Metodilla ei ole mitään velvollisuuksia.
- Jos metodia voi kutsua epäluotettava taho (esimerkiksi RMI:n tms. avulla), esiehdon tulee olla triviaali (aina tosi) ja lähtötilan oikeellisuus tulee tarkistaa iffittelemällä.

Jälkiehto

- metodin (ehdollinen) velvollisuus ja kutsujan (ehdollinen) oikeus
  - Ehto on kummassakin tapauksessa se, että esiehto päti metodiin tultaessa.
- Metodin on huolehdittava siitä, että jälkiehto on voimassa, kun metodin suoritus päättyy.
  - Metodi vapautuu tästä vastuusta, jos esiehto ei päde metodiin tultaessa.
- Kutsuja voi luottaa siihen, että metodin palatessa jälkiehto pätee, jos esiehto päti metodiin tultaessa.
- Jos jälkiehto ei päde metodista poistuttaessa, vaikka esiehto päti siihen tultaessa, se on metodin bugi.
  - ilmenee jälkiehtoväittämän laukeamisena, jos väittämät ovat päällä.

Luokkainvariantti

- ehto, joka karakterisoi luokan olioiden *sallitut tilat*.
- Luokan muodostimen tehtävänä on saattaa luokkainvariantti voimaan.
- Kukin metodi voi olettaa, että suorituksen alkaessa luokkainvariantti on voimassa.
- Vastaavasti jokaisen metodin tulee huolehtia, että invariantti on voimassa suorituksen päättyessä.
- Onko luokkainvariantti osa esi- ja jälkiehtoa?
  - Luokan kirjoittajan kannalta on.
  - Luokan käyttäjän kannalta ei ole.
  - Luokkainvariantti on luokan sisäinen asia!
  - Jos joku ulkopuolinen pääsee käsiksi luokan attribuutteihin, tällä on samat velvollisuudet kuin luokan sisäisillä otuksilla!

Ohjelman oikeellisuustodistuksesta

- Sopimusohjelmointi on parhaimmillaan formaaliuden tasolla 2.
- Väittämät ovat dynaamisia tarkistuksia silloin, kun väittämät ovat kytkettynä.
- Periaatteessa esi- ja jälkiehtojen sekä invarianttien voimassaolo voidaan tarkistaa staattisella analyysillä. Tällöin ohjelma *todistetaan oikeelliseksi* esi- ja jälkiehtojen ja invarianttien suhteen.
- Perusidea:
  - Käännetään ohjelmakoodi loogisiksi kaavoiksi, jotka ovat tosia jos ja vain jos koodi on oikeellista.
  - Todistetaan ko. kaavat teoreemoiksi.
- Näin saadaan aikaan tason 3 formaali menetelmä.

Formaali spesifiointi

- Myöhään löydetyt viat ovat iso ongelma.
- Mitä myöhemmin vika löytyy, sitä kalliimpaa se on korjata.
- Pahimmat johtuvat puutteellisesta vaatimusmäärittelystä.
- Eräs formaalien menetelmien merkittävä sovelluskohde on *formaali spesifiointi*.
  - Ideana on yrittää kirjoittaa kerättyjen vaatimusten perusteella formaali malli rakennettavasta systeemistä.
  - Johtaa vaatimusten täsmälliseen kuvaamiseen aikaisessa vaiheessa.
  - Vaatimusten epäselvyydet ja puutteet tulevat havaituiksi, kun kunnollista formaalia speksiä ei synny.
  - Formaali speksi on nopeampi tehdä ja myös helpompi lukea kuin vastaava ohjelmakoodi, koska se voi olla abstrakti.

Speksin analysointi

- Formaalin speksin etu verrattuna perinteiseen analyysi- tai suunnitteludokumenttiin on sen käyttämä kieli.
- Formaali speksi voidaan syöttää ohjelmalle, joka analysoi sitä etsien heikkouksia ja ristiriitaisuuksia.
- Formaali speksi voidaan jopa *animoida*, jonka tuloksena syntyy karkea ajettava malli rakennettavasta systeemistä, jonka toimintaa voidaan kokeilla ja testata.
- Formaalia speksiä voidaan analysoida matemaattisesti johtaen siitä ominaisuuksia, jotka speksin kuvaamilla järjestelmillä välttämättä on mutta joita ei speksistä voi suoraan lukea.
- Lyhyesti sanoen speksi auttaa suunnitelman validoinnissa jo projektin alkuvaiheissa.

Speksi ja koodi

- Speksi toimii mainiona vertailukohtana toteutusvaiheen tuotoksille.
- Speksi on erinomainen testioraakkeli.
- Kriittisten komponenttien tapauksessa koodin voidaan jopa todistaa täyttävän sille speksissä kuvatut vaatimukset.

Sopimussuunnittelu

- engl. *design by contract*
- Otetaan pohjaksi sopimusohjelmointi.
- Sallitaan luokkien perintä ja abstraktit luokat.
- Idea: järjestelmän suunnitteleminen määrittelemällä abstrakteja luokkia sopimuksineen.
- Myöhemmin suunnitelmaa voidaan tarkentaa perimällä uusia luokkia olemassaolevista.
- Lopulta jopa itse ohjelma voidaan saada aikaan perimällä toteutusluokat suunnitteluluokista.
- Perintä mallittaa *sopimusten tarkentamista*
  - aliluokan metodi voi väljentää esiehtoa (”vaatia vähemmän”) ja tiukentaa jälkiehtoa (”luvata enemmän”)
  - toisin päin ei onnistu
- Näin speksi voidaan kirjoittaa osaksi itse ohjelmaa!

Esimerkki

Pihvi I: Esiehto

- osa toimivan yksikön dokumentaatiota
  - käyttötapauksissa...
  - metodeissa niin suunnittelu- kuin toteutusvaiheessa...
- looginen kaava
- Yksikköä saa käyttää, jos esiehto on tosi.
- Jos yksikköä käyttää, kun esiehto on epätosi...
  - kaboom!
  - yksikkö vapautuu kaikista vastuista ja velvollisuuksista
- esimerkki ISO/IEC 14882:1998(E) 21.3.4 (PDF:n sivu 417)

Pihvi II: (Tila-)invariantti

- osa tilallisen yksikön dokumentaatiota
  - olioissa...
  - luokissa...
  - ...
- looginen kaava
- Yksikkö on rikki, jos invariantti ei ole voimassa.

Pihvi III: Sopimus

- Sopimuksessa on kaksi osapuolta.
  - palvelun tarjoaja
  - palvelun käyttäjä
- Sopimus antaa osapuolille oikeuksia ja velvollisuuksia.
- Sopimus on *rajapinta*.
- Rajapinta on sopimus.

Kysymyksiä?