Automaattisen dokumentaation apuvälineet

Saku Hujala (sphujala@cc.jyu.fi),
Ville Saarinen (vilsaar@cc.jyu.fi),
Juhani Honkala (juhnu@st.jyu.fi),
Miika Nurminen (minurmin@st.jyu.fi)

16. marraskuuta 2001

Sisältö

1 Johdanto

Dokumentoinnin onnistuminen on yksi ohjelmistokehityksen kulmakivistä. Nykyisissä menetelmissä suunnitelma ja lähdekoodi nähdään täysin erillisinä ilmentyminä samasta ohjelmasta. Koodaus- ja erityisesti ylläpitovaiheessa muutoksia tehtäessä vaarana on, että ne eivät pysy yhtenevinä. Ihannetilanne olisi, että koodauksen edistyessä tietoa saataisiin tulkittua koodin pohjalta, jolloin erot dokumentaation ja koodin välillä saadaan minimoitua.

Olemme tutkineet eräitä CASE- ja dokumentointiohjelmia, jotka mahdollisesti tuovat apua tähän ongelmaan. Aiheen ulkopuolelle olemme jättäneet CASE-välineistä ns. piirto-ohjelmat, jotka perustuvat kuvien piirtelyyn vailla sen suurempaa semanttista sisältöä. Käsin tehdystä kuvien päivittämisestä tahdomme juuri päästä eroon. Yhtenä pohdiskelun aiheena voisi olla, kuinka suuren riskin tiettyyn CASE-välineeseen sitoutuminen aiheuttaa projektille. Onko riskinä käytännössä koko tehdyn työn menetys, jos halutaan vaihtaa CASE-välineestä toiseen?

Tutkimme myös hieman teoreettisempaa lähestymistapaa automaattiseen dokumentaation generoimiseen käyttäen NLG(Natural Language Generation)-menetelmää. NLG-menetelmä poikkeaa olennaisesti CASE- ja dokumentointi ohjelmista, sillä NLG-järjestelmät tuottavat dokumentaation tietokannassa sijaitsevasta tietämyksestä. NLG-menetelmä ei ole vielä kovin yleisesti käytössä, mutta asiaa tutkitaan monessa yhteydessä.

2 Automaattisen dokumentoinnin apuvälineet

Olemme jakaneet tutkimamme ohjelmat kolmeen eri luokkaan, jotka kuvaavat pääpiirteissään niiden ominaisuuksia ja käyttötarkoitusta. Jako on tehty

tarkastelemalla dokumentointiprosessin kulkua. Ensimmäiseen ryhmään olemme valinneet ohjelmia, jotka pystyvät generoimaan korkeamman tason dokumentteja lähdekoodin pohjalta (reverse engineering). Toisessa ryhmässä on ohjelmia, joissa prosessia käsitellään perinteisen vesiputousmallin mukaisesti suunnitelmasta toteutukseen (forward engineering). Viimeisenä ryhmänä on joukko ohjelmia, jotka pyrkivät yhdistämään nämä kaksi suuntausta luomalla kaksisuuntaisen yhteyden dokumenttien ja lähdekoodin välille (round-trip engineering).

Olemme tutkineet näiden ohjelmien soveltuvuutta eri käyttötarkoituksiin ja arvioineet niiden toimintaa. Tarkoituksenamme ei ole ollut tehdä täydellistä analyysiä eri ohjelmien ominaisuuksista, vaan muodostaa yleiskäsitys alan ohjelmatarjonnasta. Arviointi perustuu lähinnä ohjelmien kotisivuilta saatuihin dokumentteihin ja demoversioiden testaukseen.

2.1 Käänteistekniikkaohjelmat (*reverse engineering*)

Dokumentointiohjelmat generoivat olemassaolevasta lähdekoodista referenssidokumentaation, yksinkertaisimmillaan listan ohjelmassa käytetyistä luokista, attribuuteista ja metodeista. Hyvänä esimerkkinä tästä on Java API:n dokumentaatio, joka on kokonaisuudessaan tehty lähdekoodin pohjalta Javadoc ohjelman avulla. Lähdekoodin lisäksi dokumentti perustuu siinä olevien kommentteihin. Kommenttien täytyy olla ohjelman vaatimassa muodossa, joka aiheuttaa hieman lisätyötä ohjelmoijalle. Toisaalta hyvin määritelty kommentointitapa parantaa luettavuutta ja selkeyttää ohjelmaa. Varsinaisten kommenttien kirjoitusta varten on kehitetty myös omia apuohjelmia.

