

Vaatimusmäärittely ja automatisoitu testauskehys

Ari Heinonen

Helsinki 29.10.2009

Pro gradu -tutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Tiedekunta/Osasto – Fakultet/Sektion – Faculty/Section		Laitos – Institution – Department	
Matemaattis-luonnontieteellinen tiedekunta		Tietojenkäsittelytieteen laitos	
Tekijä – Författare – Author			
Ari Heinonen			
Työn nimi – Arbetets titel – Title			
Vaatusmääritys ja automatisoitu testauskehys			
Oppiaine – Läroämne – Subject			
Tietojenkäsittelytiede			
Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages	
Pro gradu -tutkielma	29.10.2009	69	
Tiivistelmä – Referat – Abstract			
<p>Ohjelmiston laatuun liittyvien ongelmien ratkaisemiseen käytetään ohjelmistotuotannossa paljon aikaa. Järjestelmälliseen laadun parantamiseen pyritään prosessimalleilla, joita on kehitetty 70-luvulta lähtien. Testaaminen on tavallisin tapa ohjelmiston laadun varmistamiseen.</p> <p>Testaukseen käytetään merkittävä osa ohjelmistokehityksen resursseista. Suuri osa testaamisesta suoritetaan ohjelmistokehityksen loppupuolella, jolloin virheiden korjaaminen on kallista. Viimeisenä testivaiheena suoritettavaan hyväksymistestaukseen on olemassa ohjelmistokehitys Fit, jolla vaatimukset kerätään projektin alkuvaiheessa testitapauksiksi. Ohjelmistokehitys saa tarkemman käsityksen halutusta toiminnallisuudesta, kun vaatimukset on tallennettu yksikäsitteisessä muodossa testitapauksiksi.</p> <p>Vaatusmääritys koostuu neljästä osasta; vaatimusten kartuttamisesta, analysoinnista, spesifioinnista ja validoinnista. Fit -menetelmässä spesifiointivaiheessa vaatimukset tallennetaan taulukkomuotoon. Taulukossa kartutus- ja analyysivaiheessa käsitelty tieto kootaan yksiselitteiseen muotoon esitettäväksi skenaarioina taulukon riveillä.</p> <p>Hyväksymistestit voidaan suorittaa automaattisesti kehittäjien taulukoihin laatimien testikerrosten avulla. Näin testitapaukset ovat kattavat ja automaattisesti suoritettavissa sekä uudelleen toistettavissa myöhemmin regressiotestauksessa.</p>			
ACM Computing Classification System (CCS):			
D.2.1 [Requirements/Specifications]			
D.2.5 [Testing and debugging]			
Avainsanat – Nyckelord – Keywords			
Hyväksymistestaus, vaatimusmääritys, testauskehys, Framework for Integrated Testing, FIT			
Säilytyspaikka – Förvaringställe – Where deposited			
Muita tietoja – Övriga uppgifter – Additional information			

Sisältö

1 Johdanto	1
2 Ohjelmistoprosessien mallit	3
2.1 Vesiputousmalli.....	3
2.2 V-malli.....	5
3 Vaatimusmäärittely	10
3.1 Vaatimusmäärittelyn vaiheet.....	12
3.2 Vaatimusten luokittelu.....	18
3.3 Vaatimusten spesifiointi ja taulukot.....	19
4 Hyväksymistestaus osana ohjelmistoprosessia	22
4.1 Ohjelmistotestaus ja laatu.....	23
4.2 Hyväksymistestaus osana V-mallia.....	24
4.3 Kombinatorinen testaus.....	25
4.4 Taulukot testitapausten esittämisessä.....	25
4.5 Automatisoitu testaus.....	26
5 Vaatimusmäärittely ja testitapausten suunnittelu – Framework for Integrated Testing (FIT)	32
5.1 FIT ja vaatimusmäärittelyn ongelmat.....	34
5.2 FIT:n rakenne.....	34
5.3 FIT:n rajoitteet.....	38
6 Ohjelmistokehys vaatimusmäärittelyn ja hyväksymistestitapausten automatisoimiseksi	40
6.1 Ohjelmistokehykset.....	40
6.2 Testauskehysten arkkitehtuuri.....	42
7 Ohjelmistokehysten analysointi	54
7.1 Kehysten arviointi.....	55
7.2 Mittarit.....	59
8 Yhteenveto	61
Lähteet	65

1 Johdanto

Ohjelmistojen virheistä 85 % johtuu puutteellisista määrittelyistä [MRM04], testaukseen käytetään 60 % ohjelmistokehityksen resursseista [HiT00] ja ohjelmistokehityksen hyväksymistestausvaiheessa ongelmien korjaaminen on 50 kertaa kalliimpaa kuin, jos ongelma havaitaan määrittelyvaiheessa [HiT00]. Puutteelliset vaatimukset ja testitapaukset sekä testaamiseen käytetty aika ovat ongelmia, jotka motivoivat perehtymään vaatimusmäärittelyn ja testaamisen aihepiiriin.

Ensimmäisessä luvussa käsitellään lyhyesti ohjelmistoprosessien taustaa ja esitellään v-malli, joka on ensimmäinen ohjelmistotestauksen keskeiseen asemaan nostanut prosessimalli. Malli on edelleen tavallisesti käytetty ja vastaavia ominaisuuksia on havaittavissa myös myöhemmin esitellyissä prosessimalleissa.

Toinen luku perehtyy v-mallin aloitusosioon eli vaatimusmäärittelyyn. Vaatimusmäärittely jakautuu kartutus-, analysointi-, spesifointi- ja validointivaiheisiin. Näissä osissa ongelmasta koostetaan ohjelmoimalla ratkaistavissa oleva kokonaisuus. Spesifointivaiheessa määritellään ongelman ja sen sidosryhmien liittymät toteutukseen. Huolellisesti tehty vaatimusmäärittely ja tiedon perusteellinen kerääminen vähentävät myöhemmässä kehityksessä vastaantulevia yllätyksiä. Tämän tutkielman tavoitteena on osoittaa, että spesifointivaiheessa vaatimukset on mahdollista esittää riittävällä täsmällisyydellä, jotta hyväksymistestaaminen on mahdollista automatisoida. Automatisointi ei riitä ratkaisemaan puutteellisten vaatimusten ongelmaa, mutta hypoteesina on, että täsmällisiä vaatimustapauksia tehtäessä havaitaan paremmin mahdollisesti vastaan tulevat ongelmat.

Kolmas luku keskittyy hyväksymistestaukseen. Uusien ominaisuuksien testaaminen vaatii paljon resursseja, mutta vastaavasti olemassa olevien toimintojen regressiotestaukseen on varattava aikaa. Päällekkäiseltä työltä vältytään, kun testitapaukset tallennetaan ja suoritetaan automaattisesti. Ohjelmistokehityksessä aikaa käytetään vaatimusten mallintamiseen, testitapausten suunnitteluun ja testien suorittamiseen. Edellä mainitut tehtävät käsittelevät käyttäjien vaatimuksia eri näkökulmista. Ohjelmistokehityksessä on mahdollista säästää aikaa, kun vaatimukset mallinnetaan vaatimusmäärittelyn yhteydessä muotoon, jota voidaan soveltaa testitapausten yhteydes-

sä, suorittaa testaus automaattisesti ja tallentaa testitapaukset jatkokäyttöä varten. Kustannuksena on lisääntynyt ohjelmointityö vaatimusmäärittelyvaiheessa, mutta oletuksena on, että lisääntynyt ohjelmointityö maksaa itsensä takaisin pienempänä työ-määränä hyväksymistestauksessa sekä myöhemmin regressiotestauksessa. Takaisinmaksuaika kehitetylle testiohjelmalle on tyypillisesti 2-3 kehityskierrosta [Bin99].

Neljännessä osassa esitellään Framework for Integrated Testing (Fit) -ohjelmistokehys hyväksymistestien automatisoimiseen. Tässä kappaleessa käydään läpi Fit:n toiminnallisuutta käyttäjille sekä kehittäjille. Fit koostuu kolmesta eri testityypistä, joilla sovellusta voidaan testata periyttämällä Fit:n testikerroksia (fixture). Kappaleessa esitellään esimerkkejä testitapauksista, joita Fit:llä voidaan testata.

Viidennessä osassa esitellään Fit:stä johdettu ohjelmistokehys FitX. Ohjelmistokehyksestä esitetään arkkitehtuuri komponenttien tasolla sekä komponenttien välinen vuorovaikutus. Lisäksi oleellisista komponenteista on esitetty uml -kaavioilla staattinen sekä dynaaminen rakenne. FitX -kehyksellä pyritään parantamaan Fit:ssä havaittuja ongelmia. Kehyksen käyttöönottoa on tuettu paremmalla kehitysympäristötuella. FitX tukee sovelluksen testaamista kiinteänä osana perinnejärjestelmäympäristöä. Lisäksi FitX tukee jäsenneltyjä testitapauksia sallimalla viittauksia rakenteellisiin testitapaus-tauluihin.

Viimeisessä osassa arvioidaan ohjelmistokehityksen toteutusta ja analysoidaan sen heikkouksia ja vahvuuksia erilaisissa ohjelmointitehtävissä. Tukena käytetään vaatimusmäärittelyssä sovellettuja ongelmakehityksiä. Lisäksi kappaleessa esitellään mit-tareita, joilla valmista ohjelmistokehystä voidaan arvioida.

Ohjelmiston vaatimusmäärittelyssä voi jäädä havaitsematta ominaisuuksia tai ympäristö voi vaatimusmäärittelyiden jälkeen muuttua, jolloin tarvitaan uusia ominaisuuksia. Muuttuvien vaatimusten ongelmaa ei voida ratkaista, mutta puutteet voidaan minimoida huolellisella käytännön työllä ja tulevaa ennakoimalla. Vaatimusten keräämiseen ja mal-lintamiseen käytettävää työmäärää on mahdollista pienentää ja tiedon ylläpitoa helpottaa. Hypoteesina on, että vaatimusmäärittelysten yhteydessä laaditut testitapaukset vähentävät puutteita vaatimuksissa ja nopeuttavat testaamista. Käyttäjien tekemällä käyttöliittymätestillä ja ohjelmistokehittäjien yksikkötestauksella on edelleen suuri mer-kitys. Vaatimukseen vastaamisen osalta hyväksymistestauksen automatisoinnilla on

paljon annettavaa. Regressiotestauksessa taulukkojen avulla toteutettu ohjelmistotestaus lisää ohjelmiston luotettavuutta ja vähentää testaamiseen käytettävää työmäärää.

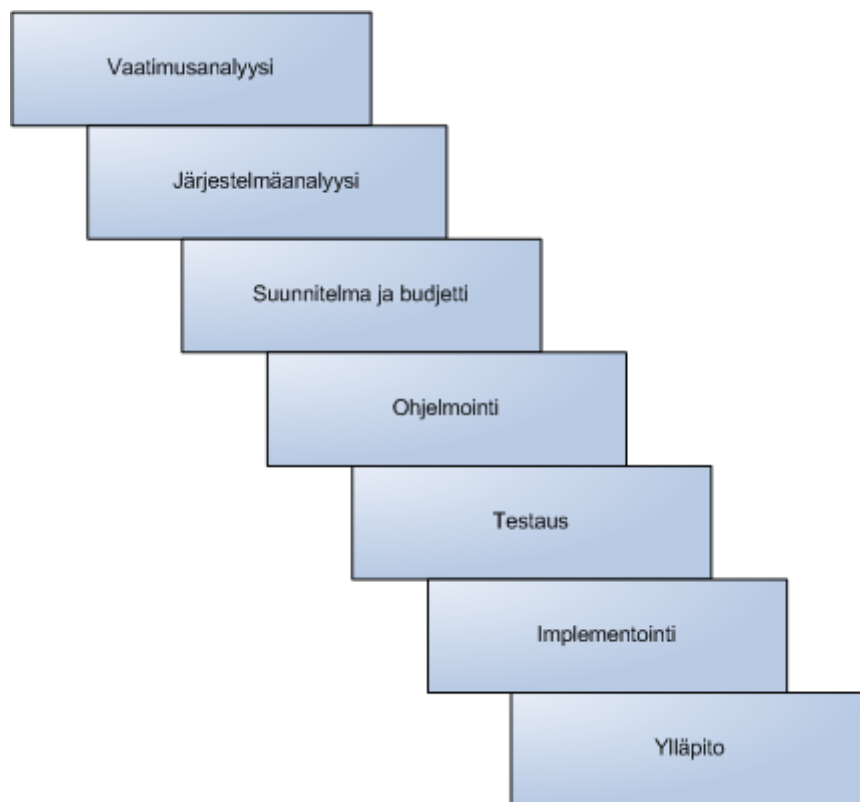
2 Ohjelmistoprosessien mallit

Ohjelmistoilla hallitaan liikennettä, kaupankäyntiä, terveydenhuoltoa, yrityksiä ja koulutusta, jotka kaikki ovat yhteiskunnan kannalta tärkeitä toimintoja. Viimeisen vuosikymmenen aikana ohjelmistot ovat muuttuneet entistä laajemmiksi kokonaisuuksiksi, joiden kehitykseen käytetään suuret määrät resursseja. Suureksi osaksi ohjelmistojen nopean kehittymisen on mahdollistanut laitteiden tekninen kehitys, jonka ansiosta suorittimet ovat pystyneet toteuttamaan samassa ajassa entistä enemmän toimintoja. Sen sijaan ohjelmistotuotantoa on pidetty kehityksen jarruna, koska ohjelmistokehitys ei ole pystynyt tehokkaasti käyttämään hyväkseen laitteiden kehitystä ja on olemassa suuri joukko esimerkkejä epäonnistuneista, viivästyneistä ja perusteellisesti väärin budjetoiduista ohjelmistotuotantoprojekteista [HiT00]. Tätä ohjelmistotuotannon ongelmaa on pyritty korjaamaan ohjelmistotuotannon malleilla, jotka tuovat näkökulmia siihen, kuinka asiakkaalta kerätyt vaatimukset tulee muuntaa kuvauksiksi, ohjelmaksi ja lopulta toimivaksi tuotteeksi.

2.1 Vesiputousmalli

Vesiputousmalli on ensimmäinen ohjelmistotuotantoon julkaistu prosessimalli [Roy70]. Se on johdettu muissa insinööritieteissä käytetyistä malleista ja perustuu vesiputouksen mukaiseen etenemiseen vaiheesta toiseen, jossa iterointi aikaisempien vaiheiden jälkeen on mahdollista vasta järjestelmän ylläpitovaiheessa. Käytännössä vesiputousmallin jokaisessa vaiheessa dokumentointi allekirjoitetaan hyväksytyksi, minkä jälkeen prosessissa edetään seuraavaan vaiheeseen. Vesiputousmallin hyvänä puolena on selkeä työnjako prosessin vaiheiden välillä ja jokaisesta vaiheesta tuotettu dokumentaatio, mikä vastaa yleisesti insinööritieteissä tunnettua mallia [Som07]. Huonona puolena on sen joustamattomuus. Vesiputousmallissa projekti päättyy usein tilanteeseen, jossa valmista tuotetta testattaessa ja otettaessa tuotantoon havaitaan, ettei tuote tee mitään sen toivottiin tekevän. Sen vuoksi ylläpidossa palataan prosessin aikaisempiin vaiheisiin ja korjataan tekemättä jääneitä asioita. Tämä johtaa helposti hankaliin ja tehottomiin ohjelmiston rakenteisiin ja tyytymättömiin käyttäjiin [Som07].

Vesiputousmallissa aloitetaan käyttäjien vaatimusten analysoinnista, jotka kuvataan järjestelmäanalyysivaiheessa toteutettaviksi komponenteiksi. Järjestelmäanalyysin jälkeen pystytään arvioimaan toteutuksen vaatimat resurssit, aika ja työmäärä. Ohjelmointi aloitetaan suunnitelmien pohjalta. Ohjelmoinnista tulevat osat siirtyvät testattaviksi. Testaamisessa pyritään osoittamaan, että ohjelma toteuttaa sen, mitä aikaisemmin on suunniteltu. Testauksessa havaitut muut kuin ohjelmavirheet siirretään seuraavan vesiputouksen käsiteltäväksi ylläpidossa. Testauksen jälkeen ohjelma implementoidaan käyttöön ja siirretään ylläpidettäväksi. Jokaisen vaiheen jälkeen tuotetaan kattava dokumentaatio, joka pysyy konsistenssina, koska aikaisempiin vaiheisiin ei enää palata ennen ylläpitoa [Gil85].



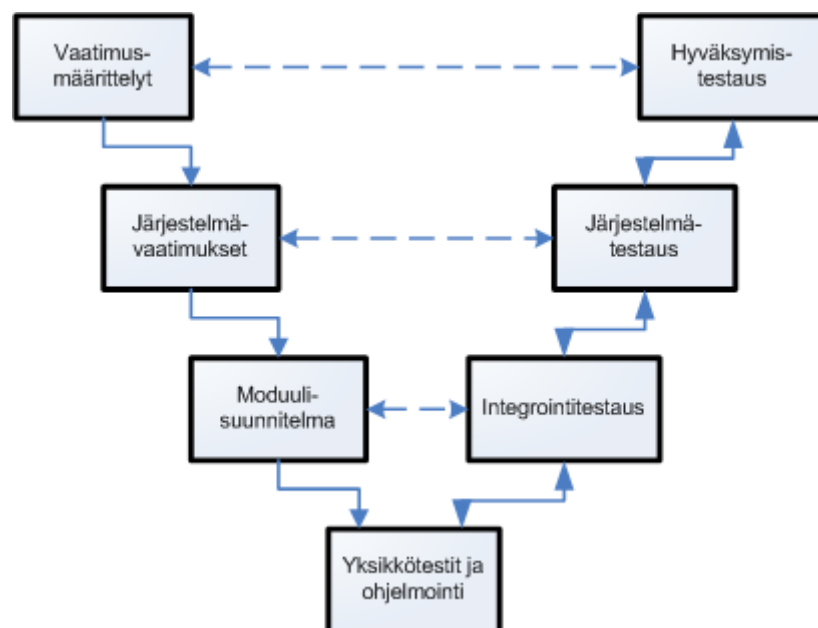
Kuva 1: Vesiputousmalli [Gil85]

Vesiputousmallista on useita variaatioita, mutta kaikille niille on tyypillistä yksityiskohdainen suunnittelu ennen ohjelmointia sekä yhteen ennalta määrättyyn implementointipäivään tähtääminen [Gil85]. Vesiputousmallin tyypillinen ongelma on, että ohjelman sidosryhmät ovat vaatimusten asettamisen jälkeen täysin ohjelmistokehitysryhmän armoilla. Vesiputousmallissa kestää usein kuukausia tai jopa vuosia, ennen kuin käyttäjät näkevät valmiin tuotteen. Testausvaiheessa muutokset arkkitehtuuriin tai logiikkaan vaarantavat koko järjestelmän toteutuksen. Kehittäjät eivät pitkissä projek-

teissa välttämättä näe käyttäjiä tai asiakkaita, eivätkä siis osaa eläytyä ohjelmiston käyttäjän rooliin [Gil85]. Malli soveltuu parhaiten laajojen ongelmien ratkaisuihin, joissa vaatimukset tunnetaan hyvin etukäteen. Malli on edelleen yleisesti käytössä, mutta siitä on lukuisia variaatioita ja kehityssuuntia joustavampiin prosesseihin.

2.2 V-malli

V-malli on perinteisen vesiputousmallin laajennus, jonka keskeisenä tarkoituksena on tuoda esille ohjelmiston suunnittelun ja testauksen välinen ero ohjelmistokehityksessä [Mye79]. V-mallin etuihin kuuluu sen helppokäsitteisyys. V-mallin vasemmalla puolella keskitytään ohjelmiston perinteiseen vesiputousmallin osaan eli suunnitteluun ja toteutukseen.



Kuva 2: V-malli

Oikealla puolella keskitytään tuotteen testaamiseen ja laadunvarmistamiseen. Vasemmalta puolelta valmistunutta ohjelmaa testataan kyseisen abstraktiotason osalta. Yksiköiden yhdistäminen kokonaisuuksiksi on luotettavaa, kun testaamalla on osoitettu matalamman abstraktiotason osat laadukkaiksi.

2.2.1 V-malli ja ohjelmiston laatu

V-mallin soveltamisen tavoitteena on saavuttaa parempilaatuinen ohjelmisto testaamalla

ohjelmistoa jatkuvasti sen eri kehitysvaiheissa. Ohjelmiston tavoiteltavaan laatuun vaikuttavat ohjelmiston koko sekä käytön kriittisyys. Lentokoneessa käytettävät ohjelmat tulee olla laadultaan varmatoimisempia, kuin esimerkiksi pienelle ryhmälle toteutettu keskustelukanava. Toimivaa ohjelmistoa on vaikea todistaa hyvälaatuiseksi, mutta huonolaatuisen ohjelmiston huomaa nopeasti virheellisestä ja epävarmasta toiminnasta. Ohjelmistoprojektin alussa tulee määritellä tavoiteltava laatu sekä varata ohjelman laadun testaamiseen riittävät resurssit.

Ohjelmiston laadulle on määritelty lukuisia joukko attribuutteja. Esimerkkejä laatuattribuuteista ovat käytettävyys, testattavuus, suorituskyky, turvallisuus, saatavuus ja muunneltavuus. V-mallissa käytettävyys toteutetaan vaatimus ja suunnitteluvaiheessa. Riittävän kattava ja oikeellinen suunnitelma toteuttaa hyvän käytettävyyden. Johdonmukainen, selkeä ja helposti tulkittava ohjelmointi tekee ohjelmasta testattavan [ABC82]. Testitapausten tulee olla mitattavat, helposti ymmärrettävissä sekä ohjelman tavoitteisiin nähden riittävän kattavat. Testitulosten perusteella pystytään toteamaan ohjelman suorituskyky, turvallisuus ja saatavuus vaatimuksiin nähden. Muunneltavuus riippuu koko prosessin onnistumisesta. Selkeä rakenne ja ohjelmointi sekä helposti toistettavat testitapaukset tekevät ohjelmistosta muunneltavan.

2.2.2 Validointi

V-mallissa on tyypillistä, että tasojen välillä tehdään dokumentointiin katselmuksia ja tarkistuksia, joilla pyritään paikallistamaan puuttuvat vaatimukset jo ennen kuin siirrytään kehityksessä eteenpäin. Tätä työvaihetta kutsutaan validoinniksi.

Ohjelmistolle päätettyjen laatuavoitteiden perusteella tehdään päätös validoinnin laajuudesta ja tarpeesta [ABC82]. Tavallisesti validointiin kuuluvat ryhmäanalyysit (walk-through), tarkistukset ja dokumenttien hyväksymiset.

Osana validointia on testauksen suunnittelu, joka aloitetaan siirryttäessä tasolta seuraavalle. Tyypillisesti testitapausten suunnittelussa tulee esiin yksityiskohtia, joita on jäänyt huomaamatta suunnittelussa. Puutteet lisätään edelliselle tasolle ja ohjelmaan ennen varsinaisten testien aloittamista. Tasojen välillä on oltava testauksen koordinoitua, jotta vältetään päällekkäisiltä testitapauksilta [PRI03].

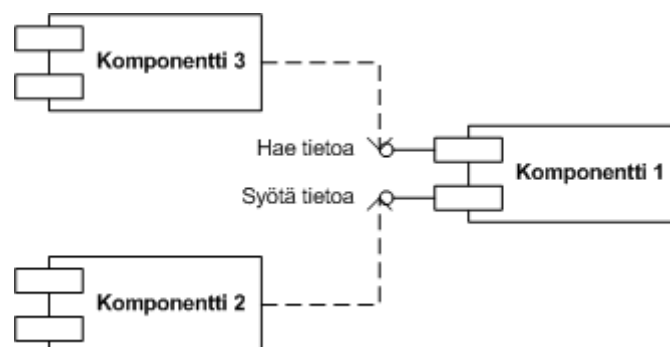
2.2.3 Ohjelmointi ja yksikkötestaus

Toiminnallisten- ja moduulikuvausten perusteella tehdään ohjelmointi, jota testataan yksikkötestauksella. Yksikkötestauksessa ohjelmoija tekee ensin määrittymiset ohjelmoimalla yksikölle annettavat syötteet ja siltä odotettavat palautteet testitapaukseen. Ohjelmoijan kuvattua määrittymiset ohjelmointikielellä, alkaa varsinaisen sovelluksen ohjelmointi. Kehitettävää ohjelmaa voidaan jatkuvasti testata määriteltyjä testitapauksia vasten automaattisesti, jolloin ohjelmoitavan yksikön laatu tunnetaan suoritettujen yksikkötestitapausten osalta.

Yksikkötestauksessa käytetään pääasiassa rakenteellisia testimenetelmiä, joilla ohjelmaa suoritetaan raja-arvoilla muutoskohdissa. Yksikkötestauksessa ohjelmoija suunnittelee testitapaukset itse, joka poikkeaa muista testauksen vaiheista. Objektiivinen testaus on haastavaa, kun ohjelmoija testaa itse omaa ohjelmaansa [Hum89]. Käytännössä yksikkötestauksessa sovelletaan usein pariohjelmointia, jossa testitapaukset tehdään ensin parityönä ja varsinaisen ohjelmointi sen jälkeen [Bec03]. Näin testeihin saadaan puolueeton näkökulma.

