Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

KANDIDAATINTYÖ

Ohjelmointiprosessin tehostaminen Java Enterprise Edition -ympäristössä

Tatu Kosonen

Kandidaatintyön aihe on hyväksytty 4.2.2009

Työn tarkastajana toimii prof. Kari Smolander

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto

Teknistaloudellinen tiedekunta

Tietotekniikan koulutusohjelma

Tatu Kosonen

Ohjelmistoprosessin tehostaminen Java Enterprise Edition -ympäristössä

Kandidaatintyö

2009

36 sivua, 11 kuvaa, 0 taulukkoa, 0 liitettä

Tarkastaja: prof. Kari Smolander

Hakusanat: Ohjelmointi, automaattinen, Java Enterprise Edition -ympäristö, työryhmä

Keywords: Programming, Automatic, Java Enterprise Edition Environment, Work group

Tietokoneiden laskentatehon nopea kasvaminen on mahdollistanut parempien ohjelmointitekniikoiden kehittämisen. Ohjelmien ylläpidettävyys on noussut tärkeämmäksi kuin niiden mahdollisimman nopea toiminta. Tarkoitukseen kehitettyjä suunnittelutekniikoita on monia, mm. suunnittelumallit, ohjelmistoarkkitehtuurit, sekä malli- ja testivetoiset kehitystekniikat. Ohjelmoinnin automatisointi on toinen tavoite, johon on pyritty jo 1950-luvulta lähtien.

Tämän työn tarkoituksena on laatia vaatimusmäärittely sellaista ohjelmaa varten, joka mahdollistaa ohjelmointiprosessin automatisoinnin. Yksi päätavoitteista on ketterien menetelmien periaatteiden toteutuminen, sekä generoidun ohjelman dokumentaation automaattinen päivitys myös ohjelman ylläpitovaiheessa. Vaatimusmäärittelyn määrittelemän ohjelman käyttäjien oletetaan olevan ohjelmoijia. Generoidut ohjelmat toimivat Java Enterprise Edition -ympäristön palvelimissa.

ABSTRACT

Lappeenrannan University of Technology

The faculty of Technology Management

The department of Information Technology

Tatu Kosonen

Enhancing the Programming Process in the Java Enterprise Edition Environment

Bachelor Thesis

2009

36 pages, 11 figures, 0 tables, 0 appendices

Supervisor: prof. Kari Smolander

Keywords: Programming, Automatic, Java Enterprise Edition Environment, Work group

The fast growth of the computer calculation speed has made it possible to develop better programming techniques. Programs with good maintenance have become more important than programs executing as fast as possible. There are many techniques developed to meet this requirements, e.g. patterns, programming architectures, and model-/test-driven development. Making programming more automatic has been another goal to achieve since the 1950's.

The purpose of this work is to create a specification for a program making automatic programming possible. The program should fulfill the requirements of agile software development. One of the main targets here is to make the documentation of the generated programs update automatically, also in the maintenance period. The users of this program are supposed to be programmers. The generated programs will work with Java Enterprise Edition servers.

Alkusanat

Tämä kandidaatintyö on tehty Lappeenrannan teknillisen yliopiston tietotekniikan osastolle keväällä 2009. Työn tarkoitus on laatia vaatimusmäärittely ohjelmointiprosessin tehostamiselle Java Enterprise Edition -ympäristössä automaation keinoin. Työ perustuu kirjoittajan pitkään työkokemukseen räätälöityjen ohjelmistojen parissa.

Kiitokset työn toteutumisesta esitän työn ohjaajalle, sekä kaikille asiakkailleni vuosien varrelta. Ilman haasteita ja hyviä neuvoja työn idea olisi jäänyt syntymättä tai ainakin jäsentymättä sopivalle tasolle. Vaimolleni annan erityiskiitoksen vaatimuksesta rajoittaa työni sivumäärää. Työ selkeytyi tämän vaatimuksen ansiosta huomattavan paljon.

Brysselissä 26.4.2009

Tatu Kosonen

Sisällysluettelo

TIIVISTELMÄ, ABSTRACT........II

ALKUSANAT........IV

SYMBOLI- JA LYHENNELUETTELO........2

1. JOHDANTO........4

1.1. Tausta........4

1.2. Tavoitteet ja rajaukset........6

1.3. Työn rakenne........7

2. OHJELMOINNIN TEHOSTAMISEN TEKNIIKOITA........8

2.1. Yleistä........8

2.2. Suunnittelumallit........8

2.3. Ohjelmistoarkkitehtuurit........9

2.4. Mallivetoinen kehitys........10

2.5. Testivetoinen kehitys........11

2.6. Ohjelmoinnin automatisointi........12

2.7. Visuaalinen ohjelmointi........12

2.8. Ketterät menetelmät........14

2.9. Oliotietokannat........14

2.10. Geneettinen ohjelmointi........15

2.11. Johtopäätökset........15

3. VAATIMUSMÄÄRITTELY........18

3.1. Ongelma........18

3.2. Tavoite........18

3.3. Rajaukset........25

3.4. Toimintaympäristöt........26

3.5. Jatkokehitys........27

4. YHTEENVETO........28

LÄHTEET........30

Symboli- ja lyhenneluettelo

Business-logiikka	Käytännössä: jonkin ohjelman sisältämä toiminnallisuus.
CSS	*Cascading Style Sheets* on WWW-sivuille kehitetty ulkoasuohjeisto
CVS	*Concurrent Version System* on eräs avoimen lähdekoodin toteutus ohelmistojen versionhallinnasta.
Fortran	*Formula Translator* on varhainen ohjelmointikieli
EAR	*Enterprise Archive* on käytännössä yksi Java EE -sovellus paketoituna yhteen tiedostoon. EAR koostuu usein WAR- ja EJB JAR -paketeista.
EJB	*Enterprise Java Bean* on Java EE -arkkitehtuurin määrittelemä sovelluspalvelimessa toimiva komponentti
HTTP	*Hyper Text Transfer Protocol* on protokolla, jota käytetään WWW-sivujen siirtämiseen.
HTTPS	*Hyper Text Transfer Protocol Secured* on HTTP-yhteys, joka on suojattu TLS-menetelmällä.
JAR	*Java Archive* on tiedosto, johon on paketoitu ajettavia luokkia. EJB JAR -tiedosto sisältää sovelluspuolen komponentteja (vrt. WAR)
MDD	*Model-Driven Development* on ohjelmointitapa, jossa ohjelman tietorakenne suunnitellaan ennen varsinaista ohjelmointia. Ohjelmointitapa sopii hyvin suunnittelumallin Malli-Näkymä-Kontrolloija (Model-View-Controller) toteutukseen.

Pattern	Suunnittelumalli, hyväksi koettu tapa ratkaista jokin standardiongelma.
SQL	*Structured Query Language* on kieli, jota yleensä käytetään tietokantojen kanssa.
TDD	*Test-Driven Development* on ohjelmointitapa, jossa ohjelman testaussuunnitelma tehdään ennen varsinaista ohjelmointia.
TLS	*Transport Layer Security* on suojausprotokolla, jonka avulla tiedot voidaan salata luotettavasti ennen niiden lähettämistä, eivätkä ulkopuoliset siis pääse tutkimaan liikenteen sisältöä. Protokollan tarjoaman suojan todisteena voidaan pitää sitä, että kaikki suomalaiset Internet-pankkiyhteysohjelmat luottavat siihen.
UML	*Unified Modelling Language* on kaavioihin perustuva mallinnuskieli.
WAR	*Web Application Archive* on tiedosto, johon on paketoitu luokkia (ks. JAR). WAR-tiedosto sisältää erityisesti käyttöliittymäpalvelimella ajettavia komponentteja.
Webbiselain	Ohjelma, jonka avulla luetaan WWW-sivuja
WWW	*World Wide Web* on Internetissä toimiva, hajautettu hypertekstijärjestelmä.