Dokumentaation ulkoasuun ja sisällytettäviin tietoihin voi yleensä vaikuttaa. Tämä mahdollistaa eri abstraktiotasolla olevat mutta yhtenevät dokumentit samasta projektista. Yleisimpiä tuettuja formaatteja ovat HTML, RTF,

PDF, LaTeX, PostScript tai käyttäjän määrittelemä XML-murre. Dokumentaatioon voidaan ohjelmasta riippuen määritellä myös ylimääräistä tietoa, kuten kaavioita (esim. ote luokkakaaviosta) tai linkkejä eri asiayhteyksien välille. Dokumentti voidaan kerätä yhteen tiedostoon tai hajauttaa. Dokumentointiohjelmia on saatavissa kaikille yleisimmille kielille (C++, Java, Pascal, C...).

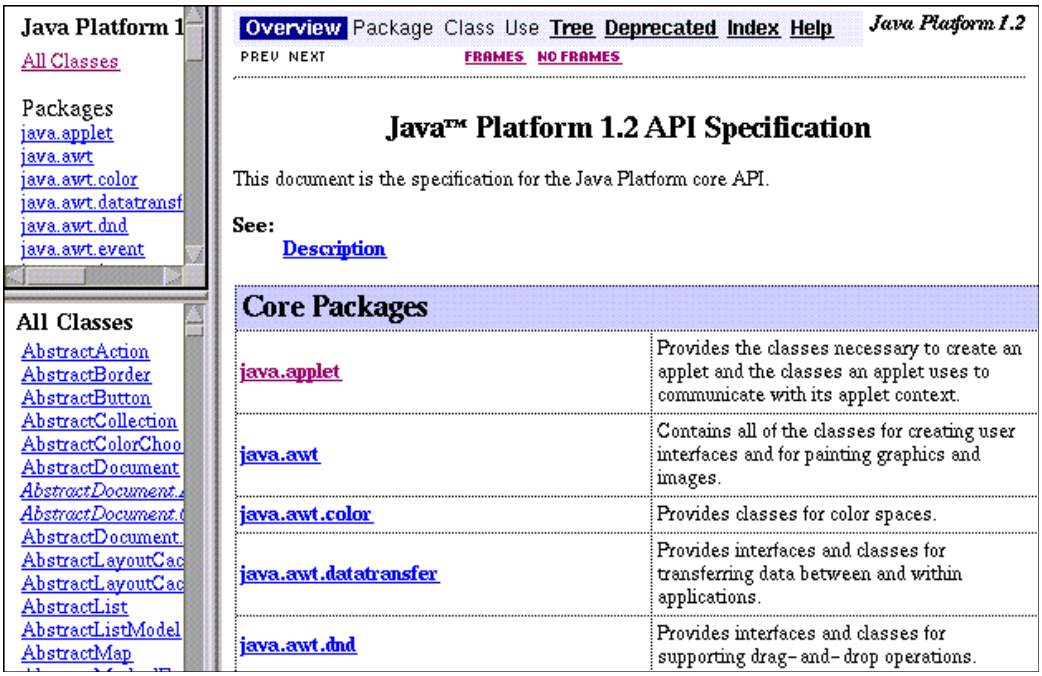
Dokumentointiohjelmien selkein etu on, että referenssidokumentit voidaan helposti pitää ajan tasalla ohjelmaa päivitettäessä sekä toteutus- että ylläpitovaiheessa. Koodin dokumentointitapaan ei vielä ole yleistä standardia, joten tietyn dokumentointiohjelman valitseminen voi sitoa ohjelmakoodin kommentit tiettyyn muotoon. Useimmat ohjelmat ymmärtävät tosin Javadocin käyttämää syntaksia.

Dokumentointiohjelmien ongelmana on generoidun dokumentin 'kertakäyttöisyys'. Koodista generoituun dokumenttiin ei voi tehdä muutoksia, jotka säilyisivät lähdekoodin pohjalta uudelleengeneroidussa dokumentissa. Ohjelmien generoimia dokumentteja ei voi käyttää hyödyksi CASE-ympäristössä, ellei CASE-järjestelmä tue generoitua formaattia. Tosin useimmissa CASE-ympäristöissä on oma dokumentointijärjestelmä.

