Uudelleenkäyttö ja komponentit

Jonne Itkonen

17. marraskuuta 2003

Sisältö

1 Yleistä

Uudelleenkäyttö on ollut ohjelmistotekniikan päämäärä jo hamoista alkuajoista lähtien. Kaikki, mitä tällä alalla tehdään tähtää uudelleenkäytettävyyteen. Ohjelmakoodi pitää kirjoittaa niin, ettei sieltä löydy toistoa, vaan uudelleenkäytetään ohjelman osia. Vanhoja ohjelmia ei heitetä menemään, vaan ne uudelleenkäytetään, uudistetaan, uusiksi ohjelmiksi. Uusia ohjelmia ei saa (saisi?) kirjoittaa kertakäyttöisiksi, vaan aina uudelleenkäytettäviksi.

Dokumentaatio tulee tehdä uudelleenkäytettäväksi, tästä hyvänä esimerkkinnä vertailkaa Nokian puhelinten 3650 ja N-Gage käyttöohjeita toisiinsa. Ohjelmistoprosessit tulee dokumentoida niin, että prosessit ovat uudelleenkäytettäviä. Päälle heitetään vielä suunnittelumallit, dokumentaatiot tiettyihin ongelmiin hyviksi havaituista ratkaisu, ja soppa on valmis.

Komponentit ovat jo määritelmänsä mukaan uudelleenkäytettäviä. Ne ovat mielekkäitä ohjelmayksiköitä, joita yhdistelemällä voi kasata koko ohjelmiston. Komponentin voi myös vaihtaa toiseen toiminnallisuuden kärsimättä. Komponentteja saa kaupanhyllyiltä, joko valmiiksi sopivina malleina tai asiakkaan tarpeisiin muokattavina.

Seuraavassa on käyty hieman läpi uudelleenkäyttöä yleisesti, sen termistöä, toteutustapoja, saavutuksia ja ongelmia. Sitten esitellään ohjelmistokomponentteja, niiden toteuttamista, saavutuksia ja ongelmia. Mikään kattava esitys tämä ei ole, kehoitan tutustumaan lähdekirjallisuuteen ja tulevaisuudessa ilmeisesti järjestettävään komponenttiohjelmoinnin kurssiin.

2 Uudelleenkäyttö

Uudelleenkäyttö on jo tuotetun ohjelmiston osan hyödyntämistä uuden osan tuottamisessa tai vanhan parantamisessa. Uudelleenkäyttöä voidaan pitää kaiken ohjelmistoteknisen toiminnan yhtenä peruspiirteenä. Uudelleenkäyttö voidaan ulottaa kaikkiin ohjelmiston osiin, mutta seuraavassa keskitytään lähinnä ohjelmiston suunnittelun ja toteutuksen uudelleenkäytettävyyksiin.

2.1 Ohjelmakoodin uudelleenkäyttö

Ohjelmakoodi on kenties tutkituin uudelleenkäytön alue. Tämän huomaa jo katsottaessa ohjelmointitekniikoiden kehittymistä. Ensin tietokoneita ohjelmoitiin konekielellä ja myöhemmin assembler-kielillä, jolloin ohjelmointi oli hyvin matalan tason ohjelmointia, kaikki tehtiin itse, silmukat, tietorakenteet, toiminnot, usein jopa raa'asti vain kopioimalla ohjelmakoodia vanhasta paikasta uuteen.

Pian kuitenkin huomattiin joidenkin rakenteiden toistuvan useasti. Näistä rakenteista kasvoi ensin kontrollirakenteita, ehto- ja toistolauseita, myöhemmin tietorakenteita ja algoritmeja. Ehto- ja toistorakenteiden kirjoittamista helpottivat huomattavasti uudet, assembler-kieliä korkeamman tason kielet. Kun ehto- ja toistorakenteet alkoivat toistumaan, syntyivät aliohjelman ja funktion käsitteet.

Kehitystä voidaan seurata eteenpäin samasta näkökulmasta helposti. Kun aliohjelmat ja funktiot huomattiin ryhmittyviksi samankaltaisiin kokonaisuuksiin, esimerkiksi toistoaliohjelmat, matemaattiset funktiot, syntyivät aliohjelmakirjastot. Kun tietynlaisen tiedon havaittiin sopivan yhteen tietynlaisten aliohjelmien kanssa, ja kun havaittiin tarve tiedon abstrahoinnille, syntyivät moduulit ja abstraktit tietotyypit. Nämä edelleen vauhdittivat olioiden muokkautumista nykyiseen vallallaolevaan luokkaperustaiseen oliotyyliin. Olioiden käyttösopimukset, yhdistettynä modulaarisuuden tiukkoihin rajoihin taas synnyttivät komponenttiohjelmoinnin.