1. Johdanto

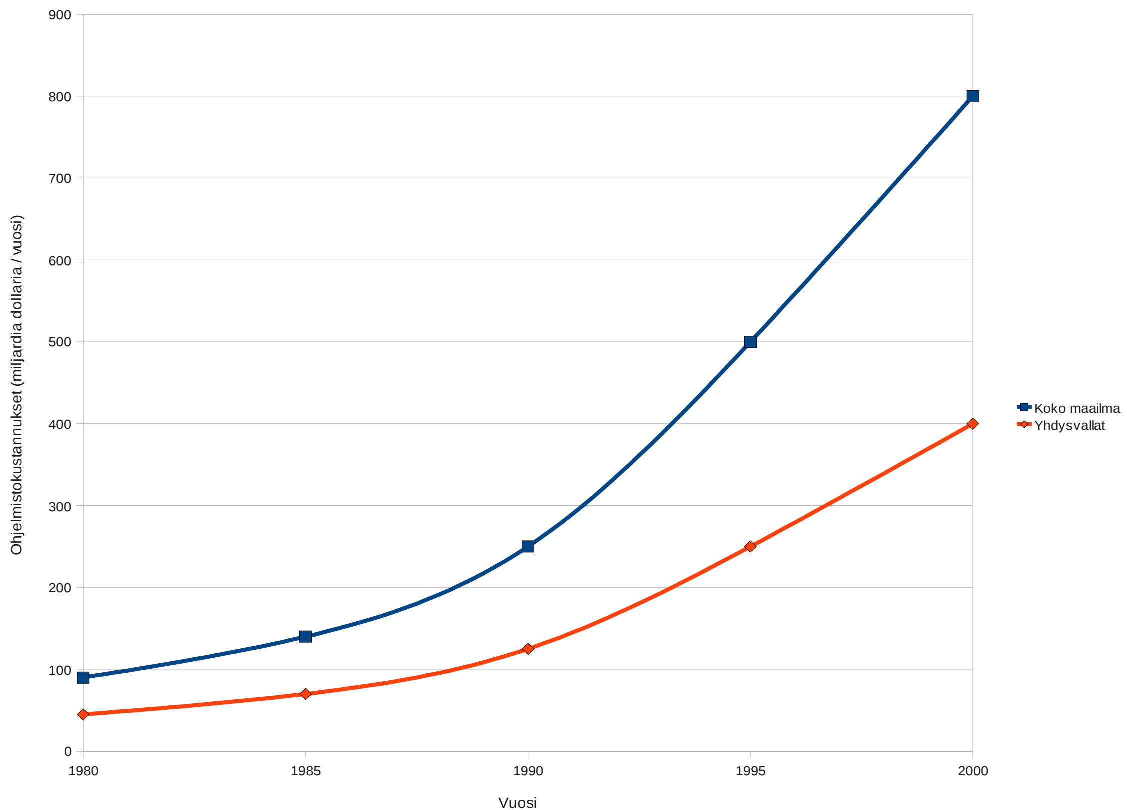
1.1. Tausta

Tietokoneiden laskentatehon nopea kasvaminen on mahdollistanut ohjelmistokielien ja -tapojen kehittymisen. Suurimmassa osassa ohjelmistoja ei enää ole tarpeen miettiä ensisijaisesti niiden optimaalisen nopeaa toimintaa, vaan painopiste on voitu siirtää ylläpidettävän ohjelmointitavan ja selkeästi jäsennellyn lähdekoodin laatimiseen. Erityisesti olio-ohjelmointikielien kehittyminen ja ohjelmistoarkkitehtuuri-käsitteen syntyminen on helpottanut ja nopeuttanut ohjelmistojen ylläpitoa sekä niihin jälkikäteen tehtäviä muutostöitä. Tämä on merkittävä asia, koska noin 67% ohjelmistoon tehtävästä työstä tehdään siihen sen ylläpitoaikana [1]. Ohjelmistojen ongelmien korjaaminen toimituksen jälkeen on 100 kertaa kalliimpaa kuin niiden havaitseminen ja oikaisu jo määrittelyvaiheessa. Ylipäänsä vain 15% ohjelmistoprojektiin tehtävästä työstä on varsinaista ohjelmointia, ja 20% ohjelmoijista tekee 80% tästä työstä. [2]

Organisaatio, prosessimalli ja menetelmät vaikuttavat toisiinsa. Niiden yhtenäistämisellä tavoitellaan ohjelmistokehityksen tehokkuuden parantamista, ohjelmistojen laatutason nostamista ja työmotivaation kohottamista [3]. Ohjelmistojen kustannukset myös kasvavat jatkuvasti, tätä kasvua on esitelty kuvassa 1. Kustannusten kasvu on sen vuoksi tärkeä leikata ohjelmistojen tuottavuutta parantamalla. [4]

1.2. Tavoitteet ja rajaukset

Työn tarkoituksena on tutkia, miten ohjelmointiprosessiin osallistuvan työryhmän kokoa voidaan pienentää automatisoinnin välinein. Tavoitteena on laatia vaatimusmäärittely ohjelmointityön automatisointia varten. Automatisointi tehdään arkkitehtuurin menetelmin, tavoitteena vähentää sekä itse prosessiin ja sen hallinnointiin kuluvaa aikaa, että pienentää varsinaiseen ohjelmointiin ja dokumentointiin kuluvaa työmäärää. Tärkeimmät strategiat ohjelmistokustannusten pienentämiseksi on esitelty kuvassa 2. Ohjelmoinnin

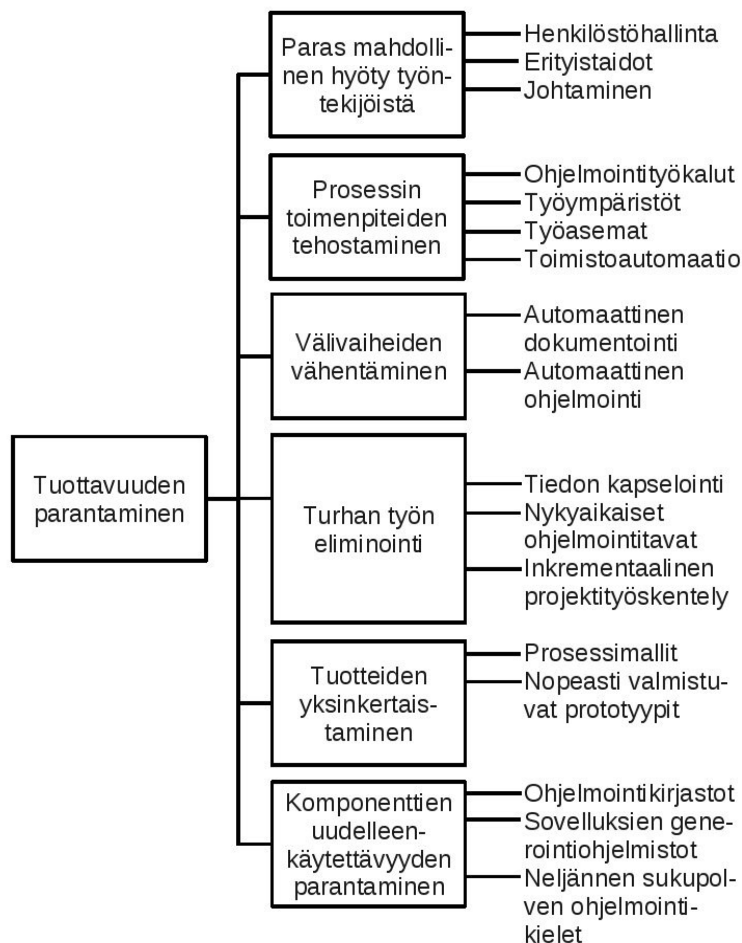


Kuva 1: Ohjelmistokustannusten kehittyminen ja suuntaus [4]

automatisointi ja automaattinen laadunhallinta, säännöllisesti toistuvien rutiinitöiden karsiminen ja ohjelmointikirjastoiden hyödyntäminen ovat tärkeitä keinoja kustannussäästöihin. [4] Tässä työssä oletetaan nimenomaan ohjelmointityön automatisoinnin olevan työkalu laadun parantamiseen. Laadukkaaseen ohjelmistoon liittyy aina myös kattava dokumentointi, jonka oletetaan myös olevan toteutettavissa osittain automaation menetelmin.

Tutkimus koskee erityisesti suuria, tietokantapohjaisia ohjelmistoja. Niiden prosessi on perinteisesti edellyttänyt suurta ohjelmointityöryhmää, vaikka näennäisesti erilliset työt voivat olla keskenään hyvinkin yhteneväisiä. Suuri työryhmä tuo yleensä mukanaan lisätyötä hallinnointi- ja tiedotustarpeidensa vuoksi, eikä ryhmän ohjelmoijien taso luultavasti ole yhtäläinen. Jos oletetaan, että resurssien lisääminen ohjelmistoprojektiin

nopeuttaa sen läpivientiä korkeintaan 25% [2], voidaan suuri työryhmä kokea ennen kaikkea yhdeksi prosessin riskitekijäksi. Parhaimman tuloksen saaminen ihmisiltä edellyttää monien tekijöiden yhtäaikaista onnistumista. Henkilöstöhallinnan haaste on osaavien henkilöiden käyttäminen työssä, vaikka heidän pyytämänsä palkkio olisikin suuri. Tehokkaiden laitteistojen ja hyvien työtilojen tarve on selkeä, mutta kasvattaa taas kustannuksia. Erityisen suuri haaste on projektin onnistunut johtaminen; varsinkin huolellinen suunnittelu ja valmistelu ennen varsinaisen ohjelmoinnin aloittamista on tärkeää. Huippuosaajat etenevät yleensä johtajiksi, jolloin varsinainen ohjelmointityö lankeaa vähemmän taitaville henkilöille. [4] Toisin päin ajateltuna voidaan todeta, että jos hyvien ohjelmoijien ei tarvitsisi käyttää aikaa johtamiseen, he ehtisivät ohjelmoida paljon enemmän. Yleisesti voidaan todeta pienemmän työryhmän lisäävän työn tehokkuutta jo



Kuva 2: Ohjelmistoprosessin tehokkuuden kehittämispuu [4]

pelkästään sen vuoksi, että keskinäisen kommunikaation (esim. työryhmän sisäisten palavereiden) tarve pienenee. Mitä laajempi työryhmä on kyseessä, sitä tarkemmin on työn välivaiheet ja eteneminen tiedotettava koko ryhmälle.

Työ rajataan koskemaan vain jo olemassa olevia ratkaisuja ja kirjallisuutta. Tekniikan osalta tutkimus rajataan koskemaan Java Enterprise Edition -arkkitehtuurin mukaisia ohjelmistoja.

1.3. Työn rakenne

Ensimmäisessä luvussa kuvataan ratkaistava ongelma ja perustellaan ratkaisun tarvetta.

Toisessa luvussa kuvataan suunnittelu- ja ohjelmointitekniikoita, jotka palvelevat ongelman ratkaisua, ja joita lopullisessa vaatimusmäärittelyssä tullaan edellyttämään.

Kolmas luku on varsinainen vaatimusmäärittely.

Neljäs luku on yhteenveto.

2. Ohjelmoinnin tehostamisen tekniikoita

2.1. Yleistä

Erilaiset suunnittelutekniikat ovat yksi työkalu ohjelmointiprosessin tehostamiseen. Voidaan todeta, että ylipäänsä suunnittelemalla työ huolellisesti ennen sen toteuttamista lopputulos on parempi ja luultavasti nopeammin valmis kuin vain aloittamalla toteutus välittömästi määrityksen valmistuttua. Suunnittelun avuksi on kuitenkin laadittu tekniikoita, jotka entisestään tehostavat työskentelyprosessia.

