TESTITAPAUSTEN SUUNNITTELU UML-MALLINNUKSEN AVULLA

Marko Jäntti

Pro gradu -tutkielma

Tietojenkäsittelytiede

Kuopion yliopisto

Tietojenkäsittelytieteen laitos

Elokuu 2003

Tiivistelmä

KUOPION YLIOPISTO, Informaatioteknologian ja kauppatieteiden tiedekunta

Tietojenkäsittelytieteen koulutusohjelma

Ohjelmistotekniikka

JÄNTTI, MARKO J.: Testitapausten suunnittelu UML-mallinnuksen avulla

Pro gradu -tutkielma, 92 sivua, 1 liite.

Ohjaaja: FT Anne Eerola

Elokuu 2003

Avainsanat: testaus, testitapaus, UML, testausmalli

Testitapausten suunnittelu tulee aloittaa mahdollisimman aikaisessa vaiheessa ohjelmistotuotantoprosessia. Tällä tavoin ohjelmistoista voidaan löytää virheet tehokkaasti ja minimoidaan virheenkorjauksista aiheutuvat kustannukset. Eri testausmenetelmille ja testaustasoille, kuten moduuli-, integrointi-, järjestelmä- ja hyväksymistestaukselle suunnitellaan erilaiset testitapaukset.

Testitapausten suunnittelun on oltava järjestelmällistä, keskitettyä ja automatisoitua. Järjestelmällisyys tarkoittaa sitä, että testitapaukset valitaan huolellisesti, jokaista syöteyhdistelmää kokeillaan ja tulokset raportoidaan. Testaus tulee keskittää sinne, missä virheitä esiintyy useimmin ja virheet ovat vakavimpia. Testauksen priorisoinnilla on suuri merkitys, koska resurssit eivät riitä kaikkien syöteyhdistelmien testaamiseen. Testauksen automatisointi on tärkeää, jotta pystytään tuottamaan ja suorittamaan suuri määrä johdonmukaisia ja toistettavia testitapauksia.

Testauksen avuksi testaaja voi muodostaa testausmallin, joka kuvaa testattavan kohteen käyttäytymistä ja rakennetta sekä ympäristöä. Testausmalleina käytetään muun muassa tilakoneita, kombinaatiologiikkaa ja UML-kaavioita. Määrittelypohjaisen testauksen tutkimus on nykyisin painottunut suurelta osin testitapausten johtamiseen UML-kaavioista, esimerkiksi tilakaavioista.

Tämän tutkielman tavoitteena on selvittää, miten testitapauksia suunnitellaan UML-kaavioiden pohjalta. Tutkielmassa esitellään jokaisen UML-kaavion yleistarkoitus, notaatio ja merkitys ohjelmistotestauksen näkökulmasta. Testausprosessia varten muodostetaan UML-pohjainen testausmalli, joka tunnistaa testauksen eri ulottuvuudet ja auttaa testaajaa jäsentämään testitapaukset loogiseen järjestykseen. Testausmalli toimii ohjenuorana testaustiimille koko testauksen ajan.

Esipuhe

Tutkielma on tehty Kuopion yliopiston tietojenkäsittelytieteen laitokselle lukuvuonna 2002-2003 PlugIT-Teho -tutkimushankkeessa. Tutkimuksen rahoittajina ovat TEKES sekä useat ohjelmistotalot ja sairaanhoitopiirit. Kiitän ohjaajaani professori Anne Eerolaa hyvästä ohjauksesta ja koko tietojenkäsittelytieteen laitoksen henkilökuntaa mukavasta työilmapiiristä. Kiitokset Elinalle tuesta ja kannustuksesta kotona.

Kuopiossa, 20.8.2003

_____

Marko Jäntti

Sisällysluettelo:

LIITTEET

1 Johdanto

Mitä maailmassa tapahtuisi ilman testausta ja tarkastuksia? Virheellisten lennonjohtojärjestelmien ansiosta lentokoneet törmäilisivät yhteen kentällä tai ilmassa. Katsastamattomat autot ilman jarruja, valoja tai muita turvallisuuslaitteita aiheuttaisivat lukuisia liikennekuolemia. Pilvenpiirtäjät ja sillat sortuisivat tarkastamattomien rakennevirheiden vuoksi. Sairaalan potilaat joutuisivat vääriin hoitotoimenpiteisiin tai järjestelmät jättäisivät heidät kokonaan ilman hoitoa kadottamalla lähetteitä ja potilastietoja. Maailma olisi siis varsin turvaton paikka elää.

Testitapausten suunnittelu ohjelmistotuotannossa ja käyttöönotossa vaatii testaajalta paljon mielikuvitusta, kokemusta ja järjestelmällisyyttä. Kokematon testaaja ei testausta aloittaessaan pysty hahmottamaan itselleen, miten valtavaksi testitapausten määrä voi kasvaa jo yksinkertaisen ohjelman kohdalla. Kokenut testaaja tunnistaa testauksen eri tasot ja ulottuvuudet sekä tietää jo projektin alussa, missä ohjelmoijille sattuu virheitä todennäköisimmin. Testaajan asiantuntemusta kannattaa käyttää jo ohjelman suunnitteluvaiheessa esimerkiksi virhetilanteiden ja poikkeusten käsittelyn suunnitteluun tai ohjelmiston hyväksymiskriteerien laadintaan.

Ohjelmistojen testaus on yrityksissä usein ohjelmistotuotantoprosessin heikoimpia lenkkejä. National Institute of Standards and Technology (USA) on arvioinut, että suuren luokan ohjelmistovirheet maksavat USA:n kansantaloudelle vuositasolla jopa 59.5 miljardia euroa [Ger02]. Kolmasosa näistä kustannuksista (n. 22.2 mrd. euroa) olisi tutkimusten mukaan eliminoitavissa parantamalla testausinfrastruktuuria, joka keskittyy aikaisessa vaiheessa tapahtuvaan virheiden tunnistamiseen ja poistamiseen. Testausta ja tarkastuksia lisäämällä virheet löydetään ohjelmista nopeammin ja helpommin, mikä säästää virheiden korjaukseen kuluneet rahat vaikkapa tuotekehitykseen. Testaus tulee huomioida heti ohjelmistoprojektin alkuvaiheessa, koska virhe on helpompi korjata suunnittelupöydällä kuin valmiiseen sovellukseen, joka on jo toimitettu monille asiakkaille.

Vuonna 2000 Suomessa tehdyssä SYTYKE-jäsenkyselyssä *Systeemityön nykytilan kartoitus* saatiin vastauksia (yht. 75 kpl) projektipäälliköiltä, suunnittelijoilta ja asian-

tuntijoilta sekä johto/esimiesasemassa olevilta henkilöiltä [Ham00]. Vastausten mukaan projektinhallinnassa, prosesseissa ja määrittelyissä oli eniten kehitettävää, kun taas testaus löytyi vasta kuudennelta sijalta. PlugIT-projektissa tehty kysely ohjelmistotuotannon nykytilasta osoitti testauksen ongelmiksi seuraavia asioita: Testausympäristö ei vastaa tuotantoa ja virheet tulevat esille vasta, kun järjestelmää käyttävät monet henkilöt samanaikaisesti. Suorituskykytestaukset tehdään vasta tuotannossa ja on olemassa monta tapaa tehdä testausta [Plu03]. Testauksen merkitys tulee todennäköisesti korostumaan vuosi vuodelta enemmän ohjelmistokehitystyössä hajautettujen tietojärjestelmien laadunvarmistuksen ja arkkitehtuurien kehittymisen myötä.

Testitapausten suunnittelun on tapahduttava järjestelmällisesti pohjautuen asiakasvaatimuksiin sekä asiakkaan kanssa laadittuihin määrittelydokumentteihin. Testausmallilla tarkoitetaan kaiken sen tiedon keräämistä ja kuvaamista, millä on merkitystä testattavan ohjelman käyttäytymiseen tai rakenteeseen liittyen. Testausmallissa voidaan käyttää hyväksi vaatimusmäärittely- ja suunnitteluvaiheen tuloksia. UML (Unified Modeling Language) on oliokeskeisen systeemin mallintamiskieli, jota käytetään ohjelmistoprojekteissa määrittelyyn, ongelman tai toimintojen visuaaliseen kuvaamiseen ja dokumentointiin [Kru00]. UML-pohjainen testausmalli perustuu siis määrittelyihin ja suunnitteludokumentteihin, joissa on käytetty hyväksi UML-notaatiota. Asiakasvaatimuksista, määrittelydokumenteista ja UML-kaavioista johdetaan testitapaukset, jotka tallennetaan testitietokantaan tai testitietovarastoon, jotta niitä voidaan myös uudelleenkäyttää.

Tutkimuksen taustana on Kuopion yliopistossa ja Pohjois-Savon ammattikorkeakoulussa 1.10.2001 alkanut TEKES-rahoitteinen PlugIT-projekti, jonka tavoitteena on tutkia ja kehittää menetelmiä terveydenhuollon sovellusintegraatiota varten. Projektissa on mukana asiakkaina terveydenhuollon tietojärjestelmiä toimittavia yrityksiä ja sairaanhoitopiirejä sekä työntekijöinä neljä osaprojektia, joista PlugIT-Teho toimii Tietojenkäsittelytieteen laitoksen ohjelmistotekniikan tutkimusalueella. PlugIT-Tehon tehtävänä on tutkia vaatimusten hallintaa, sovellusintegraatiossa käytettäviä testausmenetelmiä ja komponenttiteknologioita. Tämä tutkielma liittyy testausmenetelmien tutkimiseen ja on osa PlugIT-Teho -osaprojektin tuloksia.

Tutkielman tavoitteena on selvittää, miten testitapauksia suunnitellaan ja miten suunnittelussa voidaan käyttää apuna UML-mallinnusta. Tutkielma tarjoaa aloittelevalle testaajalle tai ohjelmistosuunnittelijalle ohjeita testauksen eri muodoista, testauksen dokumentoinnista, automatisoinnista sekä UML-kaavioihin perustuvasta määrittelypohjaisesta testauksesta. Tarkoituksena on yhdistää ohjelmiston testaukseen liittyvät asiat testausmalliksi, jolla kehitetään yritysten ja organisaatioiden ohjelmistotestausta. Testausmallia tai sen osia voidaan käyttää hyväksi ohjaamaan ja selkeyttämään monimutkaisten ja laajojen järjestelmien testausta. Testausmallin yleinen rakenne on kuvattu käsitekartan avulla.

Tutkielma etenee työvaiheittain seuraavasti: Johdannossa selvitetään tutkimuksen taustatekijät ja tavoitteet. Toisessa luvussa annetaan tietoa testauksen pääperiaatteista ja käydään läpi erilaiset testausmenetelmät ja testaustasot. Lisäksi kuvataan testitapausten suunnittelun perusasiat: miten testitapaus määritellään, mistä lähteistä testitapauksia kannattaa etsiä ja mistä virheitä löytyy eniten. Kolmas luku käsittelee testauksen hallintaa kuvaten, miten testausta dokumentoidaan, mitä tarkoittaa testiympäristö ja mitä erilaisilla testaustyökaluilla voi tehdä. Neljäs luku sisältää UML-pohjaisen testausmallin kuvauksen, jossa selvennetään, mihin testausmallia tarvitaan ja mistä asioista malli koostuu. UML-kaavioiden käsittely on jaettu ohjelmistotuotantoprosessin elinkaaren mukaan (vaatimusmäärittely, suunnittelu, toteutus ja käyttöönotto) ja jokaisesta kaaviosta selvitetään sen merkitys yleisellä tasolla, notaatio, esimerkki kaavion käytöstä sekä merkitys testauksen kannalta.

Viidennessä luvussa kuvataan Rationalin testaustyökalujen toimintaa käytännössä. Toiseksi viimeinen luku sisältää pohdinnan, jossa analysoidaan käytännön kokemuksia Kuopion yliopistollisessa sairaalassa tehdystä toiminnallisen testauksen kokeilusta, jossa testauksen kohteena oli potilashallinnon tietojärjestelmä. Yhteenveto ja arvio tutkielman tuloksista ovat viimeisessä luvussa, jossa kuvataan myös jatkotutkimusaiheita.

2 Testauksen periaatteet

2.1 Yleistä testauksesta

Testitapausten suunnittelua varten kannattaa ensin selvittää testauksen tärkeimmät periaatteet, käsitteet ja ulottuvuudet. Haikala ja Märijärvi ovat määritelleet testauksen seuraavasti: *Testaus* (software testing) on järjestelmällistä virheiden etsintää ohjelmaa tai sen osaa suorittamalla [HaM97]. Jacobsonin mukaan ensimmäinen opittava asia testauksesta on se, että ei voida koskaan osoittaa, ettei ohjelma toimi väärin. Voidaan osoittaa vain se, että ohjelma sisältää virheitä [Jac92].

Testauksen ja ohjelmiston *tarkastamisen* (inspection) tavoitteena on tukea laadunvarmistusta keräämällä tietoa aikaisemmista virheistä ja välittämällä tieto ohjelmoijille ja suunnittelijoille, jotta samoja virheitä ei tehdä ohjelmistokehityksessä uudelleen. On tutkittu, että virheen löytämisen ja korjaamisen kustannukset nousevat dramaattisesti, mitä pitemmäksi aikaa virhe jää ohjelmaan. Testausprosessin täytyy alkaa mahdollisimman aikaisessa vaiheessa ohjelmistokehitystä eli mielellään jo vaatimus- ja systeeminmäärittelyn aikana.

Testauksen tutkimuksen on kohdattava uusien teknologioiden asettamat haasteet. Joachim Hauser kertoi ICSTEST-konferenssissa auton mobiiliteknologian kehittymisen vaativan BMW:ltä aivan uudenlaisia ohjelmistokehitysmetodeja ja testausmenetelmiä [Hau03]. Mittausdataa tulisi voida siirtää entistä enemmän auton teknisistä komponenteista ohjelmistokomponenteille (esim. renkaiden pyörimiseen liittyvää dataa ja polkimien painalluksista tulevaa dataa). BMW:n tehtävänä on varmistaa, että myös alihankkijat tekevät laadukasta työtä. Testausta ja tarkastuksia tarvitaan kaikilla toimialoilla: autoteollisuudessa, pankki- ja vakuutussektorilla, päivittäistavarakaupassa, rakennusalalla ja terveydenhuollossa.

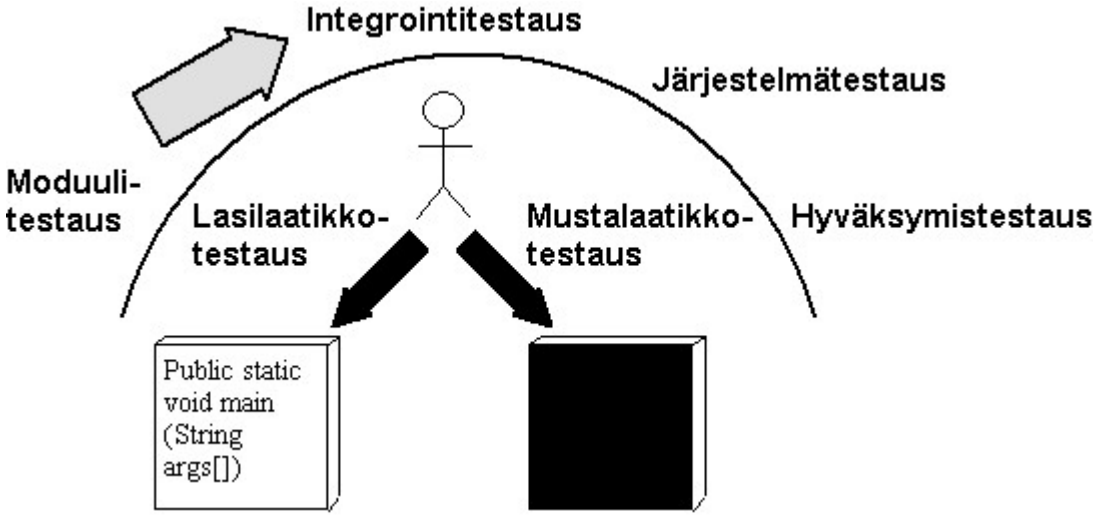
Ohjelmiston testauksesta on vastuussa koko projektitiimi yhdessä, vaikka varsinainen testaustyö kohdistetaan tiettyjen henkilöiden tehtäväksi. Testausprosessiin osallistuvat työntekijät voidaan jakaa kahteen eri rooliin: *testauksen suunnittelijoihin* (test designer) ja *testaajiin* (tester). *Rational Unified Process* (RUP) määrittelee testauksen suunnittelijan tehtäviksi testauksen suunnittelun ja valmistelun kuten testaussuunni-

telman laatimisen, testitapausten suunnittelun ja rakentamisen, testauksen kattavuuden, testitulosten ja testauksen tehokkuuden arvioinnin. Testaaja on vastuussa testien suorittamisesta ja suorittamiseen liittyvistä valmisteluista, testauksen suorittamisen ja virheistä toipumisen arvioinnista sekä muutospyyntöjen kirjaamisesta. [Kru00]

Testausstrategia tai testaustekniikka on järjestelmällinen menetelmä testien valitsemiseen tai luomiseen testauskehystä varten. *Testauskehys* (test suite) on valittu joukko saman tyyppisiä testitapauksia tietyn toiminnon testausta varten [Paa00]. Testausstrategia antaa säännöt, joiden avulla voidaan määritellä, täyttääkö testi asetetut vaatimukset. Strategia on tehokas, mikäli sen testitapaukset paljastavat riittävästi virheitä testauksen kohteessa. Usein käytettäviä perusstrategioita testauksessa ovat seuraavat [MaK00]:

- *Käyttäytymiseen pohjautuvat* (behavioral) testausstrategiat, joita kutsutaan myös mustalaatikko- tai toiminnalliseksi testaukseksi. Nämä perustuvat vaatimuksiin ja ne voidaan teoriassa (mutta ei käytännössä) tehdä ilman tietoa testauskohteen rakenteesta.
- *Rakenteelliset* (structural) testausstrategiat eli lasilaatikkotestaus, jossa keskitytään testattavan kohteen sisäiseen rakenteeseen.
- *Hybridi-testausstrategiat* (hybrid), jotka ovat yhdistelmiä kahdesta ensimmäisestä testausstrategiasta.

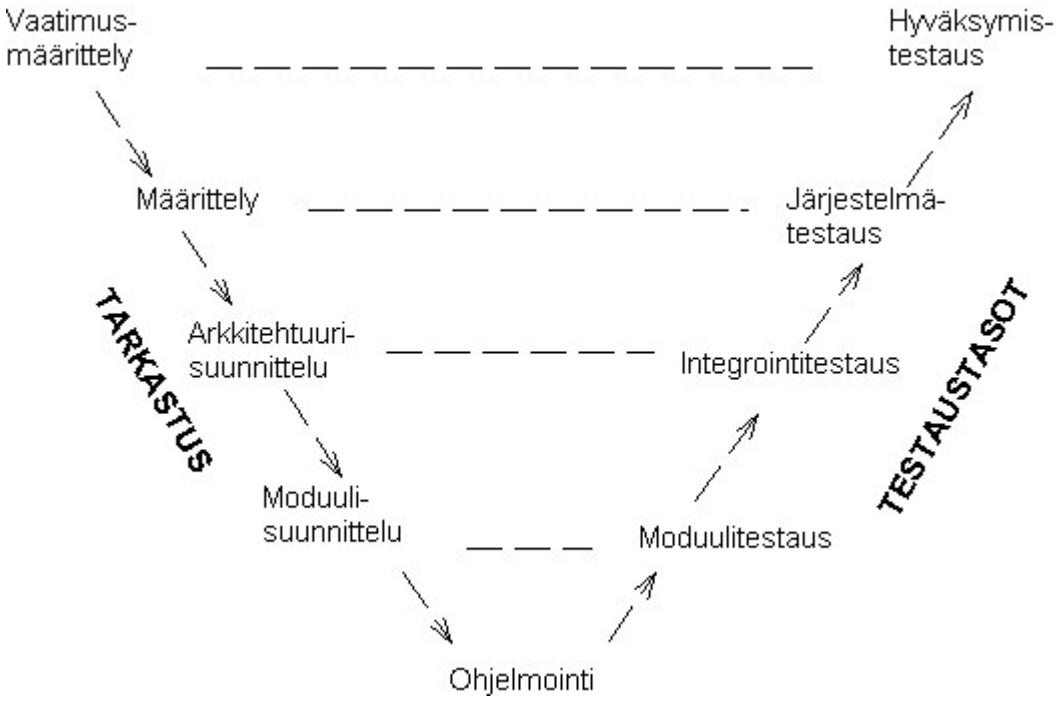
Testaus voidaan jakaa erilaisiin testaustasoihin ja –menetelmiin (tekniikoihin) seuraavasti (Kuva 1): Testauksen tasoja ovat yksikkötestaus, integrointitestaus, järjestelmätestaus ja hyväksymistestaus. Testausmenetelminä voidaan käyttää joko mustalaatikkotestausta tai lasilaatikkotestausta.



Kuva 1: Testaustasot ja -menetelmät

2.2 Testauksen tasot

Testausprosessin eri tasot voidaan kuvata V-mallin mukaisesti (Kuva 2). *Testauksen V-mallin* tarkoituksena on esittää, miten testauksen vaiheet heijastuvat ohjelmiston rakennusvaiheisiin. V-mallin vasen puoli kuvaa rakennusvaiheita ja oikea puoli testauksen eri vaiheita. Abstraktiotaso kasvaa noustaessa alhaalta kolmion kärjestä kohti kolmion kantaa. Moduulitestauksessa painopiste on itsenäisten ohjelmamoduulien koodissa. Tämä moduuli voi olla ohjelma, luokka tai komponentti. Integraatiotestaus keskittyy moduulien yhteistoimintaan ja järjestelmätestaus testaa systeemiä kokonaisuutena. Ylimmällä tasolla oleva hyväksymistestaus painottaa vaatimusten täyttymistä eli testauskohteena ovat järjestelmän tarjoamat palvelut.



Kuva 2: Testauksen V-malli

V-malli osoittaa, että testaus voidaan ja myös pitäisi ottaa huomioon ennen kuin mitään valmista koodia on tarjolla. Jotta testaus olisi kustannustehokasta ja niin tehokasta kuin mahdollista, määrittely- ja suunnitteluvaiheissa tulee valmistella testausta suunnittelemalla testausstrategia ja testitapaukset määrittelyjen ja suunnittelun tulosten tarkastamiseksi sekä vaatimusten toteutumisen varmistamiseksi [Paa00, s.7].

2.2.1 Yksikkötestaus

Yksikkö- tai moduulitestauksessa (unit/module testing) jokainen ohjelmistomoduuli (esim. Java-luokka) testataan erikseen ohjelmakooditasolla. Termi *moduuli (module)* tai *yksikkö (unit)* viittaa käsitteeseen, joka toteuttaa loogisen joukon palveluita. Tavallisesti ohjelmoijat rakentavat moduuleja itsenäisesti, joten niitä voidaan testata erikseen. Koska moduuli ei yleensä ole itsenäinen ohjelma, tarvitaan moduulien testaamiseen ajureita ja tynkämoduuleita: *Ajuri* (driver) esittää pääohjelmaa, joka kutsuu testattavaa moduulia ja *tynkämoduuli* (stub) esittää aliohjelmaa, jota testattava moduuli kutsuu.

Yksikkötestaus mahdollistaa sen, että seuraavilla testaustasoilla voidaan luottaa yksittäisten ohjelmaosien toimivuuteen. Tämän vuoksi yksikkötestaukseen tulee kiinnittää erityisen paljon huomiota. Jotta kattava testaus on mahdollista, ohjelmamoduulin

koodimäärä ei saa olla liian suuri (korkeintaan 1000 riviä) ja koodin tulee olla rakenteeltaan jäsenneltyä ja hyvin dokumentoitua. Tehokas virheiden tai huonosti toteutettujen ratkaisujen etsintäkeino on *paritarkastus*, jossa toinen ohjelmoija käy läpi toisen tekemää koodia ja kyselee eri ohjelmaosien tai ratkaisujen tarkoitusta.

2.2.2 Integrointitestaus

Integrointitestauksen (integration testing) tarkoituksena on varmistaa, että ohjelman osat toimivat yhdessä oikein. Tämä testaustaso keskittyy moduulien välisten *rajapintojen* toimivuuteen. Integraatiotestausta tarvitaan, koska jo testattujen alisysteemien yhdistelmä ei välttämättä toimi oikein, vaikka alisysteemit toimivatkin hienosti erikseen. Virheellisestä toiminnasta on esimerkkinä tilanne, jossa tietoa häviää moduulien rajapinnan tai yhteensopivuusongelman vuoksi.

Integrointitestaus voidaan tehdä liittämällä kaikki moduulit kerralla yhteen (Big Bang). Parempi tapa on testata järjestelmää inkrementaalisesti liittämällä moduulit yhteen joko top-down - tai bottom up –periaatteella [Paa00, Vir02]. Inkrementaalisen testauksen hyötynä on virheiden helpompi paikantaminen alisysteemeistä. Lisätyötä aiheuttaa se, että testauksen suorituksessa tarvitaan ajureita ja tynkämoduuleja. Integraatiotestaus pohjautuu arkkitehtuurisuunnitteluun ja siihen liittyviin dokumentteihin. Arkkitehtuurisuunnittelussa määritellään karkean tason ohjelmistomoduulit ja niiden väliset yhteydet [Bos00]. Integrointitestaus käyttää hyväkseen yksikkötestauksen tuloksia.

2.2.3 Järjestelmätestaus

Järjestelmätestauksessa koko systeemi (ohjelmisto, laitteisto ja tietokannat) testataan kokonaisuutena yhdessä olettaen, että ohjelmiston osat on edeltä käsin testattu itsenäisesti. Järjestelmätestaus voi paljastaa ongelmia, kuten alhainen suorituskyky, yhteen sopimaton laitteisto- tai komponenttikokoonpano, heikko toipuminen virhetilanteista tai poikkeuksista sekä riittämätön turvallisuus. Järjestelmätestaus pohjautuu integrointitestauksen tuloksiin.

Järjestelmätestausta esiintyy seuraavissa muodoissa [Paa00]:

1) *Volyymitestaus* (volume testing) määrittelee, pystyykö järjestelmä käsittelemään riittävää määrää tietoa tai pyyntöjä.
2) *Kuormitustestaus* (load/stress testing) tunnistaa kuormitushuiput, joissa järjestelmä voi epäonnistua käsittelemään tietoa annetuilla aikarajoilla.
3) *Turvallisuustestaus* (security testing) osoittaa, että järjestelmä täyttää sille asetetut turvallisuusvaatimukset esimerkiksi viruksia tai tunkeilijoita vastaan.
4) *Suorituskykytestaus* (performance testing) määrittelee, täyttääkö järjestelmä sille asetetut suorituskykyvaatimukset.
5) *Resurssien käytön testaus* (resource usage testing) tarkistaa, käyttääkö järjestelmä liikaa resursseja (muistia, levytilaa ym.).
6) *Konfiguraatiotestaus* (configuration testing) näyttää, toimiiko järjestelmä oikein, kun ohjelmistot, laitteisto, tietokannat ja ulkoiset laitteet on liitetty yhteen.
7) *Asennettavuustestaus* (installability testing) testaa, johtaako asennusprosessi epäonnistuneeseen asennukseen.
8) *Toipumistestaus* (recovery testing) osoittaa, miten järjestelmä täyttää sille asetetut vaatimukset uudelleenkäynnistämiseen johtavan virheen tai muun virheen jälkeen.
9) *Luotettavuus/saatavuustestaus* (reliability/availability testing) määrittelee, täyttääkö järjestelmä sille asetetut luotettavuus- ja saatavuusvaatimukset. Tutkitaan esimerkiksi, miten usein verkkopohjainen palvelu on käytettävissä.

2.2.4 Hyväksymistestaus

Ennen järjestelmän käyttöönottoa asiakas ja käyttäjä tai molemmat yhdessä tarkistavat *hyväksymistestauksessa*, vastaako järjestelmä sille asetettuja, sopimuksessa määriteltyjä vaatimuksia. Hyväksymistestaus voi olla joko *alfa-* tai *beta-testausta*. Alfatestausta tekevät tulevat käyttäjät ja testaus suoritetaan toimittajan tiloissa, kun taas beta-testaus tapahtuu yleensä oikeassa suoritusympäristössä asiakkaan omissa tiloissa. Hyväksymistestauksen ei välttämättä tarvitse pohjautua mihinkään määrittelydokumenttiin, vaan käyttäjällä saattaa olla mielessään malli, johon hän vertaa ohjelman toimintaa.

Hyväksymistestauksen yksi tärkeä muoto on *käytettävyystestaus*, jonka tavoitteena on parantaa tuotteen käytettävyyttä. Hyvän käytettävyyden ansiosta käyttäjät oppivat ohjelman nopeammin, pystyvät työskentelemään tehokkaammin ja muistavat toimin-

not paremmin käyttötaukojen jälkeen. Hyvä käytettävyys helpottaa tuotteen myyntiä ja pitää asiakkaat tyytyväisenä. Käytettävyys muodostuu useasta tekijästä tuotteen lisäksi kuten käyttäjästä, käyttöympäristöstä, suoritettavasta tehtävästä ja tuotteen ympärillä olevasta tekniikasta [Imm03]. Käytettävyystestauksessa ovat mukana järjestelmän tulevat käyttäjät, jotka suorittavat työhön liittyviä tehtäviä. Käyttäjien toimintaa ja kommentteja tarkkaillaan ja kirjoitetaan muistiin käytettävyystestin aikana. Testaustulokset analysoidaan ja niiden pohjalta suoritetaan korjaus- tai kehitystoimenpiteet.

Käytettävyystestaus aloitetaan määrittämällä päämäärä eli, mitä testauksesta halutaan saada selville. Päämääränä voi olla esimerkiksi se, että aloittelevan käyttäjän tulee selvitä tilauksen tekemisestä websivulla vähintään 20 minuutissa. Testiin osallistujien valinnassa käytetään hyväksi käyttäjäprofiileja eli mietitään olennaiset käyttäjien ja ohjelman piirteet, minkä jälkeen käyttäjäprofiileihin perustuen valitaan testausryhmät. [Wol02]

Testauksen kohteeksi valitaan tehtäviä tai toimintoja, joista oletetaan löytyvän käytettävyysongelmia. Koska kaikkea ei ehditä testaamaan, tehtävät priorisoidaan ja asetetaan tärkeysjärjestykseen. Ennen testausta tehdään päätös, miten kyseisessä testissä mitataan käytettävyyttä. Testimateriaalin valmisteluun kuuluvat salassapitosopimukset, ohjeet testaajille tehtävien suorittamista varten ja testauksen tarkistuslista. Testausympäristön valmistelussa käydään läpi testauksessa tarvittavat laitteet, ohjelmistot ja mahdolliset tallennuslaitteet. Testitiimin valmistelussa päätetään testitiimin koko, tiimin henkilöt ja jaetaan tehtävät. Jokainen tiimin jäsen saa tarkistuslistan ja toimii tietyssä roolissa. Pilottitestaus on harjoittelutilaisuus varsinaista testausta varten ja sillä varmistetaan, että testausmateriaali toimii.