Tällä hetkellä kiistellään olioiden kaikkivoipaisuudesta. Esille on tullut olion moninaamaisuus, yhdeltä suunnalta olio hoitaa yhdet toiminnalliset velvoitteet, toiselta toiset velvoitteet, muilta muita, tai laadullisia velvoitteita. On syntynyt aspektin käsite, ja myös (sanahirviö) *multidimensional separation of concerns*. Olio ei olekaan enää yksiulotteinen palikka, vaan moniulotteisen verkon solmu.

Kaikissa näissä vaiheissa on eteenpäinvievänä voimana ollut aina uudelleenkäyttö. Aiemmassa mallissa on huomattu toistoa, joka on haluttu irroittaa ja nostaa työkaluksi myöhemmälle mallille.

2.2 Analyysi- ja suunnitelmavaiheen uudelleenkäyttökohteita

Jo koko ohjelmistoprosessi on sinänsä uudelleenkäytön ilmentymä. On havaittu, että tietynkaltaisten ohjelmistojen parissa tietynlaisessa ympäristössä työskennellessä toimii tietty prosessi, toisissa oloissa toinen. Jos nyt uuden projektin ympäristö vastaa aiempaa, kannattaa ainakin kokeilla entistä, hyväksi havaittua prosessimallia.

Jos tarkemmin katsotaan prosessia vanhasta rakkaasta vesiputousnäkymästä, löytyy esitutkinta-, analyysi-, suunnittelu- ja testausvaiheesta paljon uudelleenkäytön kohteita. Jos mukaan otetaan vielä ohjelmiston ylläpito, no, sehän on vain uudelleenkäyttettävyyden soveltamista ja parantamista.

Vaatimuksia voidaan uudelleenkäyttää, varsinkin laadullisia vaatimuksia. Toiminnalliset vaatimukset toimivat lähtökohtana ohjelmiston rakenteelle, mutta laadullisten vaatimusten avulla rakenteen määrittely helpottuu huomattavasti. Käyttötapauksista osa on luonteeltaan hyvin yleislaatuisia, näiden avulla voidaan usein rakentaa tarkkojakin uudelleenkäytettäviä oliomalleja.

Lähemmäs toteutuspäätä mentäessä vastaan tulevat suunnittelumallit, jotka ovat jo määritelmän mukaan uudelleenkäytettäviä suunnitteluratkaisuja tiettyyn ongelmatilanteeseen, tietyin seurauksin. Varsinaisesti toteutuspään uudelleenkäytön mekanismeja ovat funktiot, kirjastot, moduulit, oliot, komponentit ja kehykset.

Testauksen yksikkötestit ovat uudelleenkäytettäviä, ainakin projektin sisällä, mutta auttavat myös vastaavanlaisten projektien testauksen suunnittelussa ja toteutuksessa.

Sommerville jakaa uudelleenkäytön yksiköt kolmeen kategoriaan koon perusteella:

1. *Sovellusjärjestelmän uudelleenkäyttö* Sovelluksen käyttö järjestelmän osana ilman muutoksia, COTS, tai asiakkaan käytön mukaan sovitetut sovellusperheet.
2. *Komponentin uudelleenkäyttö* Tästä lisää kappaleessa 3.
3. *Toiminnan uudelleenkäyttö* Yksikön muodostaa yksittäinen funktio.

Oliot ovat olleet viime aikoina arvostelun kohteena muun muassa niiden uudelleenkäyttöön odotettua heikomman sopivuuden vuoksi. Vaikka nämä trendikritisoijat eivät useinkaan ole ymmärtäneet olio-ohjelmoinnin ajatusta, on väitteessä silti hieman perää. Oliot, päinvastoin kuin niin monesti sanotaan, ovat parhaimmillaan *ongelma-avaruuden* olioiden kuvaamisessa, eivätkä reaalimaailman olioiden kuvaamisessa, sillä reaalimaailman oliot ovat aivan liian moninaamaisia mallinnettaviksi. Yhden sovellusalueen sisällä oliot taas useinkin ovat hyvin uudelleenkäytettäviä, parhaimpana esimerkkinä kenties liiketoimintaoliot.