2.2.4 Moduulisuunnitelma ja integraatiotestaus

Toiminnallisen kuvauksen jälkeen järjestelmä usein kuvataan komponentteina, jossa järjestelmä on jaettu pieniin ohjelmoitaviin kokonaisuuksiin. Näille kokonaisuuksille on kuvattu komponenttien rajapinnat muiden komponenttien suhteen.



Kuva 3: Komponenttisuunnitelma

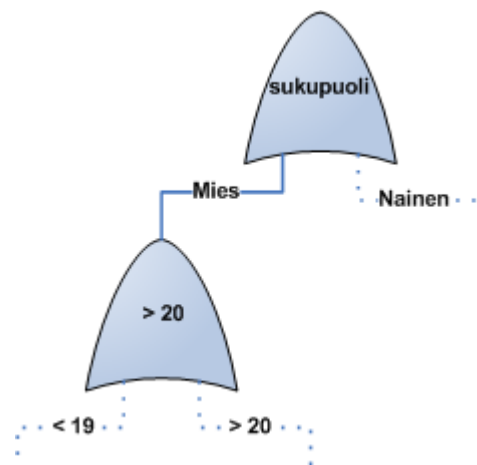
Komponenttien kuvausta kutsutaan moduulisuunnitelmaksi ja se testataan integraatiotestauksella. Integraatiotestaus keskittyy komponenttien rajapintojen sekä komponenttien välisen toiminnan testaamiseen [EiR96]. Integrointi ja sen testaus voidaan toteuttaa ylhäältä alas tai alhaalta ylös -menetelmillä. Menetelmissä liitetään

moduuliin aina uusia moduuleita suoritusjärjestyksen mukaan. Testauksessa ongelmalista on puuttuvien osien simulointi, joka toteutetaan tavallisesti oletusarvoilla tai tarkoitukseen suunnitellulla testiohjelmistolla. Käytännössä tavallinen menetelmä on liittää kaikki järjestelmän moduulit kerralla yhteen, jolloin säästytään puuttuvien moduulien simuloinnilta. Laajoissa ohjelmistoissa ongelmien paikallistaminen on tässä menetelmässä haasteellista. Liitettäessä moduulit kerralla yhteen ero järjestelmätestauksen ja integraatiotestauksen välillä hämärtyy.

2.2.5 Ohjelmiston suunnittelu ja järjestelmätestaus

Vaatimusmäärittelyn jälkeen tehdään ohjelman käsitteellinen malli tietosisällöstä sekä sisällön riippuvuuksista [Rob97]. Vaihetta kutsutaan toiminnalliseksi kuvaukseksi, jossa vaatimusten pohjalta suunnitellaan toiminnallisuus ja sen toteutus. Toteutus voidaan kuvata esimerkiksi oliomalleilla, rakenteellisella kuvauksella tai muilla kuvaustavoilla riippuen millaiselle alustalle toteutettava ohjelma tullaan rakentamaan. Toiminnallinen kuvaus validoidaan kehitysryhmän sekä mahdollisesti tilaajien kesken, jossa todetaan toiminnallisuuden kattavuus ja oikeellisuus sekä varmistetaan, että osapuolilla on yhteinen ymmärrys kehitettävästä ohjelmasta.

Toiminnallisen kuvauksen testaustasoa kutsutaan järjestelmätestaukseksi. Järjestelmätestauksessa on tiedossa rakenteet, jolla järjestelmä on toteutettu. Järjestelmätestausta kutsutaan myös lasilaatikkotestaukseksi (white-box testing), koska testaajat pääsevät käsiin ohjelman toteutukseen ja voivat suunnitella testit ohjelman kululle eri arvoilla.



Kuva 4: Ohjelman rakenne lasilaatikkotestauksessa

Testauksessa keskitytään toiminnallisten kokonaisuuksien testaamiseen. Järjestelmätes-

tissä ohjelmaa suoritetaan järjestelmän raja-arvoilla, poikkeustapauksissa, erilaisilla käyttötapauksilla ja kuormitetaan poikkeuksellisen suurilla määrillä. Erityisesti keskitytään ei-toiminnallisten vaatimusten, kuten suorituskyvyn, saatavuuden tai turvallisuuden testaamiseen, sillä näitä osia on hankala testata myöhemmin hyväksymistesteissä tuntematta ohjelmiston rakennetta [Bei84].

2.2.6 Vaatimusmäärittely ja hyväksymistestaus

V-mallissa ohjelman toteutus aloitetaan keräämällä käyttäjiltä ja muilta sidosryhmiltä vaatimukset ja käyttötapaukset järjestelmän toiminnallisuudeksi. Vaatimukset kerätään yhteen, ratkaistaan ristiriitaiset vaatimukset sidosryhmien kesken sekä muodostetaan vaatimuksista toteutettava kokonaisuus.

Kokonaisuuden valmistuessa kerätään sidosryhmät yhteen ja validoidaan vaatimusten kattavuus ja oikeellisuus. Sidosryhmien hyväksytyä vaatimukset siirrytään ohjelmiston suunnitteluvaiheeseen. Testitapausten suunnittelu aloitetaan, kun vaatimusmäärittely on valmis. Testitapausten suunnittelussa esiin tulevat puutteet lisätään vaatimusmäärittelyihin. Vaatimusmäärittelyt testataan hyväksymistestauksella. Hyväksymistestauksessa testataan täyttääkö ohjelma asiakkaan sille asettamat vaatimukset. Hyväksymistestausta kutsutaan mustalaatikkotestaukseksi, koska käyttäjä ei näe kuinka ohjelman osat on toteutettu. Käyttäjällä on pääsy vain toimintoihin, jotka ohjelma hänelle tarjoaa.



Kuva 5: Mustalaatikkotestaus

Hyväksymistestauksella on yhden suhde yhteen relaatio vaatimusten kanssa ja mikäli vaatimukset on kirjoitettu matemaattisen täsmällisesti, pystytään ohjelmiston toiminnallisuus osoittamaan oikeaksi käyttämällä matemaattisia menetelmiä. Vaatimusten oleellinen piirre on, että kaikkien ohjelmiston sidosryhmien tulee hyväksyä vaatimukset, jolloin matemaattinen esitystapa rajaa käyttäjäkuntaa. Muodollinen esitys on harvoin mahdollinen menetelmä vaatimusten esittämiseen. Vaatimusmäärittelyt koostuvat useasta osasta, jolloin osa vaatimuksista on mahdollista kuvata täsmällisesti ja osa vapaammin, jolloin ymmärrettävyys säilyy, mutta silti toiminnallisuuden oikeellisuus

voidaan osoittaa luotettavasti.

3 Vaatimusmäärittely

Vaatimusmäärittely on ohjelmistoprosessin osa, jossa kerätään yhteen ohjelmaa käyttävät sidosryhmät. Vaatimusmäärittelyn tehtävänä on kerätä sidosryhmiltä toiveet ja vaatimukset ohjelman toiminnallisuudesta. Vaatimusmäärittelyn osatehtävinä ovat kartutus, jossa kerätään vapaamuotoisesti kaikki saatavilla oleva tieto käyttäjiltä ja muista lähteistä liittyen ongelmaan. Kartutusvaiheesta saadut tiedot analysoidaan ja kerätään yhtenäiseen muotoon analyysidokumentissa. Analyysidokumentissa käyttäjien vaatimukset liitetään yhtenäiseksi kokonaisuudeksi, joka on mahdollista ratkaista ohjelmalla. Lopuksi spesifointivaiheessa määritellään rajapinnat sidosryhmien ja sovelluksen välillä analyysivaiheesta saadulle kokonaisuudelle. Spesifioinnissa määritellään järjestelmälle annettavat syötteet ja odotettavat palautteet. Rajapintoja voivat käyttää toiset järjestelmät tai ohjelmistoa käyttävät ihmiset. Vaatimusmäärittelyn laatuattribuutit ovat kattavuus ja oikeellisuus.

Ohjelmiston onnistunut toteutus riippuu siitä, kuinka tarkasti vaatimusmäärittelyllä on pystytty kuvaamaan ohjelman haluttua toimintaa. Yksityiskohtaisiin määritelmiin tarvitaan muodollista esitystapaa. Vaatimusmäärittelyt ovat usein asiakkaiden omistamia ja kirjoittamia, jolloin muodollinen esitystapa on käytössä harvoin.

Tavallisesti asiakkaat esittävät suunnitelmat liiketoiminnan vaatimuksina (business requirements). Liiketoiminnan vaatimuskuvaukset koostuvat ongelman sanallisesta kuvaamisesta sekä mahdollisista kuvista, joilla pyritään mallintamaan ongelmaa. Ohjelman suunnittelun kannalta sanalliseen kuvaamiseen liittyy ongelmia. Ensimmäinen ongelma on kuvauksissa esiintyvä häly (noise), joka aiheutuu ongelman kannalta epäolennaisen tiedon esittämisestä tekstissä. Samat asiat saattavat toistua useaan kertaan, joka vaikeuttaa olennaisen tiedon erittelemistä tekstistä. Toinen ongelma on hiljaisuus (silence). Hiljaisuudella tarkoitetaan puuttuvia vaatimuksia. Kolmas ongelma on ylimäärittely (over-specification), jolloin ongelman kuvauksessa on pyritty kuvaamaan osittain ratkaisua. Neljäntenä mainitaan toiveajattelu (wishful thinking). Toiveajattelulla tarkoitetaan vaatimuksista löytyviä epämääräisiä toivomuksia, joita voi käytännössä olla mahdoton tai hyvin hankala toteuttaa. Viidentenä on vaatimusten moniselitteisyys (am-

biguity), joka on tavallinen ongelma luonnollisessa kielessä, jossa yhdellä sanalla voi olla monta merkitystä [MRM04].

Onnistuneeksi ohjelmistoksi voidaan määritellä ohjelma, joka täyttää sidosryhmien sille asettamat tavoitteet. Vaatimusmäärittely on ohjelmistokehityksen vaihe, jossa sidosryhmien tavoitteet kerätään järjestelmällisesti ja rajataan muotoon, joka voidaan esittää asiakkaalle, käyttäjille, ohjelmoijille, testaajille, projektiorganisaatiolle ja muille sidosryhmille. Vaatimusmäärittelyn perusteella ohjelman tulee olla testattavissa oikean toiminnallisuuden varmistamiseksi. Vaatimusmäärittelyn tehtävä on määritellä ohjelmiston sidosryhmät sekä heidän tarpeensa ja tavoitteensa ohjelmiston suhteen. Sidoryhmien tavoitteet voivat olla vaihtelevat, epäselvät ja ristiriitaiset. Tavoitteet voivat olla epäsuoria, jolloin vaatimusmäärittelyiden toteuttajilta edellytetään aihealueen tuntemusta sekä kykyä havaita sidosryhmien hiljaiset tarpeet ohjelmiston suhteen.

Hyvän vaatimusmäärittelydokumentin ominaisuuksiksi katsotaan oikeellisuus, yksikäsitteisyys, kattavuus, muokattavuus, muutosten jäljitettävyyttä sekä verifioitavuus [Iee98]. Vaatimusmäärittely on oikea, kun jokainen ohjelmiston toiminnallisuus on sisällytetty dokumenttiin. Oikeellisuuden tarkistamiseksi ei ole olemassa työkaluja vaan jokainen vaatimus on liitettävä tapauskohtaisesti aihealueen lähteisiin. Yksikäsitteisyydellä tarkoitetaan, että jokaisella vaatimuksella on vain yksi tulkinta. Tällä tarkoitetaan vähintään termien avaamista yksikäsitteisesti esimerkiksi vaatimusmäärittelyiden liitteenä olevassa sanastossa. Luonnollisen kielen ongelmia on pyritty välttämään vaatimusmäärittelyä varten luoduilla logiikasta johdetuilla kielillä, kuten PSL/PSA (problem statement language/Problem statement analyzer) [NuE00] [Som05]. Näiden ongelmana on kommunikointi käyttäjien kanssa, koska käyttäjät harvoin tuntevat logiikan työvälineitä. Vaatimusmäärittelydokumentti on kattava, kun se sisältää kaikki ohjelmalle oleelliset vaatimukset. Vaatimusmäärittelyiden on oltava myös muokattavat sekä muutokset tulee olla jäljitettävissä sekä myöhemmin palautettavissa. Viimeiseksi vaatimusten tulee olla verifioitavissa. Jokaisen vaatimukset tulee olla testattavissa yksikäsitteisesti äärellisessä ajassa joko käyttäjän tai ohjelman toimesta [Iee98].

Kirjallisuudessa vaatimusmäärittelyä on kuvattu muun muassa seuraavasti: ”Vaatimusmäärittely on ohjelmistokehityksen haara, joka keskittyy käytännön elämän tavoitteisiin ohjelmistojärjestelmän toiminnallisuuden ja rajoitteiden suhteen. Vaatimusmäärittely keskittyy toiminnallisuuden ja rajoitteiden suhteiden kuvaamiseen täsmällisessä muo-

dossa niin, että ne kuvaavat ohjelmiston toimintaa ja kehitystä.” [Zav97]

Määritelmässä korostuu käytännön elämän tavoitteet, jotka motivoivat ohjelmiston kehitystä. Vaatimusten täsmälliset kuvaukset muodostavat pohjan vaatimusten analysoinnille. Analyysi puolestaan validoidaan, jolla varmistetaan sidosryhmien odotusten täyttyminen. Täsmälliset kuvaukset osoittavat asiat, joita ohjelmistokehittäjien tulee rakentaa. Täsmällinen kuvaus on helpommin testattavissa kuin epätäsmällinen. Vaatimusmäärittelyt ottavat kantaa vaatimusten kehittymiseen ajan myötä, jolloin vaatimuksia voidaan käyttää esimerkiksi tuoteperheen osana ja jotka siten vaativat jatkuvaa päivittämistä [NuE00].

Vaatimusmäärittely kerää useiden sidosryhmien erilaiset tarpeet, näkökulmat ja ristiriidat yhteen dokumenttiin niin täsmällisessä muodossa, että järjestelmän suunnittelu on sen pohjalta mahdollista. Dokumentin on oltava ymmärrettävä sidosryhmien kannalta, jotta eri sidosryhmät voivat kommentoida ja lopulta hyväksyä vaatimukset.

Vaatimusmäärittelydokumenttia voidaan kuvata kommunikointivälineeksi, jolla ohjelmiston käyttäjät kommunikoivat ohjelmiston kehittäjien kanssa. Vaatimusmäärittely luo kehittäjille raamit, joilla ohjelman toiminta voidaan perustella ja osoittaa toimivan oikein. Vaatimusmäärittelyn on oltava riittävän täsmällinen, jotta ohjelmistokehittäjät pystyvät toteuttamaan asiakkaan tavoitteet. Vaatimusmäärittelyn täsmällisyyttä rajoittaa asiakkaiden mahdollisesti ristiriitaiset tavoitteet sekä puutteellinen tietämys halutuista asioista.

Vaatimusmäärittelyn kannalta keskeisiä käsitteitä ovat tehtäväalue (problem domain) sekä ratkaisujärjestelmä (solution system) [Jac95]. Ohjelmistokehityksen päävaiheita ovat tehtäväalueen analysointi, tehtäväalueen ja ratkaisujärjestelmän vuorovaikutus sekä ratkaisun muodostaminen. Näistä analysointi ja vuorovaikutus ratkaisun kanssa kuuluvat vaatimusmäärittelyvaiheeseen ja ratkaisun muodostaminen ohjelmiston toteutusvaiheeseen.

3.1 Vaatimusmäärittelyn vaiheet

Vaatimusmäärittely on ohjelmistokehityksen avustehtävä. Sen alkuosaan kuuluvat projektin toteutuksen mahdollisuuden arviointi (feasibility assesment) sekä projektin kustannusten ja resurssitarpeiden arvioiminen. Vaatimusmäärittelyn edellyttämä työ-

määrä ja laajuus vaihtelevat ongelma-alueen laajuuden sekä monimutkaisuuden mukaan ja voi koostua useista eri vaiheista. Tyypillistä on, että vaatimukseen palataan ja niitä päivitetään ohjelmistokehityksen edetessä. Vaatimusmäärittely jaetaan kartutus- (elicitation), analysointi-, spesifiointi- ja validointivaiheisiin.

3.1.1 Kartutus

Kartutuksen tärkein tavoite on hahmottaa, mikä ongelma halutaan ratkaista ja millaisilla rajoitteilla se voidaan toteuttaa järjestelmässä [NuE00]. Alkuvaiheessa identifioidut rajoitukset määrittävät kartutukseen valittavia tekniikoita sekä sitä, mitkä sidosryhmät valitaan vaatimusmäärittelyyn mukaan.

Kartutuksen tehtävänä on määrittää sidosryhmät, jotka käyttävät, hyötyvät tai muuten joutuvat kosketuksiin ohjelman aiheuttamiin muutoksiin ympäristössä. Sidoryhmiä voivat olla organisaatiot ja yksittäiset ihmiset. Sidoryhmiin kuuluvat käytännössä aina asiakas, joka maksaa järjestelmän kehityksen sekä ohjelman tulevat käyttäjät. Erityisen haastavaa vaatimusten kartutuksessa on sidoryhmien erilaisuus. Organisaatiot voivat olla suuria tai pieniä, niissä voi olla eritasoisia ihmisiä eri rooleissa. Jotkin käyttäjistä voivat olla ohjelmistokehityksen ammattilaisia, toiset osaavat perusteellisesti oman erikoisalueensa tiedot ja osa voi olla aloittelijoita sekä ohjelmistokehityksessä että aihealueessa. Sidoryhmien piilevien tarpeiden esiintuominen ja erilaisten taustojen ymmärtäminen ja tulkitseminen ovat oleellinen osa kartutusta. Kartutuksessa pyritään hahmottamaan tehtäväalueen rajauksen lisäksi tehtäväalueen olennaiset ominaisuudet sekä sidoryhmien näkemykset näistä ominaisuuksista, niiden tarpeellisuudesta ja kii-reellisyydestä. Näiden perusteella kartutuksessa saadaan kuva siitä, miten tärkeitä ongelma-alueen eri osatekijät ovat.

Asiakkaan, käyttäjien ja tehtäväalueen asiantuntijoiden lisäksi tietolähteinä voivat toimia aikaisemmat järjestelmät, kilpailevat tuotteet, standardit tai lait [Bra02]. Kartutuksessa käytetään useita lähteitä, joista parhaat valitaan myöhempisiin tarkempaan analysointiin.

Vaatimusten kartutus on iteratiivinen prosessi, jossa eri kierroksilla tavallisesti löydetään uusia kartutuskohteita aikaisempiin löydöksiin. Tavallisia kartutustekniikoita ovat yleiset tiedonkeräämismenetelmät, kuten haastattelut, kyselyt, tutkimukset ja tehtäväalueen olemassa oleva dokumentaatio. Ohjelmiston vaatimusmäärittelyiden

kartutukseen erityisiä tekniikoita ovat muun muassa ryhmätyöskentely, jossa voidaan soveltaa käyttäjätapauksia, sidosryhmäkohtaisia tai laajempia aivoriisiä. Yhtenä esimerkkinä ryhmätyöskentelymalleista on JAD (joint application development), jossa eri sidosryhmät osallistuvat samaan tilaisuuteen ja jossa pyritään keräämään yhdellä kerralla mahdollisimman paljon tietoa tehtäväalueesta. Kartutukseen on olemassa erillisiä malleja, kuten tavoitepohjaiset mallit tai skenaariopohjaiset mallit [NuE00]. Prototyypien laatiminen kartutusvaiheessa on mahdollista erityisesti projekteissa, joissa on kysymys uudesta aihealueesta, josta on niukasti tietoa tarjolla.

3.1.2 Analysointi

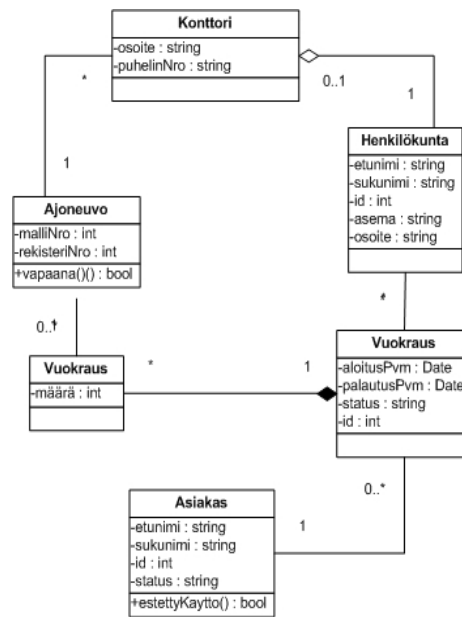
Vaatimusten analysointivaiheessa kartutusvaiheen tiedot pyritään jäsentelemään yhdeksi kokonaisuudeksi, joka koostaa kaikille sidosryhmille yhteiset vaatimukset yhdeksi kokonaiseksi ongelmaksi. Dokumentissa kerätään tehtäväalueen käsitteistö, sidosryhmien suhteet, tarpeet ja rajoitteet yhtenäisessä jäsennellyssä muodossa. Analyysin tehtävänä on jäsentää ongelman rakenne, sisältö ja yhteydet [Bra02]. Analyysivaiheen tuloksena on analyysidokumentti, jonka perusteella sidosryhmät hyväksyvät ongelman kehityksen. Analyysivaiheeseen on useita lähestymistapoja, joista monessa ongelmaa pyritään mallintamaan. Mallintamisen tavoitteena on esittää ongelmat yksiselitteisesti. Analyysivaiheen tehtävänä ovat seuraavat kokonaisuudet; tehtäväalueen rakenteen jäsentely, tehtäväalueen semantiikan ja tietovaatimusten jäsentely, toimintaa ohjaavien tapahtumien esittely sekä sidosryhmien saaman hyödyn osoittaminen [Bra02].

Vaatimusmäärittelyiden analysointi poikkeaa järjestelmäkuvauksesta, koska vaatimuksia kuvattaessa ei oteta kantaa ratkaisujärjestelmän toteutukseen [Jac95]. Analyysivaiheessa kuvataan ongelmat, joita ongelmiston tulee ratkaista, mutta ei oteta kantaa toteutukseen eikä määritetä sovelluskohtaisia muuttujia [Iee98]. Ongelman osatekijöiden suhteiden kuvaaminen ei ole yhtenevää ongelman ratkaisun kanssa. Yhdellä mallilla ei voida kuvata ongelmaa ja ratkaisua, koska kuvauksissa käsitellään eri asioita. Ongelma ja ratkaisu eivät ole käsitteellisesti yhtä suuria [Jac95]. Vaatimusmäärittelyssä kuvataan ongelmalle tyypilliset piirteet ja ohjelmoiija puolestaan soveltaa tietämystään ohjelmoinnista ratkaisun kuvaamisessa [Som05]. Käytännössä ongelman ja ratkaisun erottaminen on hankala toteuttaa. Toiselle ihmiselle jokin malli tarkoittaa kuvausta, kun taas toiselle sanallinen kuvaus ei ole riittävä esitys ratkaisun kehittämiseksi [Dav90]. Vaati-

musmäärittely ja ohjelmiston suunnittelu eivät ole erillisiä vaiheita vaan iteratiivinen prosessi, jossa molemmissa vaiheissa syvennetään ongelman tuntemusta ja palataan aikaisempiin vaiheisiin ja reagoidaan muutoksiin [Som05]. Vaatimusmäärittelyssä on muistettava, että vaatimusanalyysin ei tarvitse tietää miten ratkaisu tullaan toteuttamaan. Ratkaisuvaihtoehdot voivat koostua eri ohjelmointikielistä ja käyttöympäristöistä. Analyysissä on oleellista määrittää yksiselitteisesti ongelmakehyksen todelliset ongelmat ja rajoitteet.

Esimerkkinä vaatimusten analysointimenetelmistä on rakenteinen analyysi, olioanalyysi ja tehtäväaluepohjainen analyysi. Rakenteinen analyysi on ollut ensimmäinen lähestymistapa vaatimusmäärittelyyn, jossa on vähennetty luonnollisen kielen käyttöä ja keskitytty ongelman mallintamiseen. Mallinnusmenetelmiä on olemassa lukuisia, kuten tietovuomallit, tietohakemistomallit ja tietojen keskinäisiä suhteita kuvaavat mallit (entity relationship diagram). Rakenteinen analyysi on yleinen ja pisimpään käytetty analyysin väline [Bra02]. Rakenteisen analyysin vahvuutena on, että se pystyy kuvaamaan luonnollista kieltä täsmällisemmin ja kompaktissa muodossa tehtäväaluetta. Ongelmana rakenteisessa analyysissä on sen suhde toteutukseen. Vaatimusmäärittelijä siirtyy helposti kuvaamaan ratkaisua rakenteisessa analyysissä, jolloin varsinainen ongelma ja sen osatekijät jäävät sivuosaan. Tällöin oleellisia asioita ongelmasta saattaa jäädä mainitsematta vaatimusmäärittelyssä.