2.2. Suunnittelumallit

Suunnittelumallit (engl. pattern) ovat hyviksi koettuja tapoja ja strategioita, joilla ratkaistaan erilaisia olio-ohjelmoinnin standardiongelmia. Suunnittelumallit ovat pieniä, itsenäisiä mutta toisistaan riippuvia suunnittelukokonaisuuden osia, joita voidaan yhdistellä toistensa tai muiden tekniikoiden kanssa lopputuloksena kokonainen ohjelmisto. Suunnittelumallit on suunniteltu erityisesti olio-ohjelmointiympäristöihin, eikä niiden käyttäminen ole sidoksissa johonkin tiettyyn ohjelmointikieleen. Suunnittelumalleja on kehitetty myös ohjelmistojen laadun arvioimiseen; ne perustuvat ohjelmamoduulien vikataipumusten arviointiin ja ennustamiseen esimerkiksi tunnettujen huonojen ohjelmointitapojen (engl. bad coding practices) havaitsemisen kautta. [5, 6]

Suunnittelumallit syntyvät kokeneiden ohjelmoijien toimesta kolmiosaisessa prosessissa. Ensin on ymmärrettävä ja luetteloitava hyviksi todistetut ohjelmointitekniikat. Sen jälkeen nämä tekniikat on kuvattava selkeästi ja nimettävä ymmärrettävästi. Lopuksi toiset asiantuntijat tarkastavat ja arvostelevat uuden suunnittelumallin huolellisesti, sekä tarvittaessa muokkaavat sitä vielä. [5] Valmis suunnittelumalli takaa yhden, uudelleenkäytettävän suunnitelman jonkin tietyn ongelman ratkaisemiseen. Suunnittelumallilla vältetään luomasta olioita suoraan, erillisten operaatioiden välisiä

riippuvuuksia tai liian tiukkoja liitoksia. Tällä tavoitellaan joustavia, modulaarisia ja ymmärrettäviä ohjelmistokokonaisuuksia. Sen seurauksena erityisesti ohjelmistojen ylläpito helpottuu. [6]

2.3. Ohjelmistoarkkitehtuurit

Sellaiset ohjelmistot, joiden on tarkoitus olla pitkäikäisiä, kokevat yleensä paljon muutoksia elinkaarensa aikana. Tärkeä tavoite on ylläpitäjien työn helpottaminen ja ennen kaikkea heidän tekemiensä muutosten säilyttäminen kohtuuaikaisina ja -hintaisina. Suuret ohjelmat ovat yleensä rakenteeltaan monimutkaisia, mutta silti myös muiden kuin alkuperäisten tekijöiden pitäisi saada muutoksia tehtyä hallitusti. Ohjelmistoarkkitehtuuri huolellinen suunnittelu vähentää ohjelmiston monimutkaisuutta, ja erityisesti olio-ohjelmoinnille tyypillistä on se, että ylemmän tason elementit kirjataan ennemmin paperille kuin lähdekoodiin. Ongelmana on kuitenkin se, että tällaiset asiakirjat eivät useinkaan pysy ajan tasalla.

Yksi ohjelmistoarkkitehtuuridokumenttien tärkeimmistä tehtävistä on mahdollistaa ohjelmiston analysointi ja toiminnallisuuden mahdollisimman täydellinen ymmärtäminen ennen muutosten tekemistä. Suurimmat haasteet ovat luokkien, pakettien, alijärjestelmien ja arkkitehtuuritasojen suuressa lukumäärässä ja niiden keskinäisten suhteiden ymmärtämisessä, sekä ylipäänsä arkkitehtuurisuunnitelman ja -toteutuksen suhteuttamisessa toisiinsa lähdekoodin tasolla. [7]

Kolmitasoarkkitehtuuri on eräs yleisesti käytetty, alustariippumaton tekniikka. Se on noussut johtavaan asemaan erityisesti HTTP-pohjaisissa, webbiselainta käyttöliittymänään hyödyntävissä ohjelmistoissa. Arkkitehtuurin perusajatus on se, että kaikki toiminnallinen logiikka (engl. business logic) asennetaan ja suoritetaan palvelinpäässä. Arkkitehtuurin nimi tulee kolmesta päätasosta, jotka ovat käyttöliittymä, varsinainen sovellus ja tietokantajärjestelmä. Käyttöliittymätaso (webbiselain) ottaa yhteyden palvelimelle, eikä mitään toiminnallista logiikkaa jätetä sen varaan, vaan tämä taso toimii todellakin vain

käyttöliittymänä. Sovellustaso piilotetaan käyttäjältä kokonaan, mutta silti se vastaa koko sovelluksen toiminnoista ja tietokantapalvelimen tapaisten etäjärjestelmien hallinnasta. Tietokantajärjestelmä sisältää sovelluksen hallinnoimat tiedot. Eräs kolmitasoarkkitehtuurin valmis alusta Java Enterprise Edition -ympäristössä on Apache Struts. [8]

2.4. Mallivetoinen kehitys

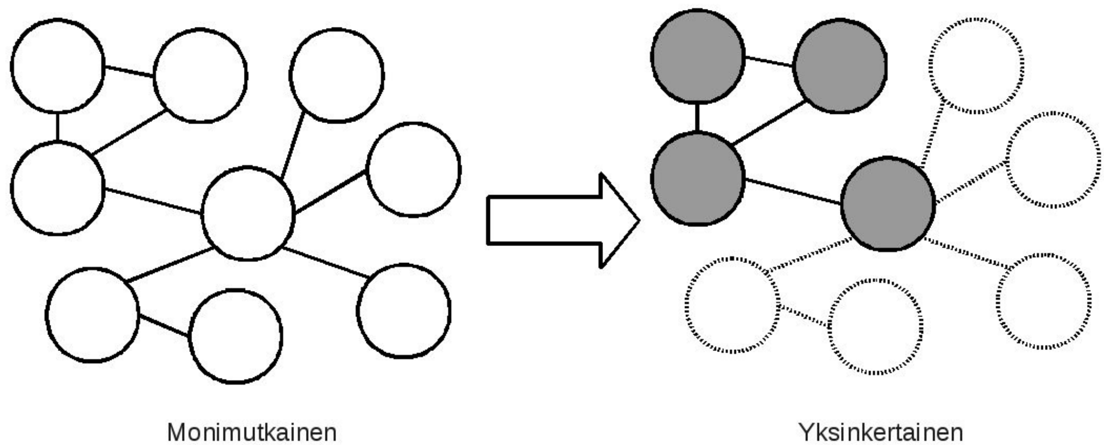
Mallivetoista kehitystä (engl. Model-Driven Development) voidaan pitää olio-ohjelmoinnin johtavana suunnittelutekniikkana. Ohjelmistojen kuvaamiseen käytetään erilaisia malleja, jotka pyritään suunnitteluvaiheessa pitämään alustariippumattomia. Sen jälkeen malli muutetaan alustariippuvaksi malliksi ja lopuksi lähdekoodiksi automaattisin työkaluin. [9] Useimmat mallivetoisen kehityksen nykyisistä työkaluista sisältävät kuitenkin vain generointityökalut mallista lähdekoodiksi, eivät siis generointityökaluja lähdekoodista malliksi tai alustariippumattomasta mallista alustariippuvaan malliin. Varsinkin jälkimmäinen puute on herättänyt paljon toiveita tekniikan kehittäjien keskuudessa. [10]

Mallivetoisen kehityksen hyvinä puolina pidetään lähdekoodin automaattisen generoinnin säästämän ajan määrää, joka on hyvin merkittävä. Korkeatkin laatuvaatimukset täyttyvät, koska generoiduissa moduleissa käytetään vain hyvin tunnettuja ja testattuja ratkaisuita. Ohjelmointialustaa vaihdettaessa vaihdetaan vain lähdekoodin generoivaa osaa, joka on paljon yksinkertaisempi toimenpide, kuin koko ohjelmiston kirjoittaminen uudestaan esim. toisella kielellä. Mallien käyttäminen helpottaa myös projektiryhmän kommunikointia ja vähentää väärinkäsitysten määrää. Kehitystekniikan haasteita ovat pientenkin osioiden laatuvaatimusten suuri merkitys koko järjestelmän kannalta, järjestelmien pitkä elinkaari (jopa kymmeniä vuosia), vanhojen järjestelmien vaikutus (useimmat uusista järjestelmistä pohjautuvat vanhoihin, jopa ikivanhoihin esijärjestelmiin), ohjelmoijille asetettu vaatimus jäsentää ongelma mallilähtöisesti, sekä tietoturvan riittävän tason saavuttaminen. [11]

2.5. Testivetoinen kehitys

Testivetoinen kehitys (engl. Test-Driven Development) on hieman nykyaikaisempi tapa suunnittella ohjelmistoja. Se on syntynyt ohjelmoijien tarpeesta kehittää ohjelmistoja paremmin ja nopeammin. Työskentelytekniikka toimii siten, että ensin kirjoitetaan suoritettavissa oleva testi, jolla tutkitaan, toteuttaako sovellus määrityksensä. Vasta testin valmistuttua kirjoitetaan varsinainen ohjelmakoodi. Useat pienet testit muodostavat kattavan testikokonaisuuden, joka helpottaa yksittäisten virheiden paikantamista jo toteutusvaiheessa ja vähentää siten virheiden etsimiseen ja ratkaisemiseen kuluvaa aikaa hyvin merkittävästi. Tämä kaikki tukee ohjelmistokehityksen lopullista päämäärää tuottaa sellaisia ohjelmistoja, jotka tehostavat ohjelmaa käyttävän yrityksen toimintaa ja pienentävät läpimenoaikoja. Tällöin myös ohjelmistojen suorat käyttökustannukset pienenevät.

Testivetoisen kehityksen lähtökohtana on erittäin yksinkertainen sääntö: ohjelmakoodia kirjoitetaan vain epäonnistuneen testin korjaamiseksi. Yhdistettynä oikeaoppiseen ohjelmointitapaan, jossa keskitytään kirjoittamaan kerrallaan vain sellaisia pieniä ohjelmanosia, jotka yhdessä kuitenkin muodostavat laajan kokonaisuuden (kuten kuvassa 3), päästään haluttuun tilanteeseen; tällöin työskennellään todellakin vain sen ongelman ratkaisemiseksi, joka alunperin oli tarkoituksenakin. Kaupallisessa ohjelmistotuotannossa tämä saattaa mahdollistaa myös sellaisen työskentelytavan, jossa edistyminen on koko ajan asiakkaan havaittavissa ja myös mahdollinen laskutusperuste sekä asiakkaan että ohjelmistoyrityksen nähtävissä. Luultavasti myös alunperin määriteltyjen aikataulujen pitäminen osoittautuu helpommaksi. Tämä kaikki on taatusti tavoittelemisen arvoista varsinkin asiakastyytyväisyyden vuoksi. [12]



Kuva 3: Monimutkaisten ongelmien ymmärtäminen helpottuu huomattavasti jakamalla kokonaisongelma ensin pienempiin osiin, jotka sitten ratkaistaan yksi kerrallaan. [12]