Samaan aikaan, kun oliot on väärinymmärretty, on muotoutunut komponentin käsite, joka on hyvin voimakkaasti juuri painottanut sovellusalueriippuvuuteen. Alusta lähtien on puhuttu esimerkiksi käyttöliittymä- ja tietovarastokomponenteista. Mitä ovat nämä komponentit, ja ovatko ne vihdoin se hopealuoti, mitä oliot(kaan) eivät olleet?

3 Komponentit

Alan perusteos on Clemens Szyperskin *Component Oriented Programming* [3].

Komponenttiajattelu juurtaa vanhoista insinöörialoista, joissa jo pitkään uusia kojeita on voitu rakentaa alkeellisempia komponentteja käyttäen. Esimerkkiinä auto, jossa on moottori, jossa on mäntiä, venttiilejä, laakereita... Vastaavanlainen koostamismenetelmä on haluttu saada aikaiseksi myös ohjelmistotekniikkaan[1].

Ohjelmistokomponentti on pieni, järkevä, toiminnallinen ja uudelleenkäytettävä ohjelmistoyksikkö. Vaikka periaatteessa mitä tahansa uudelleenkäytettävää yksikköä voitaisiin sanoa komponentiksi, on ohjelmistokomponentti sellainen ohjelman osa, joka on itsenäinen ja jolla on konsepti, kontentti ja konteksti — merkitys, toteutus ja sovellusalue. Komponentteihin yhdistetään myös monesti ajatus *"buy, don't build"*, liekö Fred L. Brooksin alkuperää.

3.1 Komponenttiko olioko komponentti?

Komponentteihin olemme jo törmänneet monesti. Delphistä ja Javasta ovat tuttuja graafisen käyttöliit-

1. Software *engineering*.

tymän komponentit, jälkimmäisestä myös Java pavut (*JavaBeans*), tavalliset sekä liiketoiminnalliset. Usein kuulee myös vertailtavan olioita ja komponentteja keskenään.

Olio ei kuitenkaan ole komponentti, eikä komponentti olio, vaikka molempia voi toisillaan toteuttaa. Tämä ero on selkeä, kun kuuntelee joidenkin komponenttipuristien mielipidettä siitä, ettei komponentilla tulisi koskaan olla pysyvää toiminnallista tilaa. Kaikki ohjelman komponenteissa käsiteltävä tieto viedään komponenteille ja saadaan niiltä. Komponenteilla kuitenkin voi olla ominaisuuksia, jotka muistuttavat hyvin paljon olioiden atribuutteja, mutta säätävät vain komponentin toimintaa, mukauttavat komponenttia. Rehellisyyden nimissä mainittakoon, että hybridimalli tilallisesta komponentista taitaa olla silti yleisin. Ihmiset eivät tunnu oikein arvostavan puhtaita paradigmoja.

Kuten oliot, komponentit toimivat toistensa kanssa tarkasti määriteltyjen rajapintojen kautta. Nämä rajapinnat toisaalta määrittävät tarkasti komponentin tarjoaman toiminnallisuuden, toisaalta piilottavat ulkopuolisilta komponentin toteutusyksityiskohdat. Komponentilla on yleensä useita rajapintoja, jotka hyvän tavan mukaan ovat tarkasti määriteltyjä toiminta-alueen mukaan, esimerkiksi rajapinta tiedolle, rajapinta liiketoiminnoille ja rajapinta järjestelmätoiminnoille.

Oliot ovat luokan instansseja. Samoin komponentista voi olla useita eri instansseja toiminnassa yhtäaikaa samassa järjestelmässä. Olioista poiketen komponenttia ei kuitenkaan tarvitse välttämättä identifioida, joskin järjestelmän toteuttamisen kannalta on helpompaa yksilöidä komponentit identiteetillä.

3.2 Komponenttijärjestelmä

Komponentin erot olioihin kasvavat näkyviksi, kun tarkastelemme komponenttijärjestelmän toteutusta. Vaikka komponentit ovatkin itsenäisiä toimijoita, tulee niiden jotenkin pystyä keskustelemaan muun ohjelmiston, ja muiden komponenttien, kanssa esimerkiksi ohjelmiston elinkaareen liittyvistä asioista. Ohjelmiston tai järjestelmän tulee myös pystyä viestittämään komponenteille mahdollisista toimintaympäristön tapahtumista. Tämä tapahtuu *komponenttisäilön* välityksellä. Säilö on kuin liitin, johon komponentti liitetään, ja jonka läpi kaikki kommunikaatio komponentin ja järjestelmän välillä kulkee hyvin määritellyn järjestelmärajapinnan lävitse.