Varsinaisessa testaustilanteessa esitellään testaustiimi ja testauslaitteisto sekä testattava tuote. Tehtävät annetaan testaajalle yksi kerrallaan. On tärkeää, ettei testihenkilöä häiritä tai ohjata suorituksessa muuten kuin täydellisessä umpikujatilanteessa. Testin lopussa testaajat saavat kertoa omia kokemuksia ja mielipiteitä tuotteesta. Päätöskeskustelussa vastataan testaajan kysymyksiin ja keskustellaan ongelma-alueista ja niiden merkityksestä sekä korjausehdotuksista. [May99]

2.3 Testausmenetelmät

2.3.1 Lasilaatikkotestaus

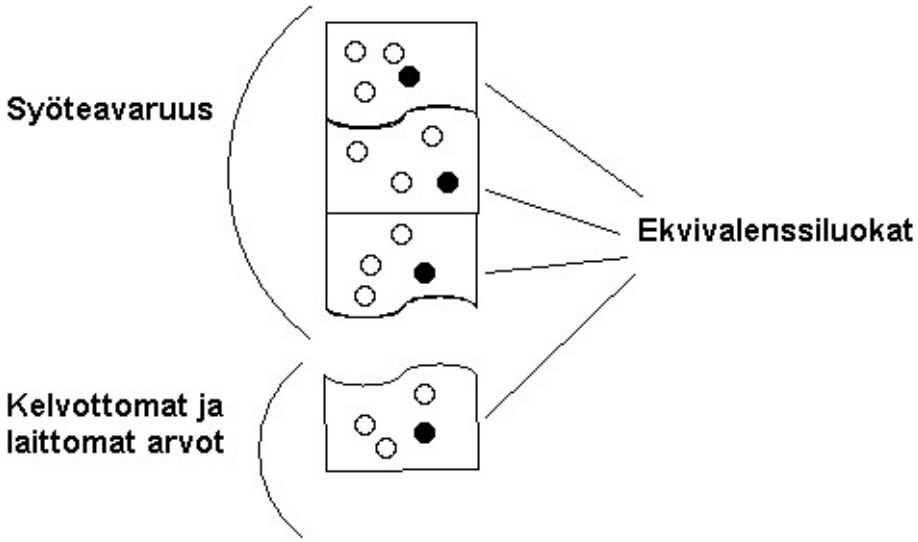
Lasilaatikkotestausta (white-box testing) suorittaa järjestelmän toimittaja tai sovellusintegraattori, jolla on ohjelmakoodi käytössä testauksen aikana. Tätä testausmenetelmää kutsutaan myös rakenteelliseksi testaukseksi, koska testaus kohdistuu ohjelman rakenteeseen. Ohjelmaa pyritään suorittamaan siten, että käydään läpi riittävän monta eri suorituspolkua. Koska lasilaatikkotestauksessa keskitytään toteutuksen yksityiskohtiin, se kuuluu yksikkötestauksen menetelmiin.

2.3.2 Mustalaatikkotestaus

Mustalaatikkotestauksessa (black-box testing) eli toiminnallisessa testauksessa ohjelmiston sisäistä toteutusta (koodia) ei ole nähtävissä, vaan ohjelmisto on ”musta laatikko”. Testaus keskittyy moduulien ja alisysteemien rajapintoihin ja erityisesti ulkoisesti tarkkailtavaan toiminnallisuuteen sekä syöte-tulos-käyttäytymiseen. Mustalaatikkotestauksen periaate on hyvin yksinkertainen:

1) Ohjelma käynnistetään tietyllä syötejoukolla x.
2) Ohjelman antamaa tulosta f(x) syötejoukolla x tarkkaillaan.
3) Todellista tulosta f(x) verrataan edeltäkäsin määriteltyyn eli oletettuun tulokseen y.
4) Jos f(x) = y, ohjelma on läpäissyt testin. Lisäksi ohjelman ja sen ympäristön pitää jäädä suorituksen lopussa oikeelliseen tilaan. Muussa tapauksessa on tapahtunut virhe.

Mustalaatikkotestauksessa on mahdollista käyttää seuraavia tekniikoita testitapausten valintaan: *ekvivalenssiluokat, raja-arvoanalyysi* ja *käyttöliittymän testaus*. Ekvivalenssiluokkien (Kuva 3) käyttö testauksessa on perusteltua, koska järjestelmää on tavallisesti mahdotonta testata kaikilla syöteyhdistelmillä. Testitapausten suunnittelija analysoi kaikkia mahdollisia syötteitä (input space) ja yrittää löytää *ekvivalenssiluokkia* (equivalence classes), joihin kuuluvia syötteitä voidaan käsitellä testauksessa samalla tavalla. Ekvivalenssiluokkia on tutkittu PlugIT-Teho -projektin aikaisemmissa tutkimuksissa [ToJ02].



Kuva 3: Ekvivalenssiluokat

Seuraavassa yksinkertaisessa esimerkissä havainnollistetaan, miten pakollista *potilaan sukunimi* -kenttää testattaessa syötejoukko voitaisiin jakaa ekvivalenssiluokkiin:

- kelvolliset sukunimet (suomalaiset nimet: Korhonen, Mäenpää; ulkomaalaiset nimet: Åkerholm, van Gogh),
- kelvottomat (Korho<nen, Van()Gogh), kun oletetaan erikoismerkkien olevan kiellettyjä ja
- laittomat (tyhjä), koska kenttä on pakollinen.

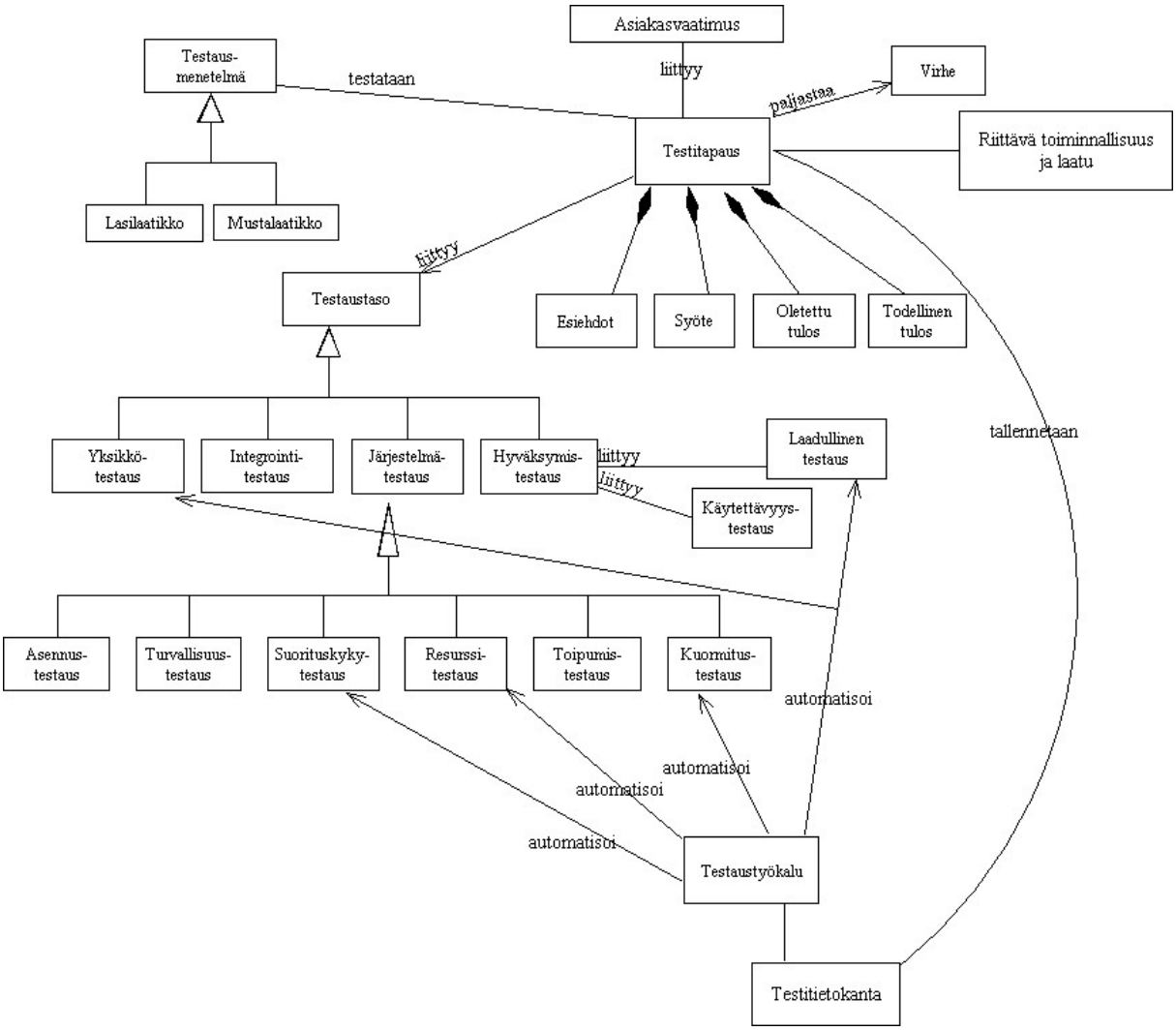
Jokaisesta luokasta poimitaan yksi edustaja, joka kuvaa kaikkia luokassa olevia syötteitä. Testauksessa säästetään aikaa, koska sukunimeä ei tarvitse testata kaikilla mahdollisilla nimillä tai erikoismerkeillä.

Raja-arvoanalyysissä (boundary analysis) testataan syötteitä, joiden pitää olla tiettyjen rajojen mukaiset. Tutkimusten mukaan ohjelmoijat tekevät paljon virheitä silmukka- ja toistorakenteissa. Testitapaukset tulisi siis valita ekvivalenssiluokkien rajoilta eikä pelkästään keskeltä. Teoreettisena esimerkkinä voisi olla järjestelmän toiminto, joka huolehtii, ettei osastolle tule liikaa potilaita. Oletetaan, että vuodeosaston potilasmäärän maksimi on 30. Järjestelmän tulee ilmoittaa, mikäli potilaiden määrä on yli 30. Kutakin rajaa kohden valitaan vähintään kolme testitapausta: rajan sisäpuolelta, rajalta ja rajan ulkopuolelta. Tässä tapauksessa testitapauksiksi tulisivat 29 ja 30 sekä 31.

Käyttöliittymän testaus kuuluu mustalaatikkotestauksen piiriin, koska järjestelmän suorittamisessa käyttöliittymätasolla sisäinen toteutus on piilossa. Erityisesti järjestelmä- ja hyväksymistestauksen tasoilla keskitytään käyttöliittymän testaamiseen. Tilannekohtaisesti tulee harkita, missä on tarvetta automaattisen työkalun käyttöön käyttöliittymän osalta ja missä taas riittää pelkkä manuaalinen testaus. Automatisointi on erityisesti tarpeen usein toistuvissa testitapauksissa, joita on suuri määrä. Käyttöliittymävirheitä voivat olla muun muassa näyttövirheet (tieto väärässä paikassa), puuttuvat toiminnot, virheellinen tai puutteellinen suoritusjärjestys (tallennus jää kesken) ja puutteellinen suorituskyky [Paa00, s.37].

2.4 Testitapausten suunnittelu

Testitapaus (test case) on ohjelmiston rakentamiseen tarvittava tiedonpala tai *tuotos* (software artifact), joka sisältää ohjelmalle annetut syötteet ja oletetut tulokset. Hyvän testitapauksen tarkoituksena on löytää ohjelmassa piilevä vika, jota ei ennen ole havaittu. Testitapausten suhteista on laadittu seuraava käsitekaavio (Kuva 4), jolla testauksen monimutkaista käsitekenttää on selvennetty.

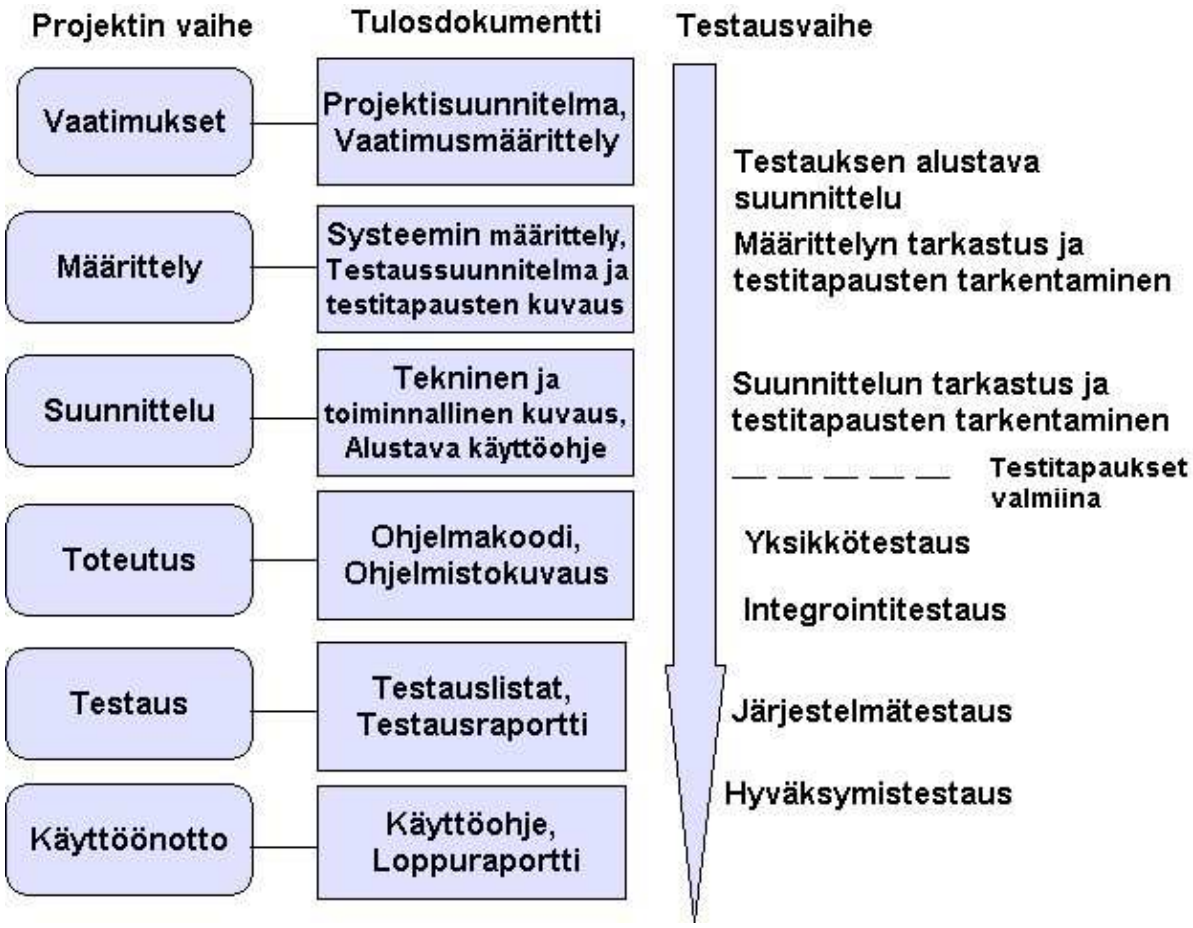


Kuva 4: Testauksen käsitekaavio

Jos testitapausta tarkastellaan jäljitettävyyden näkökulmasta, yksi iso testitapaus voi koostua monesta pienestä testitapauksesta. *Jäljitettävyydellä* (traceability) tarkoitetaan katkeamatonta ketjua asiakasvaatimuksesta lopputuotteen ominaisuudeksi [Myö02]. Testitapaus liittyy siis asiakkaan tai käyttäjän vaatimukseen. Jos ohjelmiston käyttötapauksena on *Lisää potilas*, ylätason testitapaukseksi voidaan asettaa *Lisää potilas,* joka sisältää useita alemman tason testitapauksia, kuten pakollisten kenttien ja eisallittujen syötteiden tarkistukset.

Testauksen ja testitapausten suunnittelun tulisi alkaa mahdollisimman pian ohjelmistotuotantoprosessin alussa mielellään heti vaatimusmäärittelyn yhteydessä. Ohjelmistotuotantoprosessin monimutkaisuus riippuu yrityksen toimintatavoista (noudatettavat

standardit) sekä projektin laajuudesta. Tässä tutkielmassa käytettävä ohjelmistotuotannon vaihejako (Kuva 5) on muunneltu versio perinteisestä vesiputous-mallista [HaM97].



Kuva 5: Testauksen suunnittelu ohjelmistotuotannon eri vaiheissa

Seuraavat ohjelmistotuotantoon kuuluvat toiminnot voidaan käsittää myös testauksesi [MaK00]:

- Protoilu: *Prototyyppi* on ei-valmis toteutus, joka esittää tulevan ohjelman käyttäytymistä. Proton tarkoituksena on muodostaa malli oikean ohjelmaratkaisun löytämiselle.
- Vaatimusanalyysi: Vaatimuksista pitäisi tarkastaa ovatko ne johdonmukaisia, ristiriidattomia, testattavia ja sopivia tuotteeseen.

- Formaali analyysi: Kaikkia mahdollisia syöteyhdistelmiä on mahdotonta testata. Hyvän testitapausjoukon laatiminen ohjelmistolle on monimutkaista ja vaatii aikaa.
- Suunnittelu: Jotkut suunnittelunäkökulmat (testattavuus, ohjelmiston ylläpito) eivät vaikuta suoraan ohjelmiston toiminnallisiin ominaisuuksiin, mutta vaikuttavat ohjelmiston laatuun.
- Muodolliset *tarkastukset* (inspection) ovat katselmuksia, joita pidetään virheiden löytämiseksi tuotteesta. Katselmuksen kohteena voivat olla projektidokumentit, analyysimallit, suunnittelumallit ja testit.
- Itse-testaus: Ohjelmoijat ovat vastuussa ensimmäisistä testeistä moduuli- ja yksikkötestaustasoilla.

Ohjelmistoprojekti alkaa vaatimukset-vaiheella. *Vaatimustenhallinnalla* (requirements management) varmistetaan, että ratkaistaan oikea ongelma ja rakennetaan oikea systeemi. Vaatimustenhallinnassa käytetään järjestelmällistä lähestymistapaa vaatimusten ”kalastamiseen”, analysointiin, dokumentointiin ja hallintaan [RoR99]. Tämän vaiheen tuotoksia ovat projektisuunnitelma ja *vaatimusmäärittely*-dokumentti, jossa luetellaan järjestelmän rakentamisen kannalta olennaiset laadulliset ja toiminnalliset vaatimukset. Laatuvaatimuksia ovat esimerkiksi käytettävyys, luotettavuus ja suorituskyky. Toiminnallisia vaatimuksia esitetään käyttötapauksilla, joista löytyy erilaisia tapahtumaketjuja: perustoiminta (normal flow) ja poikkeustoiminta (exceptional flow).

Määrittelyllä voidaan tarkoittaa koko järjestelmän (ohjelmisto ja laitteisto yhdessä), ohjelmiston tai ohjelmisto-osan määrittelyä. Määrittelyvaiheen (specification phase) tehtävinä ovat projektin tarpeellisuuden ja toteutuskelpoisuuden selvittäminen, tavoitteiden ja vaatimusten asettaminen sekä ratkaisumallin laatiminen [HaM97]. Testaussuunnitelman kirjoittaminen ja testitapausten kuvausten laatiminen kannattaa aloittaa jo määrittelyvaiheessa. Testaussuunnitelma määrittelee testauksen lähestymistavan, resurssit ja aikataulun. Asia, joka jää usein varsin vähälle huomiolle testausta suunniteltaessa, on *testausympäristön* kuvaaminen: mitä laitteistoja, ohjelmia ja tietoliikenneyhteyksiä tarvitaan testauksessa. Testitapausten kuvaukset voidaan liittää testaussuunnitelmaan liitteiksi. Testitapauksia on tarkoitus täydentää ja tarkentaa projektin edetessä.

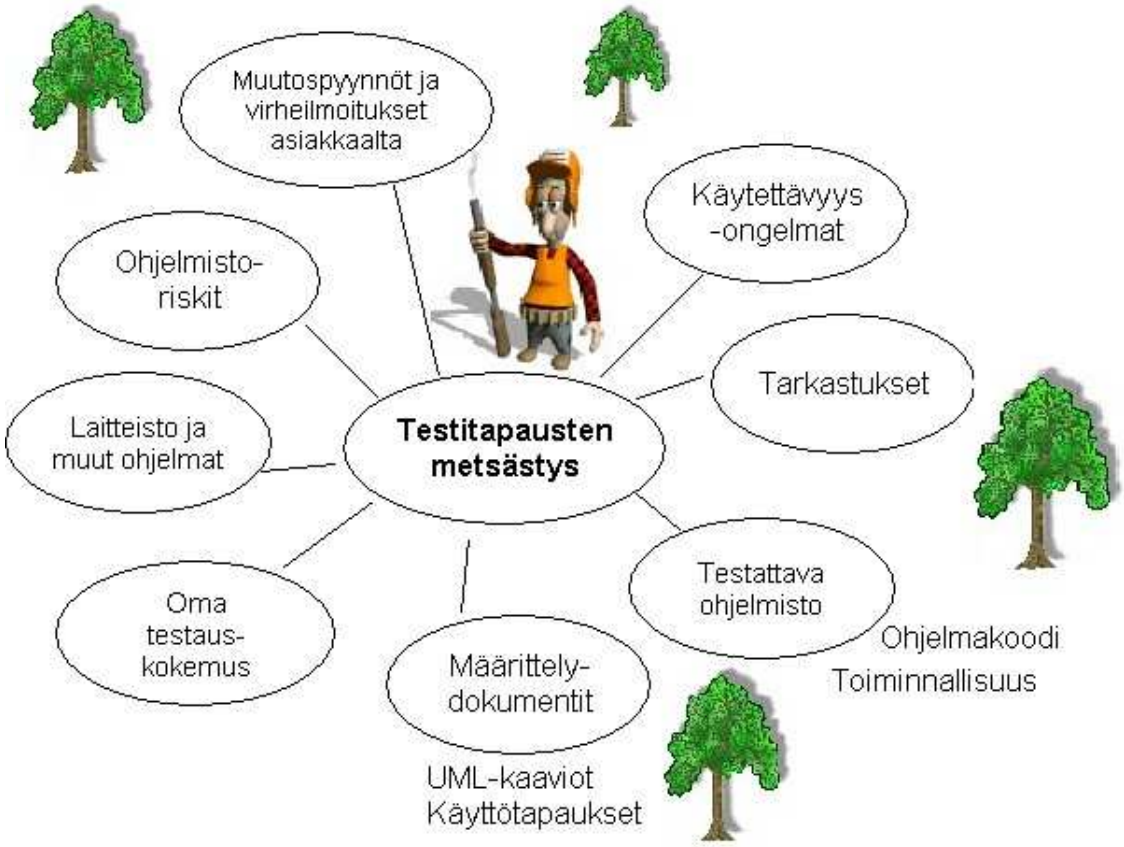
Suunnitteluvaiheessa systeemin tai ohjelmiston määrittely muunnetaan tekniseksi toteutuksen kuvaukseksi. Suunnittelu jaetaan usein arkkitehtuurisuunnitteluun ja moduulisuunnitteluun. *Arkkitehtuurisuunnittelun* (architecture design) tarkoituksena on jakaa järjestelmä moduuleihin ja määritellä moduulien rajapinnat. Moduulisuunnittelussa suunnitellaan sisäinen rakenne kullekin moduulille. [HaM97, s.61]

Projektin toteutusvaiheessa ohjelmoijat suorittavat ohjelmoinnin ohella ohjelmamoduulien yksikkö- ja integrointitestausta paritestausta hyväksikäyttäen. Toteutusvaiheen tuloksina saadaan ohjelmakoodi ja ohjelmistokuvaus, joka selventää ohjelman rakenteen. Lisäksi saadaan yksikkötestauksen tulokset tulevien testausvaiheiden perustaksi. Integraatiotestauksessa testataan, miten yksikkötestauksessa itsenäisinä palasina testatut ohjelman osat toimivat yhdessä. Projektin viimeisessä testausvaiheessa testaajat suorittavat järjestelmä- ja hyväksymistestauksen testitapauksia.

Ohjelmaa testatessa testaaja käy testauslistat järjestelmällisesti läpi ja merkitsee niihin todelliset tulokset, mahdolliset virheet tai muutospyynnöt. Testauslistat on helppo muodostaa testitapausten kuvauksista lisäämällä uusia sarakkeita (todellinen tulos, virheen kuvaus ja korjaaja), mikäli kuvaukset on tallennettu taulukkomuodossa. Testausraportissa käsitellään tarkemmin testauksessa ilmenneet virheet ja analysoidaan testauksen kattavuutta, mikäli sitä varten ei tehdä erillistä yhteenvetodokumenttia testauksesta.

2.5 Testitapausten lähteet

Mistä testitapausten etsimisen voi aloittaa? Perinteisesti testitapausten valintastrategiat on jaettu kahteen lähestymistapaan: *koodipohjaiseen* ja *määrittelypohjaiseen* testitapausten valintaan. Määrittelypohjaisessa testauksessa on nykyisin keskitytty paljon testien johtamiseen UML-mallien pohjalta [Ber03]. Testitapausten lähteinä voivat toimia myös tekijät, joita on vaikea luokitella pelkästään toiseen ryhmään (riskipohjainen testaus, virheraportit asiakkaalta jne.). Kuvassa 6 esitetään muutamia tyypillisiä kohteita, joista testitapauksia kannattaa ”metsästää” testausta varten. Eri kohteista löydetään testitapauksia eri testausvaiheisiin: ohjelmakoodin toisto- ja silmukkarakenteet tuottavat testitapauksia yksikkötestaukseen, ja laitteiston ja muiden ohjelmien kanssa tehtävät testit kuuluvat järjestelmätestaukseen.



Kuva 6: Testitapausten metsästys

Mustalaatikkotestauksessa testitapausten tärkeimpiä etsintäkohteita ovat ohjelman hyväksymiskriteerit, rakentamista varten laaditut määrittelydokumentit ja UML-kaaviot. Selkeät ja riittävän tarkalle tasolle tehdyt kaaviot ovat arvokasta materiaalia testauksen pohjaksi, koska kaavioista näkee helpommin, miten ohjelman suoritus etenee. Osan testitapauksista saa suoraan poimittua kaavioista.

Asiakkaalta tulee muutospyyntöjä ja virheilmoituksia, jotka täytyy tarkastaa ja arvioida. Osa virheistä tai korjauspyynnöistä saattaa johtua siitä, että ohjelmaa on käytetty väärin. Siitä huolimatta jokainen muutospyyntö käsitellään ja käsittelyn tuloksesta ilmoitetaan asiakkaalle. Hyvällä testaajalla on kokemusta monenlaisista sovelluksista, arkkitehtuureista, laiteympäristöistä, järjestelmien suorituskyvystä ja monesta muusta osa-alueesta. Oman kokemuksen perusteella testaaja voi keksiä testitapauksia, jotka jäisivät muilta huomioimatta. Kokeneen testaajan tulisi jakaa tieto-taitoaan nuoremmille työntekijöille esimerkiksi yrityksen sisäisessä koulutuksessa.

Riskipohjaisessa testauksessa (risk-based testing) testien suunnittelu lähtee liikkeelle ohjelmistokehitystä koskevien riskien kartoittamisella. Ohjelmiston riskienhallinnassa on tavoitteena tunnistaa ja analysoida ohjelmistoon tai tuotteeseen liittyvät riskit. Jokaisesta löydetystä riskistä arvioidaan sen esiintymisen todennäköisyys ja riskin vaikutuksen merkitys toteutuessa. Riskipohjaisessa testauksessa voidaan tunnistaa kolme erilaista ohjelmistoriskityyppiä: *projektiriski*, *prosessiriski* ja *tuoteriski*. [Rus03]

Projektiriskejä ovat resurssien rajoitteet, projektin ulkoiset rajapinnat kuten suhteet ulkoisiin toimittajiin sekä sopimusrajoitteet. Prosessiriskeihin kuuluvat poikkeamat projektin suunnittelussa ja arvioinnissa, puutteet työntekijöiden määrässä tai tietotaidossa sekä laadunvarmistuksen ja versiohallinnan puuttuminen. Tuoteriskejä ovat tuotteen kannalta kriittiset ominaisuudet, monimutkaisuus, suunnittelun ja koodin laatu, ei-toiminnalliset ominaisuudet ja vaatimusten jatkuva muuttuminen [Rus03]. Osittain tuoteriskeihin kuuluvaksi neljänneksi riskityypiksi voidaan määritellä *ympäristöriskit,* joita ovat mm. tuotteen sopeutuminen eri käyttöjärjestelmiin, erilaisiin laitealustoihin ja muuhun sovellusympäristöön.

2.6 Virheiden lähteet

Testitapauksissa kuten koko testauksessa keskitytään piilossa olevien virheiden etsimiseen. Suomenkieliselle käsitteelle *virhe* voidaan löytää englannin kielestä erilaisia nimityksiä [Paa00, s.6]:

- *Error* on ihmisen (esim. ohjelmistokehittäjän) tekemä virhe ohjelmistokehityksen aikana, mikä voi olla käyttäjävaatimusten väärä tulkinta tai pelkkä kirjoitusvirhe. *Error* aiheuttaa *Defect*-tyyppisen virheen ohjelmakoodiin.
- *Defect, Fault, Bug* on ero virheellisen ja virheettömän ohjelmaversion välillä (esim. koodausvirhe). Ohjelman suorituksen aikana *Defect* voi aiheuttaa *Failure*-tyyppisen virheen.
- *Failure* on ulkoisesti huomattavissa oleva poikkeama ohjelman toiminnan ja määrittelyn välillä (esim. väärä tulos laskutoimituksesta).

Edellä mainitut määritelmät voidaan tiivistää seuraavasti: Ohjelmistokehittäjät tekevät kehityksen aikaisia virheitä, jotka johtavat rakenne- ja koodivirheisiin, jotka puolestaan voivat ohjelman ajon aikana aiheuttaa ulkoisesti havaittavia poikkeamia. Missä virheitä esiintyy sitten useimmin? Tieto virheiden esiintymistiheydestä on arvokasta,

koska testaukseen käytettävät resurssit ovat rajallisia. Testaus kannattaa keskittää sinne, missä virheitä on eniten. Tilastojen mukaan ohjelmistovirheitä esiintyy eniten ohjelmakoodin suorituslogiikassa kuten ehtolausekkeissa, toisto- ja silmukkarakenteissa (Taulukko 1).