Toinen yleisesti käytetty analyysimenetelmä on olioanalyysi. Olioanalyysi on lähellä rakenteellisen analyysin metodeita. Olioanalyysin lähtökohtana ovat olio-ohjelmoinnin erityispiirteet, jossa rakenne kuvataan olioina, attribuutteina, metodeina ja olioiden välisinä suhteina. Oliomallissa ongelman ja ratkaisun välinen raja on rakenteista analyysiä hankalampi erottaa, koska luokkakuvaukselle on vaikea osoittaa erillistä ongelmakuvausta. Tosin käytännössä tämä on myös osoittautunut oliokuvausten vahvuudeksi, koska sillä voidaan kuvata sekä ongelmaa että ratkaisua. Oliomallinnus on tavallisesti käytetty menetelmä, ja UML:n luokkakaaviot ovat yksi sovellus aiheesta. Oliomalli on kuitenkin tavallisemmin käytetty ratkaisun kuvaamisessa kuin vaatimusmäärittelyssä. Kaikki sidosryhmien jäsenet harvoin osaavat tulkita riittävästi luokkakaavioiden sisältöä, jolloin vaatimusmäärittelyiden perimmäinen tarkoitus ei oliomallissa toteudu.



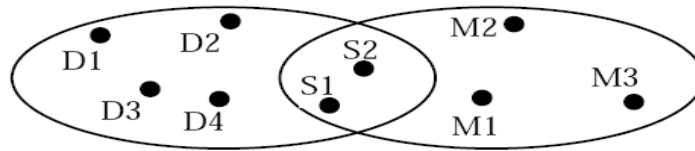
Kuva 6: UML Esimerkki olioanalyysistä

Rakenteisten kuvaamistapojen lisäksi teoriassa on esitelty tehtäväaluepohjaisia (problem frames) lähestymistapoja, joiden mukaan erilaisilla ongelmilla on yhteneväisiä piirteitä riippuen siitä millaista ongelmaa pyritään ratkaisemaan [Jac95]. Tehtäväaluepohjaisen lähestymistavan mukaan ongelmia on rajallinen määrä, jolloin jokaiselle voidaan määrittellä paras tapa vaatimusten esittämiseen. Tehtäväaluepohjaisessa analyysissä vaatimukset jaetaan pienempiin osiin, joille on mahdollista määrittää selkeä ratkaisu ja ratkaisulle yhteydet toisiin ongelmiin [HJL02]. Hyvin jäsenellyille ongelmille on mahdollista kehittää erilliset vaatimusmäärittelymenetelmät sen mukaan, millaista ongelmaa pyritään ratkaisemaan. Tehtäviksi voidaan määrittää esimerkiksi sääntöihin perustuvat ohjausjärjestelmät, tiedonmuunnosjärjestelmät tai rajapintojen kytkentäjärjestelmät. Menetelmä vaatii paljon kokemusta vaatimusmäärittelijältä eri aihealueista eikä menetelmä ole vielä vakiintunut. Hyvänä puolena menetelmässä on aikaisempien kokemusten uudelleenkäytettävyys.

3.1.3 Spesifiointi

Spesifiointivaiheessa kuvataan ongelman ratkaisun haluttu käyttäytyminen eri sidosryhmien näkökulmista sekä erityisesti ratkaisun rajapinnan tarjoamat palvelut käyttäjille. Käyttäjät voidaan jakaa ihmisiin ja toisiin sovelluksiin, joiden kanssa sovellus tulee kommunikoidaan. Lisäksi spesifiointivaiheessa jokaiselle sidosryhmälle ($\{D1, \dots, D4\}$) määritellään sen tarpeen mukaiset rajapinnat ($\{S1$ ja $S2\}$) järjestelmän ($\{M1, \dots, M3\}$) kä-

sittelyyn.



Kuva 7: Spesifiointi määrittelee ongelman rajapinnat ratkaisujärjestelmään [Jac95]

Vaikka vaatimusmäärittelyvaiheessa ei tiedetä ratkaisujärjestelmästä tai sen toteutuksesta, ohjelmistokehityksessä tiedetään, että ongelma ratkaistaan jollakin ohjelmalla. Ohjelmistoille on yhteistä, että ne tarvitsevat syötteen toteuttaakseen toiminnon. Tämä toiminto antaa tuloksena jonkin palautteen. Hyvässä spesifioinnissa määritellään täsmällisesti syötteiden ja palautteiden syntaksi ja semantiikka. Esimerkiksi muunnosjärjestelmän toimintaa voidaan kuvata seuraavasti: ”Järjestelmä saa syötteenä sovellukselta A Unicode merkin, jota ratkaisujärjestelmä yrittää muuttaa numeroksi. Ratkaisujärjestelmän tulee antaa 5 millisekunnin aikana palautteena kokonaisluku tai epäonnistuuessa null -arvo”. Edellisestä esimerkistä on huomattava, että koko tietosisältöä ei voida spesifiointivaiheessa esittää, sillä ongelman ratkaisu vaatii tavallisesti järjestelmäkohtaista lisätietoa, joka ei ole käyttäjälle välttämättä näkyvissä. Oleellista tässä vaiheessa on näyttää vain tieto, joka on ongelman kannalta tärkeää. Käyttöliittymissä palautteiden määrä voi kasvaa suureksi, koska virheellisistä syötteistä voidaan antaa lukuisia erilaisia palautteita. Virhetilanteiden palautteet eivät suoranaisesti liity ongelman ratkaisuun. Tavallisesti käyttöliittymät jätetään spesifiointivaiheen ulkopuolelle. Niistä voidaan antaa kuvaavia esimerkkejä, mutta lopullinen toteutus jätetään ratkaisujärjestelmän suunnitteluun.

3.1.4 Validointi

Vaatimusten validoinnissa etsitään ja korjataan virheitä vaatimusmäärittelyiden analyysi- ja spesifiointiosista. Validointi tapahtuu rinnakkain muiden menetelmien kanssa, joka korostaa iteratiivista lähestymistapaa vaatimusten määrittelyyn. Validoinnissa pyritään löytämään kohdat vaatimuksista, jotka eivät vastaa todellisuutta ja puutteet, joiden osalta dokumentti ei toteuta haluttuja ominaisuuksia.

Validoinnissa käytettävät menetelmät ovat esimerkiksi tarkistukset ja katselmoinnit, joissa sidosryhmät yhdessä tai erikseen tarkistavat ja lopulta hyväksyvät dokumentoin-

nin oikeellisuuden.

3.2 Vaatimusten luokittelu

Ohjelmistoilla on erityyppisiä käyttäjiä ja eri käyttäjillä on erilaiset tarpeet ohjelmiston toiminnalle. Osa vaatimuksista voi olla ristiriidassa keskenään, minkä takia vaatimukset on luokiteltava analysointia ja vertailua helpottamaan. Vaatimukset voidaan luokitella ohjelman sisäisiin ja ulkoisiin vaatimuksiin, joista ulkoisia kutsutaan tavallisesti toiminnallisiksi ja sisäisiä ei-toiminnallisiksi vaatimuksiksi. Ei-toiminnalliset laatuvaatimukset on tavallisesti jaettu luotettavuuteen, käytettävyyteen, tehokkuuteen, ylläpidettävyyteen ja siirrettävyyteen [Iso91]. Toiminnalliset vaatimukset ovat järjestelmäkohtaisia käyttäjien vaatimia toimintoja. Käytännössä vaatimusten luokittelu sisäisiin ja ulkoisiin on vaikeaa, sillä käyttäjälle on itsestään selvää, että järjestelmä on turvallinen ja vastaa käyttäjän syötteisiin ripeästi. Ei-toiminnalliset vaatimukset jäävät toteutuksen määrittäviksi, mutta on tavallista, että ei-toiminnalliset vaatimukset ovat järjestelmän käytön kannalta niin ratkaisevia, että ne on kirjattava vaatimuksina. Vaatimusmäärittelijän tehtävänä on löytää tarpeet tällaisille elementeille.

Vaatimusmäärittelyn oleellisin tehtävä on löytää järjestelmän toiminnalliset vaatimukset. Kriittisissä sovelluksissa, joissa esimerkiksi järjestelmän turvallisuus on tärkeä osa toimintaa, täytyy vaatimuksissa kirjata testattavissa olevat vaatimukset turvallisuuden suhteen. Hyväksymistestauksen kannalta ei-toiminnalliset vaatimukset voivat olla hyvin hankalasti testattavissa, sillä käyttäjillä on harvoin riittävästi tuntemusta esimerkiksi turvallisuuteen liittyvästä tekniikasta, jotta tätä voitaisiin testata. Järjestelmän tehokkuuteen tai käytettävyyteen liittyvät ei-toiminnalliset vaatimukset voidaan kirjata ja myöhemmin testata esimerkiksi lisäämällä määrällisiä vaatimuksia. Vaatimuksena voi esimerkiksi olla aika, jonka sisällä ohjelman täytyy reagoida käyttäjän syötteisiin tai aika kuinka nopeasti uusien työntekijöiden on omaksuttava ohjelman käyttö.

Luonteva sijoituskohteeksi ei-toiminnallisille vaatimuksille on spesifiointidokumentissa, jossa määritellään ohjelman rajapinnat, eli syötteet ja palautteet. Syöte/palautte -parilla päästään käsiksi järjestelmän toimintaan ja koska ei-toiminnalliset vaatimukset ovat järjestelmän sisäisestä toteutuksesta kiinni, voidaan ei-toiminnalliset vaatimukset esittää tässä vaiheessa määrittelynä. Määrälliset vaatimukset helpottavat testaamista. Sovellus, jolla varataan verkon välityksellä lippuja, on suorituskyvyltään kriittinen. Lipunvaraus-

ohjelman vaatimus voidaan määrittää: ”Käyttäjä valitsee tapahtuman, istumapaikan ja varaa lipun järjestelmästä. Vastauksena tulee vahvistusilmoitus, mikäli varaus on onnistunut. Järjestelmän ollessa ruuhkautunut, tulee odotuspalkki kuvaamaan odotuksen arvioitua aikaa. Varauksen epäonnistuessa annetaan käyttäjälle epäonnistumisesta ilmoitus. Edellä mainittujen ilmoitusten tulee ilmestyä käyttäjälle 200 millisekunnin sisällä, kun käyttäjiä on vähemmän kuin 100 sekunnissa. Maksimimäärä käyttäjiä on 5000 sekunnissa ja maksimivasteaika saa olla 5000 millisekuntia”. Skenaario on testattavissa syötteen, palautteen sekä ei-toiminnallisten vaatimusten osalta. Skenaario täyttää taulukoesityksen puitteet, jolloin vaatimus on tallennettavissa tietokantaan automaattista testausta varten.

3.3 Vaatimusten spesifointi ja taulukot

Vaatimusmäärittelyiden tarkoituksena on tuottaa kattava ja täsmällinen kuva tavoitelluista toiminnoista uudelle järjestelmälle. Ilman täsmällisiä vaatimuksia, on epätodennäköistä, että ohjelmoijat osaavat tuottaa halutun toiminnallisuuden. Kattavan ja täsmällisen dokumentoinnin määritelmän toteutuminen on vaikea osoittaa. Toisin sanoen, on vaikea näyttää vaatimusmäärittelyiden olevan valmis. Luonnollisen kielen väittämiä ei voida tarkistaa kattavuuden tai täsmällisyyden suhteen. Väittämien oikeellisuus ja yksiselitteisyys voidaan tarkistaa, jos väittämät käännetään matemaattisiksi lauseiksi, mutta se ei tee alkuperäisestä väittämästä oikeaa.

Antiikin Kreikan matemaatikoiden lähestymistavan mukaan ongelmia on kahden tyyppisiä; ongelmia jotka täytyy todistaa sekä ongelmia, jotka täytyy rakentaa olemassa olevista elementeistä [Pol57]. Tämän lähestymistavan mukaan jokainen ongelma muodostuu olemassa olevista asioista ja ratkaisutehtävästä (solution task), joita yhdessä kutsutaan ongelmakehykseksi (problem frame) [Jac95]. Esimerkiksi ongelman, jossa ”pituudet a , b ja c muodostavat kolmion”, elementit ovat tieto, tässä kolme pituutta, tuntematon, joka tässä on kolmio sekä ehto, että kolmion sivujen pituuksien täytyy olla samat kuin a , b ja c . Ratkaisutehtävänä on muodostaa tuntematon niin, että se käyttää olemassa olevaa tietoa ehtojen määrittelemällä tavalla. Vastaavasti ongelmassa, jossa todistetaan, että kolmion kulmat ovat yhtä suuret, määritellään ensin peruselementit. Näitä elementtejä ovat hypoteesi ja päätelmä. Hypoteesi tässä tapauksessa on, että sivujen pituuksista kaksi on yhtä suuria. Päätelmä on, että kulmat ovat yhtä suuret.

Ratkaisutehtävänä on osoittaa, että johtopäätelmä seuraa hypoteesia. Vastaavasti ohjelmiston peruselementit ovat sovellusalue, ohjelmisto ja vaatimuksista johdettu spesifikaatio. Tehtävänä on rakentaa ratkaisujärjestelmä niin, että vaatimusten mukaisesti rakennettu ohjelmisto täyttää sovellusalueen tarpeet, mutta ei ylitä sovellusalueen rajoitteita.

Täsmällinen esitys muodostetaan epämuodollisesta tosimaailman ilmiöstä. Esimerkiksi väittämä ”lentokone ei voi käyttää moottorijarrutusta ilmassa” on käyttäjän mielestä hyvä vaatimus, jotta lentäjät eivät vahingossa jarruta ollessaan ilmassa. Ratkaisujärjestelmän on pystyttävä erottamaan, milloin kone on ilmassa ja milloin maassa. Tähän voidaan käyttää syötteenä renkaiden pyörimisnopeutta, jonka mukaan kone on maassa ja laskeutumisvaiheessa, kun kaikkien pyörien pyörimisnopeus on vähintään 20 kierrosta sekunnissa. Tämä on todellinen esimerkki, jonka mukaan on rakennettu sovellus lentokoneeseen. Spesifikaatiossa ei otettu huomioon, että kovassa vesisateessa renkaat nousevat vesimassan päälle eivätkä pääse pyörimään. Tämän virheen johdosta lentokoneen moottorijarrutus ei ollut mahdollista kovassa vesisateessa.

Esimerkin epämuodollinen vaatimus pitää sisällään useita eri skenaarioita, joita ratkaisujärjestelmän tulee ottaa huomioon. Ensimmäinen skenaario toimii hyvin kuivalla säällä, mutta ei vesisateessa. Muodolliselle esitystavalle onkin tyypillistä, että vaihtoehtoja ylemmän vaatimuksen täyttämiseksi on useita. Skenaarioiden määrä riippuu siitä, kuinka paljon muuttujia ongelma sisältää. Edellisessä esimerkissä vaatimuksen ratkaisu muodostui yhdestä muuttujasta – renkaiden pyörimisnopeudesta. Ratkaisujärjestelmä tarvitsee tiedon mahdollisesta vesisateesta, jotta järjestelmä voi sallia moottorijarrutuksen vesisateella. Lisäksi esimerkin ratkaisujärjestelmä tarvitsee tiedon lentokoneen korkeudesta, jottei jarrutusta sallita ilmassa vesisateella. Näistä skenaarioista tulee kerätä esimerkinomaiset tapaukset kattavasti vaatimusmäärittelyihin, jotta ne voidaan ottaa huomioon ratkaisua toteutettaessa. Karteesisen tulon mukaan skenaarioiden määrä voi muuttujien määrän suhteessa kasvaa hyvin nopeasti.

$$A_1 \times A_2 \times \cdots \times A_n = \{\langle x_1, x_2, \dots, x_n \rangle | x_i \in A_i\}.$$

Olkoon $A_i = A$ kaikilla $i = 1, \dots, n$. Tällöin

$$A^n = \underbrace{A \times A \times \cdots \times A}_n.$$

Kuva 8: Karteesisen tulon määritelmä [Kaa08]

Skenaarioiden määrän kasvaessa suureksi, ongelmaksi tulee vastaan vaatimusmäärittelyn perimmäinen ongelma: vaatimukset hämärtyvät toteutuksen ja yksityiskohtien alle. Skenaarioiden määrittelyä ei voida jättää täysin ohjelmistokehittäjien päätettäväksi. Ohjelmistokehittäjillä ei ole aihealueen tuntemusta, jonka perusteella he voisivat päättää skenaarioiden tarpeellisuudesta tai kehittää kattavaa joukkoa aihealueelle tarpeellisia skenaarioita. Skenaarioiden rajaaminen on osa vaatimusmäärittelyä ja tulee toteuttaa niin, että vain oleelliset ja esimerkinomaiset skenaariot, sekä niiden antamat syötteet ja palautteet otetaan huomioon vaatimuksissa. Taulukkomuoto on yksi tapa skenaarioiden esittämiseen ja siinä pystytään näyttämään syötteet sekä niistä odotetut tulokset.

Tarve yksiselitteiselle ja helppokäsitteiselle ilmaisutavalle vaatimusten määrittelemiseksi on tunnistettu ja siihen on yritetty vastata erilaisilla sovelluksilla, kuten SCR (software cost reduction) ja petri net -menetelmillä. Taulukoita on pidetty täsmällisenä ja suhteellisen yksinkertaisena esitystapana täsmällisten vaatimusten esittämisessä. Kehittäjät ovat omaksuneet taulukkoesityksen helpommin kuin vastaavat esitystavat esimerkiksi Petri net:n tai Z:n [HAB05]. Taulukot täyttävät tarkkuudeltaan matemaattisten menetelmien ehdot, joten taulukoiden tarkasteleminen matemaattisesti on mahdollista. Taulukot ovat osoittautuneet hyvin skaalautuviksi, josta esimerkkinä on Darlingtonin ydinvoimala [PAM91] ja Lockheedin lentokoneen ohjelmisto, joka kattaa yli 1000 taulukkokuvausta [FFK94].

Taulukot helpottavat suunnittelutyötä vaatimusmäärittelyn spesifikaatioita tehdessä. Suunnittelija pystyy rakentamaan sovelluksen tarvitseman tiedon taulukon otsikoiden tasolla ja välittämään sen kommentoitavaksi käyttäjille ja muille sidosryhmille [JPZ96]. Taulukon tietosisältö rajoittaa skenaariot taulukon asettamiin puitteisiin, jolloin taulukko säilyttää yhdenmukaisuutensa. Mikäli skenaario vaatii lisää tietoa, täytyy taulukkoa laajentaa uusilla muuttujilla. Pitkään kestävässä suunnittelutyössä taulukko on yksinkertainen tapa versioiden hallintaan, kun skenaariot ja tietosisältö kasvavat samassa taulukossa [JPZ96].

Taulukon käyttö spesifioinnissa helpottaa vaatimusten validointi- ja tarkistustyötä. Validoinnissa voidaan keskittyä tietosisältöön sekä tapausten kattavuuteen ja oikeellisuuteen ”hajota ja hallitse menetelmällä” [JPZ96]. Tämä tekee tarkistuksesta helpompaa kuin luonnollisen kielen tapausten tarkistaminen.

Vaatimusten testaamisen kannalta taulukot ovat erityisen houkutteleva lähestymistapa, koska taulukointi mahdollistaa tiedon syöttämisen tietokantaan. Mikäli vaatimukset saadaan muotoon, jossa ne voidaan tallentaa tietokannassa, voidaan vaatimukset verifioida automaattisesti.

4 Hyväksymistestaus osana ohjelmistoprosessia

Hyväksymistestauksella testataan vaatimusmäärittelyissä kuvattujen toimintojen oikeellisuutta ja kattavuutta ratkaisujärjestelmässä. Hyväksymistestaaminen toimii sidosryhmille kommunikaatiovälineenä ja testien tulosten perusteella sidosryhmät voivat tehdä päätöksen järjestelmän käyttöönoton hyväksymisestä tai hylkäämisestä. Testitapaukset ovat kattavat, jos ne sisältävät kaikki vaatimusmäärittelyssä esitetyt toiminnot. Järjestelmän toiminnallisuuden kattava testaaminen on sen sijaan haastava tehtävä. Hyväksymistestaamisessa ei tunneta ohjelman rakenteita, eikä testeissä voida keskittyä rakenteiden ongelmakohtiin. Näin ollen testauksen kattavuuden osoittaminen on mahdotonta. Ei-toiminnallisten asioiden, kuten suorituskyvyn, turvallisuuden tai saatavuuden testaaminen hyväksymistesteissä on vaikeaa. Hyväksymistestien täytyy suurelta osin luottaa aikaisempien vaiheiden testien kattavuuteen.

Hyväksymistestien oikeellisuuden perustana on, että testaajat tulkitsevat vaatimuksissa esitetyt asiat samalla tavalla kuin vaatimusmäärittelijät. Tavallisesti testaajat ovat eri henkilöitä, jolloin on olemassa väärinymmärrysten riski. Vaatimusten keräämisen ja testaamisen välillä voi olla pitkä aika, joka aiheuttaa oikeellisuudelle ongelmia. Mikäli testitapaukset on tehty kattavasti jo vaatimusmäärittelyvaiheessa, ei väärinymmärrysten riskiä ole. Testien aikana tulee usein tarve uusien testitapausten kirjoittamiselle. Tällöin alkuperäisiä testitapauksia täytyy täydentää. Hyvin kuvatuista vaatimuksista saadaan testitapaukset kaivettua esiin kattavasti ja parhaassa tapauksessa vaatimukset ovat kuvattu niin, että ne ovat testattavissa automaattisesti.

4.1 Ohjelmistotestaus ja laatu

Tyypillisesti ohjelmistotuotannossa laatua on pyritty varmistamaan testauksella, joka voi kuluttaa projektin budjetista jopa 60 % [HiT00]. Projektin jäädessä aikataulusta jälkeen, testaaminen jää helposti ohueksi ajanpuutteen vuoksi. Heikkolaatuisen ohjelman todennäköisyys kasvaa, jos ohjelmistoprojekti jättää laadun hallinnan pelkän ohjelmistoprosessin loppupäässä tehtävän testauksen varaan.

Ohjelmistotuotannossa syntyneiden lukuisten heikkojen esimerkkien jälkeen on tunnistettu tarve kehittää ohjelmistoprosesseja laadun parantamiseksi. Onnistuneen ohjelmistotuotantoprojektin attribuuteiksi katsotaan kulujen- ja aikataulunhallinta sekä lopputuotteen laatu [HiT00]. Näistä attribuuteista kulut ja aikataulu ovat helposti ymmärrettävissä. Ongelmallista on määrittää hyvän laadun sisältö. Voidaan sanoa, että hyvä ohjelmisto kattaa asiakkaan vaatimukset ja tyydyttää asiakkaan ohjelmistoon liittyvät tarpeet. Laatua arvioitaessa on otettava huomioon ohjelmiston ei-toiminnalliset vaatimukset, kuten luotettavuus, suorituskyky, laajennettavuus ja käytettävyys [HiT00].

Hyvässä ohjelmistoprojektissa budjetti ja aikataulu ovat arvioitu niin, että ne pitävät paikkansa projektin lopussa. Ohjelmistoprosessi on hyvä, jos se sisältää elementit, jotka vaativat ottamaan asiakkaan toiveet ja vaatimukset huomioon prosessin eri vaiheissa ja tuottamaan ne toimivina ominaisuuksina tuotteeseen.

Tutkittaessa ohjelmistotuotannon laatutekijöitä huomataan, että budjetti ja aikataulu ovat projektin raamit ja ohjelmiston laatu on asia, joka pyritään asettamaan raamien sisään [HiT00]. Ohjelmistoteollisuus on havainnut tuotteiden laatuongelmat, ja nykyään projekteissa käytetään huomattava osuus budjetista laadunvarmistukseen. Tämä laatuun budjetoitu raha käytetään tavallisesti järjestelmätestausvaiheessa. Tässä vaiheessa virheiden ja puutteiden korjaamisen on arvioitu olevan kaikkein kalleinta.

Noin puolet järjestelmän puutteista ja virheistä syntyy määrittely- ja suunnitteluvaiheessa. Virheen korjaaminen on noin 50 kertaa kalliimpaa valmista tuotetta testattaessa kuin prosessin alkuvaiheessa [HiT00]. Ohjelmistoprosessissa, jossa virheiden paikallistamiseen ja laadunvarmistamiseen keskitytään prosessin alusta alkaen, havaitaan vähemmän virheitä testausvaiheessa. Näin testauksessa havaittujen vikojen korjaamiseen käytettävien resurssien määrä pienenee. Tällä perusteella voidaan sanoa, että laadunhallinta prosessin alkupäässä laskee ohjelmiston kokonaiskuluja.

4.2 Hyväksymistestaus osana V-mallia

Ohjelmistotestaus on prosessi tai prosessien sarja, jolla pyritään varmistamaan, että ohjelmisto tekee ne toiminnot, joita sen on suunniteltu toteuttavan eikä toisaalta tee mitään odottamatonta. Testaamisen jälkeen ohjelmiston tulee olla toiminnaltaan yllätyksetön, ennakoitava ja johdonmukainen [Mye79]. Ohjelmistotestaus on ”toimintaa, jossa ohjelmistosta etsitään virheitä” [Mye79]. Testauksella ei voida osoittaa järjestelmän olevan täysin virheetön. Yksinkertaisestakin ohjelmasta on mahdollista saada niin suuri määrä testejä, ettei kattava testaaminen ole fyysisesti mahdollista. Tämän takia testaamisella pyritään ensisijaisesti etsimään virheitä ja korjaamaan niitä niin paljon, että ohjelmiston laatuun pystytään luottamaan.