2.1.1 Javadoc

Sunin kehittämä dokumentointiohjelma Java-kielelle, ehkä yksi tunnetuimpia dokumentointivälineitä. Toimitetaan Javan kehityspaketin mukana. Kaikki Java API:n dokumentaatiot on tehty Javadocin avulla, minkä hyvänä puolena on yhtenäinen ulkoasu ja toimintalogiikka. Javadoc Tuottaa perusasetuksilla HTML-pohjaisen dokumentaation, mutta tarvittaessa Doclettien (eräänlainen Plugin) avulla tuotoksen ulkoasua ja kohdeformaattia voidaan muuttaa. Docletteja on saatavilla valmiina yleisesti käytetyille formaateille ja nii-

tä voi myös tehdä itse. Docletit voivat myös lisätä ominaisuuksia ohjelmaan esim. UML-luokkakaavion generoinnin.[sun]



Kuva 1: Javadoc

Ohjelman käyttö vaatii, että lähdekoodi on dokumentoitu tietyllä tavalla. Javadocin käyttämä syntaksi on laajalti hyväksytty ja käytössä monissa muissa vastaavissa ohjelmissa.

2.1.2 Doxygen

Doxygen on GNU-lisenssin alla julkaistu dokumentointiohjelma, joka on ominaisuuksiltaan hyvin lähellä Javadoc-ohjelmaa. Ohjelma ymmärtää Javan lisäksi myös C/C++ sekä IDL (CORBA, COM, KDE-DCOP) kieliä. Dokumenttien formaatiksi voidaan valita mm. HTML, RTF, Postscript tai PDF. Dokumenttien ulkoasua ja muotoa voi säätää monipuolisesti muuttamalla

asetuksia, mutta Doclettien tapaista tekniikkaa ohjelman laajentamiseen ei ole. Ohjelma tukee Javadocin lisäksi QT-tyylistä kommentointitapaa.

Doxygenille on tehty useita pieniä apuohjelmia, jotka helpottavat asetuksien muokkaamista ja lähdekoodin kommentointia. Ohjelman käyttö on kuitenkin sen verran helppoa, että ilman apuohjelmiakin tulee toimeen.[dox]

2.1.3 Codework CC-rider

CC-riderissa on tekstimuotoisten raporttien luomisen lisäksi mahdollisuus käydä ohjelman rakenteita läpi erillisellä selaimella. Ohjelma käy muiden dokumenttigeneraattorien tapaan lähdekoodin läpi ja koostaa siitä esityksen omaan tietokantaansa. Ohjelmalla generoi kaavoit esim. luokkahierarkiasta ja luokkien metodeista.[cw]

Oma selain mahdollistaa tehokkaan visualisoinnin verrattuna HTML:n tai muiden tekstipohjaisten formaattien ilmaisuvoimaan. Työkalua voi käyttää ohjelmiston kehityksen aikana ja ajaa taustalla koodia muutettaessa, jolloin kaaviot pysyvät ajan tasalla. Ongelmana on integrointi muiden CASE-välineiden kanssa ja kaavioiden epästandardi muoto.

2.2 Koodingeneroijat (*forward engineering*)

Koodingeneroijat on tarkoitettu analyysi- ja suunnitteluvaiheen avuksi. Ne tukevat vesiputoumallin mukaista ohjelmistoprosessia. Ohjelmassa on sisäinen malli kehitettävän sovelluksen rakenteesta. Malli määritellään kaavioilla (nykyisissä ohjelmissa yleensä UML-muotoisia). Kaaviot piirretään ohjelmaan mahdollisesti sisältyvällä piirtotyökalulla, tai ne voidaan tuoda ohjelmaan ulkopuolelta. Kaavioiden siirtoon ohjelmasta toiseen on IBM:n ja Unisysin esittämä XML-pohjainen XMI-standardi (XML Metadata Interchange). Ohjelma generoi mallin pohjalta lähdekoodin rutiininomaiset osat (esim.

luokkamääritykset, saantimetodit, funktioiden prototyypit) pohjaksi toteutukselle.[xmi]

Forward engineering-ohjelmien hyvänä puolena on mahdollisuus mallintaa ohjelman toimintaa sitoutumatta tiettyyn kieleen. Java, C#, C++, C ja Visual Basic ovat yleisimpiä tuettuja kieliä. Ohjelmat toimivat hyvin tilanteessa, jossa tarvetta suunnitelman muuttamiselle koodausvaiheen aikana ei ole. Tällainen tilanne on kuitenkin hyvin harvinainen ja käytännössä suunnitelmaa joudutaan aina muuttamaan toteutusvaiheen aikana. Syitä tähän ovat mm.