Komponenttien rajapinnat ja liittäminen järjestelmään kuvataan komponentin jakelupaketissa. Tämän sisällä voi olla useita eri komponentteja, jotka sopivat erilaisiin järjestelmäkonfiguraatioihin, ja sääntöjä, milloin mikäkin komponentti käyttöön valitaan. Esimerkiksi saman jakelupaketin sisällä voi olla komponentti niin Intel- kun PowerPC-arkkitehtuurillekin. Esimerkkinä jakelupaketeista voidaan mainita Javasta tutut jar-tiedostot.

Komponentti voi keskustella toisten komponenttien kanssa myös muita kanavia kuin säilönsä tarjoamia pitkin. Yleensä tällöin puhutaan välikerroksesta, joka pelkän viestinnän lisäksi tarjoaa muitakin komponenteille tarpeellisia palveluja, esimerkiksi palvelun toisten komponenttien etsintään haluttujen ominaisuuksien perusteella.

Komponentin elinkaari on seuraava: asennusohjelma luo komponentille säilön ja liittää komponentin tähän. Säilö alustaa komponentin halutuin parametrein, ja käynnistää tämän. Komponentti toimii, kunnes sen toiminta lakkaa joko sen oman toiminnon seurauksena tai säilön toimesta järjestelmän pyynnöstä. Tällöin komponentin parametrien mahdollisesti muuttuneet arvot ja mahdollinen komponentin tila tallennetaan pysyväismuistiin ja komponentti sammutetaan. Tarvittaessa komponentti voidaan irroittaa ja korvata uudella, johon tallennetut parametrit ja tila voidaan siirtää sellaisenaan, jatkaen näin komponentin toimintaa siitä mihin se jäi ennen komponentin vaihtoa.

Toteutetuista komponenttijärjestelmistä mainittakoon OMG:n CORBA Component Model, Javan JavaBeans ja Enterprise JavaBeans, sekä Microsoftin DCOM, mikä sen nimi nykyään lieneekin.

3.3 Mistä komponentteja saa?

Komponenttien kannattajat unelmoivat ajasta, jolloin komponentteja voi ostaa ohjelmiinsa suoraan kaupan hyllyltä. Monessa mielessä tämä haave onkin jo toteutunut, eihän siinä tunnu olevan mitään pahaa. Ei kuitenkaan koskaan tule aliarvoida yhtä ohjelmistotekniikan perusvoimaa *"not invented here"*. Oli tarjolla kuinka hienoja valmiita komponentteja tahansa, tahtovat teknisemmin suuntautuneet ohjelmistoprojektin jäsenet usein silti tehdä komponenttinsa itse. Mikäs siinä, jos ne ovat hyviä, voivat liiketoiminnallisesti suuntautuneet jäsenet tehdä niistä bisneksen myöhemmin. Jälkimmäisen ihmisryhmän parissa vain tuntuu olevan voimakkaana perusvoima *"must not be invented here"*, tai kuten aiemmin esitettiin *"buy, don't build"*.

Molemmat puolet on helposti perusteltavissa. Itsetehty tunnetaan läpikotaisin, ja sitä voidaan myydä jälkeenpäin. Ostettu on käytössä nyt heti, ja ongelmien korjaamiseen tai jatkokehitykseen ei tarvitse käyttää omia resursseja. Valittaessa toimintatapaa kannattaa kuitenkin pitää mielessä komponenttipe-

rustaisen ohjelmistokehityksen hyvät ja huonot puolet, kuten Sommerville ne kirjassaan [2] erittelee:

- Hyvää:
  - Luotettavuus paranee uudelleenkäytön seurauksena.
  - Olemassaolevan komponentin käyttö on riskittömämpää.
  - Asiantuntijoiden taidot keskitetysti.
  - Standardienmukaisuus.
  - Nopeampi kehitysvauhti.
- Huonoa:
  - Ylläpitokustannusten kasvu, jos esimerkiksi komponentin toimittaja häviää, eikä lähdekoodia ole saatavilla.
  - Työkalutuen puute. Käytetyn työkalun tukema prosessi ei tue komponenttipohjaista ohjelmistokehitystä.
  - Ei meillä keksitty -syndrooma.
  - Komponenttikirjaston ylläpito.
  - Komponenttien löytäminen ja mukauttaminen.