Hyväksymistestauksessa joudutaan luottamaan suurelta osin aikaisempien testien kattavuuteen ja oikeellisuuteen, sillä hyväksymistesteissä ei ole näkyvyyttä ongelman rakenteisiin tai ratkaisuihin. Hyväksymistestauksen roolina on vakuuttaa käyttäjät ja tilaaja ratkaisun toimivuudesta todellisessa käyttöympäristössä. Hyväksymistestauksessa verrataan järjestelmän toimintaa sen alkuperäisiin vaatimuksiin ja loppukäyttäjien sen hetkiseen tarpeeseen ohjelmiston suhteen. Ohjelmistotestauksella on sopimustekninen rooli ohjelmistokehityksessä, sillä käyttäjät hyväksyvät rakennetun ohjelmiston ja sen toiminnallisuuden, jolloin kehittäjien näkökulmasta kehitysprojekti päättyy. Nimensä mukaisesti hyväksymistestauksessa loppukäyttäjät hyväksyvät tai hylkäävät kehitetyn ohjelman toiminnallisuuden. Tämä poikkeaa muista testauksen vaiheista, joissa keskitytään virheiden paikallistamiseen [HsK97]. Hyväksymistestauksessa keskitytään ohjelmiston toiminnallisuuteen sekä käyttäjien ja muiden järjestelmien välisen kommunikoinnin testaamiseen syöte/palaute -parien avulla.

Hyväksymistestaamiseen ei ole vakiintunut menetelmiä testitapausten mallintamiseen, rakentamiseen, formalisoimiseen tai verifioimiseen. Käytännössä hyväksymistestaaminen tehdään tapauskohtaisesti. Tapauskohtaisen testauksen ongelmana on, että testitapausten suunnittelu ja testien toteutus perustuu pääasiassa testaajien kokemukseen sovellusalueesta ja ohjelmistosta [Bin99].

Hyväksymistestin tapaukset tehdään tavallisesti sovellusten käyttäjien toimesta niin, että verrataan ohjelmiston toimintaa suhteessa vaatimuksiin. Vaatimusmääritysten ollessa sanallisesti kirjoitettu, on oikeellisuuden ja kattavuuden tarkistaminen vaikeaa. Ongelmana on, että sanallisten vaatimusten perusteella on vaikea johtaa sopivaa

vaatimusjoukkoa testitapausten pohjaksi. Tämän perusteella voidaan esittää kysymys: Voidaanko vaatimukset esittää muodossa, jossa ne tukevat hyväksymistestausta paremmin?

4.3 Kombinatorinen testaus

Kombinatorinen testaus perustuu taulukoilla esitettäviin totuus- ja päätöstauluihin, joilla kuvataan testitapauksia. Kombinatorisen testauksen juuret ovat kombinatorisessa logiikassa, joka esiteltiin 70-luvulla. Kombinatoriselle testaukselle on tyypillistä, että se tarjoaa yksinkertaisen esitystavan vaatimuksille. Vaatimukset esitetään taulukossa, jossa palautteen tai palautteiden joukon oikeellisuus tarkistetaan vertaamalla palautteita taulukossa esitettyihin ehtoihin [Bin99]. Taulukoita voidaan soveltaa ohjelmiston eri kehitysvaiheissa rakenteesta moduuleihin ja aina yksittäisiin ohjelmayksikköihin asti. Taulukot tukevat sekä automaattista että manuaalista testaamista.

Tyypillisesti kombinatorisia testejä suunniteltaessa ensin mallinnetaan ongelma päätöstauluun. Tämän jälkeen testitapaukset validoidaan. Lopuksi testitapaukset suoritetaan ratkaisujärjestelmässä ja verrataan tuloksia taulukossa esitettyihin ehtoihin.

4.4 Taulukot testitapausten esittämisessä

Ohjelmiston täydellisen kattava testaaminen ja siten ohjelmiston oikean toiminnan osoittaminen on mahdotonta [DDH72]. Testaamisen luotettavuutta ohjelmiston oikean toiminnan todistamisessa on aika ajoin kyseenalaistettu ja vaihtoehdoksi on tarjottu formaalia todistusta [Hal88]. Myöhemmin on osoitettu, että formaali todistus sisältää lukuisia ongelmia. Alkuperäinen vaatimus voi olla virheellinen, jolloin todistuksella osoitetaan väärä vaatimus sisällöltään oikeaksi. Toisaalta komponentti voi olla virheellisesti toteutettu eikä välttämättä sisällä kaikkia vaadittuja elementtejä. Mikäli näitä elementtejä on edellytetty todistuksessa, todistus on virheellinen.

Tarvitaan yhdistelmä eri menetelmiä, koska testaamiseen eikä matemaattiseen todistukseen voida täysin luottaa. Toteutettavilla ohjelmilla on taloudelliset rajoitteet, jolloin testaamiseen ja oikeaksi todistamiseen on käytettävissä vain rajallinen määrä aikaa ja muita resursseja. Tästä syystä huomio on kiinnitettävä oikeisiin testi- ja verifointi menetelmiin [CIP05].

Testitapausten valinnassa hyödyllisiä arvoja ovat kohdat, joissa muuttujat vaihtuvat negatiivisesta positiiviseksi, järjestysasteikolta toiselle tai kun muu muutos tapahtuu muuttujien luonteessa. Esimerkiksi tilisiirron yhteydessä voidaan olla kiinnostuneita tilisiirron onnistumisesta positiivisella ja negatiivisella summalla, otto/pano -yhdistelmillä sekä näiden kahden muuttujan yhdistelmällä. Muuttujien määrän kasvaessa testitapausten määrä kasvaa, vaikka testitapauksissa keskitytään kiinnostaviin pisteisiin. Testaamista tarvitaan ohjelmistokehityksen eri vaiheissa, jotta välttytään liian suurelta määrältä testitapauksia tai liialliselta testien karsimiselta.

Tilisiirtoesimerkissä yksikkötesteillä voidaan tarkistaa ohjelman toiminta erityyppisillä luvuilla. Tarkistetaan myös, ettei ohjelma hyväksy muita vaihtoehtoja kuin määritellyt otto ja pano. Hyväksymistestaus on kiinnostunut siitä, toimiiko otto ja pano positiivisen ja negatiivisen luvun kombinaatiolla, kuten vaatimuksissa on esitetty. Taulukossa oleellista on sen helppokäsitteisyys eri sidosryhmille, sekä strukturoitu esitystapa. Yksikkötesteissä testitapausten suunnittelijana on ohjelmoija. Tällöin taulukko ei tuo esitystapaan lisäarvoa, sillä ohjelmointikieli on formaali esitystapa. Hyväksymistestauksessa tapausten sijoittaminen taulukkoon on perusteltua, jotta testitapausten rakenne pysyy rajoitettuna ja ohjelmaa testaavat osat rajallisena. Riittävän muodolliset, mutta helppokäyttöiset testitapaukset voidaan liittää toteutettavan ohjelmistoon testitapauksiksi, joka mahdollistaa automaattisen testauksen.

4.5 Automatisoitu testaus

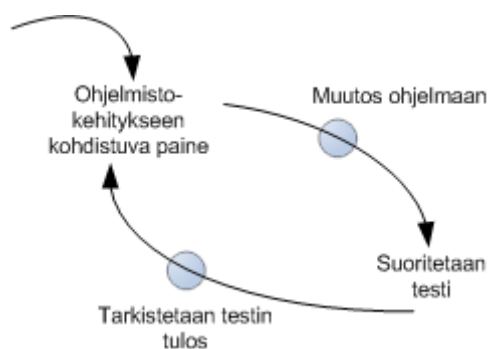
Manuaalinen testaus, jossa testaaja suorittaa testiohjelman ja käynnistää toimintoja käyttöliittymältä on aikaa vievää erityisesti, jos testi täytyy toistaa useissa eri ohjelmaversioissa ja järjestelmäympäristöissä [Mel07]. Lisäksi manuaaliset testit ovat alttiita virheille. Käyttöliittymäsovelluksissa sovelletaan nauhoitettuja testitapauksia, joilla käyttäjä antaa testiohjelman skenaarion mukaiset syötteet ohjelmalle ja kertoo testinauhurille onko vastaus oikein vai väärin. Tätä nauhoitusta voidaan suorittaa regressiotestinä myöhemmin uudelleen. Ongelmana tällaisissa testeissä on, että testitapaukset vanhentuvat pienistä muutoksista käyttöliittymällä eikä niillä voida testata logiikkaa järjestelmän taustalla [Mel07]. Suositeltavampaa on liittää testitapaukset käyttöliittymän alla olevaan ohjelmakoodiin, jota on helpompi ylläpitää muutosten yhteydessä. Näin voidaan varmistua ohjelman perusrakenteiden olevan kunnossa muu-

tosten jälkeen. Tässä menetelmässä käyttäjien hyväksymistestaamista tarvitaan edelleen käyttöliittymän testaamiseen. Käyttäjien tekemä testaaminen on hyvä suorittaa automaattisesta testaamisesta huolimatta, jotta voidaan varmistua käyttäjien hyväksynnästä.

E erityisesti regressiotestauksen automatisoiminen on hyödyllistä. Regressiotestien kuluttaessa paljon aikaa on vaarana, ettei niitä ehditä tekemään perusteellisesti. Tällöin kadotetaan tietämys ohjelmiston tilasta muutosten jälkeen. Menestyksekkäät ohjelmistoprojektit käyttävät testien automatisoimista parantaakseen ohjelmiston joustavuutta muutoksiin, ei vähentääkseen testaajien määrää [Jon97].

4.5.1 Test Driven Development

Automatisoidun hyväksymistestauksen pohjalla voidaan käyttää Kent Beckin esittelemää Test Driven Development (TDD) menetelmää. TDD:ssä on kolme periaatetta: tee testitapaukset ennen ohjelmointia, vältä päällekkäistä testausta ja automatisoi testitapaukset [Bec03]. TDD:ssä on esitetty ratkaisumalli ongelmaan, jossa kehittäjillä ei ole aikaa testaamiselle. Ohjelmakoodia kehitettäessä ohjelmoija saa testauksesta palautetta ohjelman toiminnasta. Palautteen jälkeen joko ollaan tyytyväisiä tulokseen tai korjataan ohjelmaa ja ajetaan testit uudelleen. Asia voidaan esittää kuvana.



Kuva 9: Ohjelmistokehittäjän ongelmien kehä [Bec03]

Mitä enemmän kehittäjä kohtaa painetta ohjelmiston kehitykseen, sitä vähemmän hänelle jää aikaa testaamiseen. Mitä vähemmän aikaa on testaamiselle, sitä enemmän syntyy virheitä. Mitä enemmän syntyy virheitä, sitä enemmän kuluu aikaa virheiden korjaamiseen. Tästä syntyy ohjelmistokehittäjän ongelmien kehä. Kehä voidaan katkaista aloittamalla ohjelman kehittäminen testitapauksen suunnittelulla ja vanhojen testitapauksien ajamisella. Mitä enemmän kehittäjä kohtaa aikataulupainetta muutosten tekemiseen, sitä useammin hän testaa muutoksia testitapauksilla. Testeistä muodostuu kehittäjän tukipylväs, jota hän voi käyttää saadakseen varmuutta ohjelman toiminnasta.

Muutosten jälkeen hyväksytysti ajetut testitapaukset antavat varmuuden tunteen siitä, että muutos ei ole rikkonut aikaisempaa toiminnallisuutta [Bec03].

4.5.2 Acceptance Test Driven Development

Acceptance Test Driven Development (ATDD) mallissa on ajatuksena automatisoida hyväksymistestit. Ohjelmiston laatu paranee, kun vaatimusmäärittelyt tehdään riittävän yksityiskohtaisesti aina testitapausten tarkkuudella jo vaatimusmäärittelyvaiheessa. Muutoksiin vastaaminen paranee sekä ohjelmistoprojektin aikana että tuotteen ylläpidossa, kun muutokset voidaan testata heti koko ohjelmiston tai sen osan laajuudelta [Rog04].

ATDD:tä voidaan perustella esimerkiksi tilanteella, jossa ohjelmistotuotteen myyjä kertoo tuotepäällikölle, että uusi asiakas on kiinnostunut käyttämään järjestelmässä eurojen lisäksi myös muita valuuttoja. Tämä muutos pitää saada toimintaan lyhyen ajan sisällä tai asiakas valitsee kilpailijan tuotteen, josta kyseinen ominaisuus jo löytyy. Tuotteeseen on lähiaikoina investoitu satoja tunteja kehitystä ja testausta. Valuuttamuutos tarkoittaa muutoksia useisiin olemassa oleviin näyttöihin sekä taustalla tehtäviin laskutoimintoihin. Tämä merkitsee tuotteen valuuttaominaisuuden uudelleen testaamista, koska muutos ei saa vaikuttaa negatiivisesti järjestelmän nykyisiin käyttäjiin. Mikäli hyväksymistestaus ja testaaminen yleensä on tehty käsityönä, niin kehityksessä ja sen jälkeisessä regressiotestauksessa tulee todennäköisesti kestäämään niin kauan, että asiakas ostaa palvelun kilpailijalta. Hyväksymistestausperusteisessa lähestymisessä varaudutaan jo vaatimusmäärittelyvaiheessa tuleviin muutoksiin ja automatisoidaan testitapaukset. Voidaan olettaa, että mikäli muutokset ovat arkkitehtuurin ja ohjelman yleisen rakenteen kannalta toteutettavissa, niin muutos voidaan tehdä. Laatu voidaan varmistaa automaattisella regressiotestauksella [Rog04].

Yhdeksi perusteluksi automatisoidulle hyväksymistestaukselle mainitaan vaatimusmäärittelyn laadun paraneminen. Täsmällisten testitapausten myötä saadaan kaivettua esiin yksityiskohtia, joita muuten on vaikea havaita ennen testaamista. Testitapaukset toimivat kommunikointivälineenä asiakkaan ja tuotantotiimin välillä [Rog04]. Testitapauksista tulee yhteinen sovellusalakohtainen kieli asiakkaan ja kehitystiimin välillä. Tässä kehitystiimin vastuulla on järjestää asiakkaalle mahdollisuus tuoda testitapaukset esiin riittävän täsmällisesti testien automatisoimiseksi ja asiakkaan vas-

tuulla on selventää testitapaukset kehitystiimille riittävän perusteellisesti. Asiakkaan suorittaessa testejä käyttäjät pystyvät paikallistamaan aikaisessa vaiheessa ohjelmiston epäloogisuuksia, jolloin korjaukset ohjelmaan on mahdollista tehdä aikaisessa vaiheessa asiakkaan toiveiden mukaisesti. Lisäksi etuna on vaatimusten ylläpidon helpottuminen. Muutokset testitapauksiin on aina tehtävä, koska testien tulee suoriutua hyväksytysti, ennen kuin tuote otetaan käyttöön.

Asiakas tarvitsee paljon tukea ATDD:n järjestämisessä. TDD:ssä testauksen kohteena ovat yksittäiset ohjelman osat, kun taas ATDD:ssä testataan useita taustalla suoritettavia aliohjelmakokonaisuuksia. Asiakkaalle taustalla toimivien osien tunteminen ei ole tarpeellista. ATDD:n vaatima täsmällinen esitys on tehtävä yhteistyössä asiakkaan ja kehitystiimin kanssa.

Ensimmäiseksi asiakkaan ja kehitystiimin välille on luotava yhteiset säännöt siitä, kuinka testitapaukset esitetään. Asiakkaalle luontevinta on esittää testitapaukset luonnollisena kielenä, mutta automatisoinnin kannalta ajatus ei ole käytännöllinen.

Ohjelmoijat ovat tottuneet käyttämään ohjelmistokehitysokaluja, joissa voidaan tehdä muutoksia ohjelmaan ja sen jälkeen suorittaa ohjelma oikeellisuuden tarkistamiseksi [Rog04]. ATDD:stä asiakkaalle voidaan rakentaa käyttöympäristö, jossa hän pääsee lisäämään ja muokkaamaan testitapauksia. Parhaassa tapauksessa asiakkaalla on mahdollisuus itse suorittaa testitapaukset testattavaan sovellukseen ja saada tulokset välittömästi arviotavaksi.

ATDD:n ongelma suhteessa TDD:hen on testitapausten monimutkaisuus. Yksikkötesteissä testataan vain yksittäistä osaa, jolloin testitapausten ajamiseksi riittää, että käännetään yksikkö ja verrataan tulokset. ATDD -testitapauksissa voidaan joutua kääntämään lukuisia yksiköitä, lisäämään tietoa järjestelmään ja suorittamaan aliohjelmaa. Testien suorittamisen jälkeen testin vaikutukset poistetaan järjestelmästä, mikä vaatii testiä ajavalta ohjelmalta paljon toiminnallisuutta.

Laajoissa järjestelmissä testitapausten päällekkäisyyksien hallinta on ongelmallista. Regressiotestauksessa voidaan päätyä tilanteeseen, jossa samaa ohjelman osaa testataan useilla testitapauksilla. Esimerkiksi osoitetieto ja sen käsittely voi olla toteutettuna ohjelmassa yhden kerran, mutta käyttäjä näkee saman toiminnallisuuden usealla eri käyttöliittymätoiminnolla. Tällaisessa tilanteessa samalle toiminnolle tehdään useita tes-

titapauksia. Ongelman ratkaisemiseen asiakas tarvitsee kehitystiimin tukea, jotta päällekkäisyyksiä voidaan hallita tehokkaasti. Testitapaukset voidaan osittaa vastaavasti kuin ohjelman osat, jolloin testitapauksiin tehtävät muutokset vähenevät [Rog04].

4.5.3 Hyväksymistestauksen automatisoiminen

Hyväksymistestien automatisoinnin laajuus ja mielekkyys vaihtelee sen mukaan, kuinka laaja ohjelmisto on kyseessä, kuinka paljon muutoksia siihen kohdistuu, millainen budjetti ohjelmistolla on ja miten kriittiseen käyttöön ohjelmaa sovelletaan. Käytössä oleva ohjelmointikieli vaikuttaa siihen, kuinka helposti testit ovat käytännössä automatisoivissa. Useisiin ohjelmointikieliin on tarjolla testauskehyskiä hyväksymistestien automatisoimiseksi.

Hyväksymistestien automatisoiminen johtaa tiukempaan integraatioon vaatimusmäärittysten ja hyväksymistestien suhteen [PaM08]. Hyväksymistestien automatisoimisessa on kysymys vaatimusmäärittysten kirjoittamisesta automaattisesti suoritettavassa muodossa. Hyväksymistestitapaukset ovat suoritettavissa heti, kun sovellus kytketään testitapauksiin. Testitapauksia tulkitsemalla ja tekemällä testitapausten kytkentöjä ohjelmaan, saavat ohjelmoijat yksityiskohtaisen kuvan ohjelman käyttäytymisestä suhteessa vaatimusmäärittäisiin. Ohjelmiston korkea laatu säilyy, kun testit ovat vaatimusten muuttuessa uudelleen testattavissa automaattisesti.

Automaattisesti suoritettavilla hyväksymistestitapauksilla ei tarkoiteta formaaleja matemaattisia määrittäisiä, jotka ovat usein myös automaattisesti suoritettavissa. Hyväksymistestien automatisoinnilla tarkoitetaan testitapauksia, jotka koostuvat automaattisesti suoritettavista osista sekä luonnollisesta kielestä [PaM08]. Automatisoimalla hyväksymistestit sekä asiakas että kehittäjät joutuvat esittämään vaatimukset yksiselitteisellä tavalla. Tämä poistaa useita vaatimusmäärittelylle tavallisia ongelmia, kuten häilyä ja ylimäärittelyä [PaM08]. Automatisointiin käytetyt kustannukset pystytään tavallisesti kattamaan 2-3 kehityskierroksen jälkeen [Bin99]. Automatisointi vähentää ohjelmiston riippuvuutta henkilöstöstä, joka on ollut mukana kehityksessä ja testauksessa. Automatisoidut tapaukset pysyvät yhdenmukaisina, vaikka ihmiset vaihtuisivat.

Testejä pidetään ohjelmistokehityksessä pakollisena hallinnollisena kuluna, joka ei ole tuottavaa työtä tai lisää ohjelmiston arvoa. Useat testitapaukset koetaan niin yksinkertaisina, että niiden yksityiskohtainen esittäminen tuntuu asiakkaista liioittelulta [PaM08].

Sovelluksen kasvaessa, pienten yksityiskohtien muistaminen muuttuu nopeasti mahdottomaksi, jolloin niiden tallentaminen on välttämätöntä. Testauksesta tulee tehdä mahdollisimman yksinkertaista, jotta se on mahdollista toteuttaa kiinteänä osana ohjelmiston suunnittelua ja toteutusta [Bec99]. Riittämättömällä testauksen tuella on käytännön toteutuksessa riski, että testaaminen jää tekemättä riittävällä tarkkuudella, jolloin ohjelmiston laatu kärsii. Testaus tulee automatisoida, jotta regressiotestauksessa käyttäjät voivat keskittyä testitapausten kehittämiseen ja ylläpitämiseen testitapausten suorittamisen sijaan.

Hyväksymistestauksen mustalaatikkotestaus luonteen johdosta hyväksymistestien automatisoimisessa on enemmän työtä kuin automatisoitaessa yksikkötestejä. Suora asiakaskontakti on hyväksymistesteissä välttämätöntä, mikä ei yksikkötestauksessa ole tarpeellista. Asiakkaan päivittäessä testitapauksia tai lisätessä uusia, joutuu ohjelmointitiimi tekemään vastaavat päivitykset metodikutsuihin. Tämä vaatii testitapausten ylläpitoon resursseja kehitystiimissä [And04].

Hyväksymistestauksella on ohjelmistokehityksessä oma paikkansa, joka poikkeaa yksikkötestauksesta merkittävästi. Automatisoimalla hyväksymistestit pyritään osoittamaan, että ohjelma tekee oikeita asioita, kun yksikkötesteillä pyritään osoittamaan, että ohjelma tekee sille määritellyt asiat oikein. Tästä syystä yksikkötestaus on tärkeää suorittaa automatisoitujen hyväksymistestien rinnalla. Tyypillisesti hyväksymistesteistä suurin osa epäonnistuu projektin alkuvaiheessa, joka on hyväksyttävää, koska kokonaisuus ei ole vielä valmis. Yksikkötesteillä taas on olennaista, että kaikki laaditut testit suoriutuvat hyväksytysti, ennen kuin kehitystä jatketaan eteenpäin. Hyväksymistestit keskittyvät ohjelman toiminnallisiin laatuominaisuuksiin, kun yksikkötestit keskittyvät ei-toiminnallisiin ominaisuuksiin. Hyväksymistestauksessa on tavallisesti enemmän osallistujia eri sidosryhmistä, joiden käsitykset ohjelmistosta ovat erilaiset. Tämä aiheuttaa hyväksymistestaukselle ja siihen käytetylle tekniikalle erilaiset haasteet kuin yksikkötestaukselle.

Hyväksymistestauksessa testataan useita ohjelman toiminnallisia osia yhdellä testitapauksella, testaamiseen osallistuu useita sidosryhmien jäseniä sekä vuorovaikutusta voi olla vanhojen perinnejärjestelmien kanssa. Tätä taustaa vasten on ymmärrettävää, että Unit:n kaltaista yleisesti hyväksyttyä testauskehystä ei ole hyväksymistestaukseen esitelty. Tällä hetkellä tavallisesti käytetty työkalu on Fit ja siitä tehdyt muunnokset, kuten

Fitnesse [MMC06]. Kehitystiimi voi myös rakentaa oman testausjärjestelmän, joka tallentaa ja suorittaa testitapauksia.

5 Vaatimusmäärittely ja testitapausten suunnittelu – Framework for Integrated Testing (FIT)

FIT on ohjelmistokehys, joka tukee automatisoituja hyväksymistestejä. Fit perustuu kahteen kerrokseen, jossa päällimmäisessä kerroksessa tehdään vaatimusmäärittelyt taulukoihin ja alemmassa kerroksessa ohjelmoidaan metodit, joilla varsinaista ohjelmaa kutsutaan. Kuvassa 10. on esimerkki kerrosrakenteesta Fit -menetelmällä toteutettuna.

Testitapaus: Osta liput ja tarkista palaute			
Näytös	Määrä	Palaute	Tilauksen kokonaishinta
Oopperan kummitus	2	Olet ostanut 2 lippua näytökseen Oopperan kummitus.	150 eur
Cats	1	Cats näytös on varattu loppuun.	50 eur
Othello	1	Othello näytös ei ole tällä hetkellä ohjelmistossa.	Odotettu: 79 eur Tulos: 78.99 eur

```

package lippujenVaraus;
import fit.ColumnFixture;

public class OstaLiput extends ColumnFixture{
    public int naytos;
    public int lippujenMaara;

    //Testitapaus yhteishinnan laskemiselle
    public int laskeKokonaishinta(){
        Item yksikko = new Item(naytos);
        yksikko.lisaaHinta(lippujenMaara);
        return item.kokonaisHinta();
    }
}

```

Kuva 10: Fit:llä automatisoitu testitapaus

Esimerkissä vasemmalla näkyy vaatimusmäärittelyssä kuvattu käyttötapaus, johon oikealla puolella on tehty testikerros (fixture). Testikerroksessa ohjelmoidaan metodikutsu testattavalle ohjelmalle, tarkastetaan sen antama palaute ja verrataan vaatimusmäärittelyssä annettuun syötteeseen. Käyttäjälle tulostetaan ilmoitus positiivisesta ja negatiivisesta tuloksesta.