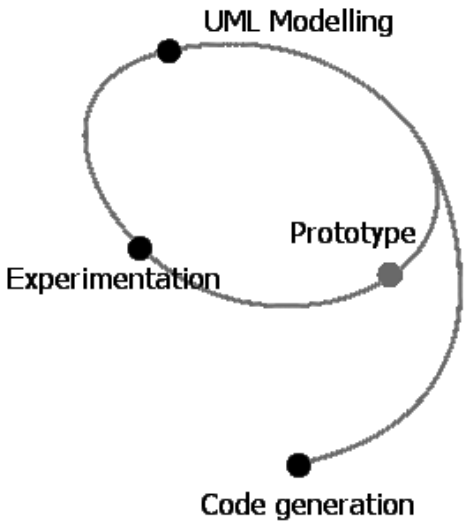
Kaavioiden ilmaisuvoiman puuttellisuus tai sen soveltumattomuus käytetyn ohjelmointikielen kanssa. Esimerkiksi jos suunnittelumenetelmä ei tue vahvasti tyypitetyn kielen rakenteita voi tuloksena olla kyseisillä kielillä vaikeasti toteutettava malli. Vaatimusmäärittely on puutteellinen tai vaatimukset muuttuvat projektin kuluessa. Toteutus aloitetaan liian aikaisessa vaiheessa suunnitelman ollessa vielä kesken. Tekniset rajoitteet voivat johtaa suuriinkin muutoksiin suunnitelmassa.

Täydellisen ja aukottoman suunnitelman tekeminen ennen toteutusvaihetta lienee siis mahdottomuus.

Yksisyyntaisessa prosessissa muutokset pitää aina tehdä ennen toteutusta ylemmän tason kaavioihin, mikä omalta osaltaan vaikeuttaa ja hidastaa toteutusta. Pahimmassa tapauksessa toteutusvaiheessa tarvittavien muutosten tekemisestä luovutaan, koska työmäärä kaavioiden ja dokumenttien muuttamisessa on liian suuri. Tällöin dokumentoinnista voi tulla yksi projektin riskitekijöistä. Ongelmana on myös suunnitelman ja toteutuksen pitäminen yhteinäisenä.[mc]

2.2.1 Delphia*Object Modeler

Tyypillinen koodingenerointiohjelma, sisältää tuen UML-muotoisten kaavioiden tekoon ja lähdekoodin generointiin. Toteutusvaiheessa esiin tulevia ongelmia on yritetty vähentää prototyypityksen tukemisella. Prototyyppi on malli sovelluksen käyttöliittymästä, joka on generoitu ohjelman omalla välikielellä. Tämä helpottaa ohjelman testausta ja käyttölogiikan suunnittelua. Mallia voidaan tarvittaessa muuttaa seuraavaa prototyyppiä varten.[dom]



Kuva 2: Delphia Object Modeler

Delphian kaltaisten ohjelmien ongelmana on, että niillä voidaan toteuttaa ohjelmia vain tiettyyn tarkoitukseen (tässä tapauksessa tietokantapohjaisiin businesssovelluksiin). Ohjelma ei reagoi toteutusvaiheessa mahdollisesti tehtäviin muutoksiin. Prototyypitys voidaan nähdä eräänlaisena 'köyhän miehen' versiona aidosta round-trip menetelmästä.

2.2.2 MetaEdit

MetaEditillä on erilainen lähestymistapa verrattuna useimpiin muihin koodingeneroijiin. Tietyn kaaviostandardin (esim. UML) sijaan Metaeditillä voidaan määritellä käsitteet ja symbolit sovellusalueen mukaan. Käsitteille voidaan määritellä omat koodigeneraattorit. Ohjelma generoi lähdekoodin lisäksi myös tekstipohjaisen dokumentaation.

UML-pohjaiseen mallintamiseen verrattuna menetelmä tarjoaa mahdollisuuden järjestelmän joustavampaan kuvaamiseen. Tämä voi mahdollisesti lisätä tuottavuutta toteutusvaiheessa, mutta saattaa aiheuttaa lisätyötä sovelluksen käsitteitä määriteltäessä. Onnistuneen projektin edellytyksenä on, että käsitteet (ja niiden koodigeneraattorit) ovat hyvin määriteltyjä.

Metaeditin huonona puolena on, että itse määritellyt kaaviomallit eivät ole yhteensopivia standardoitujen kuvauskielten, kuten UML:n kanssa. Ohjelmistoprojekti on siis dokumenttien ja kaavioiden osalta sidottu MetaEditiin.[dom]

2.3 Round-trip -ohjelmat

Tähän kategoriaan kuuluvat ohjelmat sisältävät piirteitä kummastakin aiemmin luetellusta tyypistä. Ohjelmistonkehitys nähdään suunnittelu- ja toteutusvaiheessa kaksisuuntaisena prosessina, eli muutokset koodissa välittyvät dokumentteihin ja toisinpäin. Tämä tukee iteratiivista tai inkrementaalista ohjelmistokehitystä, missä koodia sekä suunnitelmaa kehitetään samanaikaisesti. Yhtenä hyvänä puolena on dokumettien ja lähdekoodin pysyminen aina yhtenäisinä.