2.6. Ohjelmoinnin automatisointi

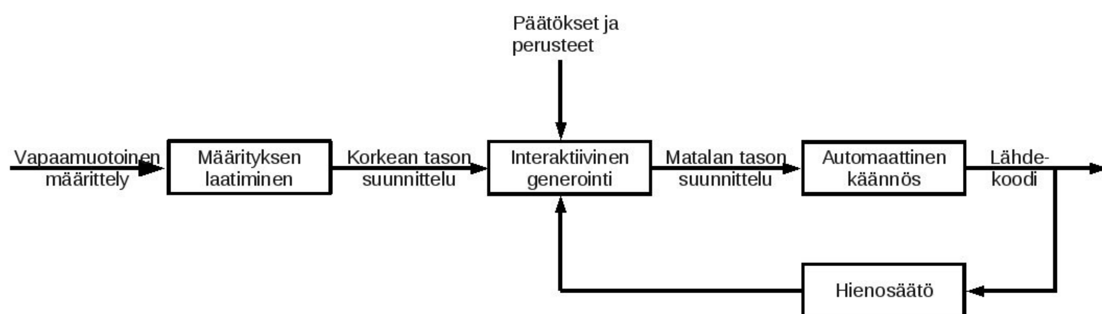
Ohjelmoinnin automatisointi on tavoite, johon on pyritty jo 1950-luvulta lähtien; termiä käytettiin ensimmäisen kerran kuvaamaan varhaisia Fortran-kääntäjiä. Termin viehätys perustuu siihen havaintoon, että ohjelmointityön voidaan todeta toimivan todellisena hidasteena tietokoneen täydelliselle hyödyntämiselle. Ohjelmiston kehitysprosessi perustuu tietämykseen ja se on hyvin riippuvainen työskentelyyn osallistuvista henkilöistä. Ohjelmistojen ylläpito kohdistuu lähdekoodiin, siis toteutettuun ohjelmakoodiin. Ohjelmointi myös on kaikkea muuta kuin vain ohjelmakoodin kääntämistä toimivaksi ohjelmaksi, erityisesti sanallisen määrityksen muuttamista toiminnalliseksi logiikaksi. Pitkäaikainen tavoite on kuitenkin ollut se, että jossain vaiheessa määrittely olisi automaattisesti muutettavissa käännetyksi ohjelmaksi (kuten kuvassa 4). [13]

2.7. Visuaalinen ohjelmointi

Eräs tietotekniikkateollisuuden suurimmista haasteista on sellaisten henkilöiden kommunikointi tietokoneen kanssa, jotka eivät ole ohjelmoijia, koska perinteisen

ohjelmoinnin opiskelu on hidasta ja jopa turhauttavaa suurimmalle osalle tietokoneiden käyttäjiä. Visuaalinen ohjelmointi on tekniikka, jonka tarkoituksena on ratkaista tämä haaste. Se perustuu ajatukseen, jonka mukaan kuvat ovat ymmärrettävissä helpommin kuin kirjoitettu teksti - varsinkin kun otetaan huomioon eri kieltä puhuvat ihmiset. Samasta ajatuksesta on syntynyt myös käyttöjärjestelmien graafinen käyttöliittymä, eikä ole sattumaa, että ensimmäiset käyttökelpoiset toteutukset kummastakin tehtiin samoihin aikoihin 1980-luvulla. [14] Varhaisia visuaalisia ohjelmointikieliä olivat mm. Pict, Formal ja IGI (ISFI's Graphical Interface). [14, 15]

Visuaalisen ohjelmointikielen avulla on suhteellisen helppo hyödyntää muutoinkin käytössä olevia hyviä ohjelmointitekniikoita, kuten olio-ohjelmointiominaisuuksia ja laajojen ongelmien jakamista pienempiin osiin hyvin luonnolliselta vaikuttavin menetelmin. Rajapintojen ja erityisesti niiden läpi kulkevan tiedon oikean muodon tarkkailu, näkymien runsas lukumäärä ja suunnittelumallien hyödyntäminen muuttuvat jopa automaattisiksi toimenpiteiksi, kun varsinainen ohjelmakoodi tuotetaan piilossa ohjelmoijalta. [15] Hyvin merkittävä hyötynäkökohta on se, että perinteisen ohjelmoinnin menetelmillä on ollut ohjelmistojen suunnittelusta syntyviä dokumentteja (kuten erilaisia kaavioita) haastavaa ylläpitää siten, että ne säilyvät myös toteutus- ja ylläpitovaiheessa ajan tasalla. Kuitenkin lähdekoodi ja dokumentit ovat vain kaksi eri esitystapaa samasta ohjelmasta. Muuttamalla jollain tekniikalla suunnitteluvaiheen kaaviot suoraan suoritettaviksi ohjelmiksi (tai päin vastoin ohjelmat automaattisesti kaavioiksi) päästäisiin



Kuva 4: Automaattisen ohjelmointijärjestelmän malli [13]

yhdistämään nämä kaksi erillistä prosessia, ja tämä olisi eräs tapa tehostaa sekä dokumentointia että ylläpitoa. [14]

2.8. Ketterät menetelmät

Ketterät menetelmät ei ole niinkään tekniikka, vaan pikemminkin ohjelmoijien antama julistus, joka annettiin helmikuussa 2001. Sen periaatteet toimivat kuitenkin hyvänä ohjeena myös automaation suunnittelussa, ja jopa automatisoinnin mallina. Tärkein periaate on saada asiakas tyytyväiseksi toimittamalla hänelle tarpeellinen ohjelmisto nopeasti, eli muuttuvien määritysten hyväksyminen jopa projektin loppupuolella, kuten varsin usein tapahtuu. On tärkeämpää reagoida muutoksiin kuin seurata suunnitelmaa. Sovelluksen tilaajan ja toimittajan tulee tämän vuoksi olla jatkuvassa vuorovaikutuksessa keskenään koko projektin ajan. Lisäksi myös ketterien menetelmien julistuksessa painotetaan ongelmien yksinkertaisuusvaatimusta, mutta myös muistutetaan tehokkaiden ja motivoituneiden ohjelmoijien käyttämisestä, joka perinteisesti onkin ollut eräs ohjelmistoprojektien suurimmista riskeistä. [16]

2.9. Oliotietokannat

Oliotietokannat ovat sekä akateemisen tutkimuksen, että ohjelmistoteollisuuden voimakkaan mielenkiinnon kohde. Yhteisten standardien luomiseksi on perustettu työryhmä, koska markkinoilla on jo olemassa useita kymmeniä olioiden tallennusta tukevaa tietokantajärjestelmää. Oliotietokanta tarjoaa vaatimusten paremman hallinnan, sekä parantaa tietojen jaettavuutta ja uudelleenkäytettävyyttä oliomaisen työskentelyn menetelmin.

Muodollisesti oliomaisessa tietokantasuunnittelussa työskennellään nykyisellä teknisellä tasolla usein siten, että oliopohjaisesta tekniikasta käytetään hyväksi tiedon järjestelyn menetelmiä, nimeämiskäytäntöjä ja tietojen välisten yhteyksien kuvaamiskäytäntöjä ja -järjestyksiä. Perinteisestä tietokantasuunnittelusta käytetään hyväksi tiedon eheyden

vaatimukset (kuten normalisointi) ja otetaan huomioon tietotyyppien rajoitukset. Perinteisen ohjelmoinnin menetelmistä käytetään hyväksi operaatioiden ja tietojen kuvaamiseen kehitettyjä menetelmiä. [17]

2.10. Geneettinen ohjelmointi

Arthur Samuel kirjoitti jo vuonna 1959 kysymyksen: *Kuinka tietokoneet saataisiin tekemään se, mitä halutaan, ilman että niille pitää kertoa täsmällisesti, kuinka se tehdään?* John Holland kuvasi projektissaan vuonna 1975, miten luonnollisen evoluution analogia voidaan kopioida hyödyntämään tiede- ja insinöörialojen ongelmien ratkaisemista. Tätä tekniikkaa kutsutaan nykyisin geneettisiksi algoritmeiksi. [18]

Kyseessä on laskennallinen prosessi, joka periaattessa tarkoittaa ohjelmien keinotekoista evoluutiota. Tämä tekniikka on osoittautunut onnistuneeksi erityisesti hajautetun laskennan sovelluksissa. Geneettisen ohjelmoinnin ongelmana on kuitenkin ollut tulosten esittäminen standardinmukaisissa muodoissa, jotta ohjelmien analysointi, siirtäminen ja testaus olisi mahdollista. [19] Pääongelmana on kuitenkin tarvittavan laskennan suuri määrä ja rinnakkaisjärjestelmien tekniset rajoitteet. Java-kielen käyttö ja Internetin hyödyntäminen geneettisessä ohjelmoinnissa on osoittautunut hyväksi vaihtoehdoksi, koska kielen siirrettävyysominaisuudet ovat varsin hyvät. [20]

2.11. Johtopäätökset

Ohjelmointiprosessin tehostamiseen järkevin (ja samalla yleisin) ratkaisu on jalostaa mallivetoinen kehityksen prosessia. Saatavilla on työkaluja rakennettujen mallien automaattiseen generointiin lähdekoodiksi, esim. UML-kieltä käyttävät kaupalliset suunnittelutyökalut Rational Rose ja Poseidon for UML. Testivetoisen kehityksen periaatteiden yhdistäminen mallivetoiseen kehitykseen on kuitenkin usein järkevää, koska nämä menetelmät eivät sulje toisiaan kokonaan pois. Kun pyritään ohjelmiston korkeaan laatuun, ovat ongelmien minimointi ja virheiden varhaiseen havaitseminen tärkeitä

tavoitteita. Valmiit suunnittelumallit ovat myös yksi keino päästä samaan tavoitteeseen, koska valmiiksi jäsennellyt ratkaisut standardiongelmiin on järkevin tapa tehdä ohjelmointityötä. Sellaista valmista työkalua, joka täyttäisi kaikki nämä vaatimukset, ei kuitenkaan vielä ole markkinoilla.