Taulukko 1: Tilastotietoa virheen lähteistä

Virheen lähde	**%-osuus**
Vaatimukset (määritelty väärin tai epätäydellisesti)	8,1
Toiminnot / toiminnallisuus (vaatimukset oletettu oikeaksi, mutta toteutus tehty väärin)	16,2
Suorituslogiikka (ehto- ja silmukkarakenteet)	25,2
Data (tiedon koodaus, määrittely, rakenne ja käyttö)	22,4
Toteutus (ristiriidat koodausstandardeihin, kirjoitusvirheet, väärät kommentit)	9,9
Integraatio (komponenttien väliset rajapinnat)	9
Arkkitehtuuri (käyttöjärjestelmäkutsut, virheistä toipuminen, suorituskyky)	1,7
Virheellinen testaus (virheet testien suorituksessa, tulkintavirheet, testidatan virheet)	2,8
Muut	4,7

Runsaasti virheitä aiheutuu myös tiedon määrittelystä ja käytöstä (tietotyypit) sekä yleensä virheellisistä toteutuksista, joissa vaatimuksia on tulkittu väärin. Taulukon 1 tilastotiedot virheiden lähteistä ovat peräisin Paakin esittämästä yhteenvedosta erään testaustutkimuksen tuloksista. Alkuperäinen tutkimus ja sen täydellinen kuvaus on löydettävissä [Bei90]. Ohjelmistovirheiden keskinäiset osuudet muuttuvat uusien teknologioiden käyttöönoton myötä. Vuonna 1999 tehdyn tutkimuksen mukaan virhejakauma ohjelmakoodille näyttää seuraavalta: Rajapinnat 29 %, data 26 %, suorituslogiikka 23 %, tiedon alustus 14 %, ja laskutoimitukset 8 % [Car99]. Rajapintavirheiden osuus on selvästi lisääntymässä komponenttipohjaisen sovellustuotannon omaksumisen myötä.

3 Testausprosessin hallinta

Testausprosessin hallinta (test process management) on kaikkien testaukseen kuuluvien toimenpiteiden ja resurssien koordinointia. Hallintaan kuuluu testauksen valmistelua, suunnittelua, suorittamista, testitulosten arviointia ja testauspäätösten tekemistä saatujen tulosten perusteella. Ohjelmistokehitystä varten tulisi laatia mittaristo, joka auttaa tehokkaampaan testausprosessin suunnitteluun ja hallintaan sekä samalla nopeuttaa tuotteen hyväksymistä markkinoille. Mittarit voidaan jakaa aikapohjaisiin ja testauskattavuus-mittareihin.

Aikapohjaisia (time-based) mittareita ovat testauksen suunnitteluaika, testien suoritusaika ja virheen löytämiseen ja korjaamiseen käytetty aika. *Kattavuusmittareita* (coverage) ovat koodikattavuus ja vaatimuskattavuus. Mittariston suunnittelussa on havaittu esiintyvän avainkysymyksiä, joihin pitää löytää vastaus, jotta tiedetään testauksen sujuvan asianmukaisesti. Mittaristo auttaa löytämään vastauksia muun muassa seuraaviin kysymyksiin [MaK00]:

- Kuinka monta testiä tarvitaan?
- Mitä resursseja tarvitaan vaadittujen testien kehittämiseksi?
- Kuinka paljon aikaa vaaditaan testien suorittamiseen?
- Kuinka paljon aikaa vaaditaan virheiden löytämiseen, korjaamiseen ja sen varmistamiseen, että virheet on todella korjattu?
- Kuinka paljon aikaa on kulutettu koko tuotteen testaamiseen?
- Kuinka suuri osa koodista on käyty läpi?
- Onko kaikki tuotteen ominaisuudet testattu?
- Kuinka monta virhettä on löydetty kussakin ohjelmistokehitysvaiheessa tai ohjelmistoa tai moduulia kohden?

3.1 Testauksen dokumentointi

Testauksen dokumentoinnissa on suositeltavaa käyttää hyväksi seuraavia dokumentteja [IEEE829]:

- *Testaussuunnitelma* (testausstrategia, resurssit ja aikataulu),
- *Testitapausten määrittely* (testitapausten syötteet, oletetut tulokset ja testausympäristö),

- *Testauslistat* (todelliset tulokset testien suorittamisesta ja kuvaukset virhetilanteista) ja
- *Testausraportti* (yhteenveto testauksen tuloksista).

Testaussuunnitelman tulisi sisältää ainakin seuraavat asiat:

- Testaussuunnitelman tunniste.
- Johdanto: Esitellään testauksen kohteet tiivistettynä ja viitteet testauksen kannalta olennaisiin dokumentteihin kuten projektisuunnitelmaan ja standardeihin.
- Testattavat kohteet: Tunnistetaan kaikki testattavat kohteet versiotasolla, tarjotaan viitteet kohteitten kannalta olennaisiin dokumentteihin kuten määrittely- ja suunnitteludokumentteihin ja käyttöohjeeseen ja luetteloidaan myös ei-testattavat kohteet.
- Testattavat ominaisuudet: Tunnistetaan kaikki ohjelmiston ominaisuudet ja niiden yhdistelmät testausta varten.
- Ei-testattavat ominaisuudet: Tunnistetaan ominaisuudet, jotka jätetään testauksen ulkopuolelle ja kerrotaan syyt pois jättämiselle.
- Lähestymistapa: Kuvataan yleinen lähestymistapa testaukselle kuten testauksen toimenpiteet ja testaustekniikat, ei-toiminnallisten vaatimusten testaaminen ja mahdolliset työkalut. Määritellään testauksen lopetuskriteerit esim. virhetiheyden tai koodikattavuuden avulla.
- Läpäisy/hylkäyskriteerit: Määritellään kriteerit, joilla päätetään onko testi läpäisty vai hylätty.
- Keskeytyskriteerit: Määritellään kriteerit, joilla testaus keskeytetään esim. työpäivän lopussa tai laitteistovirheen takia sekä keskeytyksen jälkeen toistettavat testit.
- Tuotettavat dokumentit: Tunnistetaan kaikki tuotettavat dokumentit testauksen suunnittelusta, testitapausten kuvauksista, raportoinnista jne.
- Testaustehtävät: Tunnistetaan testauksen valmisteluun ja suorittamiseen tarvittavat tehtävät esim. laatimalla testausprosessin kuvaus.
- Testausympäristö: Määritellään vaadittavat ja suositeltavat piirteet testausympäristölle: laitteistot, ohjelmistot, tietoliikenneyhteydet, ohjelmakirjastot, työkalut, käyttöoikeustasot jne.
- Vastuut: Tunnistetaan henkilöryhmät, jotka ovat vastuussa testauksen hallinnasta, suunnittelusta, suorittamisesta, valvonnasta ja testiympäristön valmistelusta.

- Työntekijät ja koulutustarve: Tunnistetaan testaajien osaamistaso ja koulutustarpeet esimerkiksi työkalujen käyttöön.
- Aikataulu: Määritellään testauksen vaiheiden aikajako kalenteriin sekä arvioidaan jokaiseen tehtävään kuluva aika ja tehtävien väliset ajalliset riippuvaisuudet toisistaan.
- Riskit: Tunnistetaan testausta koskevat riskit kuten osaavien työntekijöiden puute, tekniset ongelmat ja määritellään ennaltaehkäisevät toimenpiteet sekä toiminta riskin toteutuessa.
- Hyväksyminen: Määritellään henkilö tai henkilöt, jotka hyväksyvät testaussuunnitelman.

Testitapausten määrittely -dokumentti voi alkaa *tunnistetiedoilla* (dokumentin laatijat, projektin nimi, testaustaso, testaustekniikka ym.), joissa selvitetään dokumentin lukijalle, minkä ohjelmiston testaukseen kyseiset testitapaukset liittyvät (Liite 1, s. 12). Testitapausten määrittely -dokumentissa laaditaan testitapausten kuvaukset ja se sisältää seuraavat asiat:

- Määrittelydokumentin tunniste,
- Testattavat kohteet,
- Syötteet,
- Oletetut tulokset,
- Ympäristö (laitteisto- ja ohjelmistokokoonpano),
- Erikoisvaatimukset testin suorittamiselle (tunnetaan myös esiehtoina) ja
- Testitapausten väliset riippuvaisuudet.

PlugIT-Teho -tutkimuksessa testitapausten kuvaamiseen käytettiin Taulukossa 2 esitettyä tapaa.

Taulukko 2: Testitapausten kuvaukset

Id	Esiehdot	Syöte	Oletettu tulos
1	**Potilastiedot – lomake avoinna**	Nimi = Pertti Hetu = 041076-996T osoite = tyhjä. Paina Tallenna-painiketta	Potilaan tiedot tallennettu oikein
2	Potilastiedot – lomake avoinna	Nimi = Pertti Hetu = tyhjä Osoite = tyhjä. Paina Tallenna-painiketta	Virheilmoitus

Tunnistenumerolla (Id) erotetaan testitapaukset toisistaan. Tunnistenumero voi koostua monesta eri tasosta 1.1, 1.1.1, 1.1.1.1 riippuen testitapausten ryhmittelystä. Testitapausten tunnistenumerot voivat esimerkiksi vastata käyttötapausten järjestysnumeroita tai testattavien komponenttien versionumeroita. Tunnisteen kirjaaminen auttaa testitapausten hallinnassa ja nopeuttaa projektiosapuolten välistä keskustelua virheen paikallistamisessa. *Esiehdot* ovat ehtoja, joiden pitää olla voimassa, ennen kuin testitapauksen voi suorittaa. Esiehtoina voi olla esimerkiksi seuraavat tilanteet:

- Tietty sovelluksen osa täytyy olla avoinna (nimetään sovelluksen osa tai kenttä, jossa testitapauksen suoritus tapahtuu).
- Tietokantaan on tallennettu etukäteen tietoa (esim. tietokantaan on tallennettu tietyn nimisiä potilaita).
- Testaajalta vaaditaan riittävät käyttöoikeudet (esim. pääkäyttäjä-oikeudet).

Syötteet ovat komentoja, joita käyttäjä antaa järjestelmälle, kuten komentopainikkeet, pikapainikkeet, näppäinvalinnat, valikkovalinnat ja syötteet teksti- ja numerokenttiin. Syötteiden kohdalla määritellään riittävän tarkasti, millainen syötteen tulee olla. Testauksen nopeuttamiseksi ja helpottamiseksi kannattaa testaajalle antaa esimerkki syötteestä (”**Syöte** *nimi = Pertti”)*.

Oletettu tulos on järjestelmän antama vaste sille annettuun syötteeseen. Oletettu tulos voi olla virheilmoitus, tietojen onnistunut tallennus, ohjelman tai sen osan sulkeutuminen tai avautuminen ym. Mikäli testaaja ja testitapausten laatija ovat eri henkilöitä (kuten olisi suositeltavaa), testaajalle tulee antaa riittävän yksiselitteiset ja selkeät ohjeet testauksen suorittamiseksi. Jos testitapausten kuvauksissa viitataan ulkoisiin dokumentteihin (määrittelydokumentit), kannattaa aina mainita myös dokumentin versio ja sivunumero. Testaaja ei voi tuhlata aikaa dokumenttien ja niiden eri versioiden etsimiseen.

Virheraportti (testauslista) sisältää seuraavat asiat:

- Virheraportin tunniste,
- Yhteenveto virheestä (kuvaa virheen sisältävän testauskohteen versiotasolla ja viitteet dokumentteihin),
- Virheenkuvaus (kuvaa virhetapahtuman),
- Virheen tunniste,

- Esiehdot,
- Syötteet,
- Oletettu tulos,
- Todellinen tulos,
- Päivä ja aika,
- Ympäristö,
- Toistettavuus (voidaanko virhetilanne toistaa) ja
- Testaaja (nimi tai nimikirjaimet).

Esimerkki PlugIT-Teho -tutkimuksessa käytetystä virheraportista on liitteessä 1 sivulla 12. Edellisestä luettelosta poiketen se kertoo myös, onko testi hyväksytty vai hylätty (Pass/Fail) ja miten merkittävä on *virheen vakavuus* (severity of defect). Virheen vakavuus on luokiteltu neljään kategoriaan: 1) Kriittinen, 2) Merkittävä, 3) Keskimääräinen ja 4) Vähäinen. Kunkin virheen kohdalla kerrotaan lisäksi virheen korjaaja ja korjauspäivämäärä.

Testauksen yhteenvetoraportin sisältö on seuraavanlainen:
- Testauksen yhteenvetoraportin tunniste.
- Yhteenveto: Testauskohteiden yleisarviointi, testiympäristö, viitteet olennaisiin dokumentteihin.
- Poikkeamat: Raportoidaan ohjelman poikkeamista määrittelydokumentteihin nähden tai poikkeamista suunnitellusta testausprosessista.
- *Testauksen kattavuus*: Arvioidaan koko testausprosessin perusteellisuutta testaussuunnitelmassa esitettyihin kriteereihin nähden, tunnistetaan piirteet, joita ei testattu riittävästi ja syyt, miksi ei testattu riittävästi.
- Tulosten yhteenveto: Arvioidaan testauksen onnistumista, kuten kattavuutta.
- Arviointi: Annetaan arvio jokaisesta testauskohteesta ja kohteen rajoitteista perustuen testituloksiin ja kohdekohtaisiin hyväksymis-/hylkäyskriteereihin.
- Yhteenveto testaustoiminnoista: Tiivistelmä testaustoiminnoista, resurssien käytöstä ym.

Edellä mainitut testausdokumentit ja niiden sisältö ovat esimerkkejä IEEE-standardin mukaisista dokumenteista [IEEE829]. Käytännössä testauksen dokumentointi vaihtelee huomattavasti eri yritysten ja organisaatioiden välillä sekä toteutettavien projekti-

en koon mukaan. Pienissä projekteissa voidaan yhdistää testitapausten kuvaukset testaussuunnitelmaan liitteiksi ja käyttää testitapausten kuvausdokumenttia virheraportin pohjana.

3.2 Testiympäristö

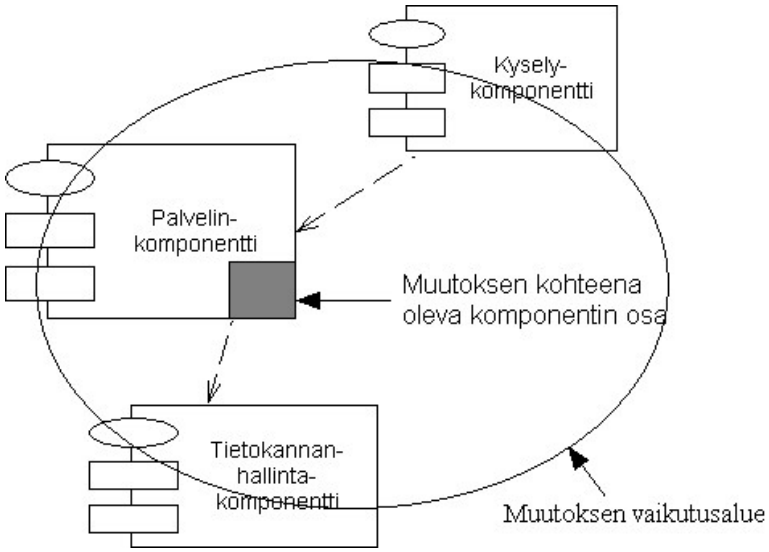
3.2.1 Yleistä testiympäristöstä

Testiympäristön rakentaminen muodostaa huomattavan osan ohjelmistokehitystyön kustannuksista. Terveydenhuollon tietojärjestelmät vaativat yhteyksiä kymmeniin tai jopa satoihin muihin järjestelmiin. Toimittajalla on testauksen aikana varsin usein käytössä erilainen laiteympäristö kuin asiakkaalla, mikä voi johtaa ennalta arvaamattomiin *yhteentoimivuus* (interoperability)- ja *asennusongelmiin* (installation) uuden tuotteen ja asiakkaalla jo olevien ohjelmistotuotteiden tai järjestelmäinfrastruktuurin välillä. Testausympäristön suunnittelussa tulisi kiinnittää huomioita seuraaviin asioihin:

- Testauksessa käytetyt ohjelmistokomponenttien versiot ja komponenteista kootut konfiguraatiot tulee dokumentoida testausdokumentteihin. Näin voidaan vertailla esimerkiksi eri kokoonpanojen suorituskykyä ja muistinkäyttöä.
- Testattavan ohjelman ja sen käyttämien oheislaitteiden yhteistoiminta tulee testata. Esimerkiksi testataan, sujuuko tulostimen valinta *Tulosta*-toiminnossa helposti tai näkyvätkö paikalliset tulostimet ylipäänsä ohjelmassa.
- Testaajalle (ei kaikille) tulisi mahdollistaa pääsy suoraan tietokannan tauluihin, jotta voidaan nähdä suorittaako ohjelma päivitykset oikein tietokantaan saakka (esim. poistaako ohjelma potilaan tiedot oikeasti pois tietokannasta). Pääsy tietokantaan edellyttää yleensä käyttäjätunnusta ja salasanaa sekä tietokannan nimen tuntemista, mikäli tietokantoja on useampia. Edelliset tiedot tulee antaa testaajalle valmiina.
- Testausympäristön rakentajan tulee tietää ajoissa, mitä käyttöjärjestelmiä ja apuohjelmia testaus vaatii. Käytännössä todettu tosiasia on, että varsinaisessa testausvaiheessa on liian myöhäistä alkaa asentaa useita eri käyttöjärjestelmiä samalle koneelle.
- Tietoliikenneyhteyksien toiminta, kuten viestin välitys hajautettujen ohjelmien välillä, tulee varmistaa (Esim. sähköpostin lähettäminen testattavasta ohjelmasta).

- Testaustyökalujen soveltuvuus testauksen eri vaiheisiin tulee arvioida tarkasti.

Testaustyökalujen käyttöönotto ja koulutus pitää tehdä ennen projektia, jotta projektin resurssit (aika ja raha) eivät kulu uuden, monimutkaisen työkalun opettelemiseen. Työkaluja voidaan käyttää muun muassa testauksen hallintaan testausta suunniteltaessa, suorituskyvyn ja muistitehokkuuden mittaamiseen yksikkötestauksessa tai testiskriptien ajamiseen regressiotestauksessa. *Regressiotestaukseksi* kutsutaan testausta, jossa keskitytään testaamaan ohjelmaa tehtyjen muutosten jälkeen [KuH98]. Sen tavoitteena on varmistaa, että muuttunut ohjelmisto toimii edelleen oikein. Toisinaan muutokset vaikuttavat vain pariin komponenttiin (pieni coupling), mutta joskus taas moniin komponentteihin (Kuva 7).



Kuva 7: Muutoksen vaikutusalue regressiotestauksessa

Muutoksen vaikutus riippuu komponenttien välisistä riippuvuussuhteista, muutoksen tyypistä ja koosta sekä muutetuista komponenteista. Olennaista regressiotestauksessa on tunnistaa muutokset ja niiden vaikutukset siten, että testaus voidaan keskittää vain muutettuihin ohjelmiin tai muutoksen vaikutuksen alla oleviin ohjelmisto-osiin. Ellei muutosten vaikutuksia pystytä kohdentamaan tietyille komponenteille, joudutaan testaamaan uudelleen kaikki ohjelman osat, mikä vie aikaa ja tulee kalliiksi. Regressiotestauksessa uudelleenkäytettäväksi rakennetut testitapaukset ja testiskriptit säästävät aikaa ja rahaa, koska niiden avulla testaajan ei tarvitse enää suorittaa manuaalisesti samoja testitapauksia uudelleen.

3.2.2 Testitietovarasto

Ohjelmistoa varten rakennetut testitapaukset järjestetään tai luokitellaan mahdollisesti jonkun työkaluohjelmiston avulla helposti saatavilla olevaan yhteiseen kirjastoon: *testitietovarastoon* tai *testikirjastoon* (testware library). Tämä tuo pitkällä tähtäimellä huomattavia säästöjä testitapausten suunnitteluun ja suorittamiseen. Kerran tehtyä työtä ei tarvitse tehdä joka projektissa uudelleen. Suuri osa testitapauksista nykyisissä ohjelmistoprojekteissa on edelleen manuaalisessa muodossa Word/Excel-taulukoissa, joten niitä ei voi ajaa skripteinä kuten automaattisessa testauksessa.

Testauksen automatisointi tarkoittaa manuaalisesti tuotettujen testitapausten automatisointia ajettavaan (tietokoneen ymmärtämään) muotoon. Automatisoinnin edellytyksenä on testitapausten tarkka dokumentointi, jotta voidaan huomata, mitä testitapauksia ylipäänsä kannattaa automatisoida. Testitapausten automatisointi saattaa olla aluksi hyvin aikaa vievää ja kallista verrattuna kertaluonteiseen, manuaaliseen testitapausten suorittamiseen. Automatisointi tuottaa hyötyjä silloin, kun monia samoja testitapauksia joudutaan ajamaan useita kertoja peräkkäin. *Automaattisen testauksen elinkaari* (Automated Testing Lifecycle Methodology) sisältää 1) päätöksen automatisoida testejä, 2) testityökalujen hankinnan, 3) automatisoinnin liittämisen kehitysympäristöön, 4) testauksen suunnittelun ja toteutuksen, 5) testien suorituksen ja hallinnan ja 6) toteutetun automatisoinnin arvion [TeK01].

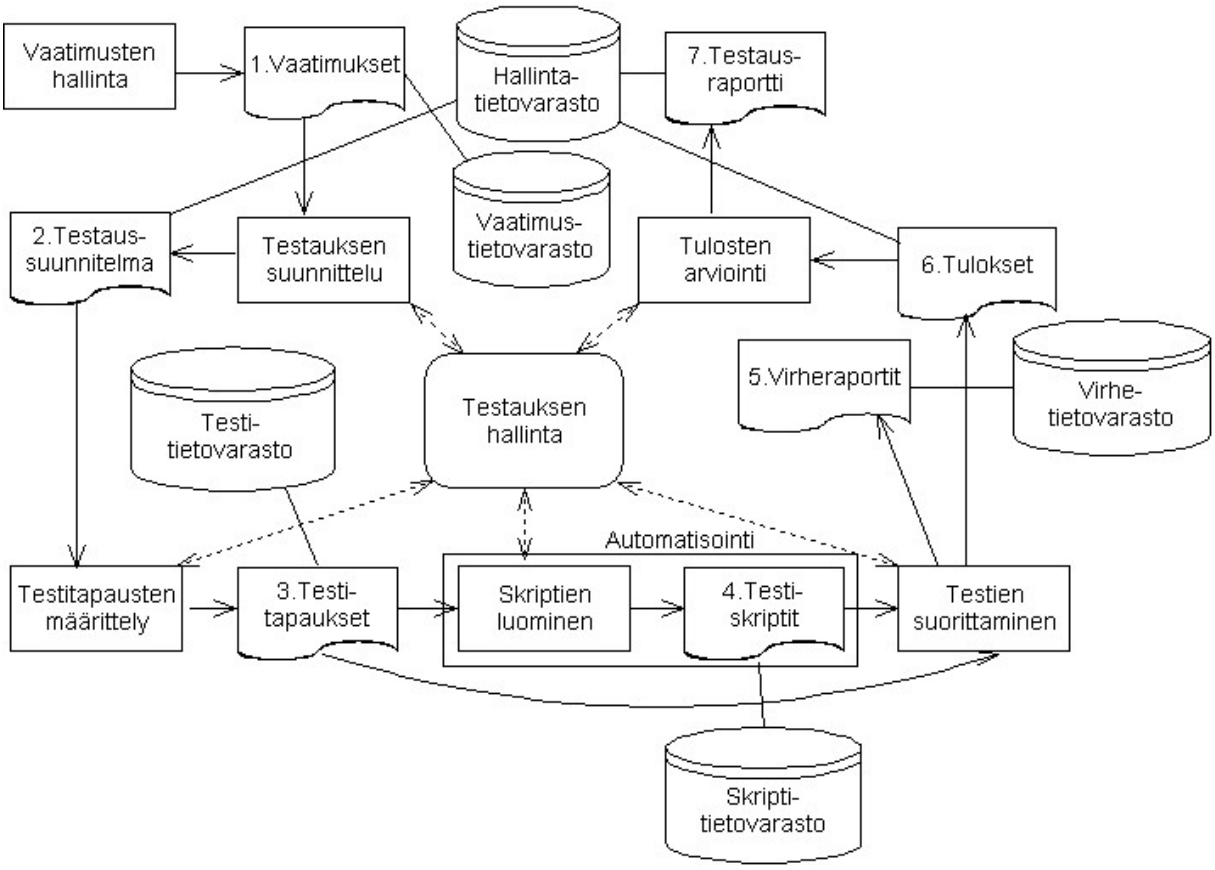
Tärkeintä olisi saada myös manuaaliset testitapaukset ryhmiteltyä siten, että niitä voidaan käyttää hyväksi tulevissa projekteissa. Kaikista testitapauksista ei kannata ryhtyä tekemään automatisoituja ja uudelleenkäytettäviä, vaan testitapausten suunnittelussa tulisi aina huomioida automatisoinnin kustannus-hyöty -suhde.

Jotta kirjasto olisi käyttökelpoinen myös muille, yhteen liittyvät testitapaukset kasataan ryhmiksi, jotka nimetään yrityksessä tai organisaatiossa yhteisesti sovittavalla tavalla. Testikirjasto voi sisältää monta ryhmää ja yksittäinen testitapaus voi kuulua yhteen tai useampaan ryhmään. Testikirjaston pitäisi sisältää: indeksisivu (aloitussivu), kuvaus kirjaston rakenteesta ja nimeämiskäytännöistä. Jokaiselle testitapausryhmälle laaditaan kuvaus ja ryhmittelykriteerit, testitapausten nimet ja komento tai käsky, jolla testitapauksia voidaan ajaa. Jokaiseen testitapaukseen liitetään kuvaus ja suo-

rituksen kuvaus sekä kaikki testitapaukseen liittyvät tiedostot esimerkiksi määrittely-dokumentit (kts. Luku 5.3). [Kit95]

3.3 Testaustyökalut

Ohjelmistot on kirjoitettu ohjelmointikielillä, joissa on hyvin määritelty syntaksi ja semantiikka. Tämän vuoksi ohjelmiston testaamisen ei tarvitse tapahtua käsityönä, vaan testauksessa voidaan käyttää apuna erilaisia työkaluja esimerkiksi testauksen hallintaan, testitapausten valintaan, testausympäristön kehittämiseen, tulosten tarkistamiseen ja testauskattavuuden analysointiin. Työkaluja on tarjolla testausprosessin kaikkiin vaiheisiin [Paa00]. Esimerkkejä testaustyökaluista löytyy luvusta 5.



Kuva 8: Testausprosessi

Testausprosessi (Kuva 8) alkaa vaatimusten hallinnasta. Vaatimusten hallintaan tarkoitetuilla työkaluilla luodaan ohjelmistoon liittyviä vaatimuksia, annetaan vaatimuksille ominaisuuksia, järjestetään vaatimukset tärkeysjärjestykseen ja kuvataan myös keskinäiset riippuvuussuhteet. Vaatimusten hallinta -vaiheesta saadaan tuotoksena Vaatimukset-dokumentti (1) ja vaatimukset tallennetaan Vaatimus-tietovarastoon.

Testauksen suunnittelussa määritellään testauksen kohde, testausstrategia, resurssit ja hyväksymis- ja hylkäyskriteerit. Testauksen suunnitteluun käytetään pitkälti samoja työkaluja kuin projektisuunnittelussa eli tekstinkäsittely- ja projektinhallintaohjelmia ja tuotoksena tästä vaiheesta saadaan testaussuunnitelma (2).

Testitapausten määrittelyssä kuvataan ohjelmistolle tehtävät testitapaukset (3) ja tallennetaan Testi-tietovarastoon. Tarvittaessa testitapauksista luodaan automaattiset skriptit eli testiskriptit (4). Testiskriptit sijaitsevat Skripti-tietovarastossa. Toiminnallisessa testauksessa on mahdollista automatisoida käyttäjän suorittamia toimintoja käyttöliittymässä kuten tekstin syöttöä, hiiren painikkeiden painalluksia ja valikkovalintoja. *Skriptien nauhoitus –työkaluilla* käyttäjän suorittamat toiminnot nauhoitetaan testiskripteiksi, jotka ovat tietokoneen ymmärtämässä muodossa olevia testitapauksen suoritusohjeita. Kun skripti ajetaan uudelleen, ohjelma toistaa täsmälleen samat toiminnot kuin käyttäjä.

Testien suorittamisessa suoritetaan testitapaukset joko manuaalisesti tai automaattisesti. Testin virheellinen tulos merkitään virheraporttiin tai testausohjelma tallentaa testituloksen tietokantaan. Löydetyt virheet tallennetaan Virhe-tietovarastoon. Testauksen tuloksena voi olla vain seuraavat vaihtoehdot: testi on mennyt *läpi* (passed) tai testi on *hylätty* (failed). Testauksen tuloksia (6) arvioidaan testausraportissa (7). Kaikki testausdokumentit (testaussuunnitelma, testausraportti, testitulokset), jotka liittyvät testauksen hallintaan, voidaan tallentaa Hallinta-tietovarastoon.

Testauksen hallinta liittyy koko testausprosessin ohjaamiseen. Testauksen hallintaan tarkoitetut työkalut sisältävät toimintoja testitapausten määrittelyyn eli luomiseen ja järjestämiseen testitapauskansioihin testitietovarastoon. Hallintatyökaluilla voidaan ajaa raportteja testauksen suorittamisesta, kuten testien kattavuudesta sekä virheellisten ja virheettömien testitapausten suhteesta eli niitä voidaan käyttää testaustulosten arvioinnissa. Testauksen hallinta -työkalu on parhaimmillaan mahdollista yhdistää vaatimus- tai analyysi- ja suunnitteluvaiheen työkaluihin siten, että asiakasvaatimusten ja testitapausten välille tulee yhteys, jolloin jäljitettävyyden tukeminen on mahdollista.

Mitä hyötyä testiskriptien nauhoituksesta on? Ajatellaan, että testauksen kohteena on webpohjainen sovellus, jonka toiminnallisuutta tulee testata sekä Internet Explorer- että Netscape Navigator -selaimilla. Ilman testaustyökalua testaaja suorittaa sovellukselle ensin esimerkiksi 400 testitapausta Explorer-selaimella, jonka jälkeen hänen on toistettava samat 400 testitapausta Navigator-selaimella. Testaustyökalulla testaaja voi nauhoittaa molempia selaimia tukevia testiskriptejä, joita voidaan ajaa automaattisesti aina, kun järjestelmään tehdään muutoksia ja tarvitsee testata, toimiiko järjestelmä samoin, kuin ennen muutoksia. Eräissä työkaluissa ohjelman käyttöliittymän objekteista muodostetaan kartta (GUI map), joka mahdollistaa *tarkistuspisteiden* (verification points) luomisen testiskripteihin. Tarkistuspisteessä katsotaan, suorittaako ohjelma jonkun tietyn toiminnon (esim. näyttää www-sivulla kuvan, linkin tai muun objektin). Mikäli ohjelma ei suorita toimintoa, testausohjelma antaa virheilmoituksen.