Analyysi- ja suunnitteluvaiheessa ohjelmaa voi käyttää koodingeneroijien tapaan rutiininomaisen koodauksen vähentämiseksi. Toteutusvaiheessa tehtävät muutokset voidaan käänteistekniikalla (reverse engineering) päivittää

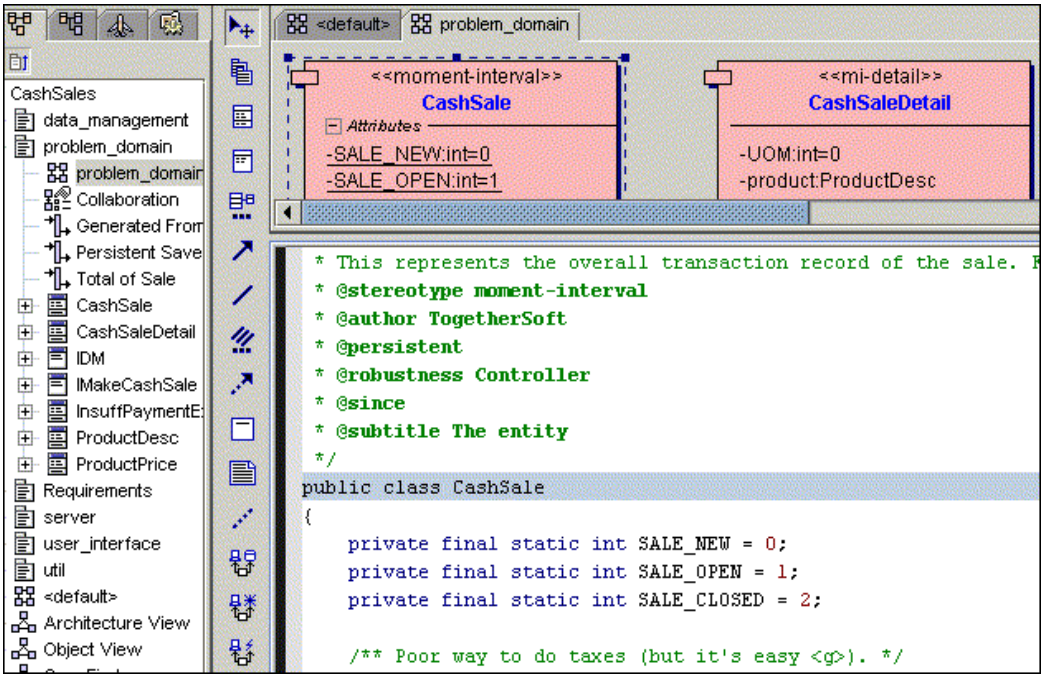
kaavioihin. Round-trip-ohjelma pystyy säilyttämään eri iteraatiokierroksilla koodatut rakenteet, vaikka kaavioita muutetaan. Kaaviota ja siihen liittyvää koodia voidaan tarkastella ja editoida samanaikaisesti.

UML-standardin mukaisista kaavioista Round-Trip-ohjelmat pystyvät päivittämään vähintään luokkakaavioita. Teoriassa myös sekvenssi- ja yhteistoimintakaavioiden päivitys onnistuu, mutta tätä ei ole toteutettu useimmissa ohjelmissa. Kaikkia UML-kaavioita (esim. käyttötapauskaavio) ei ole mahdollista päivittää lähdekoodin perusteella, koska ne eivät ole siihen suorassa yhteydessä.

Kaksisuuntainen muutosten seuranta on hankala operaatio, jonka käytännön toimivuus ohjelmissa on vaihtelevaa. Round-trip -ohjelmat ovat usein raskaita, pyrkimyksenä saada kaikki mahdolliset ominaisuudet yhteen pakettiin. Ohjelmat ovat myös huomattavan kalliita ja niiden käyttöönotto vaatii ylimääräistä koulutusta.[obj]

2.3.1 Together Control Center

Javalla toteutettu raskaan sarjan ohjelma. Tukee ohjelmistoprosessin kaikkia vaiheita UML-kuvauskielellä. Sisältää tuen hajautetulle kehitystyölle (useita projekteja, kussakin projektissa voi olla useampia kehittäjiä samaan aikaan). Pystyy analysoimaan ja generoimaan koodia Java, C#,C++, IDL ja Visual Basic -kielille. Kaaviot ja lähdekoodi ovat näkyvissä tarvittaessa samaan aikaan. Generoi HTML- PDF- tai RTF- muotoisia raportteja ja pystyy muodostamaan koodista metriikoita. Ohjelmaan saa kustomoidut näkymät eri käyttäjärooleille (esim. liiketoiminnan mallintaja, suunnittelija ja ohjelmoija). Tukee kaavioiden latausta ja tallennusta XMI-formaatissa.[tog]