Mallin ja tietokannan suunnittelussa on suositeltavaa hyödyntää oliotietokannoista saatuja tutkimustuloksia, vaikka tietokantajärjestelmä olisikin perinteinen relaatiotietokanta. Oliopohjaisuuden on hyvä ilmetä tiedon järjestelyssä, tietojen välisten yhteyksien säännöissä ja nimeämiskäytännöissä, mutta perinteinen tietokantasuunnittelu eheys- ja saatavuusvaatimuksineen on myös muistettava. Tietotyyppien rajoitukset on siirrettävä tietokannan vaatimusten perusteella myös laadittavaan ohjelmistoon. Laajemmin pohtien on erityisesti nimeämiskäytäntöjen laatiminen ja noudattaminen merkittävä asia, joka tulisikin toteuttaa mahdollisimman järjestelmällisesti koko syntyvän järjestelmän laajuudella. Jos samaa asiaa tai tietoa kutsutaan samalla nimellä niin tietokannassa kuin lähdekoodissakin, ja mahdollisesti vielä käyttöliittymässäkin, voidaan katsoa ohjelmiston olevan ainakin osittain itsedokumentoituva. Tämä palvelee erityisesti muutoskohtien nopeaa paikallistamista lähdekoodista, ja tarkoittaa selvää ajan ja rahan säästöä ohjelmistoteollisuuden (tai asiakkaan) kannalta asiaa tarkasteltaessa.

Pyrkimys sellaisten henkilöiden suorittamaan ohjelmistojen valmistukseen, jotka eivät ole ohjelmoijia, ei ole kovin realistinen. Visuaaliseen ohjelmointiin kehitettyjen tekniikoiden arvo on silti suuri, koska niiden avulla voidaan tehostaa myös ammattitaitoisten ohjelmoijien työskentelyä. Yhteys asiakkaaseen saavutetaan parhaiten ketterien menetelmien julistuksen periaatteita noudattaen, mutta suorittajana on kuitenkin oltava sellainen henkilö, joka ymmärtää sekä ohjelmistojen mahdollisuudet että niiden rajoitukset (mitä on mahdollista/järkevää toteuttaa). Hyvin tärkeä peruste visuaalisen ohjelmoinnin työkalujen mahdollisimman laajaan hyödyntämiseen on niistä lähes automaattisesti seuraava dokumentaatio. Suuri ohjelmistoteollisuuden ongelmakohta on nimittän se, että lähdekoodi ja dokumentaatio eriytyvät toisistaan ohjelmistojen elinkaaren aikana.

Täydellisesti tähän vaatimukseen pystyviä työkaluja ei toistaiseksi ole markkinoilla saatavana, vaan dokumentointi on valitusta ohjelmistoarkkitehtuureista riippuen enemmän tai vähemmän ohjelmistojen valmistajien oman aktiivisuuden varassa ja siis ohjelmoijien tai heidän esimiehiensä vastuulla. Lopullinen ohjelmisto on varsin harvoin täydellinen implementaatio laaditusta mallista tai suunnitelmasta, vaan toteutusaikana niihin usein tulee vielä käytännön sanelemia muutoksia. Kokenut suunnittelija tosin osaa väistää monia muutostarpeita jo aiemman kokemuksensa peusteella.

Geneettisen ohjelmoinnin osalta on todettava, että sen teollinen hyödyntäminen on vielä toistaiseksi saavuttamaton ajatus. Automaattisen ohjelmoinnin (ja siten ohjelmointi-prosessin tehostamisen) tekniikkana se on mielenkiintoinen, mutta sen arvo tälle työlle on olematon.

3. Vaatimusmäärittely

3.1. Ongelma

Erityisesti suurten, tietokantapohjaisten ohjelmistojen toteuttamisprosessi on perinteisesti edellyttänyt suurta ohjelmointityöryhmää, vaikka näennäisesti erilliset työt voivat olla keskenään hyvinkin yhteneväisiä. Suuri työryhmä tuo yleensä mukanaan lisätyötä hallinnointi- ja tiedotustarpeidensa vuoksi, eikä ryhmän ohjelmoijien taso luultavasti ole yhtäläinen. Suunnittelumallit ja ohjelmistoarkkitehtuurit ovat suoraviivaistaneet prosessia, mutta samalla ne tuottavat lisätyötä ohjelmakoodirivien kasvavan lukumäärän muodossa.

Ketterien menetelmien julistus on hyvä lähtökohta pienessä ryhmässä tapahtuvalle ohjelmointityölle. Asiakkaan ja ohjelmointiryhmän välinen keskusteluyhteys on korostetussa asemassa, ja muutosten käsittely jopa projektin loppuvaiheessa tulisi olla helppoa. Mitä automaattisemmaksi varsinainen työ saadaan, sitä yksinkertaisemmin lähdekoodi, määritykset ja loppudokumentaatio saadaan pysymään jatkuvasti yhteneväisinä keskenään. Vaikka ohjelmoijilta voidaankin olettaa hyvää ammattitaitoa, visuaalisen ohjelmoinnin menetelmiä kannattaa hyödyntää myös heidän osaltaan. Esimerkiksi tietokantasuunnittelu on järkevää aloittaa hahmottelemalla tarvitut entiteetit ja niiden väliset suhteet sen sijaan, että tauluja kehitettäisiin vasta ohjelmoitaessa. Tämä korostuu suurissa järjestelmissä, joissa entiteettien välisiä suhteita on paljon; erityisesti dokumentoinnin vaatimukset ovat vaativat.

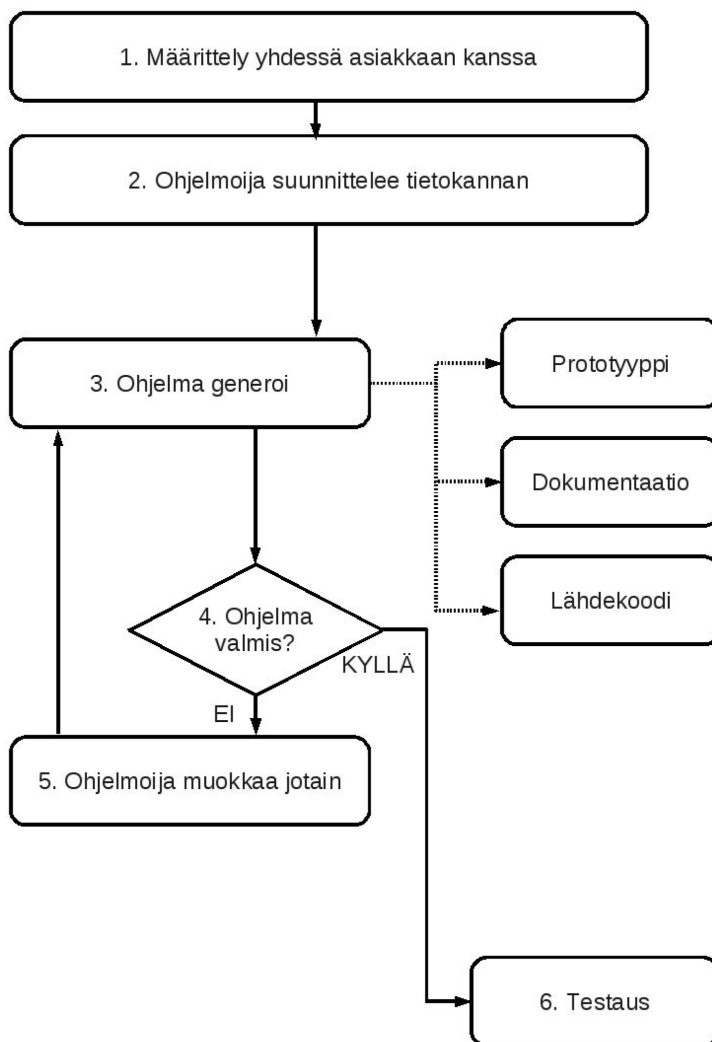
Työtä on tarkoitus jatkaa myöhemmin diplomityönä toteutuksen ja sen analyysin osalta. Diplomityön tulee toteuttaa vaatimusmäärittelyn ensimmäinen vaihe.

3.2. Tavoite

Tarkoituksena on laatia ohjelmisto, joka generoi mahdollisimman automaattisesti sellaisen kokonaisen ohjelmiston, jonka tietokannan ja perustoiminnot käyttäjä on määritellyt.

Määrittely- ja generointiprosessi on kuvattu kuvassa 5. Määrittelyn tulee sisältää ainakin syntyvän tietokannan tarkan kuvauksen mukaan lukien pää- ja viiteavaimet. Ohjelma voi ohjata käyttäjän suunnittelemien taulujen yhdenmukaisuutta lisäämällä luotaviin tauluihin standardikenttiä, tai ainakin sen on tuettava käyttöjärjestelmän leikepöydän käyttöä. Tietovarasto-tyyppisen tietokannan suunnittelu on oltava yksinkertaista ohjelmistoa hyväksikäyttäen.

Ohjelman tulee generoida ajettava ohjelmisto, joka sisältää automaattisesti ainakin