Suorituskykytestauksessa mitataan, miten paljon aikaa ohjelmiston eri osien suorittaminen vaatii. Java-työkaluilla on mahdollista nähdä, paljonko kukin luokka ja metodi kuluttaa aikaa ohjelman suorituksen aikana. *Muistin profilointi –työkaluilla* pystytään havaitsemaan sovelluskoodin muistivuotoja ja testaamaan paljonko muistia eri ohjelman osat vaativat. Järjestelmän suorituskyvyn analyysissä tulee vastata seuraaviin kysymyksiin [Bau03]:

- Kuinka kalliiksi huono suorituskyky tulee?
- Miten nopea järjestelmän pitäisi olla?
- Miten nopea järjestelmä on nykyisellä tietokuormalla?
- Miten nopea järjestelmän pitää olla tulevaisuudessa?
- Onko nykyinen tuoteversio nopeampi kuin edellinen (esim. vasteajaltaan)?
- Kuinka järjestelmä käyttäytyy raskaan tietokuorman alla?
- Kuinka järjestelmän pullonkaulat tunnistetaan ja korjataan?
- Kuinka helposti järjestelmän vasteaikaa voidaan lyhentää?
- Kuinka varmistetaan ja todennetaan suorituskyky?
- Mitkä komponentit ovat liian suuria koodimäärältään?

Kuormitustestaukseen tarkoitetut työkalut simuloivat satoja tai tuhansia samanaikaisia järjestelmän käyttäjiä. Useat työkaluohjelmat mahdollistavat käyttäjän toiminnan si-

muloinnin esimerkiksi testauksessa. Testaaja voi simuloida eBusiness-sivustojen tapauksessa muun muassa loppukäyttäjän ajatteluaikaa tai modeemin nopeutta. Kuormitustestauksesta saadaan selville järjestelmän maksimikapasiteetti ja miten järjestelmä toimii suurilla käyttäjämäärillä. Ongelmana kuormitustestauksen työkaluissa on niiden korkea hinta (satoja tuhansia euroja), mikäli testattavaa järjestelmää halutaan kuormittaa tuhansilla virtuaalikäyttäjillä. *Koodikattavuus-työkaluilla* mitataan, kuinka suuri osa ohjelmakoodista suoritetaan ohjelman suorituksen aikana ja kuinka paljon jää suorittamatta.

Manuaalisten testitapausten muuttaminen ajettaviksi tiedostoiksi ei ole helppo tehtävä. Pelkkä konekielelle kääntäminen ei riitä, vaan järjestelmä on myös saatava tilaan, josta määriteltyjen testien ajaminen voidaan käynnistää. Tätä kutsutaan usein testin esiehdoksi. Vuorovaikutteisissa järjestelmissä voidaan joutua ennen varsinaisen testin suorittamista ajamaan useita testijonoja haluttuun esiehtotilaan pääsemiseksi. Tehokas keino testien käsittelyssä on järjestää testitapaukset siten, että jokainen testi jättää järjestelmän tilaan, joka toimii esiehtona seuraavalle testille. Tällöin on varmistettava kunkin tilan oikeellisuus. [Ber03]

4 UML-pohjainen testausmalli

4.1 Yleistä UML-testausmallista

Rakennuksista piirretään ennen rakentamista rakennuspiirustukset, joista nähdään huoneiden järjestys ja eri huonetilojen vaatimukset rakentamiselle kuten sähköjohdot ja vesiputket. Isoimmista rakennusprojekteista laaditaan pienoismalleja, koska niitä on taloudellisempi ja nopeampi rakentaa asiakkaan nähtäväksi kuin oikeita rakennuksia. Samasta syystä ohjelmistoja mallinnetaan. Toimittajalla ei ole mahdollisuutta rakentaa suurta sovellusta asiakkaan kokeiltavaksi lyhyessä ajassa. Sovelluksesta laaditaan *demoversio* ilman todellista toimintaa tai toimintoja mallinnetaan UML-kaavioilla. Sovellusdemon tapauksessa toimivuus esitysympäristössä tulee testata hyvissä ajoin ennen esitystä. Samoin tulee arvioida, antaako demo riittävän hyvän kuvan järjestelmästä.

Ohjelmistojen testausta varten voidaan luoda testausmalli, joka toimii viitekehyksenä tai ohjenuorana koko testausvaiheen ajan. Mallin kehittämisen edellytyksenä on ohjelman kokonaisvaltainen ymmärtäminen, mihin El-Far on luetellut seuraavia apukeinoja [Elf01]:

- Määrittele testattavat komponentit ja ominaisuudet. Mikään malli ei ole täydellinen kuvaamaan monimutkaista tai laajaa järjestelmää. Määrittely siitä, mitä mallinnetaan, on ensimmäinen askel testausmallin hallinnalle.
- Aloita järjestelmän tutkiminen. Jos kehitys on jo alkanut, ryhdy tutkimaan viimeksi rakennettuja ohjelmarakenteita ja niiden toiminnallisuutta mahdollisia virhetilanteita silmällä pitäen.
- Kerää olennaista ja käyttökelpoista dokumentaatiota järjestelmästä. Testaajien pitää saada mahdollisimman paljon tietoa järjestelmän toiminnasta. Hyödyllisiä kohteita ovat vaatimusdokumentit, käyttötapaukset, määrittelyt, suunnitteludokumentit, käyttöohjeet ja mitkä tahansa muut tuotokset, jotka tuovat lisätietoa järjestelmästä.
- Keskustele vaatimusmäärittely-, suunnittelu- ja kehitystiimien kanssa. Puhuminen asioista projektin tiimien kanssa saattaa säästää paljon aikaa ja vaivaa varsinkin, kun on kyse mallin valinnasta ja rakentamisesta. Useat yritykset rakentavat eri

tyyppisiä malleja vaatimusmäärittelyn ja suunnittelun aikana. On turha rakentaa testausmallia tyhjästä, jos voidaan käyttää hyväksi olemassa olevia suunnittelumalleja tai niiden osia.

- Tunnista järjestelmän käyttäjät. Jokainen toimija, joka tuottaa järjestelmään tietoa tai käyttää tietoja hyväksi, on merkattava muistiin. Tarkastele käyttöliittymää, käyttöjärjestelmää, tiedostoja, tietokantoja ja ohjelmarajapintoja jne. Näiden tunnistaminen on tärkeää, jotta voidaan oppia määrittelemään tapahtumien ja tapahtumasarjojen kautta tulevat odottamattomat tulokset ja mahdolliset virheet.
- Luettele jokaisen käyttäjän (toimijan) syötteet ja tulokset. Työ voidaan jakaa esimerkiksi käyttäjän, komponenttien tai ominaisuuksien mukaan. Käyttäjät voidaan tarvittaessa ryhmitellä.
- Tutki syötejoukkoa. Hyödyllisten testien laatimiseksi kannattaa jakaa syöteavaruus osiin, esim. raja-arvoihin tai ekvivalenssiluokkiin: laittomiin, sallittuihin ja ei-sallittuihin.
- Tutki syötteiden soveltuvuutta. Mallin täytyy sisältää tietoa ehdoista, joiden on oltava voimassa, ennen kuin testitapauksia voidaan suorittaa Esimerkiksi lomakkeessa olevaa painiketta ei voi painaa ellei lomake ole auki.
- Dokumentoi olosuhteet, joissa vasteet esiintyvät. Järjestelmän antama vaste on tulos jollekin käyttäjälle tai muutos sisäisessä tiedossa, jolla on tulevaisuudessa merkitystä järjestelmän käyttäytymiselle.
- Tutki syötteiden sarjoja ja niiden mallintamista. Tämä toiminto johtaa suoraan mallin muodostamiseen ja vastaa kysymyksiin: Voidaanko kaikkia syötteitä käyttää koko ajan? Missä olosuhteissa järjestelmä odottaa tai hyväksyy tietyt syötteet? Missä järjestyksessä järjestelmän vaaditaan käsittelevän syötteet? Mitkä ovat ehdot syötesarjoille tiettyjen lopputulosten tuottamiseksi?
- Hanki ymmärrystä ulkoisten tietovarastojen rakenteesta ja merkityksestä. Tämä toiminto on erityisen tärkeää, kun järjestelmä pitää tietoja suurissa tiedostoissa tai relaatiotietokannoissa. Tieto siitä, miltä tieto näyttää ja mitä se merkitsee, on tärkeää ohjelmiston riskialueiden analysoinnissa.
- Hanki ymmärrystä järjestelmän sisäisestä tiedon välityksestä ja laskennasta. Kuten edellinen kohta, tämä lisää mallintajan ymmärrystä testattavasta ohjelmistosta ja kykyä luoda virheitä paljastavaa testausdataa. Sisäinen tietovirta eri komponent-

tien välillä on olennaista tietoa korkean tason mallien rakentamiselle testattavasta järjestelmästä.

– Ylläpidä testausmallia. Kaikista järjestelmään liittyvistä asioista ei hyödytä tehdä omaa dokumenttia testausmallia varten. Parempi vaihtoehto on pitää yllä *viitteitä* olemassa oleviin dokumentteihin. Mallia kannattaa kommentoida muistiinpanoilla ja perusteluilla, mikäli järjestelmää kuvaava paperidokumentti puuttuu. Mallin olennainen tarkoitus on helpottaa ohjelman käyttäytymisen ymmärtämistä.

Rational Unified Process (RUP) on Rational Softwaren kehittämä ja ylläpitämä ohjelmistotuotantoprosessin kuvaus. Se tarjoaa kurinalaisen lähestymistavan ohjelmistokehitystä suorittavan yrityksen tai organisaation tehtävien ja vastuiden määrittelemiseen. RUP:n mukaan testausmalli on esitys siitä, mitä pitää testata ja miten testaus suoritetaan. Edellä mainittu El-Farin luettelo sisälsi yleisiä ohjeita testausmallin kehittämiselle eli mitä testattavasta kohteesta pitää tutkia (syötteet, rajapinnat, viestien välitys). RUP:n määritelmä testausmallille puolestaan kuvaa enemmän mallin sisältöä ja rakennetta [Kru00, s. 198]:

- Testitapaukset ovat joukko testausdataa, suoritusehdot ja oletetut tulokset tiettyä testausta varten. Testitapauksia voidaan johtaa käyttötapauksista, suunnitteludokumenteista tai ohjelmakoodista. Testitapausten toteuttaminen tehdään käyttämällä yhtä tai useampaa testausproseduuria.
- *Testausproseduurit* ovat joukko yksityiskohtaisia ohjeita testien valmisteluun, toimeenpanoon ja arviointiin. Testausproseduuri toteuttaa yhden tai useamman testitapauksen tai vain osan testitapauksesta, esimerkiksi vaihtoehtoisen toiminnan käyttötapauksesta.
- Testausskriptit ovat tietokoneen luettavassa muodossa olevia ohjeita, jotka automatisoivat testausproseduurien suorituksen. Testausskripti automatisoi kokonaisen tai osittaisen suorituksen yhdelle tai useammalle testausproseduurille.
- *Testiluokat* ja *–komponentit* realisoivat testaussuunnittelut. Näitä ovat esimerkiksi ajurit ja tynkämoduulit.
- *Testausyhteydet* komponenttien välillä (test collaborations) esitetään yhteistyö- tai sekvenssikaavioilla. Näiden tarkoituksena on kuvata viestien kulkeminen aikajärjestyksessä testattavien komponenttien tai kohteiden välillä.

- *Muistiinpanot* (notes) ovat tekstimuotoinen kuvaus testausmallin rajoitteista ja muusta lisätiedosta mallia koskien. Muistiinpanoja voi liittää mihin tahansa testausmallin osaan, kuten esimerkiksi UML-kaavioon.

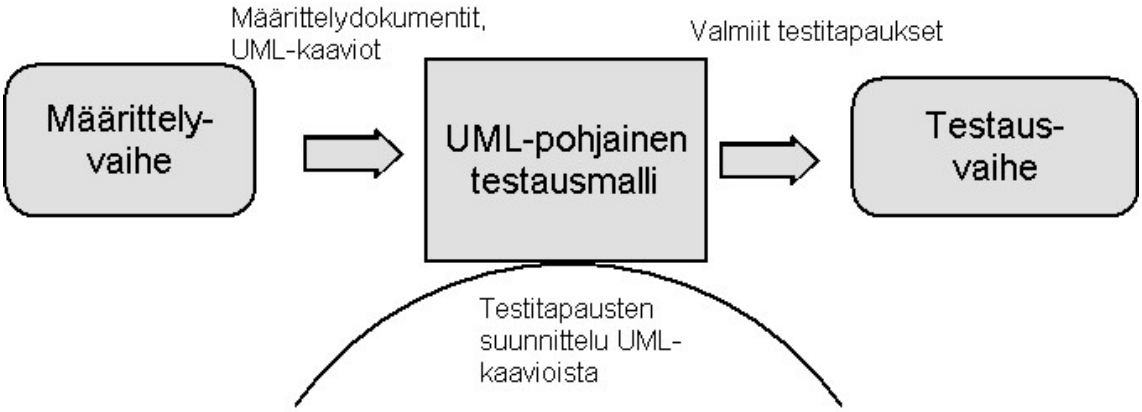
Miksi testausmalleja ylipäänsä tarvitaan? Binderin mukaan testauksen voi nähdä etsintäongelmana. Yritämme löytää miljoonien testitapausten joukosta juuri ne harvat tapaukset, jotka laukaisevat virheen ohjelmassa. Suunnittelematon testaus on ajanhukkaa, sillä testitapausten etsinnän tavoitteet ovat 1) järjestelmällisyys, 2) keskittyneisyys ja 3) automatisointi [Bin00, s.111]. Järjestelmällisyys tarkoittaa sitä, että olemme varmoja, että jokaista syöteyhdistelmää on kokeiltu. Keskittymällä testaamaan enemmän tiettyjä sovelluksen osia voimme hyödyntää tietoa, missä virheet ovat vakavimpia ja virheitä esiintyy useimmin. Testauksen priorisointi on tärkeää, koska resurssit eivät riitä kaikkien syöteyhdistelmien testaamiseen. Testauksen täytyy olla myös automatisoitua, jotta pystytään tuottamaan ja suorittamaan suuren määrän johdonmukaisia ja toistettavia testitapauksia. Automatisointia ei ole aina mahdollista eikä kannattavaa tehdä. Automatisoinnin kustannus-hyöty-suhde kannattaa arvioida huomioiden testien monimutkaisuus, lukumäärä, työmäärä ja suoritusfrekvenssi.

Mallipohjainen testaus saavuttaa kaikki edellä mainitut kolme tavoitetta: järjestelmällisyyden, keskittyneisyyden ja automatisoinnin. Testausmallin avulla testaus ei jää summittaiseksi toimenpiteeksi, vaan etenee järjestelmällisesti ja kattavasti. Malli auttaa testaajaa huomaamaan, missä osissa järjestelmän toimintaa ja rakennetta virheitä voi esiintyä. Automatisointi on helpompaa aloittaa sen jälkeen, kun testausmallissa on kuvattu tarkasti järjestelmälle annettavat syötteet ja esiehdot.

El-Farin mukaan ohjelmiston testaus vaatii jonkinlaisen mallin käyttöä ohjaamaan testausprosessin lukuisia toimintoja: testitapausten valintaa ja arviointia sekä päätöksentekoa siitä, milloin testaus voidaan lopettaa [Elf01]. Abdurazik ja Offut toteavat, että testitapausten luominen suunnitteludokumenteista auttaa testaajaa hahmottamaan itse suunnittelussa olevia puutteita ja ongelmia [AbO00]. Esimerkkeinä erilaisista testausmalleista ovat kombinaatiomallit, tilakoneet ja UML-kaaviot. *Kombinaatiomallissa* (combinational model) käytetään päätöstaulua (decision table) toteutuksen kuvaamiseksi [Bin00]. *Tilakone* (finite state machine) on abstrakti kuvaus ohjelmiston

toiminnasta, joka voidaan esittää tilojen siirtymistä kuvaavien graafien avulla. Tässä tutkielmassa keskitytään testausmallin johtamiseen UML-kaavioista.

UML-pohjainen testausmalli liittyy toiminnalliseen testaukseen (mustalaatikko), jota tehdään integrointi-, järjestelmä ja hyväksymistestauksen tasoilla ilman tietoa järjestelmän sisäisestä toteutuksesta. Testitapausten suunnittelu tehdään ilman ohjelmakoodia perustuen määrittelydokumentteihin ja määrittelyn tueksi laadittuihin UML-kaavioihin. Mallin rakenne on esitetty yksinkertaistettuna kuvassa 9.



Kuva 9: UML-pohjainen testausmalli

Määrittely- tai suunnitteluvaiheessa laadittuja ohjelmistodokumentteja analysoimalla testaaja luo mielessään itselleen kuvitteellisen mallin, jonka mukaan tulevaa ohjelmaa voidaan testata. Kun ohjelmaa ei ole vielä rakennettu, testaaja pystyy hyvin vaikuttamaan järjestelmän suunnitteluun ja ottamaan ennakoidut ongelmakohdat huomioon. UML-pohjainen testausmalli sopii siis erityisesti ohjelmistoprojekteihin, joissa rakennetaan uutta ohjelmistoa tai perinnejärjestelmän uutta osaa.

Määrittelyvaiheessa piirrettyjen UML-kaavioiden tarkoituksena on kuvata ohjelmiston toimintaa ja rakennetta eri näkökulmista. Mitä useampia kaavioita määrittelyssä on laadittu, sitä helpompaa testaajan on tarkastella järjestelmää eri näkökulmista ja saada kokonaiskuva ohjelmistosta. Testaaja voi näin ollen suunnitella myös kattavampia testitapauksia. Testitapausten johtamista UML-kaavioista on käsitelty useissa tieteellisissä tutkimuksissa, joihin viitataan tässä tutkielmassa. UML-pohjaista testausmallia on käytetty esimerkiksi integraatiotestaukseen sekvenssikaavioiden ja yhteistyökaavioiden avulla [YoC99]. Yksikkötestaus testaa ohjelmayksiköt (komponen-

tit) paikallisesti, mutta ei paljasta yksikköjen välisissä rajapinnoissa olevia virheitä. Rajapintavirheet ovat löydettävissä vain integraatiotestauksen avulla. UML-testausmalli kartoittaa testitapahtumia, joilla löydetään virheet rajapinnoista.

UML-kaaviot voidaan jakaa kahteen ryhmään: *Rakenteellisten* kaavioiden tehtävänä on mallintaa ratkaisun rakennetta, ja *toiminnalliset* kaaviot puolestaan mallintavat ratkaisun toiminnallisuutta. Rakenteellisia UML-kaavioita ovat:

- Luokkakaaviot,
- Oliokaaviot,
- Komponenttikaaviot ja
- Toimituskaaviot.

Toiminnallisten kaavioiden ryhmä sisältää:

- Käyttötapauskaaviot,
- Sekvenssikaaviot,
- Yhteistyökaaviot,
- Tilakaaviot ja
- Aktiviteettikaaviot.

4.1.1 Tutkielmassa käytetty esimerkki

Jotta UML-pohjaisen testausmallin kehittymistä ohjelmistotuotantoprosessin rinnalla olisi helpompi seurata, esitän seuraavissa luvuissa toistuvana esimerkkinä yksinkertaistetun Potilas-rekisterin suunnittelun ja testauksen. Potilaasta käytetään useissa terveydenhuollon sovelluksissa myös nimitystä henkilö tai asiakas. Kuvaus ei ole kuitenkaan toimialakohtainen, sillä potilaan tai henkilön sijaan voisimme tarkastella tuoterekisterin testausta, koska niissä on toteutuksen ja testauksen näkökulmasta selkeästi samoja piirteitä. Potilastietoihin liittyvä testaus on yleensä paljon kriittisempää. Potilasrekisterin suunnittelussa on havaittavissa aikajärjestyksessä seuraavat vaiheet:

1) Vaatimusmäärittely.
2) Arkkitehtuurin suunnittelu ja kuvaaminen (komponenttikaaviot).
3) Kohdealueen tapahtumien ja tapahtumaketjujen mallintaminen (aktiviteettikaaviot).
4) Käyttötapausten muodostaminen (käyttötapauskaaviot).

5) Järjestelmän sisäisen rakenteen ja vuorovaikutuksen kuvaaminen (luokkakaaviot, sekvenssikaaviot, yhteistyökaaviot, tilakaaviot).
6) Järjestelmän toteutuksen kuvaaminen (toimituskaaviot).

Esimerkin Potilas-rekisterin tulee täyttää seuraavat toiminnalliset vaatimukset:

- Potilaasta tulee lisätä henkilötunnus, sukunimi, etunimet, lähiosoite, postiosoite ja puhelinnumero.
- Potilasta voidaan etsiä henkilötunnuksen, sukunimen tai etunimen mukaan.
- Henkilötunnusta lukuun ottamatta potilastietoja voidaan muokata.
- Potilastiedot voidaan poistaa varmistuskyselyn jälkeen (todellisissa sovelluksissa potilastietojen poistaminen ei ole yleensä sallittua).

4.2 Vaatimusmäärittelyn UML-kaaviot

4.2.1 Arkkitehtuurit ja testaus

Ohjelmistoarkkitehtuuri kuvaa järjestelmän karkean tason ”rakennuspiirustukset” sisältäen ohjelmiston komponentit, komponenttien yhteydet toisiinsa ja ympäristöön, asianosaisten vaatimukset sekä valitut ratkaisut perusteluineen. Järjestelmäarkkitehtuuri voisi sisältää esimerkiksi Kysely-komponentin (kyselyn määrittely tietojen hakemiseksi terveydenhuollon tietojärjestelmästä) ja Palvelin-komponentin (haettujen potilastietojen palautus) sekä rajapinnan kuvauksen (potilastietojen hakukriteerit, palautettava tietosisältö ja tiedon muoto). Huonosti suunniteltu arkkitehtuuri aiheuttaa ongelmia toteutuksessa, myöhästymisiä toimituksissa asiakkaalle, puutteita suorituskyvyssä, siirrettävyysongelmia sekä hankaloittaa testausta ja ylläpitoa. Tämän vuoksi arkkitehtuurin suunnittelu ja eri toteutusvaihtoehtojen vertailu on hyvin tärkeää projektin alkuvaiheessa ja se tulisi tehdä heti vaatimusmäärittelyn jälkeen.

Arkkitehtuurin toteutuksessa voidaan erottaa erilaisia arkkitehtuurityylejä, arkkitehtuurimalleja, suunnittelumalleja, komponentteja ja rajapintoja sekä sovelluskehyksiä. Arkkitehtuurin tarkoituksena on ohjata ohjelmiston kehittämisprosessia ottamalla huomioon olennaiset laatuvaatimukset, kuten uudelleenkäytettävyys, turvallisuus, skaalautuvuus ja ylläpidettävyys. Ohjelmiston rakentamisessa hyödynnetään valitun arkkitehtuurin määrittelemiä rakenteita, mekanismeja ja periaatteita [Bos00, BuM01].

Buschmann ja kumppanit ovat jakaneet arkkitehtuurityylit neljään ryhmään [BuM01]:

a) Jäsennysarkkitehtuurit (kerrosarkkitehtuuri, putki-suodatin-arkkitehtuuri),

b) Hajautusarkkitehtuurit (broker),

c) Vuorovaikutusarkkitehtuurit (MVC = model view controller, PAC = Presentation Abstraction Controller) ja

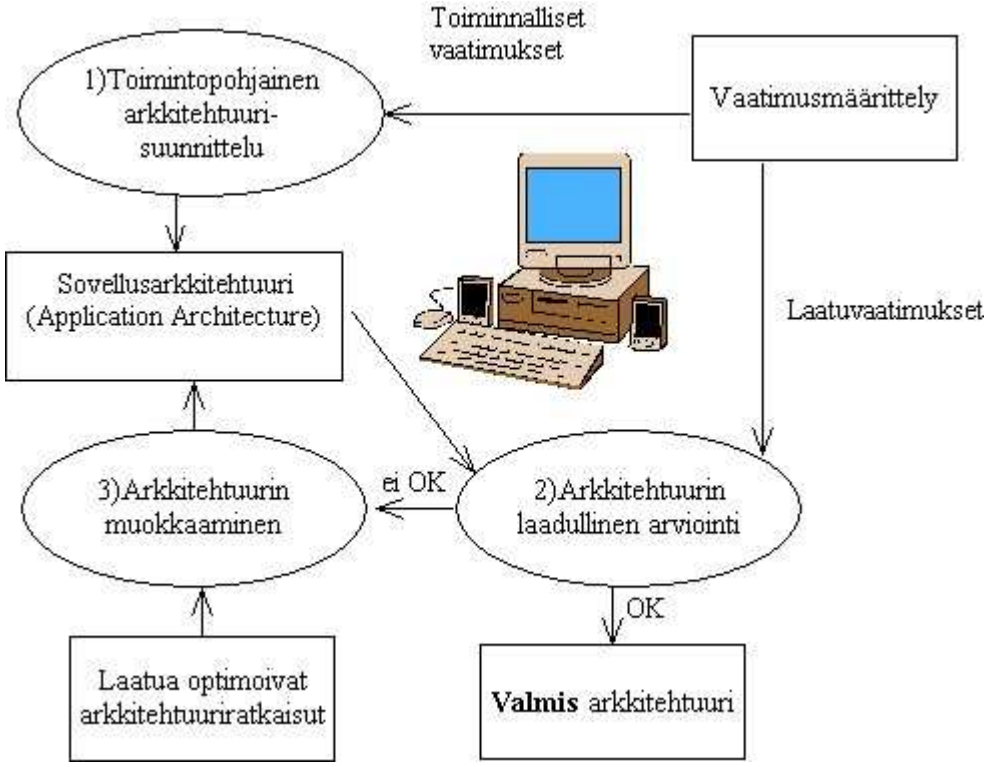
d) Muokattavuutta tukevat arkkitehtuurit.

Edellä mainittuja arkkitehtuurityylejä ja niiden sisältöä ei käsitellä tässä tutkielmassa tarkemmin, vaan kiinnitetään huomio siihen, miten arkkitehtuurin kuvauksen perusteella voidaan tehdä tulkintoja siitä, millaisia testitapauksia tarvitaan ja mihin testaus kohdennetaan. Arkkitehtuurisuunnittelu voidaan Boschin mukaan nähdä prosessina, joka saa syötteenä vaatimusmäärittelydokumentit ja josta tuloksena saadaan arkkitehtuurisuunnitelma [Bos00]. Bosch on jakanut arkkitehtuurisuunnittelun kolmeen vaiheeseen (Kuva 10):

1) *toimintopohjainen arkkitehtuurisuunnittelu*,

2) *laadullisten ominaisuuksien arviointi* ja

3) *arkkitehtuurin muokkaaminen*.



Kuva 10: Arkkitehtuurisuunnitteluprosessi [Bos00]

Toimintopohjaisen arkkitehtuurisuunnittelun tarkoituksena on kartoittaa järjestelmän toiminnalliset ominaisuudet ja rakenneosat muun muassa määrittelemällä rajapinnat ulkoisiin kohteisiin, tunnistamalla järjestelmän keskeiset abstraktiot (rakenteet), jakamalla järjestelmä keskeisiin osiin (komponentit ja tasot) ja lopuksi kuvaamalla järjestelmä edellä mainituilla rakenteilla. Monissa projekteissa arkkitehtuurisuunnittelu ja testaaminen keskittyvät liian paljon pelkkään toiminnallisuuteen ja tärkeiden laatuvaatimusten testaus unohtuu taka-alalle. [Bos00]

Laadullisten vaatimusten arvioinnissa kutakin laatuominaisuutta esim. turvallisuus arvioidaan erikseen ja laatuominaisuudelle laaditaan mittarit ja toteutumisen arviointikriteerit. Laatuvaatimusten arvioinnissa käytetään hyödyksi skenaarioprofiileja. *Skenaarioprofiilien* avulla tutkitaan järjestelmän ominaisuuksia eli laatuattribuuttien arvoja hyödyntäen systeemin käyttöä ja muita ominaisuuksia kuvaavia profiileja. *Profiili* koostuu joukosta skenaarioita. Jokaiseen skenaarioon liittyy attribuutteja kuten muutospyynnön esiintymisen arvioitu todennäköisyys tai käyttötapauksen tärkeys. Skenaariot saattavat olla osittain päällekkäisiä, joten yhteenlasketut todennäköisyysarvot voivat olla arvoltaan enemmän kuin 1. [Bos00]

Seuraavaksi on esitetty esimerkkejä erilaisista profiileista: *Käyttöprofiili* on joukko *käyttöskenaarioita* (Taulukko 3), jotka kuvaavat järjestelmän tyypillisiä käyttötapoja. Näitä voidaan käyttää toiminnallisuuden, suorituskyvyn ja luotettavuuden arviointiin. Käyttöprofiililla pyritään määrittämään, miten eri ihmiset (ylläpito, normaali käyttäjä, asiakas) käyttävät järjestelmää hyväksi eri tavoin. Käyttöprofiilit muistuttavat käyttötapauksia (use cases), mutta sisältävät käyttötapauksista poiketen arvion toimintojen suorituksen todennäköisyydestä.

Taulukko 3: Käyttöskenaario -esimerkki

Skenaario	**Todennäköisyys**
Käyttäjä lisää potilaan.	0.3
Käyttäjä etsii potilaan.	0.68
Käyttäjä poistaa potilaan.	0.02

Hazard-profiili koostuu skenaarioista (Taulukko 4), jotka kuvaavat tilanteita, joilla on vaikutusta järjestelmän turvallisuuteen. Profiilin tarkoituksena on kartoittaa järjestelmän kannalta uhkatilanteet, jotka voivat liittyä esimerkiksi ulkopuolelta tuleviin vahingoittamisyrityksiin (virukset, tiedostojen tuhoaminen tai tietosisällön muuttaminen vahingolliseksi), salaiseksi tarkoitetun tiedon laittomaan hakemiseen ja salasanojen tai tunnuslukujen hakemiseen (saaduilla salasanoilla päästään murtautumaan jonnekin muualle tekemään vahinkoa).

Taulukko 4: Hazard-skenaario -esimerkki

Skenaario	**Todennäköisyys**
Käyttäjä pyrkii sisään väärällä salasanalla.	0.7
Tunkeilija pääsee järjestelmään sisälle.	0.02
Käyttäjä yrittää päivittää tietoja toisen tallentamien tietojen päälle.	0.28

Ylläpitoprofiili sisältää *muutoskategoriat*, jotka ryhmitellään järjestelmän rajapintojen mukaan ja *muutosskenaariot* (Taulukko 5), jotka kuvaavat konkreettiset muutostarpeet. Ylläpitoprofiilin tarkoituksena on kuvata ja erottaa järjestelmän muuttuvat osat muuttumattomista osista. Järjestelmän muuttuvien osien kannalta arvioidaan muutoksiin kuluva työmäärä ja kustannusosuus. Ylläpitoprofiilin avulla voidaan arvioida järjestelmän ylläpitokustannuksia ja ottaa ylläpitokustannukset huomioon projektin kokonaiskustannuksissa.