Fit:n rakenteena on taulukkomuoto, jossa käyttäjä kuvaa järjestelmän toimintaa vapaamuotoisesti, mutta silti automaattisesti verifioitavalla tavalla [MaM08]. Tässä esitystavassa vältetään luonnollisen kielen aiheuttama häly, kun ohjelman toiminta on rajattu täsmällisempään muotoon kuin luonnollisella kielellä kuvattaessa. Kuvauksessa

ohjelmistotiimi saa käsityksen järjestelmän tietotyypeistä ja sisäisestä esityksestä. Kuvasta 10. voidaan esimerkiksi päätellä, että näytökset tulee sijoittaa erilliseen tietokantatauluun. Taulusta käyttäjä voi vetovalikolla valita vain niitä näytöksiä, joita taulussa esitetään kyseisenä ajankohtana. Määrät tulee käsitellä kokonaislukumuodossa ja tarkistaa syötteiden kelvollisuus ohjelmalle.

Taulukkoa tarkasteltaessa on vaikea havaita onko kyseessä ohjelmistoon kohdistunut vaatimus vai testitapaus. Testitapausten toteuttajan tulee voida antaa vastaavat syötteet järjestelmälle ja tarkistaa tulokset vaatimusmäärittelyiden perusteella. Taulukkomuoto mahdollistaa testitapausten automatisoinnin, jolloin testaamiseen tarkoitettu järjestelmä voi näyttää punaista tai vihreää palautetta testitapausten onnistumisen perusteella. Tätä vaatimusmäärittelyiden ja testitapausten suhdetta kutsutaan ekvivalenssihypoteesiksi: ”Formaalisuuden lisääntyessä testitapaukset ja vaatimusmäärittelyt yhtenevät” [MaM08].

Hypoteesia voidaan testata kirjoittamalla määrittelyt ensin luonnollisella kielellä ja testitapaukset sen perusteella. Siirryttäessä määrittelyiden tarkkuudessa kohti täsmällistä taulukkoesitystä, määrittelyiden tulee muuttua vaikeammin testattavaksi, jolloin hypoteesi ei pidä paikkaansa. Toiseen suuntaan yhtenevyyttä voidaan testata kirjoittamalla testitapaukset luonnollisella kielellä kuvattujen määrittelyiden pohjalta ja mikäli testitapausten johtaminen on tehokkaampaa kuin taulukosta, on hypoteesi väärä.

Parhaimmillaan vaatimusmäärittelyinä kuvatut testitapaukset ovat käyttötapauksia, joissa käyttäjä kuvaa taulukoilla ohjelman halutun toiminnan. Tällaiset käyttötapaukset ovat sekä asiakkaalle että ohjelmoijalle ymmärrettäviä ja johdonmukaisesti testattavissa. Lisäksi taulukkoesitys on päivitettävissä, jolloin testitapaukset pysyvät ajan tasalla vaatimusten muuttuessa.

Ohjelmiston käyttäjien ja tilaajien kyky suunnitella täsmällisesti ohjelmiston toimintaa vaihtelee suuresti. Fit:n lähestymistapa yksinkertaistaa asiaa, mutta se ei välttämättä riitä hyviin tuloksiin. Ohjelmistokehityksen tuki asiakkaalle on tärkeää. Vaatimusmäärittelyiden suunnittelusta täsmällisellä tasolla tehdyn tutkimuksen mukaan Fit -menetelmä auttaa asiakasta tuomaan toiveensa selkeämmin esille ja tukee vaatimusmäärittelyprosessia hyödyllisen informaation esiin tuomisessa mahdollisimman aikaisessa vaiheessa [MMC06].

5.1 FIT ja vaatimusmäärittelyn ongelmat

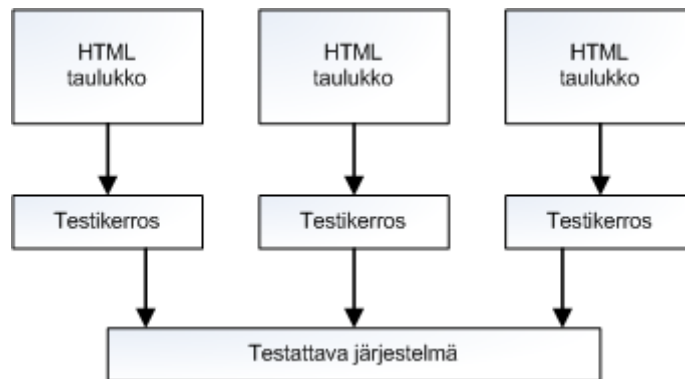
Fit -menetelmästä tehdyn tutkimuksen perustella menetelmä pystyy hyvin vähentämään vaatimusmäärittelyssä tavallisesti vastaantulevia ongelmia. Fit -testit soveltuivat hyvin kuvaamaan ohjelmoijille tarpeita, joita käyttäjät ohjelmalta haluavat. Fit -menetelmän oppimiskäyrä on matala, koska 90 % käyttäjistä palautti Fit testitapaukset, vaikka käytöstä ei ollut aikaisempia kokemuksia [MRM04].

Tutkimuksen johtopäätöksenä oli, että hälyä ja ylimäärittelyä on huomattavasti vaikeampi saada mahtumaan Fit -kehiksen asettamiin raameihin kuin luonnollisen kielen esitykseen. Häly saattaa kuitenkin aiheuttaa ongelmia, kun kehitetään jo olemassa olevaan järjestelmään uusia osia. Tällöin käyttäjä helposti lisää uuden testitapauksen ominaisuudelle, jolle on olemassa aikaisempi testitapaus edelliseltä kehityskierrokselta. Tällöin ohjelmalle voi samalle toiminnolle olla useita testitapauksia, mikä lisää dokumentoinnin hälyä ja vaikeuttaa ylläpidettävyyttä. Toiveajattelua Fit -kehys vähentää tehokkaasti, sillä taulukkoesitys vaatii testitapausten suunnittelijaa ilmaisemaan asiat taulukon antamissa puitteissa, jolloin toiveajatukset täsmentyvät vaatimuksiksi. Moniselitteisyyttä Fit -menetelmä ei kykene täysin ratkaisemaan, sillä käyttäjä voi lisätä kenttiä ja niiden sisältöä haluamallaan tavalla. Tällöin saattaa esimerkiksi syntyä tilanne, jossa taulukon sisältö on esitetty luonnollisena kielenä moniselitteisesti. Hiljaisuus sen sijaan on Fit -menetelmässä ongelma ja voi olla jopa hankalammin estettävissä kuin luonnollisessa kielessä. Fit -menetelmässä käyttäjän tulee pystyä täsmällisesti esittämään vaatimukset, jotka taas luonnollisessa kielessä on rikkaammin kuvattavissa. Fit -menetelmää käytettäessä 50 % vaatimuksista jäi testiryhmältä havaitsematta [MRM04].

5.2 FIT:n rakenne

Fit koostuu asiakkaan määrittelemistä taulukoista ja kolmesta eri testikerroksesta, ColumnFixture:sta, RowFixture:sta ja ActionFixture:sta. Fit:n osana on myös html -tiedostonkäsittelijä, joka purkaa taulukot testattaviksi metodeiksi [Cun07]. Taulukot edustavat käyttäjien näkemystä ohjelman toiminnallisuudesta [Mug04]. Taulukoihin ohjelmoijat rakentavat erityyppisiä testikerroksia riippuen asiakkaan käyttötapauksesta. Asiakas voi rakentaa taulukot esimerkiksi taulukkolaskenta- tai tekstinkäsittelyohjelmassa ja tallentaa ne html -muotoon. Fit -kehys sisältää html -tiedostonkäsittelijän, joka purkaa taulukon sisällön tietorakenteeseen. Tietorakenteen sisältöä verrataan ohjelman

tuottamiin palautteisiin.



Kuva 11: Taulukoista testikerroksen kautta automaattisiin testitapauksiin [MuC05]

Raportoinnissa Fit käyttää asiakkaan luomia taulukoita, joihin palaute lisätään upotettuna väreillä. Punainen vastaa epäonnistunutta testitapausta, jossa raportoidaan ohjelman antama palaute sekä asiakkaan kirjoittama odotettu arvo. Keltainen väri kertoo ohjelman tuottamasta poikkeuksesta. Tästä tulostetaan ohjelman lähettämä poikkeussanoma. Vihreä kertoo testin suoriutuneen onnistuneesti.

ColumnFixture:t ovat taulukoiden yksinkertaisin ilmentymä, jossa testataan sovelluksen laskutoimitusten tuloksia. Oheisessa esimerkissä sovellus laskee kertyneistä pisteistä muodostuvan bonuksen. Ensimmäinen rivi kertoo testikerroksen nimen. Seuraavalla rivillä esitellään muuttujien nimet sekä metodit, joita testikerroksen välityksellä kutsutaan testattavasta ohjelmasta. Seuraavilla riveillä annetaan ohjelmalle välitettävät arvot sekä niiden perustella odotetut tulokset. Ohjelman palauttamia tuloksia verrataan käyttäjän taulukkoon syöttämiin arvoihin.

fitTstPkg.LaskeBonus	
pisteet	bonus()
1000	0
1999	0
2000	500
2050	500
2100	1000
2200	1500
2300	2000
2350	2000
2400	2500

Kuva 12: ColumnFixture -esimerkki

Ohjelmoija täydentää käyttäjän laatimaa taulukkoa ohjelmoimalla testikerroksen, joka kertoo mitä metodeita ja kenttiä testattavasta ohjelmasta kutsutaan. Ohjelma laajentaa ColumnFixture -luokkaa fit.jar paketista ja esittelee muuttujan ”pisteet”, jonka oletetaan löytyvän asiakkaan taulukosta samalla nimellä. Lisäksi ohjelmassa on esitelty metodi bonus(), joka kutsuu testattavaa ohjelman osaa ”BonusLaskentaJarjestelma”.

```
package fitTstPkg;

import fitPkg.*;
import fit.ColumnFixture;

public class LaskeBonus extends ColumnFixture{

    public int pisteet;

    public int bonus () {

        BonusLaskentaJarjestelma l = new BonusLaskentaJarjestelma();

        return l.bonusLaskentaJarjestelma(pisteet);
    }
}
```

Kuva 13: ColumnFixturen testauskerros

Määrittelyvaiheessa kyseistä ohjelmaa ei ole vielä olemassa. Kehittäjällä on kuitenkin ollut ajatus, että tällainen osa ohjelmaan on kehitettävä, jotta laskentatehtävä saadaan suoritettua. Valmis ohjelma kytketään kiinni sekä käyttöliittymään, että Fit-testikerrokseen, jolloin taulukoista syntyy kevyt kehitystyön aikainen käyttöliittymä eri sidosryhmien testattavaksi.

ColumnFixture:lla voidaan testata vain hyvin rajallista osaa tyypillisen ohjelman toimintoista. Tietojoukkojen testaamiseen käytetään RowFixturea. Käyttäjä on esimerkiksi voinut määritellä haluavansa hakea tietokannasta tapahtumia ja määritellyt kyseiset tapahtumat taulukkoon. Varsinaisessa ohjelmassa on metodi, joka käsittelee tietokantaa ja tekee käyttäjän toivomat haut tietokantaan. RowFixture kutsuu esimerkiksi tietokantametodia, joka palauttaa joukon tietokantarivejä tai olioita. RowFixture vertaa asiakkaan html-taulukkoa tietokannasta palautettuun tietojoukkoon ja tarkistaa taulukon sisällön sekä puuttuvat ja ylimääräiset rivit.

fitTestPkg.Downl		
getIdd()	getAmount ()	getPrice()
0	0	0
1	10	20
2	50 <i>expected</i>	100 <i>expected</i>
	20 <i>actual</i>	40.0 <i>actual</i>
3	99 <i>expected</i>	188 <i>expected</i>
	30 <i>actual</i>	60.0 <i>actual</i>
4	100 <i>expected</i>	200 <i>expected</i>
	40 <i>actual</i>	80.0 <i>actual</i>
5	101 <i>expected</i>	202 <i>expected</i>
	50 <i>actual</i>	100.0 <i>actual</i>
8 <i>missing</i>	1561	3122
6 <i>surplus</i>	60	120.0

Kuva 14: RowFixturen tuloksia

Kolmas testikerros on ActionFixture, jolla testataan esimerkiksi käyttäjän käyttöliittymältä syöttämiä tapahtumia. ActionFixture sopii hyvin laajojen tapahtumakulkujen testaamiseen. Fit tarjoaa seuraavat tapahtumat, joita voidaan käyttää ensimmäisessä kolumnissa; start, enter, check ja press. Start-komennolla määritellään testitapaus, jota kutsutaan. Toisessa kolumnissa määritellään toiminnot ja esitellään muuttujien nimet. Ohjelmoijan on käytettävä samoja nimiä testit suorittavassa testikerroksessa. Kolmannessa kolumnissa käyttäjä antaa ohjelmalle syötteet sekä mahdollisesti odotetut paluuarvot.

fit.ActionFixture		
start	fitTestPkg.AddData	
enter	maara	0
enter	hinta	0
check	addData	true
enter	maara	1
enter	hinta	2
check	addData	true

Kuva 15: ActionFixturen tuloksia

Fit mahdollistaa taulukoiden liittämisen yhteen html-dokumenttiin, jolloin yhdelle sivulle voidaan kerätä kokonaisia liiketoimintaprosesseja. html-dokumentilla kuvataan esimerkiksi tiedon lisäämistä, sen prosessointia taustajärjestelmissä, tiedon muokkaamista sekä hakuja [GKS08]. Samalle html-dokumentille voidaan lisäksi syöttää tekstiä, koska Fit hakee tiedon taulukoista sen otsikon perusteella. Näin yhdelle html-dokumentille voidaan kuvata kokonaiset vaatimusmäärittelyt sanallisten selitysten ja mahdollisesti selkeyttävien kuvien kanssa. Fit poimii hälyn seasta testikerroksessa esiteltyt taulukot.

5.3 FIT:n rajoitteet

Fit tukee hyväksymistestien automatisoimista ja tehostaa ohjelmistokehitystä [And07]. Hyväksymistestitapaukset tulee olla koko kehitysryhmän työkaluna, josta vaatimuksia ja käyttötapauksia voidaan tarkistaa ja päivittää. Kehittyneeseen tiedonjakoon Fit ei kuitenkaan yksin sovellu. Fit:n ympärille on rakennettu joukko työkaluja tukemaan käytettävyyden ongelmia, kuten Fitnessse ja Fitclipse. Fitnessse on wikisovellus, jossa käyttötapaukset ovat jaettavissa verkon välityksellä ja joihin voidaan päivittää käyttötapauksia, taulukoita ja muita määrittelyjä luonnollisella kielellä. Fitclipse tallentaa historian aikaisemmin ajetuista testitapauksista, jotka voidaan tarvittaessa myöhemmin palauttaa [Den07]. Tämä madaltaa kynnystä muutosten tekemiseen testitapauksiin ja testikerrokseen, kun vaatimukset ja testit ovat helposti palautettavissa takaisin alkuperäiseen tilaan. Kehittäjille on Fitrunner -sovellus, joka helpottaa Fit:n käyttöä Eclipsen kehitysalustalla. Fitrunnerin avulla kehittäjän ei tarvitse huolehtia taustalla olevien tie-

dostojen käsittelystä ja testien ajamisesta. Fitrunner sisältää konfiguraatio -tiedoston, jossa tiedostorakenne määritellään. Konfiguraatiodokumentin avulla testitapaukset ovat nopeasti suoritettavissa kehitysympäristöstä.

Fit:n ympärille on rakennettu merkittävä joukko apusovelluksia tukemaan vaatimusmäärittelyjen tekemistä sekä kehittäjien tueksi. Tarvitaan kuitenkin jotakin, joka siirtää automatisoidun hyväksymistestauksen seuraavalle tasolle [PaM07]. Hyvistä vaatimusmäärittely- ja testaustyökaluista on selkeä puute, eikä Fit yksin ratkaise kaikkia näitä ongelmia. Kehittäjät ovat tottuneet käyttämään kehitysympäristöä, kuten Eclipse. Asiakkaalle ja vaatimusmäärittelydokumenteille on oltava vastaava ympäristö, jossa määritykset muokataan, päivitetään sekä säilytetään historia näistä dokumenteista. Asiakkaiden on pystyttävä itsenäisesti suorittamaan testitapauksia sekä lisäämään uusia tapauksia taulukoihin. Kehittäjillä tulee olla näihin taulukoihin reaaliaikainen pääsy. Uusien toimintojen kehittämiseen tarvitaan luonnollisesti asiakkaan ja kehittäjien kommunikaatiota, sillä kehittäjillä on oma tehtävänsä vaatimusten kytkemisessä valmisteilla olevaan sovellukseen. Asiakkaalla ja kehittäjillä tulee olla yhteinen pääsy vaatimusdokumentteihin, jotta kommunikaatio pysyy yhtenäisenä. Fitnessen tuki asiakkaille on oikean suuntainen, mutta ei vielä kata kaikki käyttötapauksia.

Fit:n rakenteellinen ongelma liittyy taulukoiden kaksiulotteisuuteen, joka rajoittaa vaatimusten ja testitapausten ylläpidettävyyttä. Kaksiulotteisessa taulukossa on mahdollista esittää suurin osa ohjelmalle asetetuista vaatimuksista, mutta se vaatii jo yksinkertaisissa sovelluksissa lukuisan joukon rivejä taulukkoon. Tietorakenteen tiedot on voitava tallentaa erilliseen taulukkoon, johon tarvittaessa voidaan tehdä viittauksia muista testitapaustaulukoista. Esimerkiksi asiakastietokannassa asiakkaat voidaan jakaa eri ryhmiin. Ryhmiä päivitetään ajan kuluessa, jolloin testitapauksia täytyy lisätä jokaiseen testitapaukseen, jossa asiakastyyppejä on käytetty. Mikäli näissä testitapauksissa voidaan viitata asiakastyyppeihin, säästetään virheiltä päivitystyötä.

Fit:ssä on ongelma, joka osoittautuu myös eduksi. Ongelma on hämärtyvä raja vaatimusten ja toteutuksen suhteen. Taulukoita määritettäessä toteutetaan vaatimukset ilman tietoa lopullisesta tavasta ratkaista ongelma. Kuitenkin viimeistään ohjelmointikieltä päätettäessä on tiedettävä onko kyseiseen kieleen saatavissa Fit:n toteutusta. Fit ei ole vielä riittävän geneerinen yleisen ohjelmistoprosessi tueksi, mutta siinä on toteutettu monia asioita hyvin. Hyväksymistesteissä keskeinen asia on tukea yksinkertaisuutta,

joka on eri sidosryhmien omaksuttavissa lyhyessä ajassa tuntematta ohjelmointitekniikkaa. Tämä vaatimus on ristiriidassa ohjelmistojen monimutkaisuuden kanssa, joka aiheuttaa ongelmia vaatimusten kattavuuteen [MuT03]. Seuraavissa kappaleissa esitetään ajatuksia edellä mainittujen asioiden ratkaisemiseksi.

6 Ohjelmistokehys vaatimusmäärittelyn ja hyväksymistestitapausten automatisoimiseksi

Kappaleessa kuvataan Fit:n pohjalle jatkokehitettyä ohjelmistokehystä, josta jatkossa käytetään mainintaa kehys. Kehys keskittyy muuttamaan Fit:n rakennetta aikaisempaa ylläpidettävämmäksi tarjoamalla mahdollisuuden sisällyttää taulukkoon uusia alitaulukoita, jolloin tietoa voidaan ylläpitää yhdessä taulussa, vaikka taulukon tietoa käytetään useammassa yhteydessä. Kehys käyttää syötteiden vastaanottamiseen ja tulostamiseen xml-tiedostoja, jotka tukevat paremmin moniulotteisuutta. Xml-tiedostojen rakenteelliseen käsittelyyn löytyy valmiita työkaluja, jolloin ei tarvitse ylläpitää erillistä html-jäsentelijää.

Kehyksen käyttöliittymällä pyritään antamaan käytettävyydelle parempi tuki. Fit:n tueksi on jo aikaisemmin rakennettu Fitnessse, joka käyttää wiki -sivuja verkon ylitse jaetulle kohderyhmälle. Verkkosovellukset asettavat toistaiseksi tiettyjä rajoituksia käyttöliittymän toiminnalle. Kehyksessä ei siksi eritellä käyttöliittymätoimintoja, vaan verkkosovellukset voivat toteuttaa toiminnoista osajoukon, joka sillä hetkellä on kyseisessä teknologiassa käytössä.

Lisäksi kehukseen on rakennettu tuki IDE-ympäristöille, jotta kehittäjät pystyvät seuraamaan testien suorittamista sekä soveltamaan kehystä ilman perusteellista perehtymistä kehysten toteutukseen. Kehittäjille on rakennettu viestirajapinta perinnejärjestelmiin, joista voidaan lukea vastaavia xml-tiedostoja kuin käyttöliittymällä on syötetty.

6.1 Ohjelmistokehykset

Ohjelmistokehyksellä (framework) tarkoitetaan toisiinsa liitettyjen luokkien kokoelmaa, joka muodostaa sovelluksen rungon. Runkoon on jätetty aukkoja tulevaa erikoistamista varten [KoM05]. Ohjelmistokehyksistä on tullut suosittu tekniikka oliomaailmassa. Eriyisesti tuoterunkoarkkitehtuurissa on sovellettu kehyksiä. Tuoterungossa sovelluksen

perusosat on rakennettu yhdelle alustalle, mutta siitä erikoistetaan yksittäisiä sovelluksia rajapintojen kautta. Kehyksiä on käytetty runsaasti kehitysympäristöissä niin kutsuttuina plug-in sovelluksina, joilla tuodaan kehitysympäristöön tai rakennettavaan sovellukseen lisää ominaisuuksia. Fit on esimerkki plug-in kehiksestä.

Ohjelmistokehys on luokka-, komponentti- ja/tai rajapintakokoelma, joka toteuttaa jonkin ohjelmistojoukon yhteisen arkkitehtuurin ja perustoiminnallisuuden. Ohjelmistokehiksen runko koostuu toisiinsa tiukasti sidoksissa olevista luokista, mutta kehys ei ole täydellinen ilman erikoistavaa osaa. Erikoistaville osille kehys tarjoaa erikoistamisrajapinnan, joka antaa arkkitehtuurille sovelluskohtaisen toteutuksen ja toiminnallisuuden. Erikoistamisrajapinta on mutkikas kokonaisuus, joka edellyttää useiden luokkien toteuttamista ja yhdistämistä kehikseen. Erikoistamiskohdalla (hotspot) tarkoitetaan kohtaa, joka myöhemmin täytetään sovelluskohtaisella osalla [Pre95].

Erikoistaminen toteutetaan käytännössä perinnällä, koostamisella, rajapinnoilla sekä generisillä rakenteilla [KoM05]. Tavallisesti kehystä erikoistetaan useammilla näistä menetelmistä.

6.1.1 Erikoistamismenetelmät

Erikoistamismenetelmistä tavallisimmin käytetty on periytyminen. Tässä kehys tarjoaa kantaluokan, jonka tuotekohtainen luokka perii ja samalla antaa osalle sen operaatioista uuden toteutuksen [KoM05]. Toinen tavallisesti käytetty menetelmä erikoistamiseen on rajapintojen käyttö, jossa kehiksen tuotekohtainen toteutus otetaan käyttöön tähän tarkoitukseen kehitetyn luokan tai komponentin kautta. Rajapinnoissa kehys näkee tuotekohtaisen luokan ilmentymän tai komponentin rajapinnan kautta ja käyttäessään jotakin rajapinnan palvelua käyttää alla olevaa ohjelmakoodia. Molemmat näistä menetelmistä ovat varsin joustavia, sillä niillä kehikseen voidaan liittää täysin uutta ohjelmakoodia halutun sovelluksen aikaansaamiseksi.

Javan, C# ja C++:n tarjoamat generiset luokat mahdollistavat erityyppisten erikoistamismenetelmien toteuttamisen. Geneerisyys toteutetaan tavallisesti antamalla kehikselle parametreiksi joitakin sen omia luokkia, joita erikoistetaan geneerisyydellä erikoistamisrajapinnan välityksellä [KoM05]. Lisäksi erikoistamisessa voidaan käyttää kielen tarjoamia työkaluja, kuten luokkien tai parametrien nimiä, joita kysymällä kehikselä sovellukselle tai päinvastoin voidaan ohjata sovelluksen logiikkaa.