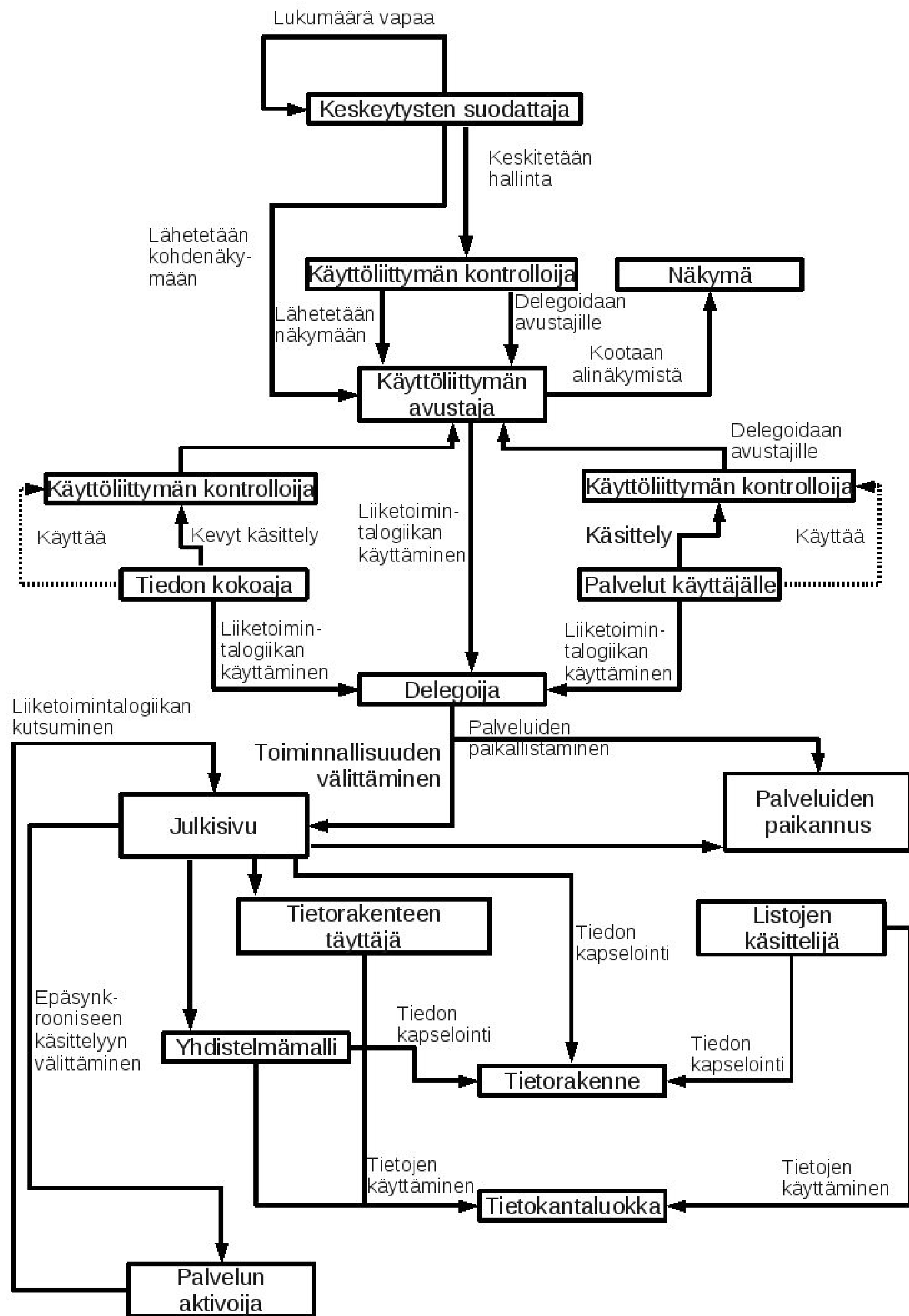


Kuva 5: Ohjelman generoinnin prosessi

normaalit tietokannan perustoiminnot, kuten uuden rivin luominen, vanhan muokkaaminen, muutosten tallentaminen tai kumoaminen, riittävät hakutoiminnot ja tietokantarivin poistaminen (tai sen piilottaminen tietovarastomaisesti). Yhden rivin muokkaaminen samanaikaisesti kahden eri käyttäjän toimesta on estettävä. Viiteavaimena toimivien taulujen osalta on ohjelmalle voitava määritellä se, käytetäänkö viiteavaimia erillisten taulujen tapaan (esim. henkilökuntaluetteloa ylläpidetään erillisessä Henkilöt-näkymässä), vai alarivien tapaan (esim. tarjoukseen lisätään tuoterivejä Tarjous-näkymässä). Jälkimmäisen tapauksen mukaisten rivien osalta generoidaan automaattisesti myös alarivien siirtotoiminnot; siis rivien keskinäistä järjestystä on oltava mahdollista muuttaa jälkikäteen. Generoitujen ohjelmistojen tulee hyödyntää mahdollisimman paljon Java Enterprise Edition -järjestelmien tunnettuja suunnittelumalleja (kuva 6). EJB-luokkien (Enterprise Java Bean) osalta käytetään standardin kolmatta versiota, koska se on selkeämpi ja kevyempi ratkaisu kuin aiempi toinen versio. [22] Generoidun ohjelmiston sovelluspalvelinosion tulee toimia siten, kuin kuvassa 7 on esitetty.

Generoidun ohjelmiston on sisällettävä peruskäyttöliittymä, jota käytetään webbiselaimella. Ulkoasu on toteutettava siten, että sen graafisia elementtejä voidaan helposti parannella jälkikäteen esim. tyylitiedostoja (engl. Cascading Style Sheet) muokkaamalla. Ensimmäisessä vaiheessa riittää, että tämä muokkaus voidaan tehdä jälkikäteen, ja tyylitiedosto sitten asentaa palvelimelle. Toisessa vaiheessa myös ulkoasun muutokset toteutetaan tietokantapohjaisesti. Käyttöliittymä toteutetaan siten, ettei generoidun ohjelman toiminnallista logiikkaa ole tarpeen siirtää siihen lukuunottamatta kevyitä javascript-koodilla toteutettavia toimintoja, kuten esimerkiksi käyttäjälle esitetty kysymys “Oletko varma” poistettaessa tietoja. Generoidun ohjelmiston käyttöliittymäosion tulee toimia siten, kuin kuvassa 8 on esitetty.

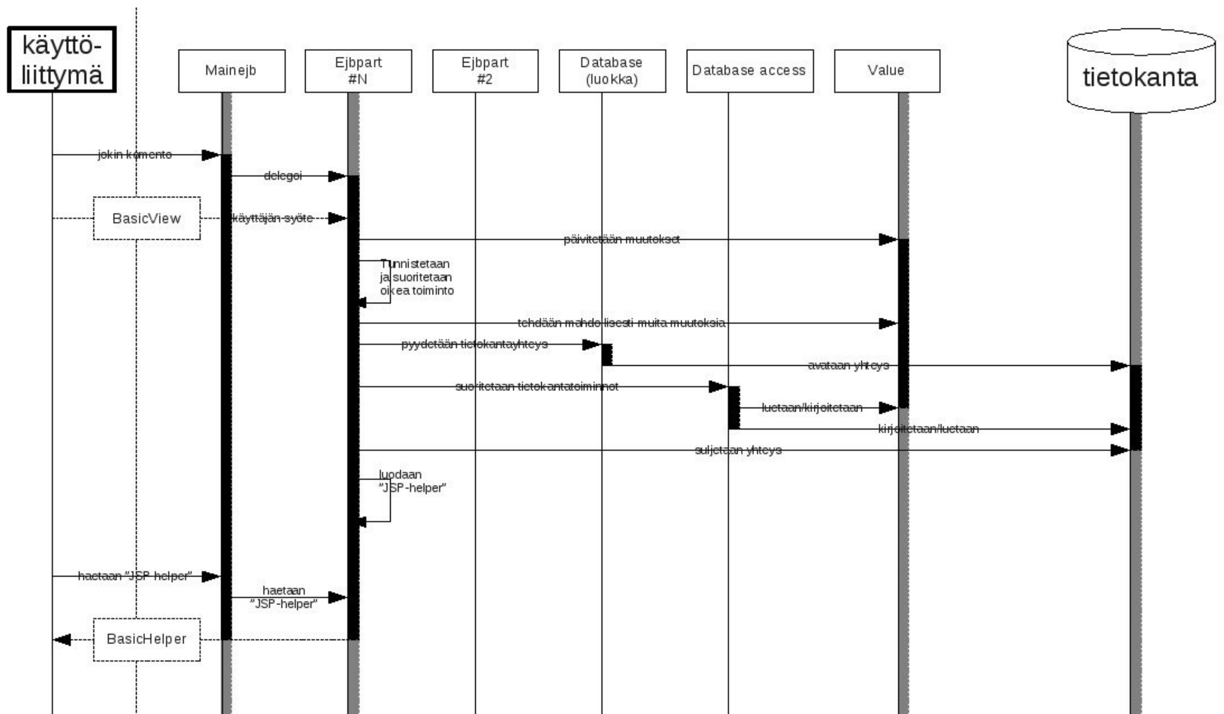
Ohjelman tulee generoida lähdekoodi jäsenneltynä kolmeen eri pakettiin: 1) *business*-pakettiin luodaan jokaista osasovellusta varten toiminnallisuuden sisältävä luokka, jota tässä kutsutaan nimellä Ejbpart. Pääohjelma toteutetaan alustatasolla toimivana EJB-



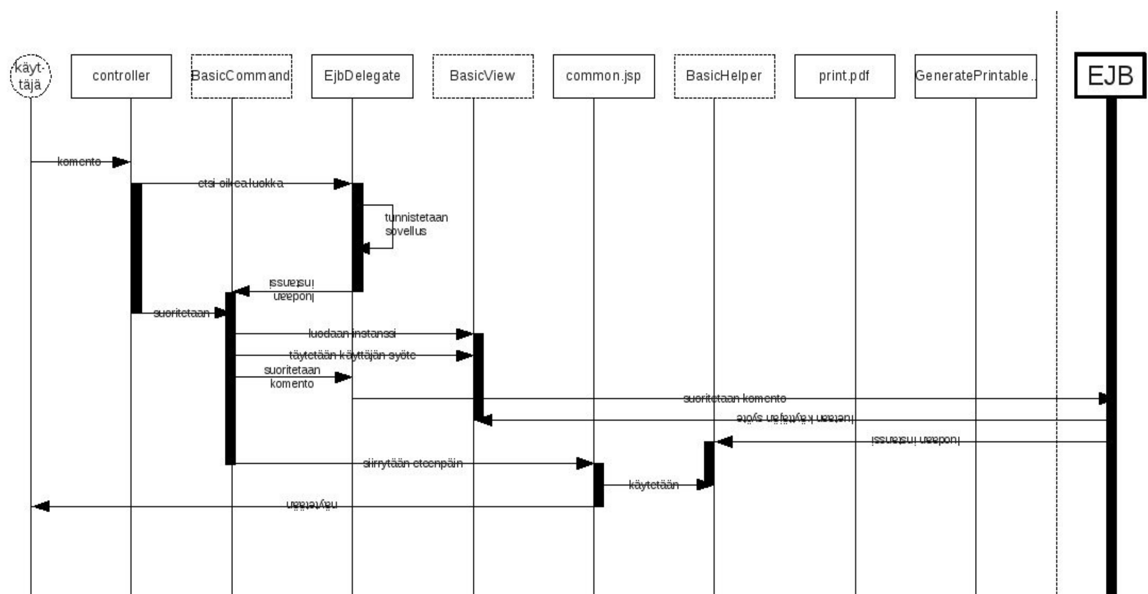
Kuva 6: Java Enterprise Edition -arkkitehtuurin tunnetut suunnittelumallit ja niiden väliset suhteet [21]

luokkana, joka on riippuvainen käyttäjän istunnosta. Tämä EJB-instanssi toimii kuitenkin vain alustana, joka delegoi tehtäviä aliluokille, joita ovat juuri nämä Ejbpart -tyyppiset luokat. Pakettiin *2)* v*iew* generoidaan kolme luokkaa jokaista näkymää varten: *JspHelper,* joka sisältää käyttöliittymän ja sen rakentamiseen tarvittavat tiedot. *View*, joka sisältää käyttäjän antaman syötteen vietäväksi sovelluspalvelimelle. *Command*, joka kerää käyttäjältä nämä tiedot ja luo View-luokan instanssin. Pakettiin *3) database* generoidaan jokaista tietokantataulua varten tietorakenneluokka, jota tässä kutsutaan nimellä *Value*. Luokka sisältää ns. set- ja get-metodit jokaista kenttää varten, sekä clone() -metodin instanssin kopiointia varten. Lisäksi generoidaan luokka, joka lukee ja siirtää tietoa tietokannan ja ohjelmiston välillä, tässä kutsuttuna nimellä *Database Access*. Huomattavaa on se, että nämä luokat ovat riippuvaisia käytetystä tietokantapalvelimesta, koska ne sisältävät SQL-kielisiä komentoja (*Structured Query Language*) ja joillekin tietokantatoiminnoille on eri tietokannoissa erilaisia komentoja (esimerkiksi automaattisesti kasvavan numeroarvon hakeminen tietokantataulun pääavaimen ainutlaatuisuuden varmistamiseksi). Nämä luokat on siis generoitava erikseen jokaista käytettävää tietokantapalvelinohjelmistoa varten.