Taulukko 5: Muutosskenaario -esimerkki

Skenaario	Todennäköisyys	Muutoksen vaikutus (koodirivejä)
Perustiedot-lomake muuttuu	0.4	40
Tietojen haku –lomake muuttuu	0.3	50
Lisätiedot-lomake muuttuu	0.2	30
Uusi lomake tehdään	0.1	200

Arkkitehtuurisuunnittelu ja suunnitteluprosessin tuloksena saatu *arkkitehtuurin kuvaus* tarjoavat testausta ja ylläpitoa varten tietoa järjestelmän laadullisista ominaisuuksista esim. suorituskyky, ylläpidettävyys, luotettavuus ja turvallisuus. Suorituskykyyn ja turvallisuuteen liittyvät arvioinnit ja testit ovat tärkeä osa järjestelmätestausta.

Suorituskykyä eli systeemin tehokkuutta voidaan mitata toimintojen läpimenoajalla (skenaariot/aikayksikkö) tai vasteajalla. Profiilina käytetään käyttöprofiilia, joka sisältää joukon järjestelmän käyttöskenaarioita. Käyttöskenaario kuvaa yhden käyttäjän käyttökerran. Järjestelmän sisältämistä komponenteista ja niiden toiminnallisuudesta saadaan suorituskykytietoa. Arkkitehtuurikuvauksesta voidaan tutkia järjestelmän suorituskykyä kussakin käyttöskenaariossa, kun kuvaukseen kirjataan jokaisen komponentin keskimääräinen ja maksimi suoritusaika sekä synkronoinnin ja järjestelmän yleiskuormituksen vaatima aika. Tätä voidaan käyttää hyväksi ohjelmiston laadun parantamisessa ja järjestelmätestausvaiheessa järjestelmän tehokkuuden ja suorituskyvyn arvioinnissa.

Ylläpidettävyyttä kuvataan ylläpitoprofiililla, joka testaa muutosten vaikutukset arkkitehtuuriin. Arkkitehtuurikuvauksesta saadaan selville arvioitu muutettavien koodirivien määrä eli miten suuri muutostarve eri järjestelmän osilla on. Luotettavuutta voidaan tarkastella käyttöskenaarioilla, jotka selvittävät komponentin ajonaikaisen vuorovaikutuksen toisten komponenttien kanssa. Käyttöskenaariot, joissa käytetään useampaa komponenttia, ovat luotettavuudeltaan alhaisempia kuin yhden komponentin skenaariot.

Turvallisuutta kuvataan Hazard-profiililla, jolla arvioidaan järjestelmän turvallisuuteen liittyviä asioita kuten tiedon saantioikeuksien ylitykset. Arkkitehtuurikuvaukseen

saadaan Hazard-profiilista järjestelmän turvallisuustaso ja suojaukseen liittyvien toimintojen suoritusaika kuten salasanojen kirjoittamiseen kuluva aika [Bos00].

Arkkitehtuurin muokkaamisvaiheessa tunnistetaan ensiksi laatuominaisuudet, jotka eivät ole täyttyneet ja selvitetään tarkemmin kohdat, joissa vaatimuksista poiketaan. Tämän jälkeen tilanteeseen valitaan sopiva korjausvaihtoehto ja suoritetaan muokkaustoimenpiteet. Muokkauskeinoina voivat olla:

- *Arkkitehtuurityylin säätäminen*: Valitaan esimerkiksi kerrosarkkitehtuurin tilalle hajautusarkkitehtuuri. Erilaisen arkkitehtuurityylin valinnalla voidaan vaikuttaa mm. järjestelmän turvallisuuteen, luotettavuuteen suorituskykyyn ja muunneltavuuteen.
- *Arkkitehtuurimallin säätäminen*: Arkkitehtuurimallin muokkauksessa laajennetaan ja muutetaan peruskomponenttien käyttäytymistä, mutta ei muuteta niiden perusrakennetta. Muokkauksella voidaan vaikuttaa samanaikaisuuteen ja pysyvyyteen.
- *Suunnittelumallit*, joilla on tarkoitus saada arkkitehtuuriin paikallisia ominaisuuksia esimerkiksi järjestelmän uudelleenkäytettävyyttä ja hallittavuutta koskien. Esimerkiksi Sovitin (Adapter) -suunnittelumallia käytetään, kun halutaan saada toimimaan yhdessä sellaiset luokat, joilla on yhteensopimattomat rajapinnat. Suunnittelumallien käytössä kannattaa seurata samalla, etteivät järjestelmän suorituskyky ja reaaliaikaisuus kärsi toimenpiteistä. Myös käytettävyyttä voidaan parantaa suunnittelumallien avulla [Imm02].
- *Laatuominaisuuksien muuttaminen osaksi järjestelmän toiminnallisuutta*. Esimerkiksi asiakkaalle myydyn webportaalin ylläpidettävyyttä voidaan parantaa ohjelmoimalla toiminto, jolla voidaan hallitaan asiakkaalla olevia portaalin toimintoja. Ylläpitäjä voi webin kautta lisätä asiakkaan tuotteeseen uuden toiminnon.

4.2.2 Komponenttikaaviot

4.2.2.1 Komponenttikaavion merkitys ja notaatio

Arkkitehtuuria voidaan kuvata *UML-komponenttikaaviolla*, joka osoittaa, mitkä komponentit kuuluvat järjestelmään. Komponenttikaaviosta käy selville myös järjestelmän sisältämien komponenttien karkean tason riippuvaisuudet eli komponenttien väliset suhteet. Komponentti määritellään itsenäiseksi yksiköksi, jota käytetään ohjelmistojen

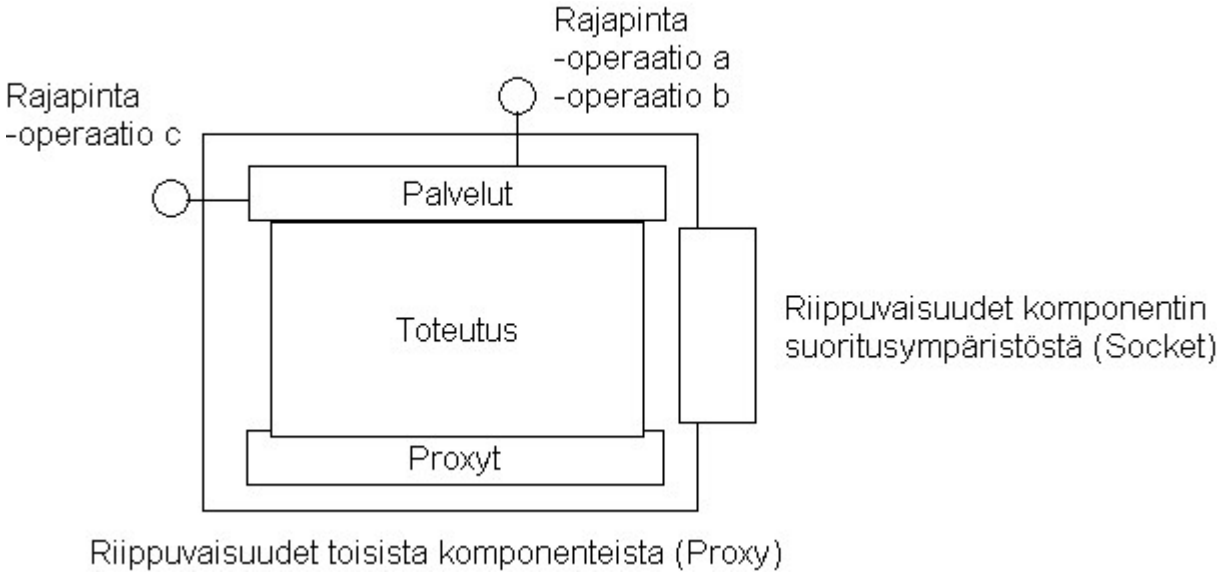
koostamisessa. Se voidaan toimittaa itsenäisenä yksikkönä (mustana laatikkona) asiakkaalle tai integraattorille, joka huolehtii komponenttien yhdistämisestä. [ToE02]

Komponenttien koostamisessa käytetään tarkasti määriteltyjä *rajapintoja* (interfaces), jotka ovat joko komponentin tarjoamia tai komponentin kutsumia rajapintoja. Rajapinta on järjestelmän toimijoiden tai osien välinen kosketuspinta. Komponenteilla on neljäntyyppisiä rajapintoja [Sam97]:

- Käyttäjärajapinta (komponentin ja loppukäyttäjän välillä).
- Datarajapinta (komponentin ja sen käyttämän tietovaraston välillä).
- Kompositiorajapinta (komponentin ja toisten komponenttien välillä). Kompositiorajapinta voi olla a) *komponenttiliittymä*, jolla komponentti tarjoaa palveluitaan muille tai b*) riippuvuus* (dependency), jolla komponentti käyttää toisten komponenttien palveluita hyväkseen.
- Komponenttialusta: komponentin tarvitsema ohjelmistoinfrastruktuuri.

Rajapinnan avulla järjestelmän eri osat voidaan liittää yhteen. Komponentin tarjoamat ja tarvitsemat rajapinnat yhdessä komponentin suoritusympäristön kanssa muodostavat kutsutun komponentin ja komponentin kutsujan välisen *sopimuksen*. Komponentilla voi olla samanaikaisesti useampia sopimuksia eri asiakkaiden kanssa. [ToM02]

Komponentit voivat olla rakeisuudeltaan eri tasoisia: hajautettuja komponentteja, toimialakomponentteja tai laajoja komponenttisysteemejä [HeS00, Myk00]. *Hajautettu komponentti* (Distributed Component) on rakeisuudeltaan alimman tason komponentti (Kuva 11), jolla on hyvin määritelty rakentamisen aikainen ja ajonaikainen rajapinta, johon pystytään viittaamaan verkosta käsin. Hajautettu komponentti siis kapseloi hajautuksen.



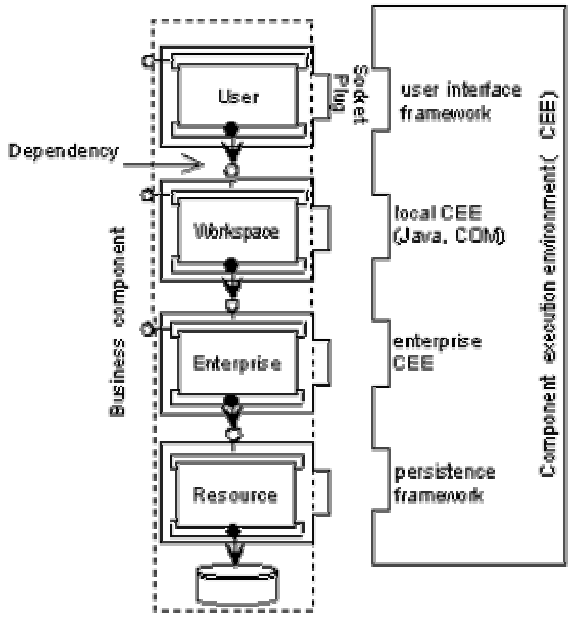
Kuva 11: Hajautettu komponentti (DC)

Rajapintoja voi olla yksi tai useampia ja niissä on määritelty komponentin tarjoamat toiminnot eli operaatiot sekä komponentin kutsussa tarvittavat parametrit. Hajautettu komponentti voi olla riippuvuussuhteessa muihin hajautettuihin komponentteihin. Käyttöliittymän toteutus tulisi pitää erillään toimintalogiikasta ja tietokantakutsuista, joten hajautettu komponentti jaetaan kerroksiin: käyttöliittymä, työtila, toimintataso ja resurssi. Hajautetusta komponentista voisi olla esimerkkinä laboratoriopyynnön syöttäminen.

Toimiala- tai liiketoimintakomponentti (Business Component, Kuva 12) on rakennettu hajautetuista komponenteista. Toimialakomponentti toteuttaa hyvin *koossapysyvän* (cohesion) loogisen kokonaisuuden. Se sisältää eri kerroksia, kuten käyttöliittymä-, työtila-, toiminto- ja resurssikerrokset kapseloiden siis kerrosarkkitehtuurin. Käyttöliittymä- ja työtilakerroksia käyttää yksi käyttäjä (yhden käyttäjän alue). Työtilakerroksessa käyttäjä voi esimerkiksi laskea tarjouksia, mutta toiminnoista ei tallennu järjestelmän tietovarastoon mitään tietoja. Toiminto- eli sovelluslogiikkakerros ja resurssikerros muodostavat puolestaan monen käyttäjän alueen [HeS00].

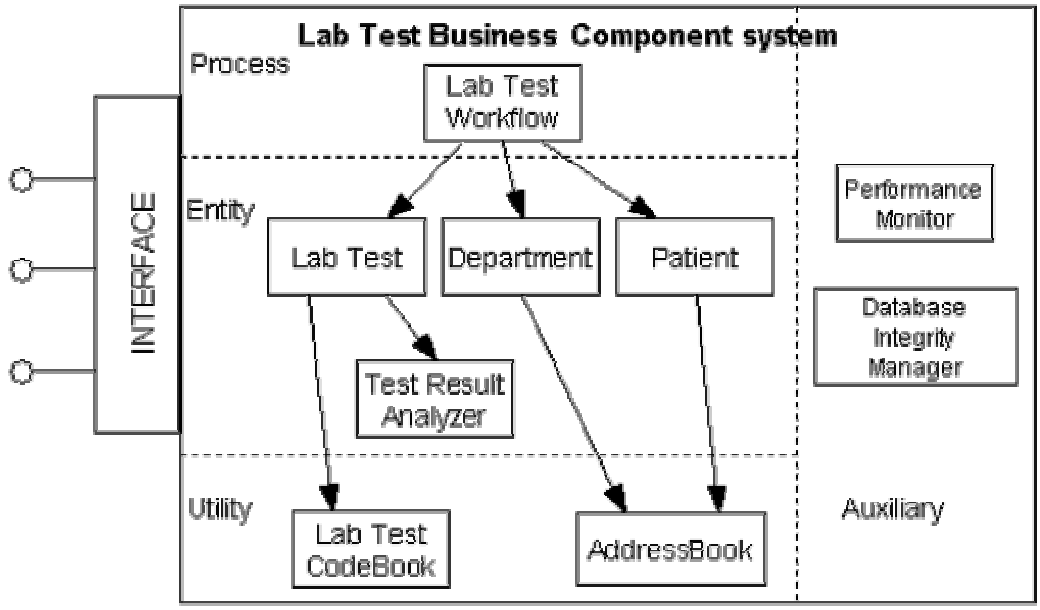
Toimialakomponentin sisältämät hajautetut komponentit suorittavat viestien välitystä alemman tai saman kerroksen hajautettujen komponenttien kanssa. Ajonaikainen rajapinta toimialakomponentille on kooste kaikkien niiden hajautettujen komponenttien

rajapinnoista, jotka ovat näkyviä toimialakomponentin ulkopuolelle. Esimerkkinä toimialakomponentista voisi olla laboratoriotesti.



Kuva 12: Toimialakomponentti (Business Component)

Komponenttisysteemi (Business Component System) rakentuu toimialakomponenteista. Esimerkkinä komponenttisysteemistä on kuvassa 13 esitetty laboratoriotestauksen komponenttisysteemi. Komponenttisysteemin rajapinta muodostuu niiden toimialakomponenttien rajapinnoista, jotka näkyvät systeemin ulkopuolelle. Toimialakomponentit voidaan ryhmitellä komponenttisysteemin sisällä erilaisiin toiminnallisiin kerroksiin kuten prosessi-, entiteetti-, apukomponenttikerros ja ulkoiset komponentit.



Kuva 13: Komponenttisysteemi (BCS) [ToM02]

PlugIT-Teho –projektissa komponenttisysteemin testausta on tutkittu toimintoketjutestauksen avulla. Toimintoketju (action flow) kuvaa toimintoja useiden ihmisten ja järjestelmien välillä. Toimintoketjut johdetaan vaatimusmäärittelyn perusteella ja niistä testataan vain tärkeimmät. Olennaista tässä lähestymistavassa on se, että toiminnot seuraavat toisiaan tietyssä järjestyksessä. Esimerkiksi potilas täytyy ensin luoda järjestelmään ja vasta tämän jälkeen voidaan lisätä häneen liittyviä laboratoriotuloksia. Lab Test -komponenttisysteemin yksi toimintoketju voisi olla seuraava (toimijat esitetty suluissa):

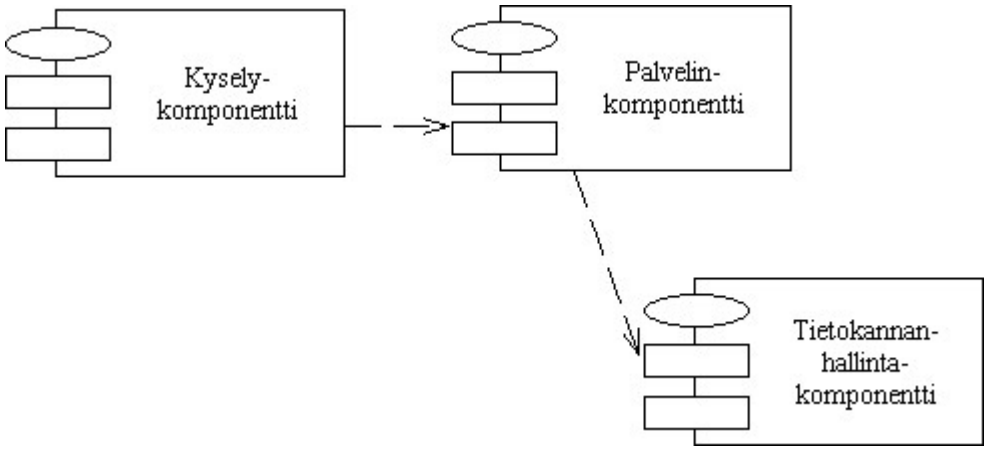
- Luo tai valitse potilas (sovelluksen käyttäjä ja Patient BC).
- Tutki potilas.
- Luo laboratoriopyyntö (sovelluksen käyttäjä ja Lab Test BC).
- Lähetä laboratoriopyyntö (sovelluksen käyttäjä, Lab Test BC ja Department BC).
- Vastaanota potilas.
- Ota näyte (ihminen).
- Analysoi näytteet ja palauta tulokset (Test Result Analyzer BC).
- Johda viitearvot (Lab Test Codebook BC).
- Tallenna laboratoriotulokset ja viitearvot (Lab Test BC).

Komponenttisysteemin testitapaukset johdetaan toimintoketjujen pohjalta (kts. Liite 1. s.8-9). Komponentteja voidaan kuvata yhtä hyvin perinteisillä laatikko-viiva -kaavioilla kuin ohjelmistotuotannon UML-kaavioilla. UML-komponenttikaavion notaatio on seuraava:

- Komponentti on järjestelmän sisältämä itsenäinen ohjelmiston osa.
- Riippuvaisuus kuvaa suhdetta komponenttien välillä. Riippuvaisuuden suuntaa voidaan merkitä nuolella.
- Rajapinta (pieni ympyrä, josta lähtee viiva) on järjestelmän toimijoiden välinen suhde. Komponentti voi rajapintojen kautta joko käyttää itse toisten komponenttien palveluita tai tarjota palveluita toisille.

4.2.2.2 Esimerkki komponenttikaavion käytöstä

Seuraavassa arkkitehtuuriesimerkissä (Kuva 14) on hyödynnetty Yoonin artikkelin esimerkkiä, missä kuvataan UML-mallin käyttöä komponenttien väliseen integrointitestaukseen [YoC99]. Kuvattu järjestelmä koostuu kolmesta hajautetusta komponentista: Tietokannan hallinta –komponentti ylläpitää potilaan tietoja tietovarastossa eli mahdollistaa lisäykset, päivitykset ja poistot. Palvelin-komponentti tarjoaa palveluna potilaan tietoja ja Kysely-komponentti mahdollistaa potilaan tietojen etsimisen.



Kuva 14: Komponenttikaavio

4.2.2.3 Komponenttikaavion merkitys testauksen kannalta

Komponenttikaavio tarjoaa siis kuvauksen järjestelmän sisältämistä komponenteista ja niiden välisistä suhteista. Koska komponentteja on eri tasoisia, myös testauksen ja

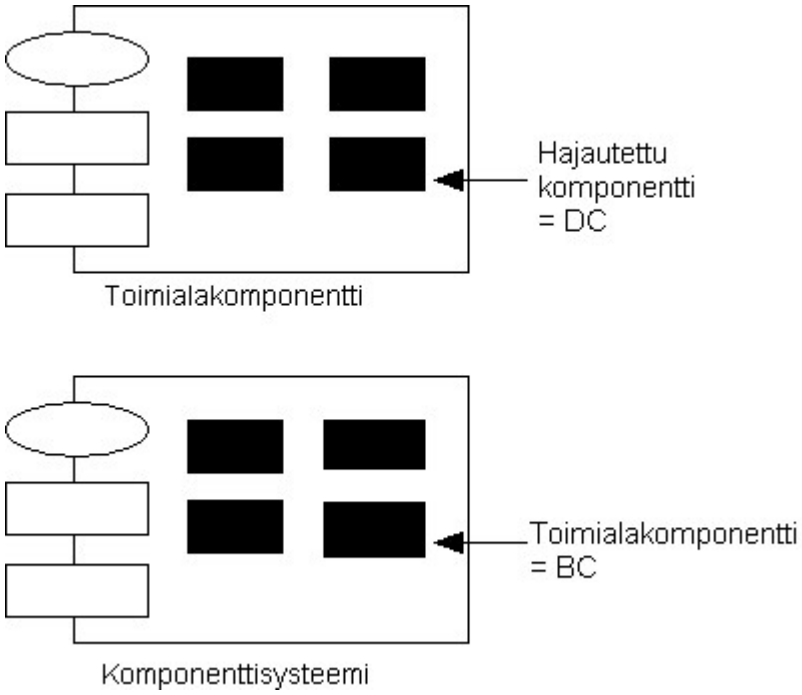
testitapausten suunnittelun on vaihdeltava eri tasojen mukaan. Komponenttisysteemien testausmenetelmässä mustalaatikko- ja lasilaatikkotestausta tehdään vuorotellen ja testaus aloitetaan pienimmän rakeisuuden omaavista komponenteista kohti karkeampijakoisia komponentteja eli siirrytään hajautetuista komponenteista kohti komponenttisysteemejä. [ToM02]

Yksikkötestausvaiheessa komponentin sisäinen rakenne on näkyvissä. Komponentin toiminnallisuutta testataan siis ”lasilaatikkona” ohjelmakoodin perusteella. Seuraavaksi testataan komponentin ulkoinen rajapinta toisiin komponentteihin nähden. Tämän jälkeen komponentti voidaan käsittää mustana laatikkona ja se voidaan testata oikeassa ajoympäristössä. Integrointitestausvaiheessa komponenttia pidetään kokoelmana komponentin sisäisiä komponentteja. Ensin testataan sisäisten komponenttien (mustat laatikot) toiminta yhdessä ja seuraavaksi testaukseen tulee kokoelmakomponentin rajapinta.

Komponenttipohjaisessa sovellustuotannossa testaajalla havaitaan olevan eri tyyppisiä rooleja, joilla ei tässä tarkoiteta perinteistä jaottelua testaajan ja testauksen suunnittelijan tehtäviin, vaan jakoa komponentin tarjoajan, integraattorin ja asiakkaan rooleihin [ToE02]:

- Komponentin tarjoaja käyttää testausmenetelmänä lasilaatikkotestausta varmistuakseen siitä, että komponentti täyttää sopimuksessa määritetyt ominaisuudet ja mustalaatikkotestausta varmistaakseen sen, että rajapintaa voidaan kutsua sopimuksessa määritetyistä ympäristöistä rajapinnan dokumentaation määrittelemällä tavalla.
- Integraattori liittää itse tehdyt ja muualta ostetut komponentit yhdeksi kokonaisuudeksi eli komponenttisysteemiksi. Integraattori tarvitsee mustalaatikkotestausta tarkistaakseen, että hän saa vaatimukset täyttävän tuotteen tai komponentin. Lasilaatikkotestausta tarvitaan varmistamaan itse tehtyjen komponenttien toimivuus.
- Asiakas, jolle tarjotaan valmis komponenttisysteemi. Asiakas ei pääse tutustumaan komponenttisysteemin sisäiseen rakenteeseen, joten hän käyttää hyväksymistestauksessa mustalaatikkotestausta.

Itse tehtyjen ja ostettujen komponenttien integraatiotestauksessa testataan komponenttien välisten kutsujen toimivuutta. Toimialakomponentin sisäisestä testauksesta on mahdollista käyttää lasilaatikkonimitystä, vaikka hajautettujen komponenttien koodia ei ole käytettävissä. Samoin komponenttisysteemin sisäistä testausta voidaan kutsua lasilaatikkotestaukseksi (Kuva 15), vaikka sisäisten komponenttipalasten ohjelmakoodia ei ole saatavilla [ToM02].



Kuva 15: Lasilaatikkotestaus mustalaatikkotestauksena

Komponenttisysteemin testausta ja testitapausten suunnittelua on käsitelty PlugIT-Teho-projektin tutkimuksissa (Liite 1, s. 6-10).

4.2.3 Aktiviteettikaaviot

4.2.3.1 Merkitys ja notaatio

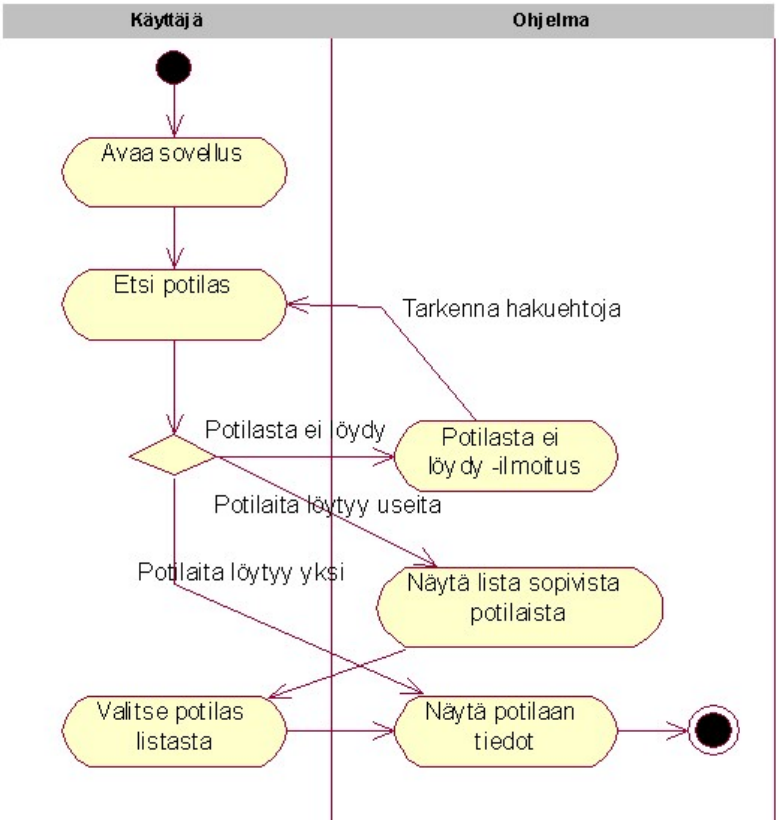
Aktiviteettikaavioiden avulla voidaan kuvata järjestelmän sisällä tai ulkopuolella tapahtuvaa työnkulkua [Eer01]. Aktiviteettikaavion solmut ovat tehtäviä tai toimintoja, joita järjestelmä suorittaa yksin (dialogin avaaminen) tai vuorovaikutuksessa käyttäjän

kanssa (painikkeen painaminen). Kaavio voi sisältää useita peräkkäisiä toimintoja tai rinnakkaisia toimintoja, joista kustakin voidaan siirtyä (nuoli osoittaa siirtymissuunnan) uuteen toimintoon. Aktiviteettikaavio kuvaa siis itse asiassa vain yhtä aktiviteettia tai kiinteästi toisiinsa liittyvien aktiviteettien kokonaisuutta.

Aktiviteettikaavioita voidaan muodostaa joko ennen käyttötapausten laatimista tai sen jälkeen. Ennen käyttötapausten muodostamista projektissa on tarve kehittää prosesseja ja esittää kohdealueella tapahtuvien toimintojen järjestys korkealla tasolla. Tähän aktiviteettikaavio sopii mainiosti. Kun järjestelmästä on saatu enemmän tietoa ja käyttötapaukset on rakennettu, asteeltaan tarkempia aktiviteettikaavioita käytetään selventämään käyttötapausten kuvauksia. Koska aktiviteettikaavio tukee samoja elementtejä kuin vuokaaviot, sitä voidaan käyttää kontrollikaavioiden tapaan testausmallin luomiseen. Testaaja pystyy vertaamaan aktiviteettikaaviossa kuvattua ohjelman toimintojen suoritusjärjestystä todelliseen toimintojen järjestykseen.

Aktiviteettikaaviossa käytetään seuraavaa notaatiota [Bin00, s. 294]:

- *Toiminto* (activity) on kokoelma käyttäjien tai ohjelman työnkulun tapahtumia.
- *Päätössolmua* (decicion) kuvataan salmiakilla. Päätössolmussa toiminnon suoritus haarautuu kuvatun ehdon mukaan.
- *Synkronointipalkin* kohdalla tarkastetaan, että rinnakkaiset aktiviteetit ovat tehtävien suorituksen kannalta ja toisiinsa verrattuna sopivassa vaiheessa.
- *Uimalinjat* (swimline) osoittavat toiminnon suorittajan kuten järjestelmän, osaston tai henkilön, joka käyttää järjestelmää.



Kuva 16: Aktiviteettikaavio Etsi potilas -toiminnosta

4.2.3.2 Esimerkki aktiviteettikaavion käytöstä

Kuvan 16 aktiviteettikaavio kuvaa todellista tilannetta, missä käyttäjä (esim. sairaanhoitaja) etsii potilaan tietoja potilashallinnon järjestelmästä. Käyttäjä avaa potilastietoja sisältävän sovelluksen ja syöttää suoraan lomakkeelle hakukriteerit: henkilötunnuksen tai sukunimen ja etunimen ja käynnistää haun. Mikäli potilasta ei näillä kriteereillä löytynyt, ohjelma antaa *Potilasta ei löydy* -ilmoituksen ja avaa lomakkeen tarkennetulle haulle. Tarkennetussa haussa potilasta voi hakea henkilötunnuksen, sukunimen tai etunimen, kutsumanimen, entisen nimen, kotikunnan, sukupuolen tai syntymäajan mukaan.