Kuva 3: Together Control Center

2.3.2 Rational Rose -tuoteperhe

Ominaisuuksiltaan lähellä Together -ympäristöä. Tukee UML:ää ja Boochin metodia. Yhden ohjelman sijaan toiminnallisuus on jaettu monelle eri työkalulle (esim. omat työkalut projektin hallintaan, tietokannan ja olioiden mallintamiseen, koodin generointiin, debuggauksen ja dokumentointiin). Rational Rosesta on erilliset versiot Windows- ja yleisimpiin Unix-ympäristöihin.[rs1]

Rational Rose tukeee erityisesti automaattista dokumentointia. Dokumentteihin sisällytettävän tiedon ja ulkoasun voi määritellä, kuten dokumentointivälineissä yleensäkin. Lisäksi generoituun dokumentaatioon voi tehdä omia muutoksia - esim. lisätä kuvia ja kaavioita, jotka jäävät dokumentteihin uudelleengeneroinnin jälkeen. Voidaan siis puhua eräänlaisesta iteratiivisesta dokumenttikehityksestä.[rs2]

2.4 Yhteenveto

Koodingeneroijat ja erityisesti Round-trip-ohjelmat ovat usein hankalia käyttää. Useimmilla ohjelmilla on aivan oma käyttölogiikkansa, joka poikkeaa Windows-käyttöliittymästandardista. Tämä vaikeuttaa ohjelmien nopeaa käyttöönottoa. Monesti tuntuu, että on helpompi kirjoittaa pari riviä koodia perinteiseen tapaan kuin määritellä se CASE-ohjelmalla. Ohjelmien käänteistekniikan toimivuus vaihtelee riippuen oman projektin laajuudesta ja käytetyistä tekniikoista. Yleensä CASE-välineet toimivat parhaiten kehitettäessä ohjelmia tarkasti määritellylle sovellusalueelle.

Mielestämme nykyisten 'monolliittisten' CASE-ympäristöjen sijaan niiden toimintoja pitäisi hajauttaa usealle kevyelle (ja helppokäyttöiselle) työkalulle, jotka tarvittaessa pystyvät kommunikoimaan keskenään. Tätä kehitystä tukevat UML-kuvauskieli ja XMI-standardi. Tällöin analyysivaiheessa kaaviot voidaan piirtää halutulla piirto-ohjelmalla UML:n mukaisesti ja siirtää suunnittelu- ja toteutusvaiheessa Round-Trip ohjelmalle. Tekstimuotoiset dokumentit voidaan tuottaa esim. HTML-muodossa halutulla dokumentointivälineellä.

3 NLG menetelmän edut ja haitat

Tässä kappaleessa tutkimme automaattisen dokumentoinnin hyötyjä ja haittoja. Kappaleessa käsitellään erityisesti NLG (Natural-language Generation) menetelmällä tuotettua dokumentaatiota [reiter92][reiter93], mutta myös muita mentelmiä käsitellään hiukan.

NLG menetelmässä tuotetaan dokumentaatiota tietämyskannassa (engl. knowledge base) olevasta informaatiosta. Tietämyskanta sisältää faktoja tuotteesta

"neutraalissa" muodossa, joista generoidaan selkokielistä informaatiota tietämyskannassa olevan sanaston ja kielioppisääntöjen perusteella. Näin voidaan generoida haluttujen sisältö- ja muoto-standardien mukaisia dokumentteja automaattisesti. Tietämyskantaan lisätään tietoa sitä mukaa kuin tuotetta kehitetään.

NLG menetelmä soveltuu parhaiten teknisten, ohjekirjamaisten, dokumenttien tuottamiseen, mutta sitä voidaan käyttä mahdollisesti myös muuhunkin dokumentointiin täydentämään nopeasti muuttuvia asioita.

3.1 Edut

Automaattisen dokumentoinnin etuja verrattuna perinteiseen dokumentointitapaan ovat seuraavat:

3.1.1 Dokumentaation ylläpito- ja tuottamiskulut vähenevät

Mikäli suurin osa dokumentointiin tarvittavasta tiedosta on jo olemassa, joko tietokannassa tai jonkinlaisessa tietämyskannassa, voidaan dokumentoinnin tuottamiseen tarvittavaa aikaa ja työmäärää vähentää merkittävästi NLG-lähestymistavalla. Eo. pitää paikkansa vaikka dokumentoinnin generointijärjestelmä olisikin tarpeellista syöttää lisäinformaatiota pelkästään dokumentoinnin tuottamista varten tai mikäli ilmenee tarvetta generoidun dokumentaation jälkieditointiin.