6.1.2 Kehyslajit

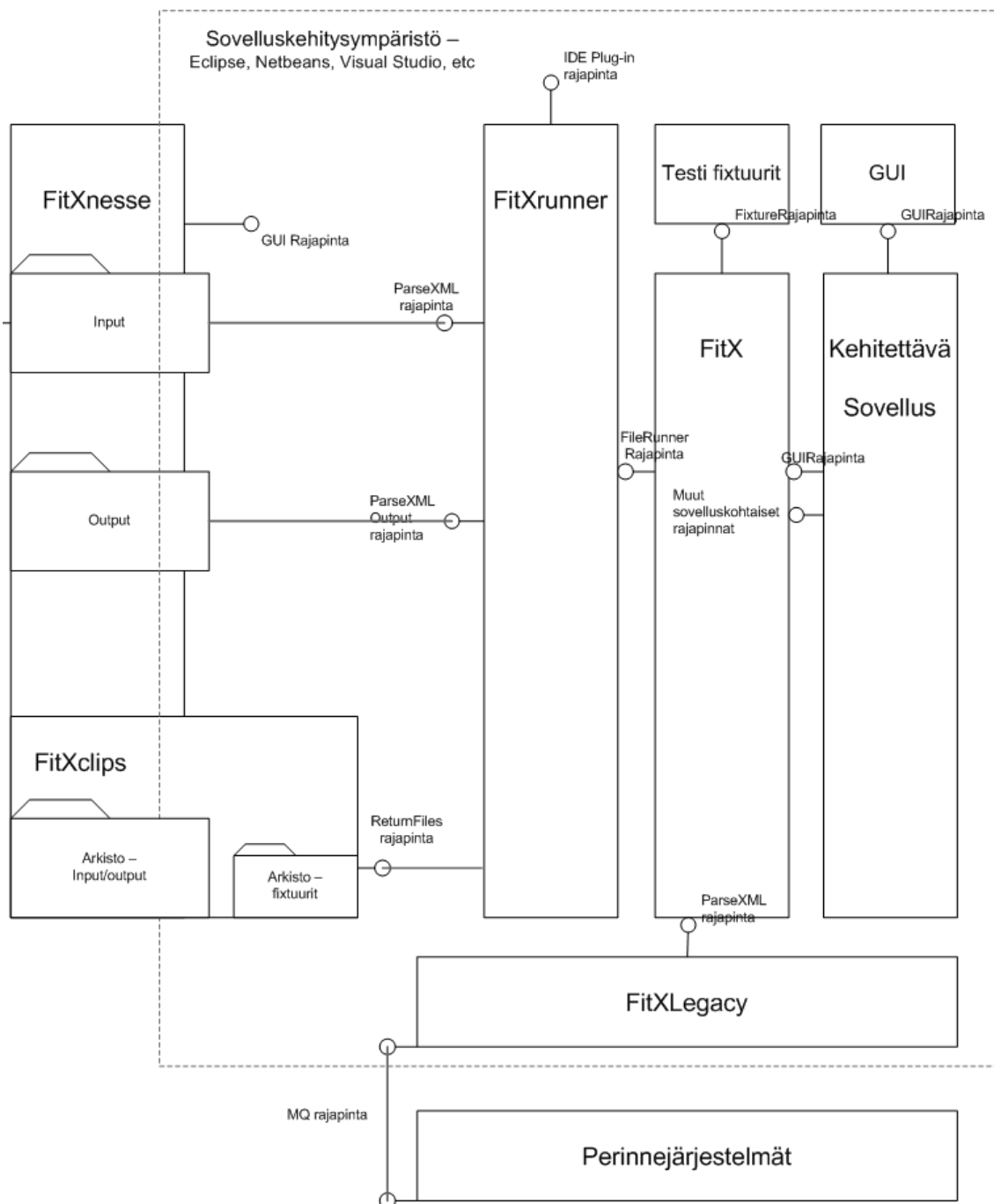
Tavallisesti käytettyjä kehyslajeja ovat abstraktit kehykset, muunneltavat ja koottavat kehykset sekä plug-in kehykset [KoM05]. Näistä abstraktit kehykset eivät sisällä lainkaan ohjelmakoodia vaan ainoastaan rajapintoja. Abstraktit kehykset eivät rajoita eivätkä tue ohjelmoijaa sen suhteen, kuinka alla olevaa kehystä tulee käyttää. Abstrakteja kehyksiä käytetään usein kerrosrakenteessa, jossa kehyksellä on oma kerroksensa. Kerroksen kautta voidaan soveltaa eri kehyksiä abstraktin kehyksen luomien standardien mukaisesti.

Muunneltavissa kehyksissä (white-box framework) sekä koottavissa kehyksissä (black-box framework) suoritusajan kontrolli on pääsääntöisesti kehyksellä. Muunneltavissa kehyksissä erikoistamisrajapinnat tarjoavat kantaluokkia, joita sovelluskohtaiset aliluokat käyttävät hyväkseen. Näillä aliluokilla ohjelmoija voi antaa kehykselle parametreja. Koottavissa kehyksissä erikoistamista ei tehdä luokilla, vaan ainoastaan muuttamalla kehyksen erikoistamislukien parametreja, joka rajoittaa kehyksen joustavuutta.

Plug-in kehystä käytetään erikoistamalla rajapintoja. Kehys erikoistetaan toteuttamalla kehyksen rajapintoja sovelluskohtaisilla laajennusyksiköillä (plug-in) ja rekisteröimällä toteutukset kehykselle [KoM05]. Laajennusyksiköt voivat koostua yhdestä tai useammasta luokasta, jotka erikoistavat kehystä. Tyypillinen esimerkki plug-in kehyksestä on Eclipse, joka on Javalla toteutettu ohjelmistokehitysympäristö.

6.2 Testauskehyksen arkkitehtuuri

Testauskehyksen arkkitehtuuri on rakennettu suurelta osin Fit:n ja sen ympärille rakennettujen työkalujen Fitnessen, Fitrunnerin ja Fitclipsin päälle. Alla olevassa kuvassa näkyy korkean tason arkkitehtuuri, josta FitX, FutXrunner, FitXnesse sekä FitXclips vastaavat Fit:n toteutuksia melko suoraan. Arkkitehtuuriin on lisätty liittymä perinnejärjestelmiin. Perinnejärjestelmien kommunikaatiosta vastaa FitXLegacy, joka tarjoaa viestirajapinnan perinnejärjestelmille, sekä kääntää viestit FitX:n ymmärtämään xml-muotoon.



Kuva 16: FitX:n arkkitehtuuri

Fit:ssä on muuttunut sen nykyinen parse -luokka, jota käytetään html -dokumenttien jäsentelyyn. Tämä luokka on muuttunut parseXML -luokaksi, joka käyttää ohjelmointikielten valmiita xml-jäsentelijöitä. FitX pystyy käsittelemään xml-dokumentin taulukoita sisäisillä viittauksilla niin, että käyttötapaustaulun solussa tai useissa soluissa voidaan viitata toisen taulun sisältöön. Fit:n testikerros-rakenne säilyy ennallaan ja perustuu kolmeen olemassa olevaan testikerrokseen sekä sovelluskohtaisiin testikerroksiin.

FitXrunner tarjoaa rajapinnat kehitysympäristöille, joka helpottaa testikerroksien sekä input ja output -tiedostojen liittämistä kehitettävään sovellukseen. FitXrunner ylläpitää konfigurointitiedostoa, jonne tallennetaan tiedostojen sijainti. Konfiguraatitiedoston avulla kaikki sovelluksen testaamiseen liittyvät tiedostot voidaan suorittaa kerralla kokonaiskuvan saamiseksi.

FitXclips ylläpitää arkistoa suoritetuista testeistä, sekä niiden tuloksista. FitXclips tallentaa jokaisen testin alkaessa input sekä testikerros-tiedostot sekä testin päättyessä output -tiedoston arkistoon. FitXclips pitää yllä versiohistoriaa, jolloin tilanne voidaan palauttaa aikaisempaan versioon.

FitXnessä ei käsitellä tässä työssä tarkemmin, mutta korkean tason arkkitehtuuriin sille on varattu paikka, koska se on käytettävyyden kannalta oleellinen osa arkkitehtuuria. FitXness tuottaa ja lukee xml-dokumentteja, mutta itse käyttöliittymä voidaan toteuttaa html -dokumentteilla. Xml-dokumentit sisältävät enemmän tietoa dokumentin rakenteesta ja tukevat paremmin automaattista käsittelyä. Käyttöliittymän täytyy tukea taulukkorakennetta ja linkitysten muodostamista, jonka tuloksena muodostetaan FitXrunnerille:lle xml-dokumentit. FitXness:n täytyy pystyä tulostamaan xml-tiedostoista palautteet html -tiedostoiksi, joihin tarvitaan oma luokkansa FitXness:n sisälle. FitXness:een on jätetty rajapinta kehitysympäristölle, josta se voi käyttää hyväkseen tarjolla olevia käyttöliittymäelementtejä. Esimerkiksi .net-ympäristössä on lukuisa määrä käyttöliittymäelementtejä tarjolla työasemasovelluksiin, joista vain pieni osa on käytettävissä .asp-verkkosovelluksissa. FitXness on rakennettu niin, että se tukee laajinta mahdollista käyttöliittymäelementtivalikoimaa mutta, josta voidaan tarpeen mukaan käyttää osajoukkoa riippuen käyttöliittymäraajapinnan tarjonnasta ja tarpeesta.

Tähän asti mainittuihin osiin on oltava pääsy kehitysympäristöstä, jolloin ohjelmoijalla on mahdollisimman vähän vaivaa FitX:n päivittäisestä käytöstä. Ohjelmoijalla ja asiakkaalla tulee olla jaettu pääsy input/output -tiedostoihin, jotta heillä on yhteinen käsitys testien etenemisestä.

6.2.1 Vaatimukset

Ohjelmistokehityksen tärkeimpänä vaatimuksena on olla helposti käytettävä sekä asiakkaalle että ohjelmoijille. Tämä vaatimus saavutetaan, kun käyttöliittymä tarjoaa riittävän tuen vaatimusten kuvaamiseen asiakkaille sekä sanallisessa muodossa että täsmällisillä

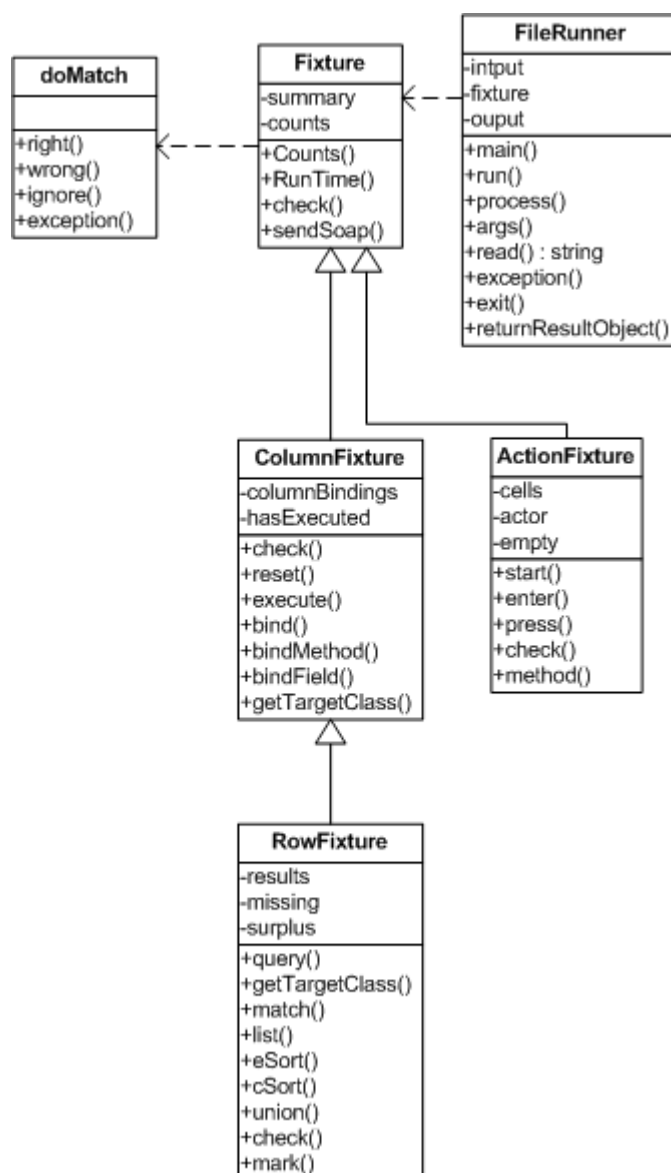
taulukkoesimerkeillä. Ohjelmoijille käytettävyys tarkoittaa riittävää kehitysympäristötu-
kea, jota FitXfilerunner kehityksessä tarjoaa. Kehyksen täytyy tukea tavallisia
kehitysympäristöjä, kuten Eclipseä, Netbeansia sekä Visual Studiota.

Käytettävyteen ja ylläpitoon liittyy FitX:n rakennemuutos useiden taulukoiden linki-
tyksiä tukevaksi. Asiakkaiden tulee voida lisätä käyttöliittymältä taulukoita eri
käyttötapauksille, sekä kytkeä taulukoista tietoa toisiin taulukoihin. FitX tukee fixture
yliluokassa geneerisiä rakenteita, jolloin ohjelmoija voi määrittellä esimerkiksi RowFix-
turessa tyypitettyjä joukkoja ja käyttää niitä täsmäytyksessä. Uusissa sovelluksissa
joukot palautetaan usein geneerisinä ja mikäli FitX ei pysty näitä käsittelemään, joudu-
taan joukko tyypittämään ennen täsmäytystä sekä päivittämään tehtäessä muutoksia
palautteisiin.

FitXLegacy tarjoaa FitX:lle soap -rajapinnan, josta se voi tehdä vastaavia metodikutsuja
perinnejärjestelmälle, kuin se tekee samalla alustalla kehitetylle sovellukselle. FitX erit-
telee perinnejärjestelmät omassa arkkitehtuurissaan ja lähettää soap -metodikutsuja
tähän tarkoitukseen varatulle komponentille. Perinnejärjestelmältä tulevat palautteet lue-
taan viestijonoon, josta ne käsitellään eteenpäin FitX:n käsiteltäväksi. FitX odottaa
palautetta vastaavassa muodossa, jota tavalliset metodit palauttavat.

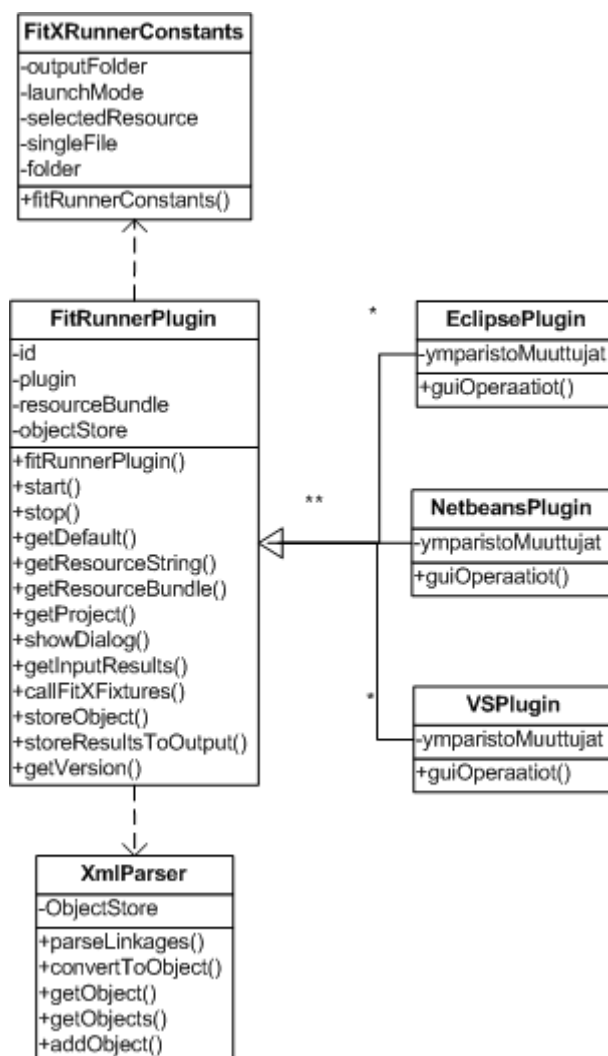
6.2.2 Staattinen kuvaus

Staattisessa kuvauksessa esitellään tärkeimpien moduulien rakenne oleellisten muuttu-
jien ja metodien osalta. Lisäksi kuvauksessa esitellään perintähierarkia ja
riippuvuussuhteet luokkien välillä.



Kuva 17: FitX:n staattinen kuvaus UML:llä

FitX:n staattinen kuvaus mukaillee pitkälti Fit:n rakennetta. Siinä Fixture-luokka on taulukoiden ylliluokka, joka perii kolme suorittavaa luokkaa toiminnolle, laskennalle ja joukoille. Näille kolmelle joukolle ohjelmoija voi antaa ohjelmaa kutsuvia testikerroksia periyttämällä jonkin Action-, Column- tai RowFixture -luokista. Ohjelmistokehysten eriyttämisessä periytyminen on tavallisimmin käytetty menetelmä. Html -tiedoston jäsentelemisessä on Fit:ssä parse -luokka, joka jäsentelee html -tiedostosta taulukot erillisiin olioihin. FitX:ssä odotetaan palautteena luokka -objekteja sekä array -tyyppisiä listoja RowFixturen joukkojen täsmäyttämiseen. Testattavalta sovellukselta tulevat generiset luokat normalisoidaan FitX:ssä tavallisiksi luokiksi ja array -kokoelmiksi, jolloin FitX:n testikerroksilla on täsmäytettävänä samantyyppisiä luokkia ja kokoelmia.

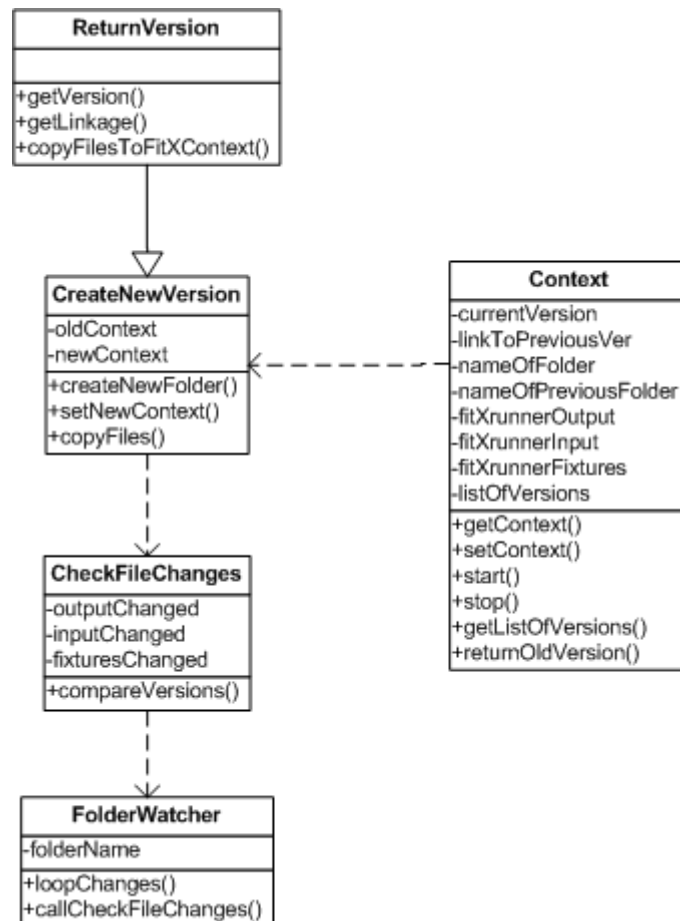


Kuva 18: FitXrunner:n staattinen kuvaus UML:llä

FitXrunner toimii sihteerinä FitX:lle. FitXrunner tallentaa tiedon kontekstista, jossa FitX kulloinkin toimii. Sillä on tieto input- ja output -tiedostoista, sekä kehitysympäristöstä. FitXrunner aloittaa testitapausten suorittamisen, jolloin se käy läpi joko koko input-kansion tai erillisen tiedoston, riippuen konfiguraation asetuksista. FitXrunner käyttää apunaan xmlParser -metodia, joka purkaa kansion tauluista linkitykset muihin tauluihin ja koostaa näistä erilliset xml-objektit ilman linkityksiä. Xml-objektit käännetään xmlConverter -metodilla luokka -objekteiksi, jolloin niitä on suoraviivaisempi käsitellä myöhemmin ohjelman kulussa.

FitXrunner tallentaa input -kansioista käännettyt objektit array -muuttujaan, josta lähetetään objektit yksittäin FitX:n täsmäytettäväksi varsinaisen ohjelman kanssa. Tällä tavalla saadaan arkkitehtuurista modulaarisempi, kun FitX keskittyy vain tulosten täsmäyttämiseen ja FitXrunner palvelee sitä hoitamalla rutiininomaiset konversiot sekä

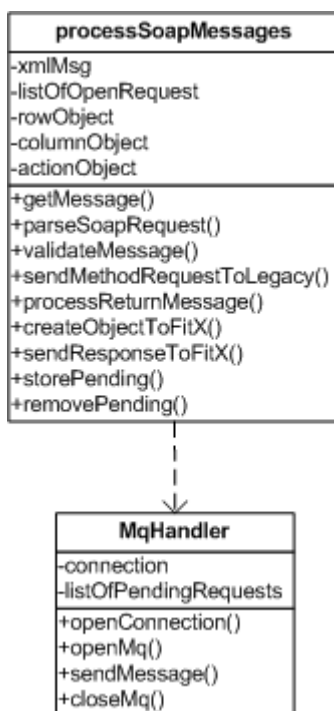
liitynnät eri kehitysympäristöihin. FitXrunner sisältää plug-in osat eri kehitysympäristöihin, Plug-in osat koostuvat kehitysympäristön vaatimista konfiguraatio tiedostoista, sekä käyttöliittymän tekemiseen tarvittavista operaatioista. Käytännössä plug-in luokilla on käyttöliittymän lisäksi lukuisa joukko muita operaatioita, mutta niitä ei käsitellä tässä työssä.



Kuva 19: FitXclips

FitXclips hoitaa kansiorakennetta input, output ja testikerros -tiedostoille. FitXrunner suorituksen jälkeen FitXclips tarkistaa, onko input, output tai testikerros -tiedostoissa muutoksia verrattuna viimeisimpään versioon. Tiedostoille luodaan uusi kansio ja linkitetään se aikaisempaan versioon linkList -tyypillä, mikäli checkFileChanges palauttaa true -arvon. FitXrunner tarjoaa kehitysympäristöön toteutetun käyttöliittymän, jolla voidaan valita aikaisempi versio suoritettavaksi viimeisimmän version sijasta. FitXrunnerin käyttöliittymältä voidaan vaihtaa konteksti aikaisempaan versioon, jolloin halutun version tiedostot kopioidaan FitXrunner-oletuskansioihin. Tässä yhteydessä palautetaan myös linkitys versioon, joka edelsi palautettua versiota.

FitXclipsillä voi selata vanhoja versioita myös palautuksen jälkeen, koska versiot tallennetaan fyysisesti uusiin kansioihin. Rakenteesta tulee puurakenne, josta voidaan valita versio johon palataan.