3.4 Komponentit ja uudelleenkäyttö

Tämä on muokattu ote Ohjelmistoarkkitehtuurit-kurssin luentomonisteesta.

Ohjelmistojen uudelleenkäyttö komponentteina ei ole mikään kovin uusi keksintö, sillä tätä ehdotti McIlroy jo vuonna 1969. Tätä ohjelmistotekniikan pyhää päämäärää on sen jälkeen pidetty hyvänä vaihtoehtona edelläalueteltujen ongelmien korjaamiseen. Boschin [1] mukaan tässä tavoitteessa onnistuneita komponentteja ovat käyttöjärjestelmät, tietokannanhallintajärjestelmät, kääntäjät, graafiset käyttöliittymät, komponenttien välisen vuorovaikutuksen standardien toteutukset, sekä uusimpana verkkopalvelimet ja -selaimet.

Lista komponenteista saattaa näyttää oudolta, mutta komponenttien elinkaaren pohdinta auttaa ymmärtämään sen. Alussa komponentti on ollut kiinteä osa ohjelmaa. Seuraavassa vaiheessa on huomattu sen eriävyys muusta ohjelmasta. Komponentti on eriytetty omaksi osakseen, ja sen yleistämistä on tutkittu. Siitä on tehty markkinoitava kokonaisuus, joka sitten lopulta suuren yhtiön toimesta on integroitu osaksi järjestelmää. Monet muistavat tämän parhaiten graafisen käyttöliittymän, www-selaimen ja Microsoftin kohdalta.

Josko uudelleenkäytettävyys tavallaan näiden komponenttien kohdalla onkin onnistunut, sovellusalueen komponenttien kohdalla se ei ole. Komponentti on yhä muokattava sovellusalueelleen, tai sitten se on niin yleinen, että se kasvaa liian massiiviseksi ja hankalaksi käyttää. Yksinkertaisia, uudelleenkäytettäviä komponentteja löytyy vain pienistä yksinkertaisista sovelluksista.

Vaihtoehtona ohjelmien uudelleenkäytölle Bosch tarjoaa ohjelmistoarkkitehtuureja ja ohjelmistotuotelinjoja. Arkkitehtuurit vaikuttavat positiivisesti laatuun, tuotelinjat laadun lisäksi kehitys- ja ylläpitokuluihin sekä ohjelman valmistumisaikaan.

4 Mitä jäi jäljelle?

Jos kerta komponentit ovat huonoja uudelleenkäytön välineitä, samoin kuin oliot, ja uudelleenkäyttö on jo perusteiltaankin mahdotonta, niin mitä tästä sitten jäi jäljelle?

Vaikka teilaan yllä uudelleenkäytön ja komponentit, ei se tarkoita sitä, etteivätkö ne joissain olosuhteissa, tietyillä ehdoilla, toimisi vallan hienosti ja jopa kehitystä nopeuttaen. Yleiskäyttöisiksi hopealuodeiksi niistä ei kuitenkaan ole. Ehkä sellainen keksitään, ehkä ei, mutta sitä odotellessa on parasta opetella monien eri työkalujen käyttö, jotta osaa oikeassa tilanteessa käyttää oikeita työkaluja. Komponentit ovat yksi työkalu, oliot toinen. Parasta uudelleenkäyttöä onkin siten työkalujen uudelleenkäyttö, ei työstettyjen artefaktien.

Viitteet

[1] Jan Bosch: *Design and use of software architectures: adopting and evolving a product-line approach*, ACM Press/Addison-Wesley Publishing Co., 2000, ISBN 0-201-67494-7.

[2] Ian Sommerville: *Software engineering (6th ed.)*, Addison-Wesley Longman Publishing Co., Inc., 2001, ISBN 0-201-39815-X.

[3] Clemens Szyperski: *Component Oriented Programming*, 1. painos, Addison-Wesley, 1997, ISBN 0201178885, the book is expected to be out in November 97.
URL `http://cseng.aw.com/bookdetail.qry?ISBN=0-201-17888-5&p%type=0`