Jopa suuremmat säästöt kuin dokumentaation tuottamisessa voidaan saada dokumentaation ylläpitopuolella. Suurin ongelma dokumentaation ylläpidossa ei suinkaan ole kielioppi- tai kirjoitusvirheet vaan dokumentaation ajantasalla pysyminen. Esimerkiksi ohjelman rakennetta muutettaessa voidaan

muutos esittää käytetyssä tietokannassa, josta automaattinen dokumentointisysteemi generoi ajantasalla olevan dokumentaation.

Myös dokumentaation lähdekoodista generoivat ohjelmat helpottavat dokumentaation ylläpitoa ja tuottamista. Itseasiassa lähdekoodissa olevat kommentit voidaankin melkein rinnastaa tietämyskannassa olevaan informaatioon.

3.1.2 Dokumentaation yhtenäisyys rakenteen kanssa

Usein ohjelmistosuunnittelija joutuu esittämään ohjelmansa dokumentaation useammassa kuin yhdessä eri muodossa. Tällainen saman tiedon moninkertainen esiintyminen on ensiksikin kallis tuottaa, mutta myös hyvin vaarallinen tekijä dokumentaation yhtenäisyyden kannalta. Automaattisen dokumentoinnin avulla suunnittelijan tarvitsee ylläpitää vain suunnittelutietokantaa. Tämä tapa vähentää merkittävästi yhtenäisyysongelmia saman tiedon eri esiintymien kanssa.

3.1.3 Dokumentaation ja standardien yhdenmukaisuus

Usein dokumentaation vaaditaan noudattavan tiettyjä kirjoitus- ja sisältöstandardeja. Kirjoitusstandardit on suunniteltu takaamaan dokumentaation kielen yksikäsitteisyys ja helppo ymmärrettävyys. Eräs esimerkki tällaisesta kirjoitusstandardista on AECMA Simplified english.

Sisältöstandardit taas määrittelevät mitä informaatiota tiettyjen dokumenttien tulee sisältää (esim. ylläpito- ja turvallisuusmenettelyt). Sisältöstandardit ovat harvoin tarkoin määriteltyjä, mikä taas hankaloittaa niiden automatisointia.

Ohjelmoimalla NLG järjestelmään tarkoituksen mukainen sanasto ja kielioppi voidaan se pakottaa noudattamaan haluttuja kirjoitusstandardeja. NLG järjestelmä voidaan myös asettaa noudattamaan sisältöstandardeja mikäli niiden noudattamiseen tarvittava tieto on tietokannassa.

3.1.4 Monikielisyys

NLG järjestelmä saadaan tuottamaan dokumentaatio monilla eri kielillä asettamalla sille halutun kielen syntaksi ja sanasto määritykset. Monikielisen dokumentaation tuottaminen ei ole teknisesti triviaali ongelma, mutta teknisten dokumenttien tuottaminen ei ole niin suuri ongelma varsinkin, jos käytetään jo aikaisemminkin mainittuja kirjoitusstandardeja. Monet teknisten dokumenttien kirjoitusstandardithan kieltävät monimutkaiset lauserakenteet, mikä helpottaa dokumentoinnin automatisointia.

Automaattinen monikielisyys vähentää dokumentaation käännöskuluja (mikäli sellaisia on), mutta ei vielä voi poistaa niitä. Käytännössä dokumentaation on tarkastettava ja jälkieditoitava.

3.1.5 Monimuotoisuus

NLG menetelmällä voidaan myös määritellä generoitavan dokumentaation esitysmuoto tai yhdistelmä esitysmuodoista. Pelkän tekstin lisäksi voidaan generoida kaavioita (esim. UML) ja muita graafisia esityksiä, sekä tuottaa interaktiivisia viitteitä muuhun liittyvään tietämykseen (esim. hyperteksti).

Kuten monikielisyydessäkin, myös monimuotoisuudesta johtuen dokumentaatio on yleensä tarkastettava.

3.1.6 Dokumentaation räätälöinti

NLG järjestelmä sallii dokumentaation dynaamisen muokkaamisen kontekstin mukaan. Tämä tarkoittaa, että dokumentaatiosta saadaan eri näkymiä tarpeesta riippuen, esimerkiksi koodausportaalle teknisemmät dokumentit ja johtoportaalle selkeämpi sanaiset.