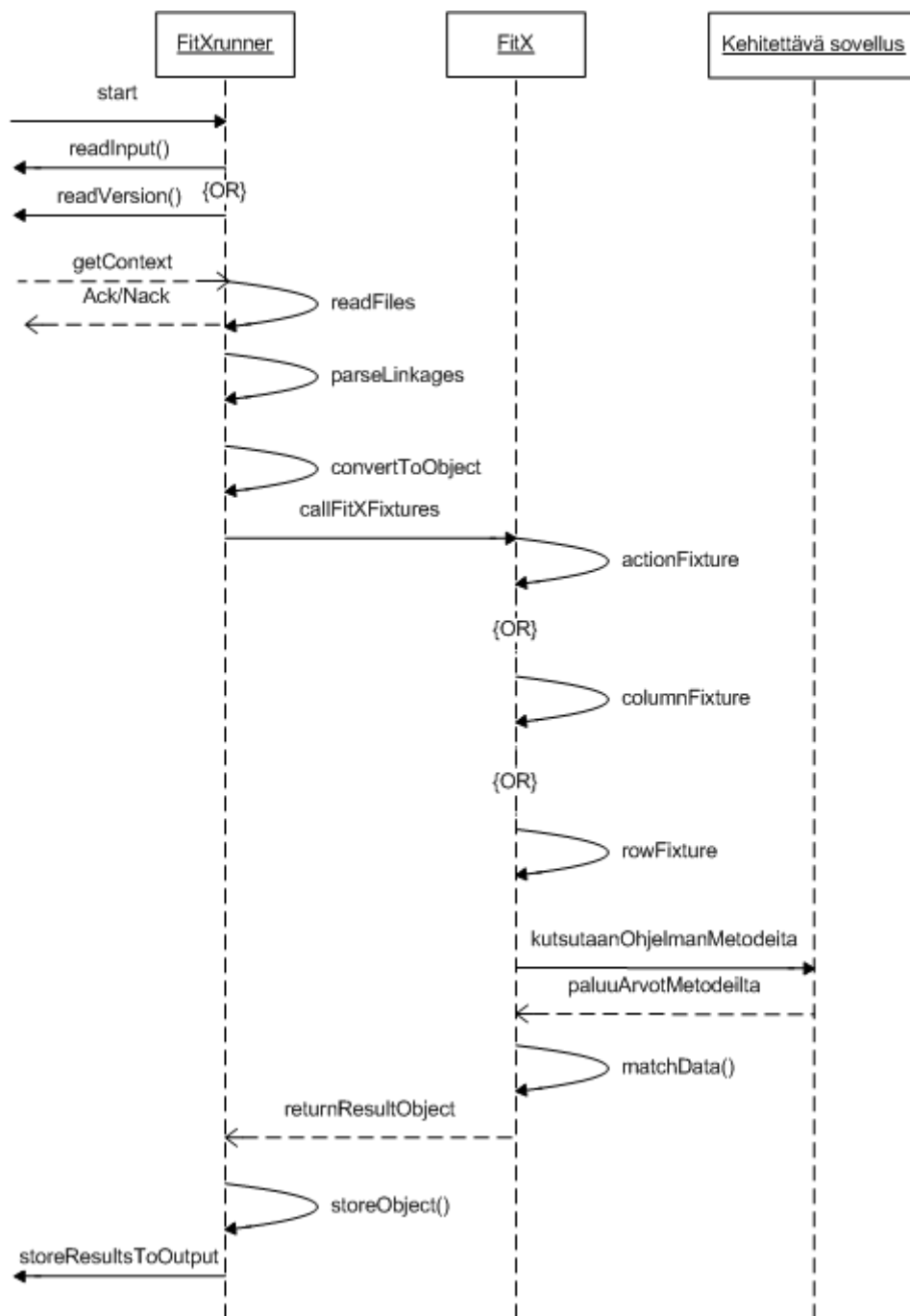


Kuva 20: FitXlegacy

FitXlegacy kommunikoi suoraan FitX:n kanssa, koska FitXrunnerilla ei ole tietoa alla olevien testikerrosten metodikutsuista. Testikerroksessa määritellään kutsut perinnejärjestelmiin lähettämällä FitXlegacy-komponentille soap -dokumentti, joka sisältää metodikutsun. FitXlegacy purkaa ja tarkistaa dokumentin sekä lähettää sen eteenpäin viestijonolle, josta se ohjataan alla olevalle perinnejärjestelmälle. Viestijono jää odottamaan vastausta viestiin ja sen saatuaan lähettää soap -sanoman eteenpäin luokalle, joka purkaa sanoman ja muodostaa siitä FitX:lle ymmärrettävän objektin. Tämän jälkeen FitX tekee täsmäytyksen, kuten tavallisilta metodeilta saaduille palautteille.

6.2.3 Toiminnallisuus

Toiminnallisuudessa esitellään kehyksen oleellisten osien toiminta sekvenssikaavioiden tarkkuudella. Oleellisiksi osiksi on valittu täsmäytusrutiini asiakastestien ja varsinaisen ohjelman välillä sekä perinneohjelmien kanssa kommunikointi. Plugin-ominaisuuksia, arkistoinnin yksityiskohtia tai käyttöliittymää ei esitellä yksityiskohtaisesti tässä työssä.

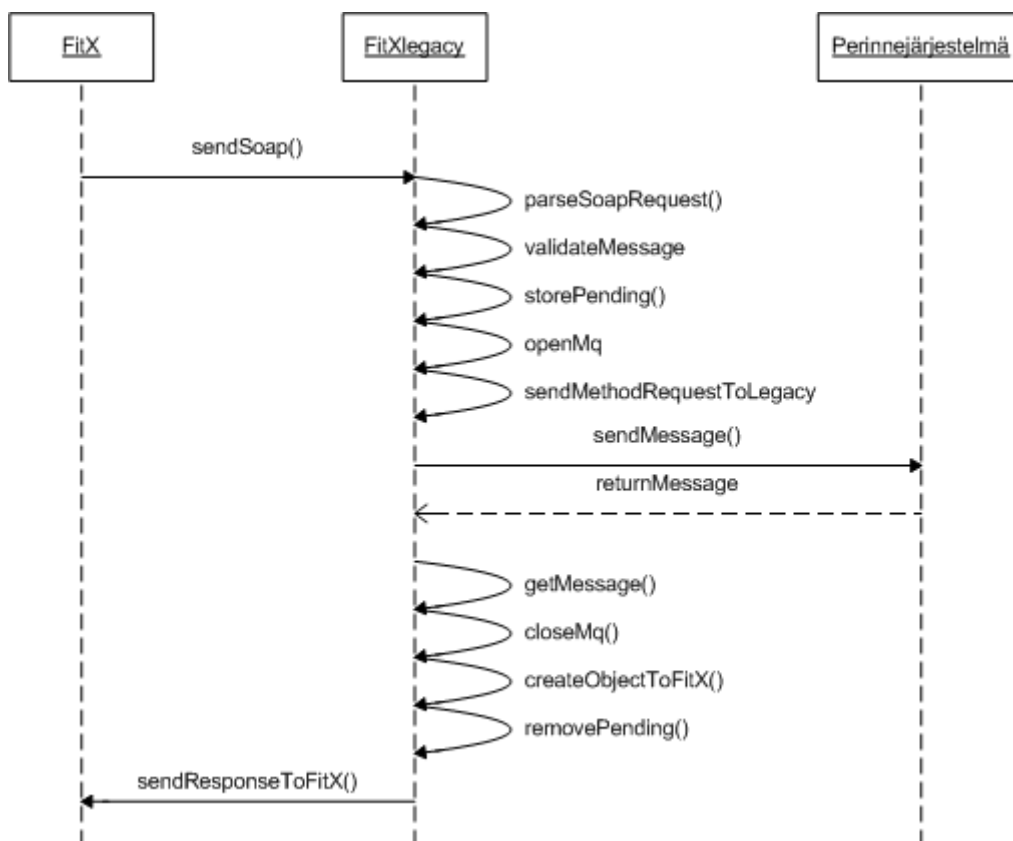


Kuva 21: Taulujen ja ohjelmien täsmäytys -sekvenssikaavio

Kuvassa 21 esitellään FitXrunnerin, FitX:n sekä testattavan sovelluksen kommunikointi suorituksen aikana. Suoritus alkaa start-komennolla kehitysympäristöstä, jota ennen FitXrunnerille on määritelty tiedostojen sijainti. Mikäli halutaan suorittaa FitXclipsin aikaisempia versioita, voidaan suoritus aloittaa oletuskontekstin sijasta historiaan tallennetusta kontekstista. Tiedostojen oikeellisuus tarkistetaan ja lähetetään FitXclips:lle palaute prosessin jatkosta. Tämän jälkeen input-kansion tiedostoista puretaan linkitykset taulukoiden välillä ja tuotetaan xml-tiedostot, jotka pitävät sisällään alitaulu-

jen tiedot. ParseLinkages -metodi tuottaa selkokieliset ja validit xml-dokumentit, joiden perusteella convertToObject -metodi tuottaa tauluja vastaavat luokat. Valmiit luokat välitetään FitX -komponentille täsmäytettäväksi.

FitX suorittaa komponentin parametrien mukaisen testikerroksen, joka kutsuu testattavaa sovellusta. Testikerrokset voivat olla action-, column- tai rowFixtuureita. Testattava ohjelma lähettää palautteena arvot, jotka täsmäytetään doMatch-metodilla FitX:ssä. DoMatch luokittelee täsmäytyksen tulokset oikeisiin, väriin, poikkeuksiin sekä sivuutettuihin tapauksiin. Tuloksista luodaan xml-palautetiedosto, joka tallennetaan alkuperäisen kontekstin mukaiseen kansioon.



Kuva 22: FitXlegacy -sekvenssikaavio

FitXlegacy saa Soap -sanoman FitX:ltä, jonka sisällön se validoi, lähettää eteenpäin viestijonon kautta perinnejärjestelmälle ja tallentaa odottamaan käsittelyä. Perinnejärjestelmä on rakennettu vastaanottamaan soap -metodikutsuja, se prosessoi viestin sisältämän metodin ja palauttaa soap -palautteen viestijonolle. FitXlegacy hakee viestin viestijonosta ja sulkee jonon, jonka jälkeen se purkaa viestin. Puretusta viestistä luodaan objekti -luokka, jota FitX käyttää täsmäytyksessä kuten tavallisilta metodeilta saatuja palautteita.

6.2.4 Toteutus

Useiden taulukoiden rakenne toteutetaan saman dokumentin sisällä esitettyinä taulukoina, sekä nimeämällä taulukon alkuosat joko `staticData` tai `staticStructure`, joka kertoo taulukoiden sisältävän staattista tietoa muiden taulukoiden käytettäväksi. Käyttöliittymät voivat tuottaa joko html -dokumentteja tai xml-tiedostoja.

Kuvan 23 esimerkissä kuvataan testitapausta, jossa asiakkaan lisäämisessä suoritetaan asiakkaalle luokittelu, joka on esitetty erillisessä taulussa ylläpidon helpottamiseksi sekä saman tiedon toistamisen välttämiseksi.

Testitapaus 1.

Lisää asiakas

Asiakkaita on kolmea eri tyyppiä. Tyyppejä voi tulla lisää myöhemmin.

<code>staticData.asiakasTyyppi</code>
tukkuasiakas
jälleenmyyjä
yksityisasiakas

Lisäksi asiakkaaseen liittyy useammassa yhteydessä yhtenäiseksi sovittu rakenne osoitteesta, jonka tulee sisältää samat tiedot läpi koko prosessin.

<code>staticStructure.osoite</code>	oletusArvot
katuosoite	testiosoite
kaupunki	espoo
postinumero	00260
maa	suomi

Asiakas lisätään seuraavilla tiedoilla.

<code>fit.ActionFixture</code>		
<code>start</code>	<code>fitTestPackage.AddData</code>	
<code>enter</code>	<code>nimi</code>	Petteri Kuono
<code>enter</code>	<code>staticStructure.osoite</code>	
<code>enter</code>	<code>yryitys</code>	testiyryitys
<code>enter</code>	<code>staticData.customerType</code>	
<code>check</code>	<code>addData</code>	true
<code>enter</code>	<code>nimi</code>	Milla Magia
<code>enter</code>	<code>staticStructure.osoite</code>	
<code>enter</code>	<code>yryitys</code>	testiyryitys
<code>enter</code>	<code>customerType</code>	tukkuasiakas
<code>check</code>	<code>addData</code>	false

Kuva 23: Esimerkki testitapauksesta

Edellisessä taulukossa nähdään, että viimeinen `addData`-komento odottaa saavansa `false`-palautteen. Tämä esimerkki kuvaa tilannetta, jossa staattisten taulukoiden riveistä odotetaan toisistaan poikkeavia tuloksia. Tällaiset poikkeukset staattisissa taulukoissa tulee määritellä erikseen. Edellisessä esimerkissä `tukkuasiakas` -tyyppi tuottaa palaut-

teen false, koska sitä ei hyväksytä asiakastyypiksi. Tämä määrittellään erillisenä testitapauksena poikkeuksena muihin asiakastyyppeihin.

Html -taulukot puretaan xml-tiedostoiksi, joissa linkitykset esitetään yksittäisinä testitapauksina. Kuvan 24 esimerkissä näytetään testitapaus, joka on johdettu kuvan 23 tapauksesta. Tässä osoitteen rakenne ja oletustiedot on haettu osoite-taulusta sekä asiakastyypin ensimmäinen rivi asiakasTyyppe-tilusta.

```

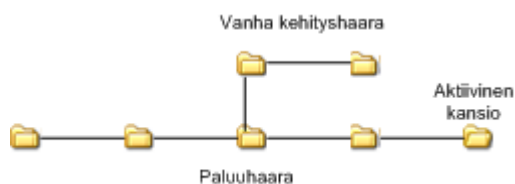
<ActionFixture>
  <Start>fitTestPackage.AddData</Start>
  <Enter>
    <Nimi>Petteri Kuono</Nimi>
  </Enter>
  <Enter>
    <katuosoite>Testiosoite</katuosoite>
  </Enter>
  <Enter>
    <kaupunki>Espoo</kaupunki>
  </Enter>
  <Enter>
    <Postinumero>00260</Postinumero>
  </Enter>
  <Enter>
    <maa>Suomi</maa>
  </Enter>
  <Enter>
    <Yritys>testiyritys</yritys>
  </Enter>
  <Enter>
    <AsiakasTyyppe>TukkuAsiakas</AsiakasTyyppe>
  </Enter>
  <check>addData</check>
</ActionFixture>

```

Kuva 24: Esimerkki XML-dokumentista

FitXrunner luo xml-dokumentin pohjalta olion, jonka se välittää FitX:lle. Tämän jälkeen FitX kutsuu addData-testikerrosta, josta kutsutaan varsinaista ohjelmaa. Ohjelmasta saatavat palautteet täsmäytetään xml-dokumentin sisältöön. Palautteet rikastetaan xml-dokumenttiin, joka lähetetään käyttöliittymälle joko html- tai xml-muodossa. Palautteissa ei enää esitetä usean taulukon rakennetta, vaan näytetään yksittäisten testirivien tulokset alitaulujen tiedoilla lisätynä.

FitXclips:n versionhallinta toteutetaan linkitettyinä puurakenteena. Tässä jokaisen suorituskerran jälkeen tarkistetaan onko tiedostojen sisältö muuttunut ja tallennetaan uusi versio uuteen kansioon, mikäli muutoksia havaitaan.



Kuva 25: Versionhallinnan puurakenne

Mikäli kehittäjä haluaa palauttaa vanhan version ja jatkaa kehitystä uuteen haaraan, linkitetään aikaisempaan tiedostoon uusi haara, jolloin kehittäjälle jää käyttöön aikaisempi haara, jonka kehitys jäi kesken.

7 Ohjelmistokehityksen analysointi

Testauskehysiltä edellytetään helppokäyttöisyyttä testien laatimisessa ja suorittamisessa. Parhaimmillaan testitapauksista on tukea määrittelyvaiheessa, toteutuksessa ja testien suorituksessa. Tällöin käyttäjien ei tarvitse käyttää aikaa suoritukseen, vaan voivat keskittyä testien suunnitteluun ja tulosten analysointiin. Ihannetilanteessa testauskehys tarjoaa tuen prosessin vaiheesta seuraavaan siirryttäessä, jolloin ohjelmiston ominaisuudet ja testitapaukset tarkentuvat aikaisempiin määrittelyihin. Hyvä testauskehys on geneerinen, jotta sen metodeita voidaan soveltaa erityyppisiin ohjelmiin ja toteuttaa eri ohjelmointikielillä ja -ympäristöissä.

Päivittäisessä käytössä kehityksen testitapausten tulee olla helposti päivitettäviä ja suoritettavia. Muussa tapauksessa testitapausten kirjoittamisesta tulee ylimääräinen rasite, joka heikentää testitapausten laatua [And07]. Testitapausten tulee olla helposti luettavissa ohjelman kaikille sidosryhmille, jotta ohjelmiston käyttäjät osaavat arvioida vaatimusten ja testitapausten laatua. Testauskehityksen testitapausten tulee olla helposti ylläpidettävissä ja paikallistettavissa, jotta ohjelmaa kehitettäessä kyseiset testitapaukset ovat helposti löydettävissä.

Kehityksen on oltava hyvin dokumentoitu, helposti asennettavissa, sen käyttöön on oltava riittävästi tukea sekä käytöstä on oltava riittävästi esimerkkejä, jotta kehitys voi saada yleisen hyväksynnän [NaN0]. Ohjelmistokehityksiryhmälle on tärkeää, että testauskehys on saanut aikaisemman käytön myötä vakiintuneen statuksen ja siitä löytyy aikaisempia kokemuksia, esimerkkejä ja dokumentaatiota. Pienissä ohjelmistoprojekteissa voidaan hyväksyä vielä kehitysvaiheessa oleva työkalu, mutta tällöin on riskinä, että testiohjel-

maan ei ole enää tukea saatavilla ohjelmiston kasvaessa. Tässä tilanteessa testitapaukset voidaan joutua rakentamaan uudelle alustalle. Testikehystä valittaessa jatkuvuus on tärkeä tekijä.

7.1 Kehyksen arviointi

Ohjelmistotehtävät voidaan luokitella ongelmakehyksiin (problem frames), riippuen siitä minkälaista ongelmaa ohjelma tai sen yksittäinen osa ratkaisee [Jac95] [LBJ04]. Tyypilliset ohjelmistolla ratkaistavat ongelmat ovat tekstinkäsittely-, ohjausjärjestelmä-, tietojärjestelmä-, muunnos- ja yhteystehtävät.

7.1.1 Ongelmakehykset

FitX:n suoriutumista arvioidaan tekstinkäsittelylle tyypillisten ongelmien kautta, kuten kirjoittaminen, tallentaminen, tulostus, kielentarkistus, ja muotoilu. FitX:llä on käytössä kolme menetelmää, joita voidaan testitapauksissa soveltaa; toiminta-, rivi- ja kolumnitestit. Tekstinkäsittelyn kirjoittamisessa sovellus ottaa käyttöjärjestelmältä vastaan käyttäjän antamia syötteitä näppäimistöltä. Syötetyt merkit ovat erilaisia riippuen maasta ja käytettävästä merkistöstä. Testi- ja määrittelytapauksissa FitX:stä voidaan käyttää kolumnitesterrosta. Tässä merkkiä painettaessa, näppäimistöltä annetaan arvo. Testikerroksessa kutsutaan ohjelman metodia, otetaan vastaan ohjelman palauttama arvo ja verrataan sitä taulukossa annettuun arvoon. Tallentamisessa voidaan soveltaa toimintatesterrosta, joka palauttaa käyttäjälle tosi-arvon onnistuneesta tallennuksesta. Tallentaminen on tyypillinen esimerkki vaatimusmäärittelylle kohdistuvista testaustarpeista verrattuna ohjelmoijan testaustarpeisiin. Käyttäjälle riittää tieto, että tallennus on onnistunut virheettää. Ohjelmoijan tarpeena on testata, että kaikki tallennettu tieto on todella tallennettu sellaisena kuin käyttäjä on tiedon ohjelmaan syöttänyt. Yksityiskohtainen testaus on järkevää tehdä yksikkötestausvaiheessa. Testaustyö on perusteltua jakaa käyttäjätesteihin ja yksikkötesteihin. Funktionaaliset toiminnot, kuten tulostus, kielentarkistus ja muotoilu ovat toteutettavissa toimintatesterroksella. Näistä jokaiselle funktiolle annetaan syöte, jonka jälkeen ohjelma suorittaa prosessin ja antaa siitä palautteena arvon onnistumisesta tai epäonnistumisesta. FitX:n testaus ei ota kantaa käyttöliittymään, joka on tärkeässä asemassa tekstinkäsittelyohjelmissa. Tästä johtuen hyväksymistestit tulee jakaa kahteen osaan. Ensimmäisessä osassa testataan järjestel-

män sisäinen toiminta määrittämiä vastaan. Toisessa osassa käyttäjät testaavat käyttöliittymän toimintoja ja tarkistavat, että toiminta vastaa määrittämiä myös käyttöliittymällä.

Ohjausjärjestelmällä ohjataan jonkin prosessin kulkua. Ohjausprosessi on tyypillinen järjestelmän alijärjestelmä. Ohjausprosessi on joko täysin automaattinen tai se voi edellyttää käyttäjien päätöksiä. FitX:n testitapaukset voivat kohdistua esimerkiksi viestijärjestelmään, jossa alijärjestelmä ottaa ulkoiselta järjestelmältä vastaan viestejä ja ohjaa toiselle järjestelmälle jatkokäsittelyä varten. Prosessiin tyypillisesti kuuluu viestien validointia, jonka jälkeen virheelliset viestit ohjataan käyttäjien arvioitaviksi ja korjattaviksi. Ohjausjärjestelmän perusrakenne koostuu tavallisesti taulukosta, joka kertoo mihin yksittäiset objektit tulee ohjata. Tällöin FitX:stä voidaan käyttää kolumnitestikerrosta, jossa kerrotaan objekti, tässä tapauksessa viesti, sekä sille osoitettu jatko-osoite. Testitapaus voidaan suorittaa antamalla ohjelmalle syötteenä viesti, jonka ohjelma käsittelee ja palauttaa viestille jatkokäsittelyosoitteen. Testi on hyväksyty, jos osoite täsmää käyttäjän antamaan arvoon taulukossa. Testitapaus mutkistuu hieman, kun lisätään käyttäjien osallistuminen prosessiin. Hyväksymistestit tulee jakaa kahteen osaan käyttöliittymän ja toimintalogiikan suhteen, kuten tekstinkäsittelyssä. FitX:lle määritellään toimintatestikerros, jossa käyttäjä muuttaa viestin sisältöä ja lähettää viestin ohjainjärjestelmälle uudelleen käsiteltäväksi. Taustalla oleva logiikka voidaan tarkistaa syötteiden ja uudelleen lähettämisen osalta FitX:llä.

Tavallinen ohjelmointitehtävä on tiedon tallennus levyille luotettavalla tavalla. Tämä voi olla esimerkiksi tietokantasovellus. Sovelluksessa tallennetaan tietoa tietokantaan, jota käyttäjät tai muut järjestelmät hakevat tarvittaessa. Tavallisesti tietojärjestelmään on liitetty toiminnallisuutta, joka raportoi tietoa eteenpäin automaattisesti järjestelmään tallennetun säännösten mukaan. FitX:n rivitestikerros on tietokantarivien testaamiseen soveltuva osa. Rivitestikerrokselle annetaan arvot, joita testattavan järjestelmän metodien tulee antaa palautteena. Tyypillisesti nämä palautetaan luokkien get -metodeilla. Käyttäjä määrittelee testitaulukkoon tiedot, joita kannasta halutaan palauttaa, joko käyttäjän antamalla komennolla tai ohjelman käynnistämällä funktiolla. Toimintatestikerros soveltuu tietokantaan tallennuksen testaamiseen. Käyttäjä määrittelee testitapauksessa, mitä tietoa kantaan halutaan tallentaa. Testiä suorittaessa taulukon tiedot lähetetään ohjelman suoritettavaksi ja ohjelma palauttaa tosi-arvo, mikäli tallennus on onnistunut. Tämä on jälleen esimerkki yksikkötestauksen tarpeellisuudesta. Ohjelmoijan on varmis-

tuttava yksikkötesteissä, että jokainen alkio on todella tallennettu kantaan.

Tietokantaan tallennetaan tavallisesti tietoa, jota järjestelmä kerää tai ottaa vastaan muilta järjestelmiltä. Tämä tieto ei aina ole hyvin määriteltyä, jolloin ohjelma joutuu muuttamaan vastaanotetun tiedon muotoa ennen kantaan tallentamista. Tämä tapahtuu ohjelmaan tallennetun säännösten mukaan, jossa viestistä kerätään haluttu tieto tietokantaan tai muuhun tarkoitukseen myöhemmin käsiteltäväksi. Tiedostomuodot ja niiden muunnoskäsittely voivat poiketa merkittävästi riippuen ongelman luonteesta. Tällainen muunnoskäsittely on vaikeasti testattavissa hyväksymistestauksella, koska suuri osa työstä kohdistuu tiedoston käsittelyyn ohjelman käyttäjälle näkymättömissä osissa. Tämä käsittely voi olla hyvinkin mutkikasta, jonka lopputuloksesta käyttäjälle näkyy vain pieni osa. Tällainen sovellus voi esimerkiksi olla ohjelma, joka muuntaa csv-tiedoston xml-muotoon. Toinen esimerkki on ohjelma, joka kerää internetistä tietoa hakuohjelmalle. Testitapaukset voivat olla yksinkertaisia, joissa syötteenä annetaan tekstitiedosto ja palautteeksi annetaan tosi tai epätosi riippuen käsittelyn onnistumisesta. Käyttäjälle tämä ei kuitenkaan kerro vielä onko esimerkiksi xml-muunnoksessa kaikki tieto siirtynyt tekstitiedostolta eteenpäin. Tässä ongelmassa korostuu yksikkötestauksen merkitys. Testauksen kannalta voi olla järkevää tallentaa muunnoksen väli- ja lopputulokset tietokantaan, jolloin käyttäjä voi suorittaa testin rivitestikerroksella. Tähän sisältyy vastaava ongelma kuin käyttöliittymää testattaessa. Käyttäjä ei voi olla varma, onko tieto todella siirtynyt tekstitiedostosta toiselle, vaikka tietokannan tulokset näyttävätkin hyvältä.

Viimeisenä ongelmakehyksenä esitellään yhteysjärjestelmät. Sovellukset ovat usein linkittyneet toisten järjestelmien kanssa joko viestien tai metodien vaihdolla. Sovellusten toiminnalle aiheutuu ongelmia, jos yhteys toiseen järjestelmään katkeaa tai hidastuu. Tällainen ongelma on hyväksymistestauksen kannalta vaikeasti testattavissa, sillä se liittyy ei-toiminnallisiin vaatimuksiin. Käyttäjä voi suorittaa testin kerran tai useita kertoja peräkkäin, jolloin testi toimii moitteettomasti. Tämä ei kuitenkaan tarkoita, että yhteys on luotettava tuotantoympäristössä. Tämä esimerkki on jälleen osoitus, toiminnallisen-, moduuli- ja yksikkötestauksen tarpeellisuudesta etenkin ei-toiminnallisten ominaisuuksien osalta.