Hakua vastaavia potilaita voi löytyä yksi tai useampia. Jos hakutuloksena löytyi vain yksi sopiva potilas, hänen tietonsa tulevat suoraan lomakkeelle. Mikäli henkilöitä on useampia, ohjelma avaa valintalistan, josta haluttu potilas voidaan valita, jonka jälkeen tiedot näkyvät näytöllä.

4.2.3.3 Aktiviteettikaavion merkitys testauksen kannalta

Miten aktiviteettikaaviota voidaan käyttää hyväksi testauksessa? Aktiviteettikaavio kuvaa sarjaa tapahtumia ja vastaa perinteisen kontrollikaavion toimintamallia. Lasilaatikkotestauksessa aktiviteettikaavion elementeillä on lisäksi mahdollista kuvata ohjelmakoodin ehtorakenteita (if), silmukkarakenteita (for ja while) sekä rinnakkaisia toimintoja (säikeet) [BeP02, s. 608].

Toiminnallisessa (mustalaatikko) testauksessa aktiviteettikaaviolla voidaan mallintaa esimerkiksi dialogien suoritusjärjestystä. Aktiviteettikaavion käyttöä dialogipohjaisen sovelluksen testaukseen on tutkittu PlugIT-Teho -projektin tutkimuksissa (Liite 1, s. 1-6). Testitapauksen laatimiseksi tarvitaan syöte ja oletettu tulos. Syötteet näkyvät käyttäjän uimakaistalla ja ne ovat yleensä seuraavanlaisia:

- Käyttäjä painaa jotain painiketta (esim. Etsi).
- Käyttäjä valitsee valikosta jonkun toiminnon (esim. Tiedosto - Lopeta).
- Käyttäjä syöttää tietokenttään tietoa (esim. hakuehto).

Oletetut tulokset ovat tapahtumia, joita järjestelmä suorittaa käyttäjän antamien syötteiden jälkeen. Tuloksena voi olla jonkun dialogin avautuminen, tietojen tallennus tietokantaan, laskutoimitus tai joku muu ohjelmalle määritelty tehtävä. Oletetut tulokset näkyvät kuvan aktiviteettikaaviossa järjestelmän uimakaistalla. Käyttämällä uimakaistoja kaaviossa pystytään helposti havainnollistamaan tehtävien jakautumista eri käyttäjäryhmien ja järjestelmän kesken. Taulukko 6 kuvaa Etsi potilas -aktiviteettikaavion pohjalta johdettuja testitapauksia. Ensimmäinen testitapaus käsittelee poikkeustilanteen, jossa yritetään etsiä olematonta potilasta. Toinen testitapaus koostuu normaalista tilanteesta eli potilaan tiedot löytyvät tietokannasta. Kolmas testitapaus testaa, miten haun jälkeen järjestelmän palauttamasta potilaslistasta voidaan valita yhden potilaan tiedot.

Taulukko 6: Etsi potilas -aktiviteettikaavion pohjalta johdetut testitapaukset

Id	Esiehto	Syöte	Oletettu tulos
1	Potilasta ei olemassa	Hakuehtona hetu=041076-9999	Ilmoitus Potilasta ei löydy
2	Potilas on olemassa	Hakuehtona hetu=061083-9569	Potilaan tiedot näytöllä

3	2 samannimistä potilasta ”Korhonen” olemassa	Hakuehtona sukunimi= Korhonen	Valintalista näytöllä

4.2.4 Käyttötapauskaaviot

4.2.4.1 Merkitys ja notaatio

Asiakasvaatimusten perusteella johdetaan *käyttötapaukset* (use cases), jotka kuvaavat järjestelmän ja käyttäjän vuorovaikutusta. Käyttötapaukset edustavat eri tyyppisiä järjestelmän vaatimuksia, kuten toiminnallisia vaatimuksia, toiminnallisuuden jakautumista luokille tai olioille sekä olioiden vuorovaikutusta ja rajapintoja. Tämän lisäksi käyttötapauksia voidaan soveltaa suunnitteluongelmien analysointiin ja määrittelyyn kuten

- jäljitettävyyden muodostamiseen,
- kohdealueen käsitteiden kuvaamiseen,
- järjestelmän rajojen määrittelemiseen,
- kehittämisresurssien ja työpanosten arviointiin ja
- arkkitehtuurisuunnitteluun.

Käyttötapaukset esittävät dialogia järjestelmän ja ulkoisten toimijoiden välillä. Toimijoiden ei tarvitse olla ihmisiä, vaan käyttötapauksilla on mahdollista kuvata myös vuorovaikutusta eri tietojärjestelmien tai sähköisten mittalaitteiden välillä. Käyttötapauksen ilmentymä (esim. *Lisää potilas* tai *Näytä Potilastiedot -lomake*) määrittelee toimijoilta tulevat syötteet tai oletetut tulokset. Käyttötapauskaavioiden jokainen merkittävä käyttötapaus esitetään tarkemmin tekstimuodossa käyttötapausten skenaariokuvauksissa. Kuvaus sisältää seuraavat tiedot: käyttötapauksen tunniste ja nimi, tarkoitus, toimijat, sanallinen kuvaus, esiehdot, loppuehdot, perustoiminta, vaihtoehtoistoiminta ja poikkeustoiminta. [Bin00, Hir00]

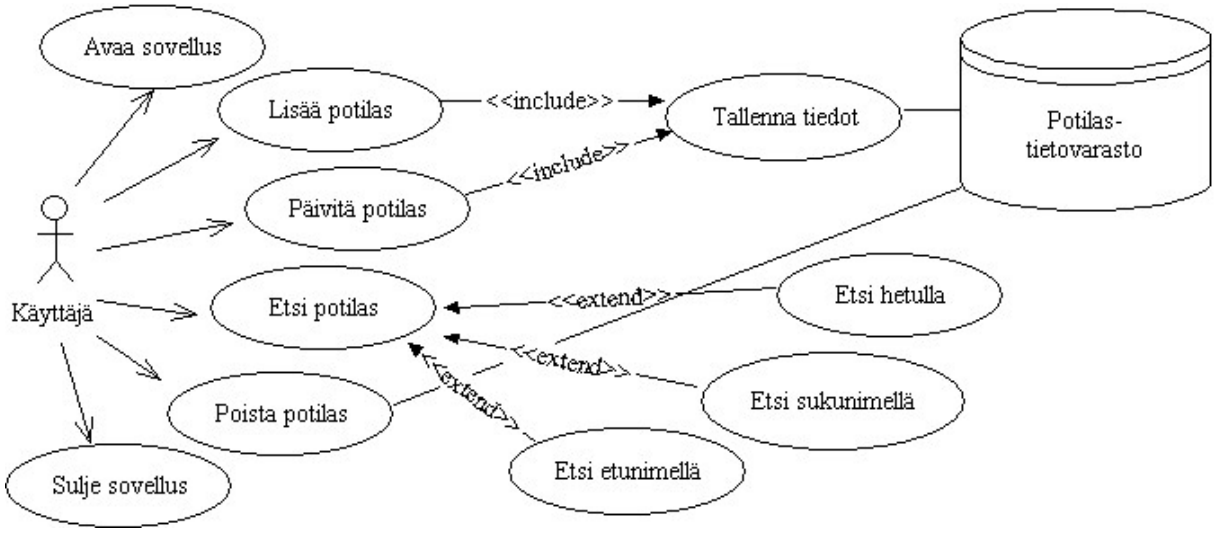
Käyttötapauskaaviossa käytetään seuraavaa notaatiota:

- *Toimija* (actor) on henkilö tai järjestelmä, joka toimittaa tietoa kohteena olevaan järjestelmään tai vastaanottaa tietoa.
- *Käyttötapauksen ilmentymä* (use case) kuvaa toimijan suorittaman toiminnan.
- Yhteys käyttötapausten välillä:

- *Sisältymissuhde* (include) tarkoittaa sitä, että käyttötapaus käyttää toista käyttötapausta toiminnassaan hyväkseen. Esimerkiksi *Lisää potilas* -käyttötapaus voisi käyttää *Tallenna* -käyttötapausta.
- *Laajennussuhde* (extend) tarkoittaa sitä, että käyttötapaus laajentaa toisen käyttötapauksen toimintaa. Esimerkiksi *Etsi nimellä* -käyttötapaus lisää ominaisuuksia *Etsi potilas* -käyttötapauksen toimintaan.
- *Luokittelusuhde* (classification, is kind of) tarkoittaa sitä, että samankaltaiset käyttötapaukset voidaan ryhmitellä kuuluvaksi saman yläkäyttötapauksen alle.

4.2.4.2 Esimerkki käyttötapauskaavion käytöstä

Potilasrekisteri –esimerkin käyttötapaukset (Kuva 17) ovat 1) Lisää potilas 2) Etsi potilas 3) Päivitä potilas, 4) Poista potilas, 5) Avaa sovellus ja 6) Sulje sovellus. Näiden lisäksi järjestelmän suunnittelija huomaa pian, että tietojen tallennusta tarvitaan sekä *Lisää potilas* - että *Päivitä potilas* –toiminnoissa. Näin ollen tietojen tallennusta varten tehdään *Tallenna tiedot* –käyttötapaus, joka sisältyy (include-suhde) *Lisää potilas* - ja *Päivitä potilas* –käyttötapauksiin.



Kuva 17: Käyttötapauskaavio Potilasrekisteri-sovellukselle

4.2.4.3 Käyttötapausten kuvaukset ja testitapaukset

Seuraavaksi tarkastellaan esimerkkiä Lisää potilas –käyttötapauksesta:

Käyttötapaus: 1. Lisää potilas

Tarkoitus: Potilaan tiedot lisätään tietokantaan.

Toimijat: Käyttäjä, joka on vastuussa potilastietojen kirjaamisesta.

Kuvaus: Käyttäjä kirjaa Lisää potilas –lomakkeelle tarvittavat henkilön tiedot ja tallentaa ne.

Esiehdot: Käyttäjä on valinnut päävalikosta Lisää potilas –toiminnon.

Loppuehdot: Potilastiedot on tallennettu tietokantaan oikein ja järjestelmässä on avoinna Lisää potilas -lomake.

Perustoiminta:

1. Käyttäjä syöttää henkilötunnuksen, sukunimen, etunimet, lähiosoitteen, postiosoitteen ja puhelinnumeron.
2. Käyttäjä tallentaa tiedot Tallenna-painikkeella.
3. Järjestelmä tallentaa tiedot tietokantaan.

Poikkeustoiminta:

a) Käyttäjä on jättänyt syöttämättä tietoa pakolliseen kenttään. Ohjelma ilmoittaa, että pakollinen kenttä on tyhjä.

b) Käyttäjä on syöttänyt virheellisen henkilötunnuksen. Ohjelma ilmoittaa, että henkilötunnuksen tulee olla muotoa ppkkvv-xxxx.

c) Käyttäjä on syöttänyt ei-sallitun merkin johonkin kenttään. Ohjelma ilmoittaa ei-sallitusta syötteestä.

d) Käyttäjä yrittää tallentaa potilaan, jolla on sama henkilötunnus kuin tietokannassa olevalla henkilöllä. Ohjelma ilmoittaa, että potilas on jo olemassa.

Lisää potilas –käyttötapauksen kuvauksesta saadaan seuraavia testitapauksia. Taulukossa 7 esitettyjen testien lisäksi henkilötunnuksesta testataan loppuosa ja muoto sekä kaikkia kenttiä testataan normaaleilla syötteillä, erikoismerkeillä ja jättämällä kentät tyhjiksi. Testitapaukset 1-3 testaavat käyttötapauksen perustoimintaa ja testitapaukset 4-11 poikkeustoimintaa. Tässä huomataan, että testitapausten suunnittelu ja kirjaaminen vie paljon aikaa jo yksinkertaisenkin ohjelman tapauksessa. Ongelma korjaantuu, kun ohjelmista tehdään paremmin testattavia. Lisäksi testitapaukset kannattaa tallentaa testitietovarastoon (kts. s. 28) uudelleenkäyttöä varten. Näin menetellen niitä ei tarvitse aina suunnitella uudelleen.

Taulukko 7 : Lisää potilas –käyttötapauksen testitapaukset

Id	Esiehdot	Syöte	Oletettu tulos
1	Lisää potilas – lomake avattu	hetu=041076-0876 sukunimi=Testaaja etunimi=Taavi osoitetiedot=tyhjä	Potilas on lisätty järjestelmään (tarkista myös tietokannan tauluista)
2	-”-	hetu=121203A0876 sukunimi=Testaaja etunimi=Tauno	Potilas on lisätty järjestelmään
3	-”-	hetu=010199+3346 sukunimi=Testaaja etunimi=Taneli	Potilas on lisätty järjestelmään
4	-”-	hetu=tyhjä ja muut kentät ok .	Pakollinen kenttä puuttuu
5	-”-	Sukunimi=tyhjä ja muut kentät ok	Pakollinen kenttä puuttuu
6	-”-	Etunimi=tyhjä ja muut kentät ok	Pakollinen kenttä puuttuu
7	-”-	hetu=**78**1276-0639, pp-virhe	Henkilötunnus virheellinen
8	-”-	hetu=06**13**76-0639, kk-virhe	Henkilötunnus virheellinen
9	-”-	hetu=3103**xw**-0639, vv-virhe	Henkilötunnus virheellinen
10	-”-	Sukunimi=@!&	Virheellinen syöte
11	-”-	etunimi=<img src	Virheellinen syöte

Lisää potilas –käyttötapauksen testitapauksia on käsitelty myös PlugIT-Teho -projektin tutkimuksessa, jossa testitapaukset etenevät askelittain eteenpäin *testitapausjonona*. *Testausaskel* (step) testaa yhden osan toimintoketjusta. Ensimmäisen askeleen tulosta käytetään toisen askeleen syötteenä. Seuraavana on esimerkki Lisää potilas -käyttötapauksen peräkkäisistä askeleista (Liite 1, s. 5-6):

1. Syötä potilaan nimi, hetu, osoite ja oireet. (Tulos 1: syötetty teksti näkyy kentissä).
2. Paina Tallenna-painiketta. (Tulos 2: Haluatko tallentaa tiedot? –dialogi avautuu).
3. Paina Kyllä-painiketta. (Tulos 3: Tiedot tallennettu –dialogi avautuu).

Uusi testausaskel alkaa aina siitä tilasta, mihin järjestelmä on edellisen askeleen jäljiltä jäänyt. Toimintoja ei ole järkevää eikä edes mahdollista testata päinvastaisessa jär-

jestyksessä. Hyvin suunniteltujen testausaskelien avulla testaukselle löydetään selkeä järjestys, joka nopeuttaa testaajan työtä. Testitapaus voidaan jakaa alitestitapauksiin tai se voi koostua useista askelista. Liitteen 1 esimerkissä Lisää potilas -testitapaus on jaettu seuraaviin alitestitapauksiin: a) potilaan tietojen onnistuneeseen tallentamiseen, b) potilaan tietojen tallennuksen peruuttamiseen ja c) potilaan tietojen epäonnistuneeseen tallentamiseen. Lisää potilas -testitapaus on suoritettu valmiiksi, kun kohdat a, b ja c on kaikki testattu.

Seuraavaksi tarkastellaan Etsi potilas –käyttötapausta:

Käyttötapaus: 2. Etsi potilas

Tarkoitus: Potilaan tiedot etsitään tietokannasta.

Toimijat: Käyttäjä, joka haluaa etsiä potilaan tiedot.

Kuvaus: Käyttäjä etsii potilaan tiedot valitsemallaan hakuehdolla.

Esiehdot: Käyttäjä on valinnut päävalikosta Etsi potilas –toiminnon.

Loppuehdot: Potilaan tiedot tulostuvat näytölle ja järjestelmässä on avoinna Lisää potilas –lomake.

Perustoiminta:

1. Käyttäjä syöttää hakuehdoksi henkilötunnuksen, sukunimen tai etunimen ja painaa Etsi-painiketta.
2. Järjestelmä etsii tietokannasta hakuehtoja vastaavan potilaan tiedot
3. Haetun potilaan tiedot tulostuvat näytölle

Vaihtoehtoistoiminta:

a) Hakuehtoja vastaavia potilaita on useita, jolloin järjestelmä tulostaa listan ehtoihin sopivista henkilöistä.

b) Haettavaa potilasta ei löydy, jolloin järjestelmä ilmoittaa, ettei haettua potilasta löytynyt.

c) Etsintä peruutetaan Sulje-painikkeella. Päälomake avautuu.

Poikkeustoiminta:

a) Käyttäjä on syöttänyt hakukenttään virheellisen henkilötunnuksen. Ohjelma ilmoittaa, että henkilötunnuksen tulee olla muotoa ppkkvv-xxxx.

b) Käyttäjä on syöttänyt ei-sallitun merkin johonkin hakukenttään. Ohjelma ilmoittaa ei-sallitusta syötteestä.

Etsi potilas –käyttötapaukselle laaditut testitapaukset on kuvattu taulukossa 8. Testitapaukset 1-3 testaavat käyttötapauksen perustoimintaa, 4-6 vaihtoehtoistoimintaa ja 7-8 poikkeustoimintaa.

Taulukko 8: Etsi potilas -käyttötapauksen testitapaukset

Id	**Esiehdot** Etsi potilas –lomake avattu.	**Syöte**	**Oletettu tulos**
1	Potilas, jolla hetu = 041076-6676 on tietokannassa	Hakukriteerinä on hetu=041076-6676	Tiedot löytyvät
2	Potilas nimeltä Markkanen on tietokannassa	Hakukriteerinä on sukunimi=Markkanen	Tiedot löytyvät
3	Potilas nimeltä Jussi on tietokannassa	Hakukriteerinä on etunimi=Jussi	Tiedot löytyvät
4	Potilasta ei ole tietokannassa	Hetu=121299-065B	Tiedot eivät löydy
5	2 potilasta nimeltä Markkanen on tietokannassa	Hakukriteerinä on sukunimi=Markkanen	Tulostaa listan sopivista henkilöistä
6	Etsi potilas-lomake on avoinna	Paina Etsi potilas –lomakkeen Sulje-painiketta	Päälomake avautuu.
7	Markkanen tallennettu tietokantaan	Hakukriteerinä on sukunimi=Markka#nen	Virheilmoitus

Seuraava tarkasteltava käyttötapaus on Päivitä potilas:

Käyttötapaus: 3. Päivitä potilas

Tarkoitus: Tietokannassa olevan potilaan tietoja päivitetään.

Toimijat: Käyttäjä, joka haluaa päivittää potilaan tietoja.

Kuvaus: Käyttäjä etsii päivitettävän potilaan, muokkaa tietoja ja tallentaa ne lopuksi.

Esiehdot: Käyttäjä on valinnut päävalikosta Päivitä potilas –toiminnon.

Loppuehdot: Potilaan tiedot on päivitetty ja järjestelmässä on avoinna Lisää potilas –lomake.

Perustoiminta:

1. Käyttäjä syöttää hakuehdoksi henkilötunnuksen, sukunimen tai etunimen ja painaa Etsi-painiketta.

2. Haetun potilaan tiedot tulostuvat näytölle.

3. Käyttäjä päivittää tarvittavat tiedot.

4. Käyttäjä tallentaa tiedot tietokantaan painamalla Tallenna-painiketta

Poikkeustoiminta:

Samat kuin Lisää potilas –käyttötapauksessa.

Päivitä potilas -käyttötapauksesta muodostetut testitapaukset (Taulukko 9) ovat osittain samoja kuin Lisää potilas -käyttötapauksessa. Molemmat käyttötapaukset on silti testattava erikseen. Testitapaus 1 testaa käyttötapauksen poikkeustoimintaa ja testitapaukset 2-5 perustoimintaa. Henkilötunnuksen päivittäminen on mahdollista potilashallinnon järjestelmissä yleensä vain luomalla uusi potilas uudella henkilötunnuksella ja yhdistämällä väärä tai tilapäinen henkilötunnus uuteen potilaaseen.

Taulukko 9: Päivitä potilas -käyttötapauksen testitapaukset

Id	Esiehdot	Syöte	Oletettu tulos
1	Lisää potilas – lomake avattu	Sukunimi=Testaaja etunimi=Taavi osoitetiedot=tyhjä	Potilastiedot on päivitetty järjestelmään (tarkista myös tietokannan tauluista)
2	-”-	Hetu=tyhjä	Päivitys ei onnistu
3	-”-	Sukunimi=tyhjä ja muut kentät ok	Pakollinen kenttä puuttuu
4	-”-	Etunimi=tyhjä ja muut kentät ok	Pakollinen kenttä puuttuu
5	-”-	Sukunimi=@!&	Virheilmoitus: Virheellinen sukunimi

Käyttötapaus: 4. Poista potilas

Tarkoitus: Tietokannassa olevan potilaan tiedot poistetaan.

Toimijat: Käyttäjä, joka haluaa poistaa potilaan tietoja

Kuvaus: Käyttäjä etsii päivitettävän potilaan ja poistaa sen tietokannasta.

Esiehdot: Käyttäjä on valinnut päävalikosta Poista potilas –toiminnon

Loppuehdot: Potilaan tiedot on poistettu

Perustoiminta:

1. Käyttäjä syöttää hakuehdoksi henkilötunnuksen, sukunimen tai etunimen ja painaa Etsi-painiketta.

2. Haetun potilaan tiedot tulostuvat näytölle

3. Käyttäjä painaa Poista-painiketta

4. Järjestelmä kysyy varmistusta poistoon. Myöntävä vastaus poistaa potilaan.

Poikkeustoiminta:

Potilasta ei haluta poistaa. Varmistuskyselyssä vastataan Ei.

Poista potilas -käyttötapausta testattaessa (Taulukko 10) tarkistetaan tietokannasta poistuiko potilas oikeasti, sillä pelkästään järjestelmän antamiin ilmoituksiin ei pidä luottaa. Yleensä poistamistoimintoa ei ole toteutettu terveydenhuollon tietojärjestelmiin, koska historiatietoja potilaan tietojen muutoksista ei saa muuttaa. Testitapaus 1 testaa käyttötapauksen perustoiminnan ja 2 poikkeustoiminnan.

Taulukko 10: Poista potilas -käyttötapauksen testitapaukset

Id	Esiehdot	Syöte	Oletettu tulos
1	Päivitä potilaan tiedot –lomake avoinna	Paina Poista-painiketta ja poiston varmistuskyselyssä Kyllä	Potilas on poistettu järjestelmästä.
2	-”-	Paina Poista-painiketta ja poiston varmistuskyselyssä Ei	Potilaan tiedot säilyvät järjestelmässä.

4.2.4.4 Käyttötapauskaavion merkitys testauksen kannalta

Käyttötapausten kuvaus kuvaa ohjelman peruskäyttötilanteen (normal flow), vaihtoehtoisen toiminnan (alternative flow) ja poikkeustoiminnan (exceptional flow). Testauksessa käydään läpi ohjelman jokainen käyttötapaus laajennus- ja sisältymissuhteineen ja kustakin käyttötapauksesta edellä mainitut perus-, vaihtoehtoinen- ja poikkeustoiminta.

Jacobson [Jac92, s.331] esittää käyttötapausten olevan hyödyllinen apuväline ohjelmistojen integrointitestaukseen, koska ne yhdistävät korkealla tasolla eri luokat tai ohjelman osat yhteen. Käyttötapauskaaviot eivät kuitenkaan yksinään tarjoa riittävän tarkkaa informaatiota järjestelmän toiminnoista testauksen avuksi. Huolellisesti laadituista käyttötapauksen kuvauksista testaaja saa johdettua hyviä testitapauksia liittyen

järjestelmän käyttäytymiseen poikkeustilanteissa. Tarkkojen määrittelykuvausten ansiosta ohjelmiston virheiden korjaus myöhemmässä vaiheessa vähentyy, kun poikkeuksiin ja virhetilanteisiin varaudutaan jo suunnitteluvaiheessa.

4.3 Suunnittelu- ja toteutusvaiheen UML-kaaviot

4.3.1 Sekvenssikaaviot

4.3.1.1 Merkitys ja notaatio

Sekvenssikaaviot kuvaavat luokkien, komponenttien tai olioiden välistä vuorovaikutusta. Yhdestä käyttötapauksesta saadaan monta sekvenssikaaviota, joista jokainen on käyttötapauksen yksi suorituspolku. Koska UML on kehitetty oliopohjaiseen suunnitteluun, olioiden välistä kommunikointia kutsutaan viesteiksi (message). Sekvenssikaaviossa oliot on järjestelty horisontaalisesti ja suoritusaika vertikaalisesti.

Sekvenssikaaviossa käytetään seuraavaa notaatiota [Bin00, s. 286]:

- *Oliot* (object) ovat luokkien ilmentymiä. Kaaviossa luokkaa kuvataan suorakulmiolla ja olion nimi on kirjoitettu luokan sisälle. Oliot voidaan korvata komponenteilla.
- *Toimijat* (actor) voivat kommunikoida olioiden/komponenttien kanssa. Niitä kuvataan tikku-ukoilla.
- *Elämälinja* (lifeline) määrittää olion/komponentin olemassaoloajan. Linjaa kuvataan vertikaalisella katkoviivalla.
- *Aktivointi* (activation) kuvaa, milloin olio/komponentti suorittaa toimintaa.
- *Viestit* (message) ovat viestejä olioiden/komponenttien välillä. Viestejä kuvataan vaakasuorilla nuolilla.

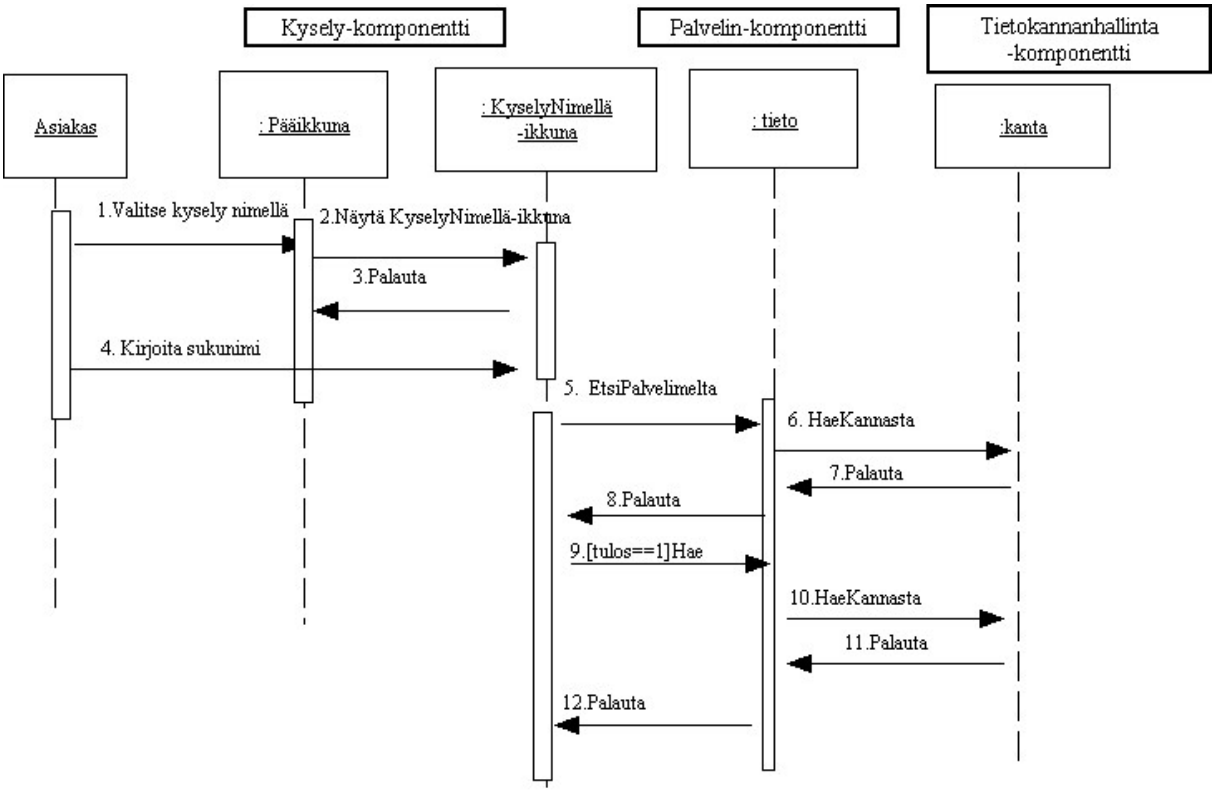
Sekvenssikaavioon voidaan liittää myös komponentit olioiden yläpuolelle osoittamaan, mihin komponenttiin kukin olio kuuluu.

4.3.1.2 Esimerkki sekvenssikaavion käytöstä

Kuvan 18 sekvenssikaavio on mukaeltu esimerkki, joka esittää potilaan tietojen etsimistä nimen perusteella Kysely- ja Palvelin-komponenttien välillä normaalitapauksessa eli kun löydetään yksi hakua vastaava potilas [YoC99]. Alkuperäiseen esimerkkiin

on lisätty Tietokannanhallinta-komponentti, joka vastaa tässä tapauksessa tietojen hakua tietokannasta.

- Asiakas valitsee ensiksi KyselyNimellä-toiminnon (1) ja
- Kysely-komponentti näyttää pääikkunassa KyselyNimellä-ikkunan (2 ja 3).
- Asiakas kirjoittaa KyselyNimellä-ikkunaan potilaan sukunimen (4), ja
- Kysely-komponentti välittää etsintäpyynnön Palvelin-komponentille (5).
- Palvelin-komponentti välittää hakupyynnön Tietokannanhallinta-komponentille (6), joka on yhteydessä tietokantaan ja joka palauttaa löytyneet tiedot takaisin Palvelin-komponentille (7),
- Palvelin-komponentti palauttaa tiedot edelleen Kysely-komponentille (8).
- Mikäli hakutuloksia löytyy vain yksi, Kysely-komponentti hakee potilaan kaikki tiedot Palvelin-komponentilta (9) ja edelleen Tietokannanhallinta-komponentilta (10), joka palauttaa kaikki tiedot takaisin Palvelin-komponentille (11).
- Palvelin-komponentti palauttaa lopuksi kyselyn tulokset Kysely-komponentin ja asiakkaan nähtäväksi (12).



Kuva 18: Sekvenssikaavio komponenttien tiedonvälityksestä

4.3.1.3 Sekvenssikaavion merkitys testauksen kannalta

Kysely- ja Palvelin-komponenttien testauksessa [YoC99] testaus kohdistetaan normaalin suorituspolun lisäksi poikkeustoimintaan eli virhetilanteita voi tapahtua ainakin, kun a) hakutuloksia ei löydy yhtään b) hakutuloksia on enemmän kuin yksi ja käyttäjä valitsee listasta potilaan.