3.2 Haitat

Automaattisesta dokumentoinnista saatavia etuja on syytä verrata sen haittoihin, jotta voitaisiin tarkastella saatavaa kokonaishyötyä. Seuraavassa tarkastellaan automaattisen dokumentaation haittapuolia.

3.2.1 Suunnittelutietokannan luonti hankaloituu

Yleensä suunnittelutietokannat eivät sisällä kaikkia automaattisen dokumentoinnin tarvitsemia tietoja oletuksena. Tämä tarkoittaa että tietokantaan täytyy lisätä automaattisen dokumentaation tarvitsemat tiedot mikä lisää suunnittelijoiden työmäärää ja näin myös kustannuksia. Toisaalta taas voidaan ajatella, että kaikkea tietokantaan lisättävää informaatiota voidaan käyttää myös muihin tarkoituksiin kuten esimerkiksi yhtenäisyyden tai eheyden tarkasteluun.

3.2.2 Tietokannan kustannukset

Automaattinen dokumentointi vaatii tietokannalta myös tiettyjä erityisominaisuuksia. Tällaisia ovat esimerkiksi kohdealueen kuvaus ja dokumentaation käyttämän 'alakielen' määritys (usein kirjoitusstandardissa määritetty). Tämän kaltaiset kulut pienenevät tehtäessä useita sovelluksia, sillä hyvin tehty tietokanta voidaan usein uudelleenkäyttää.

3.2.3 Laadun varmistus

Automaattisen dokumentaation tuottama teksti joudutaan lähes aina viemään laadun varmistuksen kautta. Laadun varmistus joutuu tarkastamaan tuotetun dokumentaation sekä sisällöllisen ja kielellisen oikeellisuuden ja usein myös muokkaamaan sitä. Tällainen jälkimuokkaus saattaa maksaa huomattaviakin summia.

3.3 Yhteenveto

Näyttää ilmeiseltä, että automaattisen dokumentoinnin käyttöönotto tehostaa ohjelmistotuotantoa. Monesti dokumentaatio koetaan toisarvoiseksi itse tuotteeseen nähden, lähinnä dokumentaation tuottamiseen kuluvien resurssien suuruuden takia. Automaattinen dokumentointi, eri muodoissaan, yleisesti vähentää dokumentoitiin vaadittavien resurssien määrää ja sitä kautta helpottaa laadukkaampien ja kattavampien dokumentaatioiden tuotantoa.

On kuitenkin huomattavaa, että ohjelmistotuotteen koosta riipuen on syytä tarkasti valita sopivat työkalut. Suuren NLG menetelmään perustuvan järjestelmän luominen, suhteellisen pientä ohjelmistoa varten ei varmastikkaan ole kustannus syistä järkevää. Mutta toisaalta pienissäkin projekteissa voidaan käyttää valmiina olevia järjestelmiä.

Kuvat

Viitteet

[reiter92] Reiter, Ehud; Mellish, Chris; and Levine, John (1992). "Automatic generation of on-line documentation in the IDAS project." Proceedings, Third Conference on Applied Natural Language Processing, Trento, Italy, April 1992, 64–71. http://citeseer.nj.nec.com/reiter92automatic.html

[reiter93] Ehud Reiter and Chris Mellish, 'Optimizing the costs and benefits of natural language generation', in Proceedings of the International Joint Conference of Artificial Intelligence, Chambery, France, pp. 1164– 1169, (1993). http://citeseer.nj.nec.com/reiter93optimizing.html

[sun] Sun Microsystems: Javadoc
<http://java.sun.com/j2se/javadoc/index.html>

[dox] Dimitri van Heesch: Doxygen
<http://www.stack.nl/~dimitri/doxygen/>

[cw] Codework: CC-Rider
<http://www.codework.com/george/product.htm>

[mod] Modelistic ltd: Kritiikkiä koodigeneroijista
<http://www.modelistic.com/overview.html>

[xmi] O'Reilly & Associates, Inc: XMI
<http://www.xml.com/pub/rg/XMI_XML_Metadata_Interchange>.

[mc] MetaCase Consulting
<http://www.metacase.com/>

[dom] Atos Origin: Delphia*Object Modeler
<http://www.ii.atos-group.com/rhone-alpes/Dom/english/>

[obj] Objects By Design, inc. (oliosuunnitteluun ja -ohjelmointiin keskittynyt sivusto)
<http://www.objectsbydesign.com/>

[tog] Together Software
<http://www.together.com/>

[rs1] Rational Software
<http://www.rational.com/>

[rs2] Rational Software: Software Document Automation: A Technical Overview
<http://www.rational.com/products/whitepapers/347.jsp>