7.1.2 Kehyksen integrointi

Testauskehys voidaan integroida kehitysympäristöihin, olemassa oleviin ohjelmistoympäristöihin ja ohjelmistoprosessin eri osiin. FitX:ssä vaatimusmäärittely ja hyväksymistestaus ovat tiukasti integroitu. Parhaassa tapauksessa vaatimukset ja testitapaukset ovat täysin yhtenevät. Muihin prosessin osiin FitX ei ole integroitavissa. Kehittäjien kannalta on hyödyllistä, jos aikaisempaa määrittelytyötä voidaan hyödyntää sellaisenaan samassa ympäristössä ja rakentaa aikaisemman työn päälle lisää määrittelyksiä. V-malli ei prosessina tue tätä lähestymistapaa, sillä prosessissa määrittelyvaihe on eriytetty suunnittelusta. Eriyttämisen tarkoituksena on määrittellä ongelma mahdollisimman hyvin ja sen ratkaisu erikseen myöhemmässä prosessin vaiheessa. Mikäli hylätään v-mallin muodollisuus ja kehitetään ketterien menetelmien mukaisesti matala prosessimalli, voidaan määrittely ja suunnittelu toteuttaa samalla dokumentaatiolla. Tällöin eri prosessin vaiheet voidaan testata automaattisesti samalla alustalla. Tämä ei ole FitX:n tavoite, vaan mahdollinen laajentumissuunta.

FitX:n on tuettava tavallisesti käytettyjä kehitysympäristöjä, jotta se voi saada riittävän hyväksynnän kehittäjäkunnassa. Työn tuottavuuden kannalta ohjelmoinnin rutiinitehtäviin ei saa kulua liikaa aikaa, jonka takia kehitysympäristöt ovat yleisiä ohjelmoinnissa. Ohjelmiston kehitysympäristöt tarkistavat koodia ja tarjoavat apua metodien nimeämisessä ja löytämisessä. Vastaavasti testien kehittäminen, testattavaan ohjelmaan liittäminen, suorittaminen ja tulosten tulkitseminen on oltava yksinkertaista ja kehitysympäristön tukemaa. Muussa tapauksessa testauksesta kertyy ylimääräinen rasite, jolloin testauksen suorittaminen voi jäädä huolimattomaksi. Integrointirajapintoja ei ole tässä työssä kuvattu yksityiskohtaisesti, mutta toiminnallisuuden kannalta ne on otettu huomioon oleellisena osana.

Perinnehjelmistoympäristöön testattava sovellus on FitX:n tuella integroitavissa sanomajapinnan kautta. Testit harvoin rajoittuvat vain ympäristöön, jota ollaan kehittämässä. Usein kehitettävä järjestelmä hakee syötteitä ja palautteita olemassa olevista perinnejärjestelmistä. Tämän huomioiminen hyväksymistestauksessa on tärkeää. Kehitettävät järjestelmät ovat kytköksissä olemassa oleviin järjestelmiin, jolloin muutokset kehitettävässä järjestelmässä voivat vaikuttaa useamman järjestelmän toimintaan ja niiden väliseen kommunikaatioon. Testaajan on voitava määrittellä tapauksia, joissa muilta järjestelmiltä haetaan tietoa, sekä välitetään eteenpäin muille perinnejärjestelmil-

le. Etenkin tietosisällön muuttuessa on tärkeää, että tarkistetaan muuttuneiden viestien käsittely olemassa olevissa järjestelmissä.

7.1.3 Erilaiset projektityypit

Automatisoitu ohjelmistotestaus lisää merkittävän määrän kehitystyötä ohjelmistoprojektiin, jonka odotetaan maksavan itsensä takaisin myöhemmillä kehityskierroksilla. Hyväksymistestauksen automatisoimien on päätös, joka tulee tehdä arvioitaessa ohjelmiston kokoa sekä elinikää. Pieneen ja suljetussa ympäristössä toimivaan järjestelmään ei välttämättä ole järkevää käyttää aikaa kattavien testitapausten integroimiseen. Hyväksymistestauksen integrointia kannattaa harkita järjestelmälle, jonka elinikä on suunniteltu pitkäksi, joka on kytketty ympäröiviin järjestelmiin ja joilla on paljon käyttäjiä. Tällaiselle järjestelmälle on todennäköisesti tulossa lukuisia testauskierroksia ja paljon regressiotestausta. Tällaisessa tilanteessa integrointi maksaa itsensä takaisin ohjelman elinkaaren aikana.

Ohjelman testausstrategia päätetään, kun tunnetaan projektin laajuus ja luonne. Ohjelmistokehityksen prosessimallilla on vaikutusta siihen, millainen testausstrategia valitaan. Laajoissa ohjelmistoissa v-malli ja perusteellinen toteutuksesta eriytetty määrittely on tavallinen lähestymistapa. FitX tukee tätä lähestymistapaa. Sen sijaan pienissä ohjelmistoissa ja uutta teknologiaa kehittävässä projekteissa on määrittelyt tehtävä lähempänä toteutusta ja toteutuksen lomassa. Tällaisissa projekteissa voidaan käyttää muita testausmenetelmiä esimerkiksi vain yksikkötestausta laajennetuilla testitapauksilla tai kehittää FitX:ää tukemaan v-mallin muita vaiheita.

7.2 Mittarit

FitX kehystä ei ole ohjelmoitu valmiiksi, joten valmiilla kehyksellä tehtyjä testejä ei ole tämän työn puitteissa mahdollista toteuttaa. Mittaamisessa tulee kuitenkin ottaa huomioon kehyksen opiskeluun ja implementointiin käytettävä aika määrällisenä mittarina. Laadullisena mittarina tulee huomioida käytettävyyden, jotta testitapauksista saadaan kattavat.

Määrällisenä mittarina voidaan käyttää implementointiin käytettävä aika verrattuna muihin testauskehyksiin, esimerkiksi unit-kehykseen. Oletettavaa on, että FitX:n implementointi on huomattavasti työläämpää, kuin monen muun testaamiseen käytettä-

vän ohjelman. Perinnejärjestelmien kanssa kommunikointiin on kehitettävä erillinen rajapinta, joka vaatii resursseja. FitX kompensoi ylimääräistä resurssitarvetta kehitysympäristötuella, jolloin kehyksen asentaminen, testien kehittäminen ja suorittaminen on yksinkertaista.

Toinen määrällinen mittari on ohjelmoijien ja määrittelijöiden menetelmän opiskeluun vaatima aika. Taulukkoesitys on melko yksinkertainen opetella, joten määrittelijälle tämä tuskin vaatii pitkää opiskeluaikaa. Sen sijaan toimintojen ajattelemisen sekvensseinä, jotka voidaan suorittaa testitapauksina voi olla hankalaa ja vaatii järjestelmällistä ajattelutapaa. Ohjelmoijalle testikerrosten ohjelmointi on tavallisten metodikutsujen ohjelmoimista, jonka opiskelu on suoraviivaista.

Laadullisena mittarina FitX:lle voidaan pitää vaatimusten ja testitapausten kattavuutta. Voidaan olettaa, että testitapausten kattavuus on hyvä, koska määrittelyt toteutetaan testitapauksina. Testitapaukset ovat kattavat, kun jokaiselle määrittelylle on testitapaus. Näin ollen kattavuus on rakennettuna menetelmässä sisään. Vaatimusten kattavuus saattaa jonkin verran kärsiä esimerkiksi luonnolliseen kieleen verrattuna, koska määrittelijä joutuu toimimaan taulukon rajaamissa puitteissa. Luonnollisella kielellä asiat pystytään esittämään vapaammin. Toisaalta taulukkoesityksessä säästytään monelta luonnolliselle kielelle tyypilliseltä ongelmalta, kuten hälyltä ja ylimäärittelyltä.

8 Yhteenveto

Laadukkaan sovelluksen ominaisuudet vastaavat käyttäjien vaatimuksia. Vaatimukset voivat olla toiminnallisia tai ei-toiminnallisia, kuten turvallisuus tai suorituskyky. Laadukkaaseen ohjelmistoon pyrittäessä on keskityttävä vaatimusten keräämiseen ja vaatimusten toteuttamiseen. Ohjelmistoprojekti on laadukas, kun se onnistuu keräämään käyttäjien vaatimukset kattavasti ja pystyy verifioimaan toteutuksen vaatimusten mukaiseksi. Ohjelmiston laatuun on kiinnitetty erityistä huomiota 70-luvulta alkaen, jolloin ensimmäinen prosessimalli esiteltiin ohjelmistojen järjestelmälliseen laadun parantamiseen.

Useita prosessimalleja on esitelty 70-luvun jälkeen mutta suuri osa malleista perustuu samoihin komponentteihin; vaatimusten keräämiseen, sovelluksen suunnitteluun ja kuvaamiseen, sovelluksen toteutukseen sekä eritasoiseen testaamiseen. Prosessimallien oleelliset erot syntyvät dokumentointiin ja suunnitteluun käytetystä ajasta sekä kehitysyksien pituudesta. Ohjelmiston testaamiseen on kiinnitetty eniten huomiota V-mallissa, jossa kehitystasoilta dokumentoidaan vaatimukset. Tasoilta tuotetut vaatimukset testataan toteutusta vastaan. V-mallin etuna on, että se huomio järjestelmään kohdistuvat eritasoiset vaatimukset. Käyttäjien vaatimukset kohdistuvat lähinnä liiketoiminnan kannalta hyödyllisiin toimintoihin. Sovelluksen toteutuksella ja ylläpidolla on omat sovelluskohtaiset ei-toiminnalliset vaatimukset. Tämän lisäksi sovelluksen sisäisellä arkkitehtuurilla on omat vaatimuksensa.

Ohjelmistotekniikassa on osoitettu, että ohjelmaa ei voida osoittaa virheettömäksi. Yksinkertaisestakin sovelluksesta voidaan johtaa niin suuri joukko testitapauksia, ettei niitä käytännössä ole mahdollista suorittaa. V-mallin etuna on, että siinä jokainen vaatimustaso testataan. Ohjelmistoa ei ole mahdollista testamalla tai muodollisesti todistamalla osoittaa täysin vaatimusten mukaiseksi, mutta testaamisella voidaan osoittaa, että ohjelma toteuttaa vaatimukset testattavilta osin. Testaamisen oleellisin tehtävä on virheiden löytäminen.

Testaamalla ohjelmisto voidaan osoittaa laadukkaaksi vaatimusten osalta, mutta ohjelmistokehityksessä ongelma on vaatimusten laatu. Vaatimusmäärittely on läsnä jokaisessa ohjelmistoprosessissa. Vaatimuksia ei aina ole muodollisesti kirjattu, mutta

pienelläkin ohjelmalla on vaatimukset, joita ohjelma on rakennettu täyttämään.

Vaatimusmäärittelyn tavoitteena on koota sidosryhmät yhteen ja kerätä heiltä tavoitteet rakennettavalle ohjelmalle. Vaatimusmäärittely rajaa vaatimukset kehykseen, joka on toteutettavissa ohjelmistotekniikan menetelmin. Vaatimukset saattavat olla ristiriitaiset. Vaatimusmäärittelydokumenttiin ristiriidat neuvotellaan ja kirjataan yksikäsitteisinä vaatimuksina. Vaatimusmäärittelydokumentti toimii kommunikaatiovälineenä koko ohjelmistokehityksen ajan aina ratkaisun suunnittelusta testaukseen.

Vaatimusmäärittelydokumentin ongelmana voivat olla häly, moniselitteisyys, ylimäärittely, toiveajattelu ja puuttuvat vaatimukset. Vaatimusmäärittely voidaan toteuttaa esimerkiksi formaaleilla malleilla, vaatimusmäärittelykielillä ja luonnollisen kielen dokumenteilla. Tavallinen vaatimusten esitystapa on luonnollinen kieli, jota täydennetään tarpeen mukaan erilaisilla kuvauksilla. Luonnollisen kielen vaatimuksissa tyypillinen ongelma on häly. Vapaamuotoisesta tekstistä on hankala kaivaa vaatimusten joukkoa ohjelmakehityksen ja testauksen tueksi. Muodolliset vaatimusmäärittelykielet ja matemaattinen esitys eivät vastaa vaatimusten kommunikointitehtävän tarpeisiin. Muodollista esitystä ymmärtää tyypillisesti vain pieni joukko kehitysryhmästä.

Vaatimusmäärittelyä ei voida osoittaa kattavaksi vastaavasti kuin testaamalla ei voida osoittaa ohjelmistoa virheettömäksi. Molemmat asiat jäävät kehitysryhmän ammattitaidon varaan. Kehitysryhmän ja sidosryhmien on kyettävä keräämään ohjelmiston kannalta oleelliset toiminnot yksiselitteisiksi vaatimuksiksi, joiden on oltava testattavissa. Testaaminen on suoritettava niin, että käyttäjien vaatimukset katetaan ei-toiminnalliset vaatimukset huomioiden.

Ohjelmistokehityksen kannalta formaalit vaatimukset ovat yksiselitteinen tapa kerätä vaatimukset. Tämä ei kuitenkaan ole usein mahdollista, koska se rajaisi vaatimusmäärittelyiden käyttäjäkuntaa. Toisaalta luonnollinen kieli jättää liikaa mahdollisuuksia tulkinnalle. Soveltamalla kombinatorisessa logiikassa esiteltyjä päätöstauluja voidaan testitapaukset esittää yksiselitteisesti mutta ymmärrettävästi. Taulukoiden lisäksi tarvitaan tavallisesti luonnollisen kielen selitys avaamaan taulukon sisältöä. Fit-testauskehyksessä vaatimukset kerätään html -taulukoihin, joista ne ovat automaattisesti testattavissa. Lisäksi vaatimukset voidaan esittää yhtenä liiketoimintaprosessina, jolloin mukaan voidaan sijoittaa luonnollista kieltä ja testitapausten ketjuja. Vaatimukset kirjataan taulukkoon, jolloin luonnollisen kielen vaatimukset voidaan tulkita

skenaarioina. Skenaario muodostuu muuttujista ja metodeista, joihin odotetaan vastinetta testattavalta ohjelmalta. Ohjelmoijat rakentavat taulukoiden ja testattavan ohjelman väliin testikerrokset, jotka suorittavat testattavan ohjelman metodit ja muuttujat. Palautteena saadaan joko onnistunut tai epäonnistunut testi. Mikäli testi on aiheuttanut poikkeuksen ohjelmassa, palautetaan kääntäjän lähettämä virheilmoitus.

Fit -kehiksen implementointi ja käyttö vaatii ohjelmakehitysryhmältä syvällistä perehtymistä kehiksen toimintaan. Fit:n ympärille on rakennettu joukko ohjelmia, jotka tukevat sen käyttöönottoa sekä käyttöä. Ohjelmat ovat kuitenkin irrallisia kokonaisuuksia, eikä esimerkiksi kehitysympäristötukea ole saatavissa kuin Eclipselle. Osaltaan tästä syystä Fit ei ole levinnyt laajempaan käyttöön.

Tässä työssä Fit:stä on kehitetty laajennus FitX, jonka tarkoituksena on integroida kehiksen toiminta eri kehitysympäristöihin. FitX:n tarkoituksena on tukea vaatimusmäärittelyä, jotta jokaista käyttötapausta ei tarvitse määrittellä erikseen taulukoon. Tätä tarkoitusta varten kehikseen on rakennettu tuki staattisille tauluille, joihin voidaan viitata muista taulukoista. Kehys on rakennettu tukemaan xml -dokumentteja, jotka ovat paremmin strukturoitavissa kuin html -dokumentit. Lisäksi FitX:ssä on tuki perinnejärjestelmien testaamiseen. Kehitetyt järjestelmät usein sijoittuvat ympäristöön, jossa käyttäjänä on sekä ihmisiä että sovelluksia. FitX:llä on mahdollista automatisoida testit ympäröivien sovellusten kanssa. Tämä aiheuttaa kehitystyötä muihin järjestelmiin soap -rajapinnan tukemiseksi. Toisaalta FitX:n käyttö tämän jälkeen on suoraviivaista, koska kehittäjillä ja käyttäjillä on pääsy samoihin testeihin sekä testien arkistoihin. Lisäksi testien lisääminen ja suorittaminen on mahdollista eri kehitysympäristöistä. FitX:llä määrittelyt on toteutettavissa yksiselitteisesti ja automaattisesti testattavasti, jolloin testeistä saadaan kattavat. Lisäksi testit tallentuvat myöhempiä kehityskierroksia varten regressiotestattavaksi. Näin ollen laadun voidaan olettaa säilyvän tasaisena myöhemmässä kehityksessä.

FitX rajoittuu vaatimusmäärittelyiden tallentamiseen ja hyväksymistestaukseen. FitX:llä ei voida testata käyttöliittymää. Käyttöliittymä on rinnakkainen kerros testikerroksille ja käyttöliittymässä voi syntyä virheitä, jotka tulee testata erikseen. FitX ei myöskään tarjoa tukea muiden kehitysvaiheiden testaamiseen. Näin ollen yksikkötesteillä, integraatiotestauksella ja järjestelmätestauksella on ohjelmistokehityksessä merkittävä rooli. Tämä rooli korostuu ei-toiminnallisten vaatimusten testauksessa, sillä hyväksy-

mistesteissä ei-toiminnallisia laatutekijöitä ei voida kattavasti todentaa.

Tulevaisuuden kehityssuuntana voidaan ajatella kehyksen laajentamista tukemaan muita ohjelmistokehityksen vaiheita. Mikäli kehystä voidaan soveltaa ratkaisun ja arkkitehtuurin määrittelyssä sekä varsinaisessa ohjelmistossa, säästetään aikaa päällekkäiseltä työltä. Vaatimusmäärittelyiden perimmäinen tarkoitus on osoittaa ongelma, joka soveluksen tulee ratkaista. Ratkaisun esittämisessä on erilaiset tarpeet kuin vaatimusten esittämisessä. Vaatimuksen ja ratkaisun esittäminen samassa yhteydessä on käytännöllistä päällekkäisen työn karsimisen kannalta, mutta ongelman ja ratkaisun esittäminen samassa yhteydessä ei ole mutkatonta. Tämä ongelma jää tulevien töiden ratkaistavaksi.

Lähteet

- ABC82 Adrion, W.R., Branstad, M.A., Cherniavsky, J.C., Validation, verification and testing of computer software. *ACM computing surveys*, USA, 1982, sivut 159-192
- And04 Andrea, J., Generative acceptance testing for difficult to test software. *Proceedings of the 5th International Conference on XP 2004*, Garmisch-Partenkirchen, Germany, 2004, sivut 29-37.
- And07 Andrea, J., Envisioning the next-generation of functional testing tools, *Software, IEEE*, 2007, sivut 58-66.
- Bec99 Beck, K., *Extreme programming explained*. Addison-Wesley, 1999.
- Bec03 Beck, K., *Test-driven development*. Addison-Wesley, 2003.
- Bei84 Beizer, B., *Software system testing and quality Assurance*, Van Norstrand, 1984
- Bin99 Binder, R., *Testing object-oriented systems*. Addison-Wesley, 1999.
- Bra02 Bray, I., *An introduction to requirements engineering*, Addison-Wesley, 2002.
- Cun07 Cunningham, W., Making fixtures, *Framework for integrated testing*, October 2007, <http://fit.c2.com/> [6.7.2009]
- Dav90 Davis, A., *Software requirements: analysis and specification*, Prentice hall, 1990.
- DDH72 Dahl, O.-J., Dijkstra, E., Hoare, C., Notes on structured programming, *Academic press*, 1972.
- Den07 Deng, C., FitClipse, a testing tool for supporting executable acceptance test driven development, A thesis for the degree of master of science, University of Calgary, 2007

- EiR96 Eickelmann, N.S., Richardson, D.J., What makes one software architecture more testable than another. *Foundations of Software Engineering, Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ACM, California, USA, 1996, sivut 65-67
- Gil85 Gilb, T., Evolutionary delivery versus the "waterfall model", *ACM sigsoft software engineering notes*, vol 10, 1985, sivut 49-61
- GKS08 Garcia, C., Kropp, M., Schwaiger, W., Fit for business process testing, *IMVS Fokus Report 2008*, IMVS, 2008, sivut 36-40
- Hal88 Hall, P., Towards Testing with respect to formal specification, *Software engineering 88., Second IEEE/BCS Conference*, IEEE, Liverpool, UK, 1998, sivut 159-163.
- HiT00 Hilburn, T., Towhidnejad, M., Software Quality: A curriculum postscript, *ACM SIGCSE bulletin*, 32, 1(2000), sivut 165-171.
- HsK97 Hsia, P., Kung, D., Software requirements and acceptance testing, *Annals of software engineering*, 3, Springer, 1997, sivut 291-317.
- Hum89 Humphrey, W.S., *Managing the software process*, Addison-Wesley, 1989
- Iee98 IEEE, *Recommended practice for software requirements specifications*, IEEE-SA standards board, New York, 1998
- Iso91 ISO/IEC, *Information technology - Software product evaluation - Quality characteristics and guidelines for their use*, International Standard ISO/IEC 9126, Geneva, 1991
- Jac95 Jackson, M., Problems & requirements. *Proceedings of the IEEE Second International Symposium on Requirements Engineering*, IEEE, York, UK, 1995, sivut 2-8.
- JPZ96 Janici, R., Parnas, D., Zucker J., Tabular representations in relational documents, *Springer-Verlag*, New York 1996.
- Kaa08 Kaarakka, T., *Algoritmit matematiikka*, Luentomoniste, 2008.

- KoM05 Koskimies, K., Mikkonen, T., *Ohjelmistoarkkitehtuurit*, Talentum, 2005.
- LBJ04 Laney, R., Barroca, L., Jackson, M., Nuseibeh, B., Composing requirements using problem frames. *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE*, Milton Keynes, UK, 2004, sivut 122-131.
- MaM08 Martin, R., Melnik, G., Tests and requirements, requirements and tests: a möbius strip. *IEEE Software*, vol 25, issue 1, 2008, sivut 54-59.
- Mel07 Melnik, G., Empirical analyses of executable acceptance test driven development, A thesis for the degree of doctor of philosophy, University of Calgary, 2007
- MMC06 Melnik, G., Maurer, F., Chiasson, M., Executable acceptance tests for communicating business requirements: customer perspective *Proceedings of AGILE 2006 Conference (AGILE'06) IEEE*, Minneapolis, USA, 2006, sivut 12-46.
- MRM04 Melnik, G., Read, C., Maurer, F., Suitability of FIT user acceptance tests for specifying functional requirements: developer perspective. *4th Conference on Extreme Programming and Agile Methods*, Springer, Calgary, Canada, 2004, Lecture Notes in Computer Science julkaisussa sivut 60-72.
- Mug04 Mugridge, R., Test driving custom Fit fixtures, *Extreme Programming and Agile Processes in Software Engineering*, 3092/2004, Springer 1997, Lecture notes in computer science julkaisussa sivut 11-19
- MuT03 Mugridge, R., Tempero, E., Retrofitting an acceptance test framework for clarity, *Proceedings of the Agile Development Conference*, Agile alliance, Auckland, New Zealand, 2003
- MuC05 Mugridge, R., Cunningham, W., *Fit for developing software*, Prentice Hall, 2005.
- Mye79 Myers G., *The art of software testing*, John Wiley and Sons, 1979.
- NaN08 Nagy, T., Nayné, A., Erlang testing and tools survey, *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, IEEE, Victoria, Canada, sivut 21-28.

- NuE00 Nuseibeh, B., Easterbrook, S., Requirements engineering: a roadmap, *Proceedings of the Conference on the Future of Software Engineering*, ACM, Limeric, Irlanti, 2000, sivut 35-46
- Pol57 Polya G., *How to solve it*, Princeton University Press, 2nd Edition, 1957.
- PAM91 Parnas, D., Asmis, G., Madey, J., Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2), 1991
- PaM08 Park, S., Maurer, F., The benefits and challenges of executable acceptance testing, *Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral*, ACM, Leipzig, Saksa, 2008, sivut 19-22
- Pre95 Pree, W., *Design patterns for object oriented software development*, Addison-Wesley, 1995
- PRI03 Pyhäjärvi, K., Rautiainen, D., Itkonen, J., Increasing understanding of modern testing perspective in software product development projects, *Proceedings of the 36th Hawaii International Conference on System, Sciences*, IEEE, Havaji, USA, 2003, sivut 10-20
- Rob97 Robinson, S., Simulation model verification and validation: Increasing the users confidence. *Proceedings of the 1997 Winter Simulation Conference*, IEEE, Atlanta, USA, 1997, sivut 53-59.
- Rog04 Rogers, R., Acceptance testing vs. unit testing: a developer's perspective, *Extreme Programming and Agile Methods – XP/Agile Universe 2004: 4th*, Springer Berlin/Heidelberg, 4(2004), Lecture notes in computer science julkaisussa sivut 22-31.
- Som05 Sommerville, I., Integrated requirements engineering: a tutorial, *IEEE Software*, 22, 1(2005), sivut 16-23.
- Som07 Sommerville, I., *Software engineering*, Addison-Wesley, 2007.
- Roy70 Royce W., Managing the development of large software systems, managing the development of large software systems, *Proceedings of the 9th international conference on Software Engineering*, IEEE WESCON 26 (August), IEEE, California, USA, 1970, sivut 1-9.

Zav97 Zave, P., Classification of research efforts in requirements engineering, *ACM Computing Surveys*, 29, 4(1997), sivut 226-235.