Ohjelman generoinnin yhteydessä tulee syntyä myös mahdollisimman kattava dokumentaatio. Yleisten toimintojen osalta voidaan hyödyntää valmiiksi kirjoitettuja vakiosivuja, mutta esim. tietokantaan jälkikäteen tehtävien muutosten osalta dokumenttien on päivityttävä reaaliaikaisesti. Dokumentaation tulee koostua ainakin tietokannan ER-kaaviosta, tietokannan kenttien kuvauksesta (käyttäjän määrittelyn perusteella), käyttötapauskaaviosta (engl. Use Case) ja kuvauksen generoituun ohjelmaan liittyvistä käyttöoikeuksista (käyttäjän määrittelyn perusteella). Toisessa vaiheessa on ohjelman dokumentaatioon voitava liittää mikä vain ulkopuolinen tiedosto. Koska kaaviot generoidaan automaattisesti, on niiden saaminen selkeäksi haaste. Taulujen lukumäärä voi myös kasvaa suureksi. Ohjelman generoiman ER-kaavion ulkonäkö ei sen vuoksi ole merkittävä tekijä. Taulut voidaan asetella aakkosjärjestykseen, ja taulujen väliset yhteydet piirtää siten, että niiden tulkinta on yksiselitteistä. Esimerkki tällaisesta ER-kaaviosta on

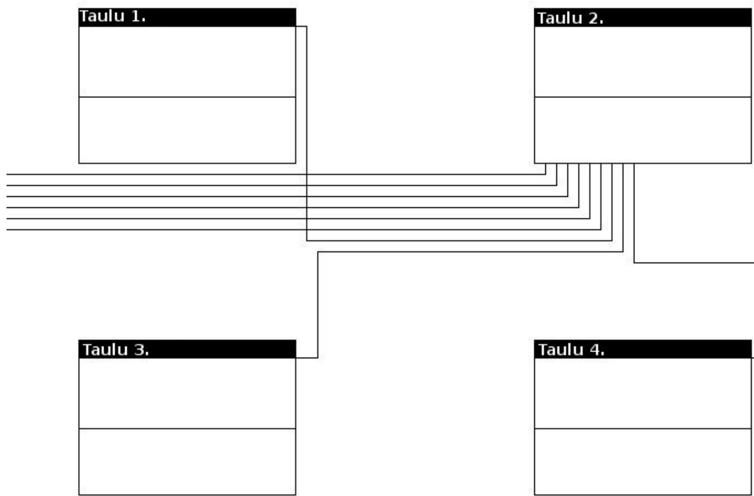


Kuva 7: Generoidun ohjelmiston toimintamalli sovelluspalvelimessa



Kuva 8: Generoidun ohjelmiston toimintamalli käyttöliittymäpalvelimessa

esitelty kuvassa 9, ja kuvassa 10 havainnollistettu esimerkin toimivuutta suurehkossa järjestelmässä (67 taulua). Esimerkki yhden tietokantataulun kenttäkuvauksesta on esitelty kuvassa 11 sillä tasolla, kuin sen tulee olla. Lisäksi ohjelman on generoitava valmiit SQL-lauseet tietokannan luomiseksi ja dokumentoimiseksi.

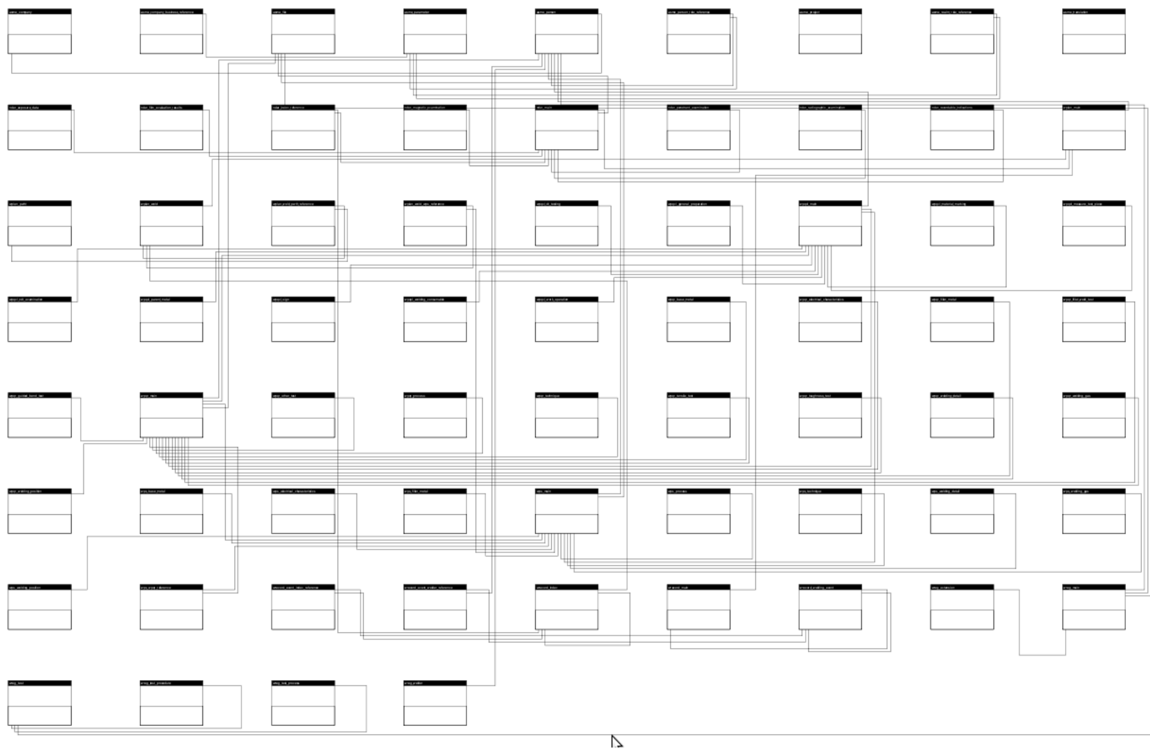


Kuva 9: Esimerkki ER-kaaviosta. Taulujen väliset yhteydet kulkevat eri linjassa toisiinsa nähden riippumatta

3.3. Rajaukset

Ohjelman käyttäjällä oletetaan olevan kohtuullisen hyvä ymmärrys tietokoneista ja erityisesti tietokantojen suunnittelusta. Ohjelma ei ota kantaa mahdollisten suunnitteluvirheiden syntymiseen, mutta lähtökohtaisesti pyritään kuitenkin oikeaoppiseen tietokantaan; esimerkiksi tiedon denormalisointi on käyttäjän itse erikseen määriteltävä.

Ensimmäisessä vaiheessa ohjelma toteutetaan vain komentorivipohjaisena eräajosovelluksena, joka lukee syötteensä vakiomuotoisesta ohjaustiedostosta. Toisessa vaiheessa se



Kuva 10: Esimerkki ER-kaaviosta suurehkossa tietojärjestelmässä

person			**Implements one person in the system**
id	numeric (10, 0)	not null	**Primary key**
deleted	numeric (1, 0)	not null	Database row is deleted (1), it isn't (0)
last_saved_date	date		Date of the last save
last_saver_reference	numeric (10, 0)		*References to person*
last_saver_name	varchar2 (102)		Last saver name (denormalized information) lastname || ', ' || firstname
employee_number	varchar2 (10)		Employee number in
last_name	varchar2 (50)		Last name
first_name	varchar2 (50)		First name
user_name	varchar2 (20)		User name
password	varchar2 (20)		Password
birth_date	date		Date of birth
birth_place	varchar2 (50)		Birthplace
company_reference	numeric (10, 0)		*References to company*
superior_reference	numeric (10, 0)		*References to person*
street_address_1	varchar2 (50)		Street address, row 1
street_address_2	varchar2 (50)		Street address, row 2
street_zip	varchar2 (10)		Street address, ZIP code
street_city	varchar2 (50)		Street address, location
pobox_address	varchar2 (50)		P.O. Box number
pobox_zip	varchar2 (10)		P.O. Box, ZIP code
pobox_city	varchar2 (50)		P.O. Box, location
phone_number	varchar2 (50)		Phone number
fax_number	varchar2 (50)		Fax number
email_address	varchar2 (50)		E-mail address

Kuva 11: Esimerkki tietokantataulun kuvauksesta. Samat kommentit siirtyvät myös lähdekoodiin ja tietokantaan, jonka vuoksi kommentointikielenä tultaneen käyttämään englantia.

sisältää graafisen käyttöliittymän joko ikkunointitekniikalla tai webbiselain-käyttöliittymällä toteutettuna. Toisen vaiheen ohjelma tallentaa tietonsa joko tiedostoon/hakemistoon, tai vaihtoehtoisesti tietokantaan. Tiedon on oltava varmistettavissa, mutta sen ei tarvitse olla siirrettävissä. Käyttäjän on voitava lisätä uusia metodeja standardimetodien lisäksi siten, että ne näkyvät lähdekoodissa ja loppudokumentaatiossa, mutta varsinainen ohjelmointi voidaan tehdä ulkoisen editorin avulla. Kolmannessa vaiheessa ohjelman on tarjottava käyttäjälle mahdollisuus metodien muokkaamiseen suoraan lähdekooditasolla ja neljännessä vaiheessa ohjelman tulee huolehtia myös mahdollisten muutosten versionhallinnasta. Tähän voidaan käyttää avuksi jo olemassa olevia tekniikoita, kuten CVS (Concurrent Version System).