Sekvenssikaavio osoittaa viestipolut olioiden tai komponenttien välillä aikaulottuvuudessa, mikä antaa hyödyllisen kuvauksen olioiden välisestä yhteistyöstä. Se on kuitenkin heikko malli verrattuna *kontrollikaavion* [Paa00, s.39] pohjalta muodostettuihin testitapauksiin, koska siinä näytetään vain yksittäinen toimintaskenaario. Kontrollikaavio kuvaa ohjelman kaikki suorituspolut ja mahdollistaa ehto- ja toistorakenteiden esittämisen. Kontrollikaavio edellyttää koodin olemassaoloa eli se sopii vain lasilaatikkotestaukseen. Yhden käyttötapauksen toteuttaminen vaatii useita skenaariokaavioita. Sekvenssikaavion puutteita ovat muun muassa seuraavat [Bin00, s. 292]:

- Monimutkaisen toiminnon esittäminen on hankalaa, koska valintojen ja toistojen notaatio on kömpelöä.
- Erottelu ehdollisen (conditional) viestin ja viivästetyn (delayed) viestin välillä on heikko.
- Dynaamista sitomista ja yliluokka-aliluokka-käyttäytymistä ei voida esittää suoraan (viestin sitominen eri luokkaan täytyy näyttää eri kaaviossa).

Sekvenssikaavion etuna on olioiden tai komponenttien välisten viestien (metodit ja parametrit) kuvaaminen aikajärjestyksessä. Sekvenssikaavio osoittaa mitkä oliot tai komponentit ovat viestinnässä aktiivisia ja passiivisia (osallistuvat /eivät osallistu viestintään). Taulukko 11 sisältää sekvenssikaaviolle tehdyt testitapaukset.

Taulukko 11: Testitapaukset sekvenssikaaviolle

Id	Esiehdot	Syöte	Oletettu tulos
1	Päälomake avoinna.	Valitse Kysely nimellä –toiminto	KyselyNimellä-ikkuna avautuu
2	Tietokannassa on Taavi Testaaja. KyselyNimellä-ikkuna on auki.	Sukunimi = Testaaja. Paina Ok.	Ohjelma palauttaa haun tuloksena Taavi Testaajan
3	Tietokannassa on useita Testaaja-nimisiä henkilöitä. KyselyNimellä-ikkuna on auki.	Sukunimi = Testaaja. Paina Ok.	Ohjelma palauttaa haun tuloksena luettelon kaikista Testaaja-nimisistä.
4	Luettelo Testaaja-nimisistä potilaista on näytöllä.	Valitse Taavi Testaaja hakutuloksista.	Ohjelma palauttaa Taavi Testaajan kaikki tiedot näkyviin.
5	Tietokannassa ei ole Tuija Testeri-henkilöä. KyselyNimellä-ikkuna on auki.	Sukunimi = Tester. Paina Ok.	Ohjelma ilmoittaa, että henkilöä ei löydy.

4.3.2 Yhteistyökaaviot

4.3.2.1 Merkitys ja notaatio

Yhteistyökaavio (Collaboration Diagram) kuvaa olioiden välistä vuorovaikutusta. Tämä kaavio on ikään kuin oliokaavion ja sekvenssikaavion risteytys. Sekvenssikaavioissa vuorovaikutusta kuvataan rivi-sarake periaatteella, kun taas yhteistyökaaviossa oliot on vapaassa järjestyksessä oliokaavion tapaan. Tästä johtuen yhteistyökaavion olion sisältämät kaikki vuorovaikutussuhteet on helpompi havaita. Yhteistyökaavio voi sisältää sekä *luokittelijarooleja* (classifierRole) että *assosiaatiorooleja* (associationRole). Luokittelijaroolit edustavat mitä tahansa oliota tai luokkaa. Assosiaatioroolit kuvaavat yhteyttä luokittelijaroolien välillä.

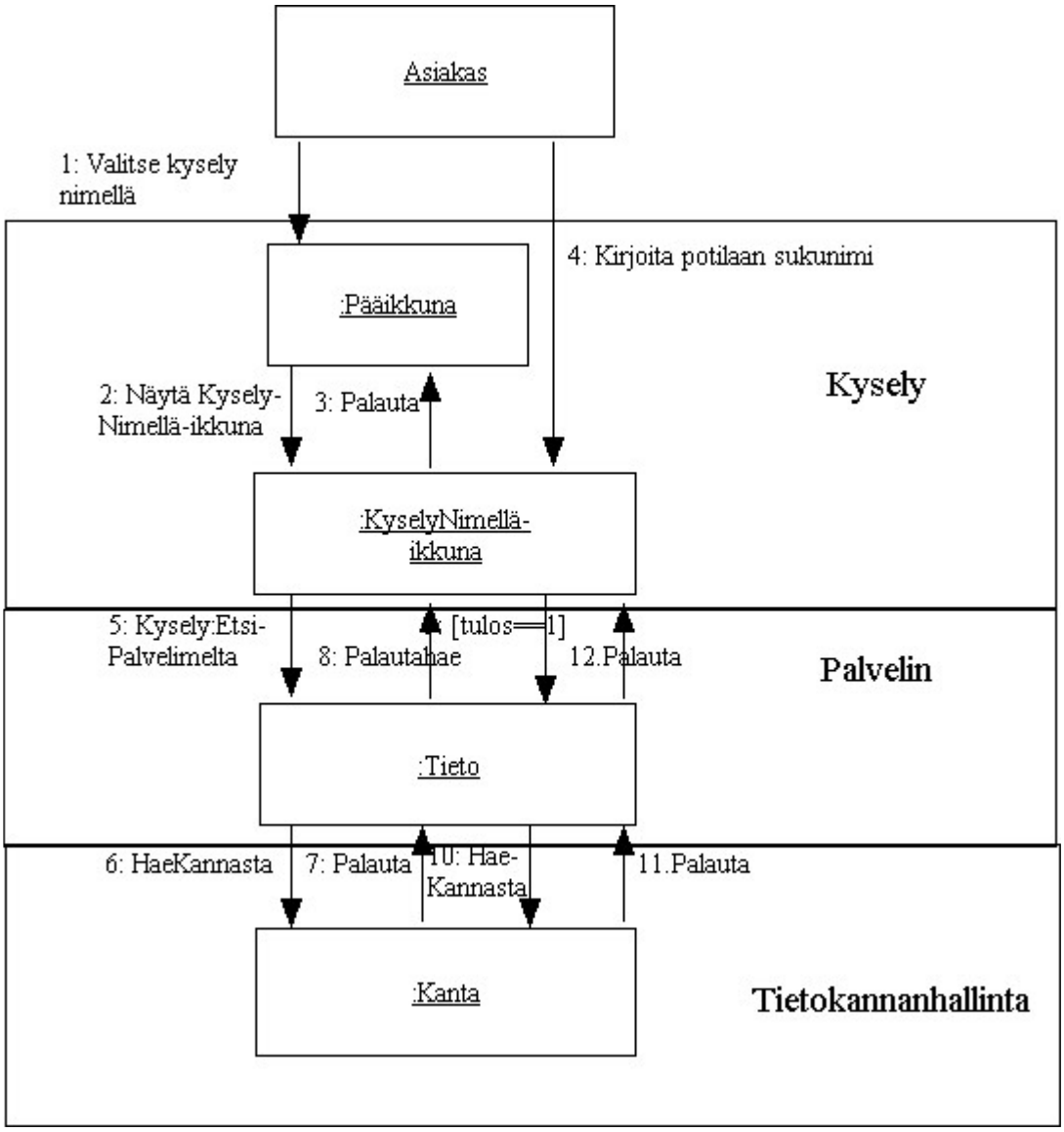
Yhteistyökaaviossa käytetään seuraavaa notaatiota:

- *Oliot* ovat luokan ilmentymiä ja voivat osallistua kommunikointiin toisten olioiden tai toimijoiden välillä. Olioita kuvataan suorakulmion muotoisilla laatikoilla.

- *Toimijat* voivat viestiä olioiden kanssa. Toimijoita kuvataan laatikoilla tai tikkuukoilla.
- *Viestejä* mallinnetaan nuolilla, jotka sisältävät myös viestin järjestysnumeron.

4.3.2.2 Esimerkki yhteistyökaavion käytöstä

Kuvassa 19 on esimerkki yhteistyökaavion käytöstä samasta sovelluksesta kuin sekvenssikaavion kohdalla kappaleessa 4.3.1.2. Pääikkuna- ja KyselyNimellä-oliot kuuluvat Kysely-komponentille, Tieto-olio kuuluu Palvelin-komponentille ja Kanta-olio Tietokannanhallinta-komponentille.



Kuva 19: Yhteistyökaavio

4.3.2.3 Yhteistyökaavion merkitys testauksen kannalta

Yhteiskaavion etuna testauksen kannalta on se, että se kuvaa sekvenssikaavion tavoin olioiden tai komponenttien välistä vuorovaikutusta. Yhteistyökaaviossa ei ole otettu huomioon vuorovaikutuksen aikajärjestystä muuten kuin viestien järjestysnumeroilla. Ainakin tämän tutkielman esimerkissä (Kysely-, Palvelin- ja Tietokannanhallinta-komponentit) sekvenssikaavio tarjoaa selkeämmän kuvan vuorovaikutuksesta kuin yhteistyökaavio, koska tapahtumien seuraaminen aikajärjestyksessä on helpompaa sekvenssikaaviosta kuin yhteistyökaaviosta.

Binder esittää seuraavia ongelmia yhteistyökaavioiden käytössä testauksen kannalta [Bin00, s. 300]:

- Kaavio esittää vain yhden palasen ohjelmistosta. Useimmissa tapauksissa ollaan kiinnostuneita koko toteutuksen testaamisesta.
- Kaavio esittää hyvin korkean tason sovellusriippumattomia vaatimuksia (rooleja) ja toteutuskohtaisia yksityiskohtia (kutsupolut, palautusarvot). Vaarana on se-kaannus abstraktion ja toteutuksen välillä.

Yhteistyökaavio saattaa olla havainnollisempi tilanteessa, jossa olioita tai komponentteja on paljon eikä tapahtumien aikajärjestyksellä ole niin paljon merkitystä. Yhteistyökaaviot osoittavat testausta varten, mitä olioita tai komponentteja järjestelmään kuuluu ja missä järjestyksessä viestit tapahtuvat. Koska yhteistyökaavion ja sekvenssikaavion erot ovat vain olioiden järjestyksen esittämisessä, niistä saadaan samoja testitapauksia (kts. Taulukko 11).

4.3.3 Luokkakaaviot

Luokkakaavioita ja niiden merkitystä käydään läpi tässä tutkielmassa vain rajoitetusti. Olio-ohjelmien testausta käsitellään tutkielmassa [Par03].

4.3.3.1 Merkitys ja notaatio

Luokkakaaviot mallintavat järjestelmän luokkarakennetta sisältäen jokaisen luokan ominaisuudet ja metodit sekä luokkien väliset suhteet kuten periytymisen. Luokkakaavio on yksi eniten käytetyistä UML-kaavioista ohjelman rakenteen kuvaamisessa, mutta testauksen kannalta se ei välttämättä tarjoa riittävästi tietoa järjestelmän suuori-

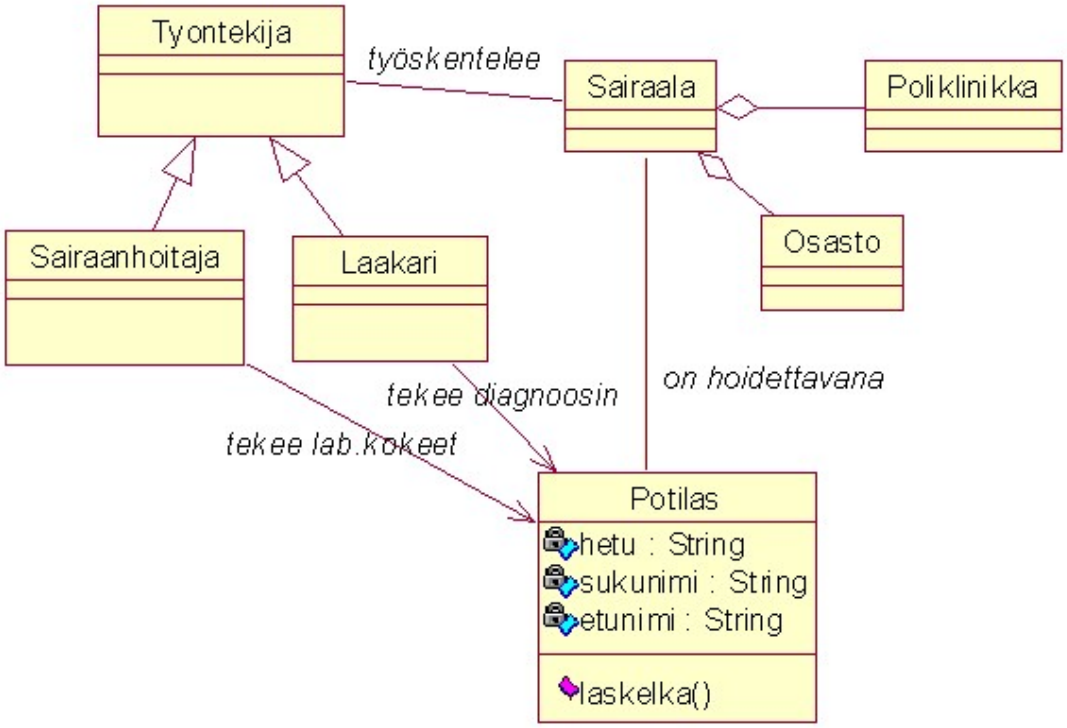
tuksesta. Case-työkaluilla on mahdollista generoida automaattisesti ohjelmakoodia luokkakaavioiden pohjalta esimerkiksi Java-, C++ tai C# -kielille, mutta on tärkeää huomata, että automaattinen koodigeneraattori muodostaa vain kehykset luokille eikä toimintalogiikkaa. Kun ohjelmakoodi on käytössä, sille voidaan tehdä koodikattavuustestejä, silmukkatestejä ja muita yksikkötestauksen kuuluvia testitapauksia.

Luokkakaaviossa käytetään seuraavaa notaatiota:

- *Luokka* on oliosuuntautuneen ohjelmoinnin peruskäsite ja sitä kuvataan kolmeen osaan jaetulla suorakulmiolla.
- *Assosiaatio* kahden luokan välillä tarkoittaa sitä, että molempien täytyy toteuttaa tietty rajoite tai riippuvaisuus. Assosiaatio voi kuvata myös suhteen tyypin (esim. yhden suhde moneen 1..n).
- *Koostumisessa* (aggregation) tunnistetaan luokat, jotka ovat kiinteästi toiseen luokkaan kuuluvia osia (esim. sairaala koostuu poliklinikoista ja osastoista).
- *Yleistäminen* (generalization) tarkoittaa yliluokka-aliluokkasuhdetta (lääkäri on työntekijän aliluokka).

4.3.3.2 Esimerkki luokkakaavion käytöstä

Kuvassa 20 esitetty luokkakaavio kuvaa teoreettisella tasolla, mitä tekijöitä, käsitteitä ja käsitteiden välisiä suhteita sairaalaympäristöstä voisi löytyä. Lääkäri ja sairaanhoitaja ovat molemmat eräänlaisia työntekijöitä eli niiden välillä vallitsee yleistämissuhde.



Kuva 20: Esimerkki luokkakaaviosta

Kuvan 20 luokkakaavion perusteella voidaan laatia seuraavia testitapauksia ja kysymyksiä:

- Onko sairaalalle mahdollista luoda osastoja ja poliklinikoita?
- Voiko sairaalan luoda ilman osastoa tai poliklinikkaa?
- Miten sairaanhoitaja eroaa ohjelman toiminnassa lääkäristä? Miten käyttöoikeudet testataan?
- Voiko potilaan tiedot tallentaa ilman lääkärin tietoja?

4.3.3.3 Luokkakaavion merkitys testauksen kannalta

Luokkakaavio tarjoaa mahdollisuuksia havaita ohjelman olioiden riippuvaisuuksia kuten luokkien välinen periytyminen, assosiaatiot ja osa-kokoonpano-suhteet. Luokkakaavio piirretään valitettavan usein epätäydellisenä, mikä voi aiheuttaa vääriä tulkintoja ohjelman toiminnasta [BaL02]. Luokkakaavio voidaan piirtää kolmesta eri näkökulmasta (FoS00]:

– Käsitteellinen luokkakaavio kuvaa reaalimaailman käsitteitä ja niiden välisiä suhteita asiakkaan ymmärtämällä kielellä. Kuvaus on ohjelmointikieliriippumaton ja

tehdään asiakkaalle tutuilla käsitteillä. Esimerkiksi käsitteellä *potilas* on ominaisuutena (attribuuttina) *potilasnimi*.

- Määrittelevä luokkakaavio kuvaa järjestelmää olioiden käyttöliittymän avulla. Olio osaa kertoa attribuuttinsa arvon: *potilasnimi = Perttu Potilas*.
- Toteutuskaavio selvittää, miten järjestelmän luokat tullaan toteuttamaan. Toteutusnäkökulmassa attribuuttia edustaa tietyn tyyppinen ja tietyn arvoalueen omistava olion kenttä ja attribuutilla voi olla joku oletusarvo: String *potilasnimi* = "".

Vaikka luokkakaavio on yksi eniten käytetyistä kaavioista ohjelman rakenteen kuvaamisessa, testauksen kannalta se ei välttämättä tarjoa riittävästi tietoa järjestelmän sisäisestä vuorovaikutuksesta, jota tarvitaan toiminnallisuuden testauksessa. Luokkakaavioiden kohdalla testattavuutta voi parantaa sillä, että luokkarakenteessa vähennetään olioiden välistä vuorovaikutusta ja monimutkaisen periytymisen tuomia hankaluuksia käyttämällä *rajapintaluokkia* (interfaces) [BaL02]. Abstrakti rajapintaluokka määrittelee joukon metodeja, mutta ei niiden toteutusta. Luokka voi taas toteuttaa monia eri rajapintoja eli antaa toteutukset kaikille metodeille kaikissa rajapinnoissa, jotka luokka ilmoittaa toteuttavansa. Yksikkötestausvaiheessa luokille ja metodeille voidaan tehdä muistiprofilointia, suorituskykytestausta ja koodikattavuustestausta.

4.3.4 Tilakaaviot

4.3.4.1 Merkitys ja notaatio

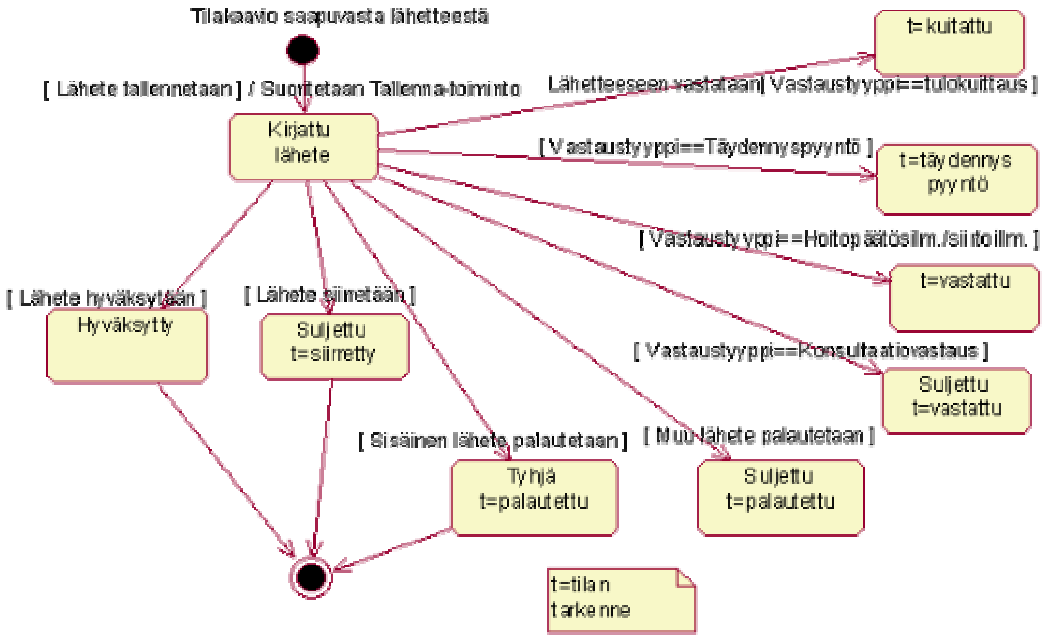
Ohjelmalla voi olla suorituksensa aikana lukuisia tiloja. *UML-tilakaavio* koostuu kolmesta eri tekijästä: tiloista, siirtymistä ja toiminnoista. *Tilalla* tarkoitetaan ohjelman, olion tai komponentin olotilaa tiettynä suoritusajankohtana. Tilakaaviolla kuvataan ohjelmiston tai sen keskeisten olioiden tai komponenttien tiloja. *Tilasiirtymä* kuvaa lähdetilan ja tulostilan suhdetta, joka saa kohteen siirtymään tilasta toiseen [LiZ99]. Tilasiirtymä voi tapahtua myös tilasta takaisin samaan tilaan. Tilasiirtymään liitetään toiminnan lisäksi ehto, jonka täytyy olla voimassa ja joka laukaisee tilasiirtymän. Järjestelmän tiloista voidaan laatia tilasiirtymätaulu (state transition table) tai tilasiirtymäpuu.

Tilakaaviossa käytetään seuraavaa notaatiota:

- *Tila* (suorakulmio) on ohjelmiston tila tiettynä suoritusajankohtana.
- *Siirtymä* (nuoli) kuvaa siirtymää tilojen välillä.
- *Toiminto* on tapahtuma, joka suoritetaan tilasiirtymässä.
- *Ehto* on sääntö, jonka täytyy olla voimassa tilasiirtymän toteutumiseksi.

4.3.4.2 Esimerkki tilakaavion käytöstä

Kuvan 21 tilakaaviossa on esitetty terveydenhuollon organisaatioon saapuneen lähetteen tilamuutoksia eräässä potilashallinnon ydinjärjestelmässä.



Kuva 21: Tilakaavio saapuvalle lähetteelle

Tällä ydinjärjestelmällä voidaan mm. ylläpitää potilaiden tietoja, varata potilaille aikoja sekä kirjata saapuvia ja lähteviä lähetteitä. Lähetteen tila koostuu itse tilasta ja tilan *tarkenteesta (t),* joka on lisämääritelmä tilalle (tilan alitila). Kun saapuva lähete tallennetaan ensimmäisen kerran, sen tilaksi tulee *kirjattu.* Kirjatun lähetteen kohdalla hyväksyminen tarkoittaa myönteistä hoitopäätöstä. Hyväksymisen jälkeen lähetteen tila on *hyväksytty.* Mikäli lähete on tullut väärään paikkaan, se voidaan siirtää oikealle vastaanottajalle, jonka jälkeen tila on *suljettu* ja tilan tarkenne on *siirretty.*

Kun sisäinen lähete palautetaan, tilaksi tulee *tyhjä* ja tarkenteeksi *palautettu.* Muun lähetteen palauttamisen jälkeen tilana on *suljettu* ja tarkenteena *palautettu.* Vastaan-

ottajan vastattua lähetteeseen tarkenteeksi tulee vastaustyypistä riippuen *kuitattu, täydennyspyyntö* tai *vastattu*. Mikäli vastaustyyppinä on ollut konsultaatiovastaus, tila muuttuu *suljetuksi* ja tarkenne *vastatuksi*. Kuvasta 21 on muodostettu seuraava tilasiirtymätaulukko (Taulukko 12).

Taulukko 12 : Tilasiirtymätaulukko saapuneelle lähetteelle

Id	Alkutila	Toiminto	Lopputila
1	Organisaatioon on saapunut lähete.	Paina Tallenna.	Saapuneen lähetteen tila on *kirjattu.*
2	Lähete on *kirjattu.*	Lähetteeseen vastataan ja vastaustyyppinä on Tulokuittaus.	Tilan tarkenne on *kuitattu.*
3	Lähete on *kirjattu.*	Lähetteeseen vastataan ja vastaustyyppinä on Täydennyspyyntö.	Tilan tarkenne on *täydennyspyyntö.*
4	Lähete on kirjattu.	Lähetteeseen vastataan ja vastaustyyppinä on Hoitopäätösilm/siirtoilm.	Tilan tarkenne on *vastattu.*
5	Lähete on *kirjattu.*	Lähetteeseen vastataan ja vastaustyyppinä on Konsultaatiovastaus.	Tila on *suljettu*, tarkenne on *vastattu.*
6	Lähete on *kirjattu.*	Muu lähete (ei sisäinen) palautetaan.	Tila on *suljettu*, Tarkenne on *palautettu.*
7	Lähete on *kirjattu.*	Sisäinen lähete palautetaan.	Tila on *tyhjä*, tarkenne on *palautettu.*
8	Lähete on *kirjattu.*	Lähete siirretään.	Tila on *suljettu*, tarkenne on *siirretty.*
9	Lähete on *kirjattu.*	Lähete hyväksytään (tehdään myönteinen hoitopäätös).	Tila on *hyväksytty.*

4.3.4.3 Tilakaavion merkitys testauksen kannalta

Muihin UML-kaavioihin verrattuna tilakaaviot soveltuvat ehkä parhaiten testausmalliin, koska ne perustuvat tilakoneisiin ja niitä käytetään kuvaamaan olioiden tai komponenttien välistä vuorovaikutusta. Tilakaaviot ovat UML:n näkökulmasta eniten formaaleja ja tarjoavat luonnollisen pohjan testitapausten luomiselle [OfA99]. Tilakaaviot kuvaavat, mitä ehtoja järjestelmän tilan vaihtuminen vaatii ja mitä toimintoja tilan vaihtumisessa tehdään. Toiminnallisessa testauksessa tilakaaviota voi käyttää hyväksi järjestelmän testauksen kattavuusmittarina. Ohjelmaa voidaan *siirtymäkattavuuskriteerin* (transition coverage) mukaan pitää testattuna, kun jokainen tilasiirtymä on käyty läpi vähintään kerran. Aloittelevan testaajan on helppo löytää testitapauksen syötteet tilasiirtymän sisältämistä toiminnoista ja oletetut tulokset lopputiloista.

Monimutkaisten ohjelmien kaikkia syöteyhdistelmiä on mahdotonta käydä läpi kattavasti ilman jonkinlaista järjestelmällistä testausmallia. UML-tilakaaviot ovat yksinkertainen tapa esittää järjestelmän käyttäytymistä eri tiloissa. Tilakaavion lisäksi järjestelmän eri tilat voidaan kuvata tilasiirtymätaulukkona. Testitapausten muodostamista tilakaavioiden pohjalta on käsitelty useissa tutkimustuloksissa. Chevalley ja Thevenod-Fosse ovat tutkimuksessaan muodostaneet testitapaukset UML-tilakaavioista käyttäen siirtymäkattavuutta testauskriteerinä. Heidän tutkimuksensa painottui automatisoidun tavan määrittelemiseen syötteiden ja tulosarvojen tuottamiseksi [ChT02].

4.4 Käyttöönottovaiheen UML-kaaviot

4.4.1 Toimituskaaviot

4.4.1.1 Merkitys ja notaatio

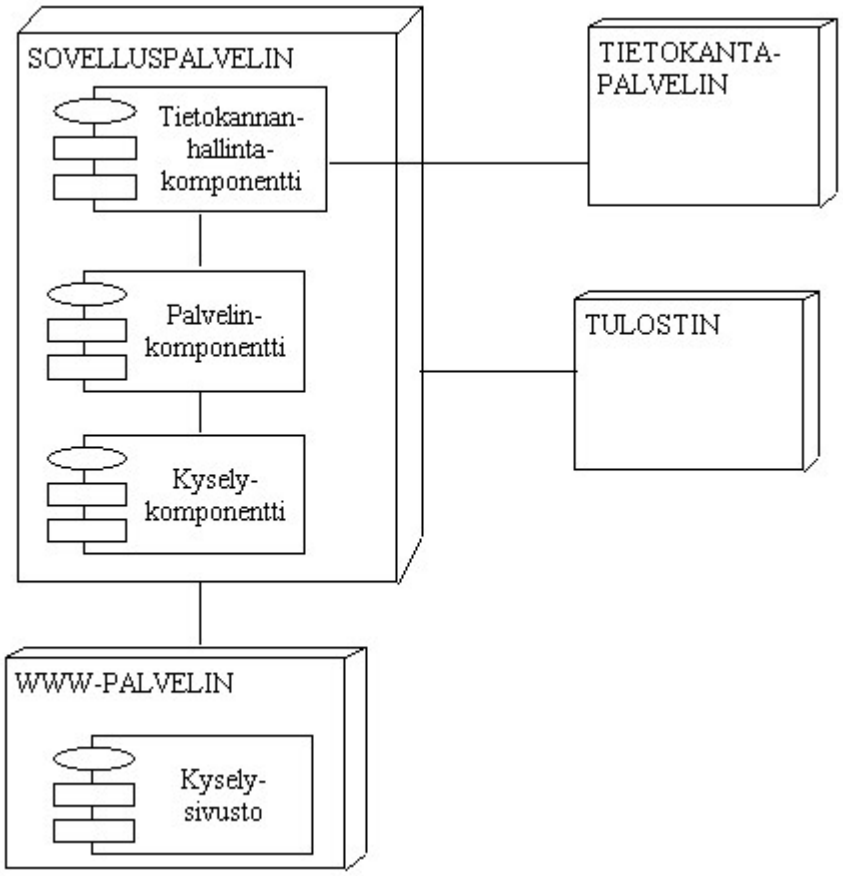
Toteutuskaavioiden ryhmään kuuluvalla *toimituskaaviolla* (deployment diagram) mallinnetaan järjestelmän käyttämää laitteistoa ja laitteistokomponenttien välisiä yhteyksiä. Kaaviolla voidaan osoittaa ohjelmistokomponenttien paikka laitteistossa. Projektin suunnitteluvaiheessa toimituskaavioilla voidaan kuvata järjestelmän arkkitehtuuria.

Toimituskaaviossa käytetään seuraavaa notaatiota:

- *Komponentti* (component) on järjestelmän sisältämä ohjelmiston osa.
- *Solmu* (node) esittää laitteiston osaa ja se kuvataan kolmiulotteisella laatikolla.

4.4.1.2 Esimerkki toimituskaavion käytöstä

Kuvan 22 toimituskaavio esittää ohjelmistokomponenttien paikan fyysiseen laitteistoon nähden. Kaaviosta voidaan nähdä Kyselysivusto-komponentin (www-lomake potilaan hakua varten) sijaitsevan webpalvelimella ja Kysely-, Palvelin- ja Tietokannanhallintakomponenttien olevan sovelluspalvelimella. Näiden lisäksi laitteistoon kuuluvat tietokantapalvelin ja tulostin.