Generoitujen ohjelmien voidaan olettaa toimivan mallivetoisen toteutuksen tavoin. Viimeistään ohjelman kolmannessa vaiheessa on kuitenkin huomioita myös testivetoisen kehityksen tarpeita siten, että generoidun ohjelman testaus on mahdollista suunnitella etukäteen, ja se liitetään ohjelman dokumentaatioon automaattisesti.

3.4. Toimintaympäristöt

Generointiohjelmisto pyritään laatimaan riippumattomaksi käyttöjärjestelmistä. Ohjelman luonteen vuoksi voidaan olettaa sen toimivan hyvin myös pienitehoisilla tietokoneilla, tai olevan jopa selainkäyttöinen.

Ohjelman generoimat ohjelmat laaditaan toimiviksi Java Enterprise Edition -ympäristöissä mahdollisimman standardinmukaiseksi paremman yhteensopivuuden aikaansaamiseksi. Minimiedellytys on generoidun ohjelman toiminta JBoss 5.0 -palvelimella, Oracle 10 XE -tietokantaa hyväksikäyttäen. Ohjelmistovalinnat on tehty perusteena ohjelmien ilmaisuus.

Ohjelmille voidaan laatia yhteinen toimintaympäristö (engl. platform), joka sisältää esim. yhteiset tietoturva- ja tunnistautumistoiminnot. Tämä toimintaympäristö kopioidaan tarpeen mukaan generoitujen ohjelmien mukaan sikäli kun niitä ajetaan erillisillä

palvelimilla. Generoitujen ohjelmien on kuitenkin toimittava siten, että samalla palvelimella ajettavat, samaan tietojärjestelmään kuuluvat erilliset ohjelmat voidaan asentaa yksitellen omissa EAR-paketeissaan (Enterprise Archive). Tällä pyritään siihen, että suuren järjestelmän kaikki ohjelmistot eivät ole poissa käytöstä samanaikaisesti. Jos vain yhtä korjataan, vaan ohjelmoijat voivat työskennellä vain yhden ohjelman kanssa kerrallaan muiden osioiden jatkaessa toimintaansa normaalisti.

3.5. Jatkokehitys

Syntyvää ohjelmaa voidaan myöhemmin laajentaa siten, että se tukee paremmin ryhmätyöskentelyä. Koska tavoitteena on työryhmän pienentäminen, tätä voidaan kuitenkin pitää prioriteetiltaan varsin toisarvoisena tavoitteena. Ryhmätyöskentelyominaisuuksiin voidaan yhdistää projektin hallinnointityökaluja, kuten resurssointi, ohjelmoijien käyttämien tuntimäärien seuranta (jopa automaattisena toimintona) sekä näistä generoituna raportteja esim. projektin etenemisestä. Liian paljon ei voi kuitenkaan painottaa tarvetta pitää ohjelmisto yksinkertaisena, jotta se palvelisi päätarkoitustaan: projektiryhmän koon pienentämistarvetta.

4. Yhteenveto

Ohjelmointitekniikoiden kehittyminen on mahdollistanut ylläpidettävyydeltään erinomaisten ohjelmointiarkkitehtuurien syntymisen. Kielteisenä ilmiönä on kuitenkin samalla tullut lähdekoodin rivien lukumäärän kasvaminen. Monet ohjelmistoarkkitehtuuriin liittyvät lähdekoodirivit ovat kuitenkin standardimaisia, ja toisaalta varsin monet muutkin tavanomaiseen ohjelmointiin liittyvät tehtävät ovat automatisoitavissa. Mallivetoisen kehityksen työkaluihin on toteutettu lähdekoodia generoivia toimintoja, mutta ongelmana on dokumentaation ja lähdekoodin eriytyminen toisistaan ohjelmointiprosessin edetessä. Ohjelmien elinkaari on usein varsin pitkä, ja erityisesti ylläpitovaiheessa dokumentaation päivittäminen usein lyödään laimin. Automaattinen yhteys lähdekoodin ja dokumentaation välillä olisi ratkaisu tähän, mutta nykyiset työkalut ovat näiltä osin varsin puutteellisia.

Tässä työssä tutkittiin olemassa olevia tekniikoita koko ohjelmointiprosessin tehostamiseen. Tutkimus koski sekä varsinaisia suunnittelu- ja ohjelmointitekniikoita, että tapoja helpottaa tietokoneiden loppukäyttäjien mahdollisuutta vaikuttaa ohjelmistoihin. Visuaaliset ohjelmointikielet on kehitetty juuri tästä näkökulmasta: mahdollistamaan sellaisten henkilöiden kommunikointi tietokoneen kanssa, jotka eivät ole ohjelmoijia. Työssä oletetaan kuitenkin, että parhaiten tämä kommunikointi on toteutettavissa ammattitaitoisen ohjelmoijan välityksellä, mutta työskentelyn nopeuttaminen ja erityisesti ketterien menetelmien julistuksen asettamat haasteet ovat tärkeä keino lähestyä alkuperäistä tavoitetta.

Työn tarkoituksena oli tutkia, miten ohjelmointiprosessiin osallistuvan työryhmän kokoa voidaan pienentää automatisoinnin välinein, tavoitteena vaatimusmäärittely ohjelmointityön automatisointia varten. Vaatimuksia on esitetty sekä lähdekoodin, että automaattisesti generoidun dokumentaation osalta. Työ on rajattu Java Enterprise Edition -ympäristöön. Tätä ympäristöä on tutkittu paljon, joten suunnittelumalleja ja jopa valmiita arkkitehtuurikirjastoja on saatavana riittävästi. Niinpä sekä generoidun lähdekoodin, että

dokumentaation osalta voidaan todeta nykyisten, tunnettujen ja tutkittujen tekniikoiden riittävän vaatimuksiksi. Ratkaistavat haasteet koskevatkin siis automaation osuutta, lopputulos, johon pyritään, on sen sijaan jo tunnettu.

Työtä on tarkoitus jatkaa diplomityönä, jolloin vaatimusmäärittelyn täyttävä ohjelmisto toteutetaan. Vaatimusmäärittely on jaettu kolmeen eri kehitysvaiheeseen, ja diplomityön osuutta on sen ensimmäinen vaihe. Ohjelman toinen ja kolmas kehitysvaihe, samoin kuin kaikki muu myöhemmin tapahtuva jatkokehitys voivat olla pohjana tutkimukselle, tai jopa työssä syntyvän ohjelmiston kaupallistamiselle.

Lähteet

1. Schach, S.R. (2002): Classical and Object-Oriented Software Engineering, 5th edition

2. Barry W. Boehm: Industrial Software Metrics: A Top -Ten List, 1987

3. Jarkko Sikiö: Ohjelmistotuotannon toteutusmenetelmien yhtenäistäminen, diplomityö Lappeenrannan teknillinen yliopisto 2005

4. Barry W. Boehm: Improving Software Productivity, Computer-lehti, syyskuu 1987

5. Dragos Manolescu, Markus Völter, James Noble: Pattern Languages of Program Design 5, Pearson Education Inc. 2006

6. Md. Abul Khaer, M.M.A. Hashem, Md. Raihan Masud: On Use of Design Patterns in Empirical Assessment of Software Design Quality, Proceedings of the International Conference on Computer and Communication Engineering, 2008

7. Carola Lilenthal: Architectural Complexity of Large-Scale Software Systems, European Conference on Software Maintenance and Reengineering, IEEE 2009

8. Michael Gutkowski, Jaroslaw Wojciechowski, Bartosz Sakowicz, Andrzej Napieralski: Thesis Management Supporting System based on J2EE Platform, CADSM'2007, 20.-24.2.2007, Polyana, Ukraina

9. Salvador Trujillo, Don Batory, Oscar Diaz: Feature Oriented Model Driven Development: A Case Study for Portlets, 29th International Conference on Software Engineering 2007

10. Markus Scheidgen: Model Patterns for Model Transformations in Model Driven Development, Proceedings of the 4th Workshop on Model-Based Development of

Computer-Based Systems and 3rd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software 2006

11. Detlef Streitferdt, Georg Wendt, Philipp Nenninger, Alexander Nyßen†, Horst Lichter†: Model Driven Development Challenges in the Automation Domain, Annual IEEE International Computer Software and Applications Conference 2008

12. Lasse Koskela: Test Driven, Manning Publications Corporation 2008

13. Robert Balzer: A 15 Year Perspective on Automatic Programming, IEEE Transactions on Software Engineering, vol. se-11, no. 11, marraskuu 1985

14. Nan C. Shu: A Visual Programming Language Designed for Automatic Programming, IEEE 1988

15. Alice L. Schafer: Graphical Interactions with an Automatic Programming System, IEEE Transactions on Systems, Man and Cyvernetics, vol. 18, no 4, heinä-elokuu 1988

16. Jari Parantainen: Tuotteistaminen - rakenna palvelusta tuote 10 päivässä, Talentum Media Oy 2007

17. Jim Davies, James Welch, Alessandra Cavarra, Edward Crichton: On the Generation of Object Databases using Booster, 11th IEEE Conference on Engineering of Complex Computer Systems (ICECCS'06)

18. John R. Koza, Forrest H. Bennett III, David Andre, Martin A. Keane: Four Problems for which a Computer Program Evolved by Genetic Programming is Competitive with Human Performance, IEEE 1996

19. Thomas Weise, Michael Zapf, Mohammad Ullah Khan, Kurt Geihs: Genetic

Programming meets Model-Driven Development, 7th International Conference on Hybrid Intelligent Systems, IEEE 2007

20. A.M.S.Zalzala, D.Green: MTGP: A Multithreaded Java Tool for Genetic Programming Applications, IEEE 1999

21. Deepak Alur, John Crupi, Dan Malks: Core J2EE Patterns, Best Practices and Design Strategies, Sun Microsystems Press 2001

22. Debu Panda, Reza Rahman, Derek Lane: EJB 3 in Action, Manning Publications Co. 2007