Kuva 22: Toimituskaavio

4.4.1.3 Toimituskaavion merkitys testauksen kannalta

Toimituskaavion pohjalta voidaan suunnitella testitapauksia (Taulukko 13) muun muassa järjestelmätestausvaiheeseen. Järjestelmätestauksessa ohjelmistoa ja sen osia sekä laitteistoa testataan yhtenä kokonaisuutena. Kuvan 22 toimituskaavion perusteella voidaan päätellä, että testitapauksia tarvitaan seuraaville laitteistokomponenttien välisille tietoliikenneyhteyksille: *sovelluspalvelin - tietokantapalvelin*, *sovelluspalvelin - web-palvelin*, *sovelluspalvelin-tulostin.*

Testeissä ei ole tarkoituksena todistaa pelkästään, että yhteydet laitteiden välillä toimivat moitteettomasti normaalitilanteessa, vaan tarkoituksena on tutkia myös järjestelmän todellisia vasteaikoja suurilla käyttäjämäärillä ja testata riittääkö laitteiston suorituskyky käsiteltäessä maksimikapasiteetin rajoilla olevia tietomääriä.

Taulukko 13: Testitapaukset toimituskaaviolle

Id	Esiehdot	Syöte	Oletettu tulos
1	Potilaan tiedot ovat Lisää potilas-lomakkeessa avoinna ja tulostin on asennettu.	Valitse Tulosta	Potilaan tiedot tulostuvat oikein tulostimelle.
2	Etsi potilas -lomake on avoinna. Tietokannassa on -100 potilasta -1000 potilasta -10 000 potilasta, joista vähintään 2 sukunimellä Korhonen.	Sukunimi= Korhonen. Paina Etsi.	Ohjelma palauttaa haun tuloksena kaikki Korhonen-nimiset alle 2 sekunnissa.
3	Lisää potilas-lomake on avoinna. 20 käyttäjää käyttää sovellusta yhtäaikaa	Etunimi = Taavi Sukunimi = Testaaja. Paina Tallenna.	Potilaan tiedot lisätty. Samanaikaiset käyttäjät voivat käyttää järjestelmää ongelmitta
4	Lisää potilas-lomake on avoinna. Sovelluspalvelin on poissa käytöstä.	Etunimi = Taavi Sukunimi =Testaaja. Paina Tallenna.	Virheilmoitus
5	Lisää potilas-lomake on avoinna. Tietokantapalvelin on poissa käytöstä.	Etunimi = Taavi Sukunimi =Testaaja. Paina Tallenna.	Virheilmoitus

5 Rational Softwaren testaustyökalut

Tässä luvussa esitellään tarkemmin vaatimusten hallintaan ja testaukseen sopivia työvälineitä, jotka kuuluvat Rational Softwaren tuotteisiin.

5.1 RequisitePro vaatimusmäärittelyssä ja Rose mallintamisessa

Vaatimusten hallinnassa on mahdollista käyttää Rational RequisitePro -ohjelmaa, jolla voidaan lisätä ja hallita eri tyyppisiä vaatimuksia. RequisitePro ei ole itse varsinainen testaustyökalu, mutta sillä laaditut vaatimukset voidaan yhdistää testauksen hallintaohjelmassa testitapauksiin. RequisitePro toimii yhdessä MS-Wordin kanssa ja vaatimusten lisääminen tapahtuu valitsemalla Word-dokumentista käyttäjän kirjaama vaatimusteksti ja määrittelemällä vaatimustyyppi.

Vaatimuksille voidaan antaa attribuutteja kuten prioriteetti ja vaatimusten välille voidaan muodostaa riippuvuussuhteita (esimerkiksi vaatimus 1 seuraa vaatimusta 2 tai vaatimus 1 sisältää vaatimukset 2 ja 3). Ohjelmalla voi tämän jälkeen tarkastella, mikä käyttötapaus ratkaisee minkäkin ongelman. Vaatimuksesta jää myös muutoshistoria talteen.

Rational Rose on ohjelmistotuotantoon suunniteltu visuaalinen mallinnus- ja kehittämistyökalu, jota voidaan käyttää vaatimusmäärittelyn lisäksi systeeminmäärittely – ja ohjelmistosuunnitteluvaiheissa. Rosen avulla voidaan mallintaa asiakkaiden ja käyttäjien vaatimukset UML-kaavioiksi ja muodostaa automaattisesti luokkien perusrakenne ohjelmakooditasolle. Rational Rose on yhdistettävissä muihin Rationalin projektityökaluihin (vaatimusmäärittely ja testaustyökalut) ja se tukee Rational Unified Process – menetelmää.

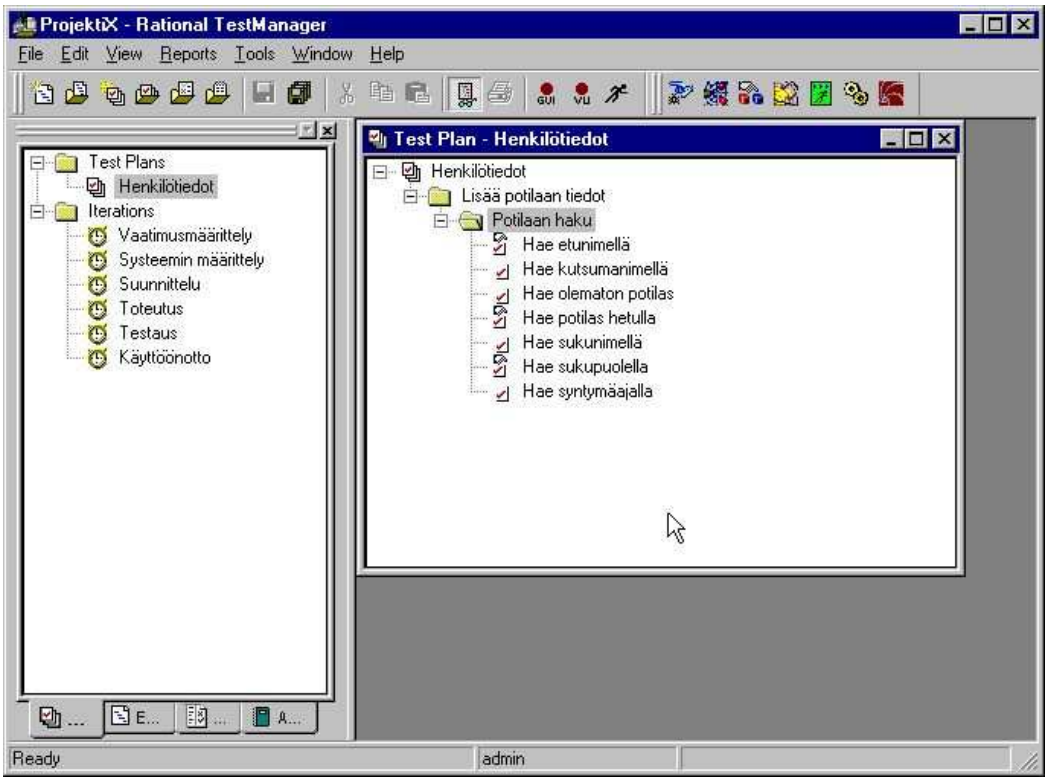
5.2 Administrator testausprojektin hallinnassa

Ennen testaustyökalujen käyttämistä pitää Rational Administrator –ohjelmalla (Kuva 23) luoda uusi testausprojekti. Administratorin avulla testausprojekti on mahdollista yhdistää muutostenhallintavälineeseen, analyysimalleihin (Rational Rosella laaditut UML-kaaviot) tai RequisitePro-vaatimusmäärittelyprojektiin. Rational Administratorin avulla hallitaan testausprojektin käyttäjiä ja käyttäjäryhmiä, jotka osallistuvat testaamiseen ja annetaan heille tarvittavat käyttöoikeudet.

Kuva 23: Rational Administrator - testausprojektin ylläpidon työkalu

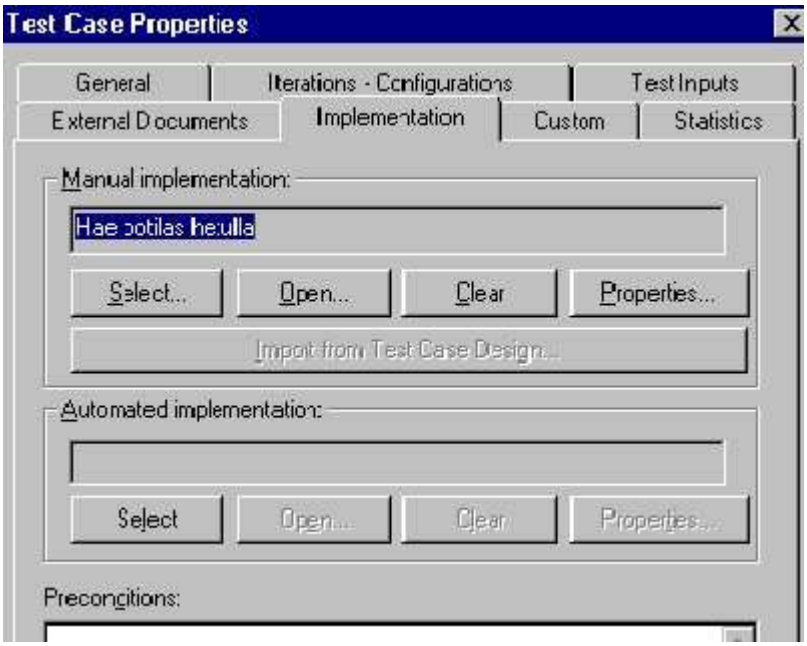
5.3 TestManager testitietovarastona

Testitietovaraston rakentaminen on mahdollista esimerkiksi Rational TestManager-ohjelmalla. TestManager on testauksen hallintaan tarkoitettu ohjelma ja sillä voidaan luoda testitapauskansioita ja testitapauksia tai ryhmitellä eri tyyppiset testitapaukset omiin kansioihin tai projektin eri vaiheisiin kuuluviksi (Kuva 24).



Kuva 24: Testitapausten hallinta Rational TestManager -ohjelmalla

Yksittäiseen testitapaukseen voidaan liittää ulkoisia määrittelydokumentteja, testitapauksen esiehdot, kuvaus sekä manuaalinen tai automaattinen skripti (Kuva 25). Skriptin nauhoitus tehdään erillisellä ohjelmalla esim. RobotJ:llä ja samoin manuaalisen skriptin kirjoittaminen tapahtuu Rational ManualTest –ohjelmalla.



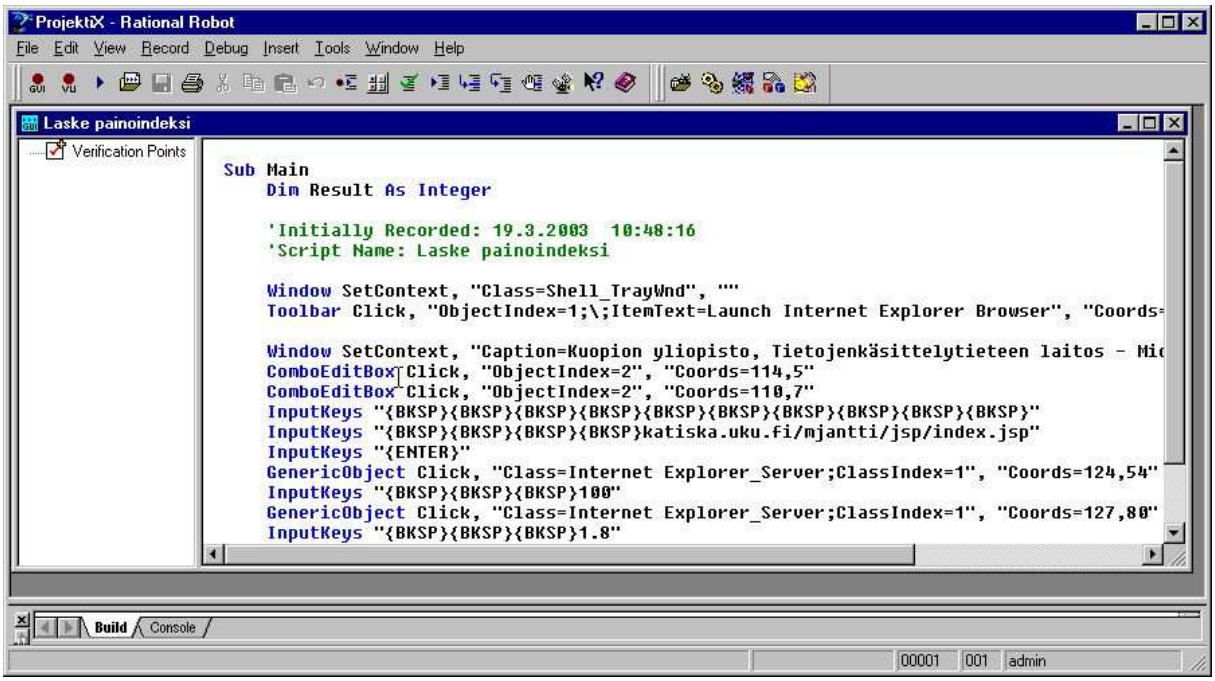
Kuva 25: Testitapauksen ominaisuuksien määrittely TestManager -ohjelmassa

TestManageria voi käyttää ensin ei-automatisoidun testauksen testitietovarastona, josta tulevaisuudessa päästään tarvittaessa siirtymään automatisoituun testaukseen. Lisäksi TestManager tarjoaa monipuolisia raportteja testauksen arviointia varten eli käyttäjä voi itse rakentaa esimerkiksi kaavion hylättyjen ja suoritettujen testitapausten jakaumasta tai yhteenvetoraportin, jossa näkyvät kaikki rakennetut testitapaukset. Raportit voidaan muuttaa MS-Word-muotoon, jolloin niihin voidaan lisätä itse uusia sarakkeita.

5.4 Robot/RobotJ testiskriptien nauhoituksessa

Toiminnallisessa testauksessa käyttäjän suorittamat toiminnot voidaan automatisoida nauhoittamalla suorituksista uudelleenkäytettäviä skriptejä. Testiskriptien nauhoitukseen voidaan käyttää RobotJ-ohjelmaa. Rational nimeää tuotteen entisen Robot-ohjelman ”siskoksi” eikä korvaajaksi. Skriptiä nauhoittava työkalu toimii siten, että ensin luodaan skripti ja annetaan sille nimi. Tämän jälkeen ohjelma käynnistää nauhoitustoiminnon, jonka aikana kaikki käyttäjän tekemät toiminnot (sovelluksen avaukset, hiiren painallukset ja kirjoitetut tekstit) tallentuvat skriptiin muistiin (Kuva 26). Nauhoitettua skriptiä voidaan muokata, päivittää ja tallentaa. Käyttäjän antamien tekstisyötteiden muuttaminen skriptissä on varsin yksinkertaista, mutta monimutkaisemmat muutokset vaativat ohjelmointikokemusta skriptin nauhoituskielellä.

Regressiotestausvaiheessa voidaan ajaa testiskripti uudelleen ja katsoa toimiiko ohjelma muutosten jälkeen toiminnallisesti oikein. Testiskriptien muuttaminen on yksi tapa saada uusia testitapauksia [Rat03].



Kuva 26: Nauhoitettu testiskripti Rational Robot -ohjelmassa

Vanhempaan Robot-ohjelmaan verrattuna RobotJ-ohjelma sisältää seuraavat erot [Rat03]:

1) Skriptikielenä käytetään Javaa, kun se Robotissa oli Visual Basicin kaltaista koodia.

2) ScriptAssure-tekniikan ansiosta skripti voidaan suorittaa, vaikka käyttöliittymäobjektin paikka tai nimi vaihtuvat. Robotissa skriptin suoritus pysähtyi, mikäli tietyn nimistä objektia ei löytynyt tarkalleen siltä paikalta, missä se nauhoituksen aikana oli. ScriptAssuressa käyttöliittymän osille eli objekteille annetaan ominaisuuksia (esim. nimi=*passwd*), joille annetaan tietty painoarvo (esim. 100). Käyttäjä voi määritellä myös raja-arvon, johon ominaisuuksien painoarvojen summaa verrataan. Esimerkiksi ohjelma suorittaa skriptin ja huomaa, ettei se löydä *java.swing.jcheckbox*-objektia nimeltä *passwd*. Käyttäjä on määritellyt skriptin hyväksymistasoksi 10000. Mikäli

objektin painoarvo ei ylitä hyväksymisrajaa, skriptin suorittamista jatketaan, vaikka *passwd*-nimistä objektia ei löytynyt. Hyväksymisrajan lisäksi voidaan määritellä hälytysraja, jossa käyttäjälle annetaan varoitus skriptin aikana havaituista muutoksista, mutta skriptin suoritusta jatketaan silti.

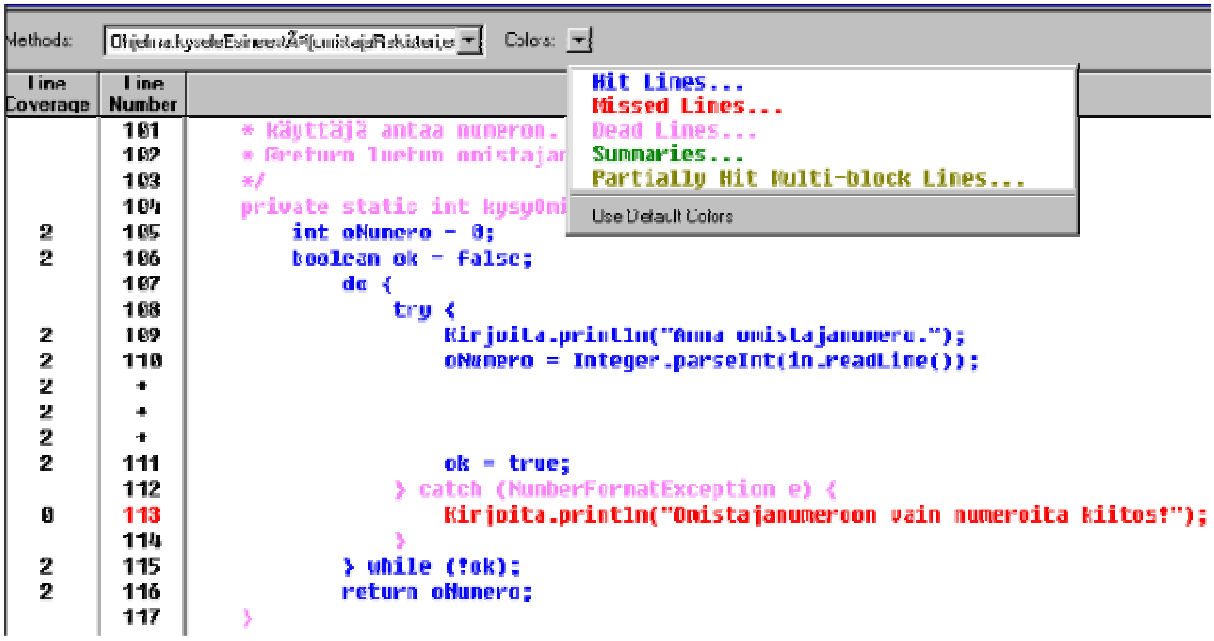
3) Dynaamisen datan/sisällön vastaavuuden testauksessa voidaan käyttää hyväksi säännöllisiä lausekkeita. Tarkastellaan esimerkiksi webpohjaista toimintoa, jolla syötetään uuden tilauksen tiedot. Joka kerralla, kun syötetään uusi tilaus, myös tilausnumero kasvaa yhdellä numerolla. Kun skripti on nauhoitettu, sivustolla ja skriptissä näkyvä tilausnumero on esimerkiksi 1000. Kun skripti ajetaan uudelleen ja vertaillaan sivun sisältöjä toisiinsa, järjestelmä ilmoittaa sisällön olevan erilainen, koska tilausnumerona on nyt 1001. Tilausnumeron johdosta sisältö on siis erilainen testiskriptin uudelleenajon jälkeen kuin nauhoituksen aikana ja skriptin suorittamisesta aiheutuu turha virheilmoitus. Muuttuva osuus websivustolla (tilausnumero) voidaan korvata säännöllisellä lausekkeella (tilausnumero koostuu 1-3 numerosta välillä 1-9). Näin ollen testiskripteistä tulee paremmin uudelleenkäytettäviä.

4) Skriptit voidaan liittää versiohallintaan

5) RobotJ tukee sekä IE- että Netscape-selainten uusimpiakin versioita eli IE-selaimella nauhoitettu testiskripti voidaan ajaa myös Netscapella.

5.5 PureCoverage koodikattavuuden arvioinnissa

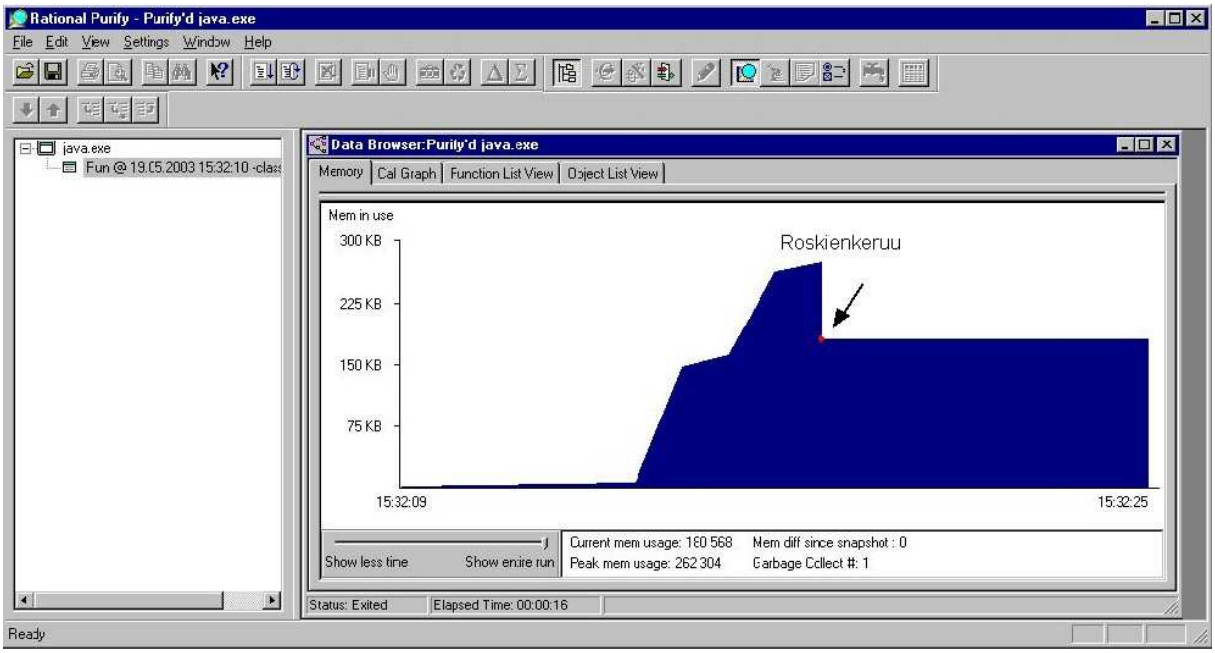
Rational PureCoverage tarjoaa koodikattavuustietoa Javalle, Visual C/C++:lle ja Visual Basicille. PureCoverage näyttää lähdekoodissa eri väreillä (Kuva 27), mitkä rivit ohjelman suorituksen aikana käytiin läpi ja mitkä puolestaan jäivät käymättä eli ohjelma soveltuu hyvin ns. kuolleen ohjelmakoodin paikantamiseen. *Kuollut ohjelmakoodi* on turhaa koodia, jota ei koskaan suoriteta esimerkiksi virheellisten ehtolausekkeiden vuoksi. Lisäksi välinettä voidaan käyttää sen varmistamiseen, että kaikki ohjelman osat on testattu.



Kuva 27: Rational PureCoverage

5.6 Purify muistinkäytön tehokkuuden arvioinnissa

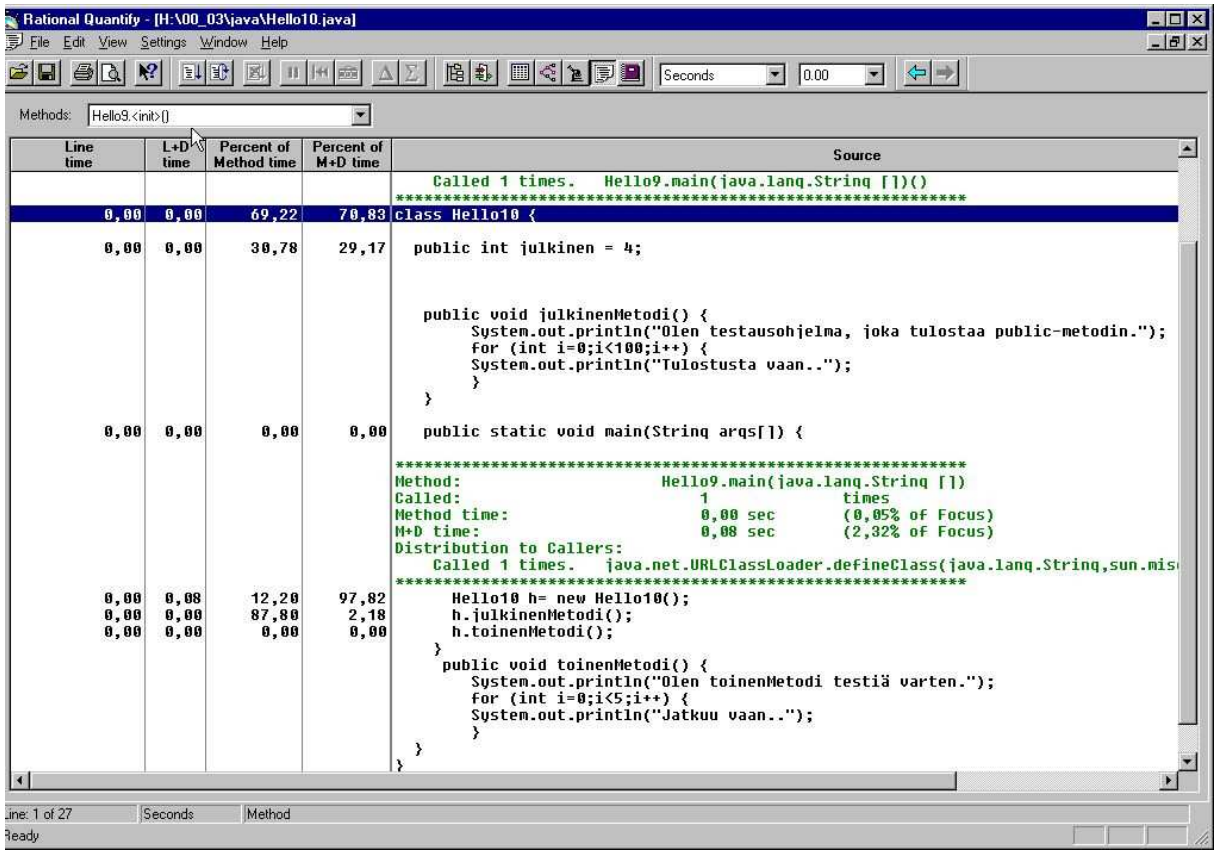
Rational Purify soveltuu muistivirheiden ja muistinkäytön tehokkuuden tarkkailuun muun muassa Javalle sekä Visual C/C++:lle. Ohjelman suorituksesta saadaan muistin käyttöä havainnollistava kaavio (Kuva 28), joka näyttää muistin kuormitushuipun sekä ajan, jolloin roskienkeruu tapahtuu (muistinkäyttökäyrällä oleva piste).



Kuva 28: Rational Purify

5.7 Quantify suorituskykytestauksessa

Rational Quantify on Java-, Visual C/C++- ja Visual Basic-ohjelmille tarkoitettu suorituskykytestauksen ohjelmisto. Quantify näyttää lähdekoodissa, kuinka paljon aikaa kunkin luokan metodit kuluttavat (Kuva 29).



Kuva 29: Rational Quantify

5.8 Muut työkalut

Testaustyökaluja on saatavilla testausprosessin jokaiseen vaiheeseen ja markkinoilta löytyy jopa useita satoja erilaisia työkaluja, joista osa on kaupallisia ja osa vapaasti kokeiltavissa [Poh02]. Kuormitustestaukseen tarvittavien työkalujen ongelmana on niiden korkea hinta. Esimerkiksi Keynoten tuote Total Performance Lab ja siitä 3-vuoden Load Testing -paketti kuormitustestausta varten sisältäen 1000 virtuaalikäyttäjää, koulutus, tekninen tuki ja päivitykset maksaa noin 225 000 euroa. Keynoten tuotteen hinnoittelu määräytyy joko virtuaalikäyttäjien määrän tai testausajan mukaan. Muita kuormitustestauksen työkaluja ovat muun muassa Mercury Interactiven LoadRunner ja Compuwaren QACenter Performance Edition.

Kattavan testitapausjoukon luominen on testausprosessin vaikeimpia tehtäviä. Automatisointikokeiluissa ongelmaksi on havaittu se, että työkalut eivät osaa tuottaa riittävän kattavaa testitapausjoukkoa, kun ohjelman rakenne monimutkaistuu [MaK00]. Automaattista generointia tekee muun muassa UMLTEST-ohjelma [OfA99], joka on integroitu Rational Rose -välineeseen. Rose tallentaa UML-määrittelyt tekstimuodossa, jota voidaan vaivattomasti lukea. UMLTEST luo testitapauksia predikaattikattavuus (predicate coverage)- ja siirtymäkattavuus (transition-pair coverage) -testaustasoille.

UMLTEST käy läpi Rose-määrittelytiedostoa (MDL-tiedosto) selvittääkseen määrittelyjen sisällön tarkoituksen ja muodostaa automaattisesti tilasiirtymätaulukon. MDL-tiedostot säilyttävät määrittelytietoa eri näkökulmia varten (looginen näkymä ja käyttötapaus-näkymä). UMLTEST koostuu kolmesta pääobjektista: UML määrittelyparseri, full-predicate -testitapausgeneraattori ja transition-pair –testitapausgeneraattori. Testitapausgeneraattorit ottavat tilasiirtymätaulukon syötteenä ja luovat testitapauksia, joissa siirtymä käydään läpi ja testitapauksia, joissa siirtymää ei toteuteta. Automaattisen generoinnin onnistumisen edellytyksenä on kuitenkin ehtojen ja toteutumisen tarkka määrittely (esimerkiksi boolean tyyppisillä attribuuteilla: true ja false).

6 Pohdinta

Tässä luvussa esitetään yleiset havainnot, jotka koskevat sekä UML-pohjaisen testausmallin käyttöä että toiminnallisen testauksen kokeilua. Testauskokeilu suoritettiin tutkielman teon yhteydessä ja siinä testattiin Kuopion yliopistollisessa sairaalassa käyttöönottovaiheessa olevaa potilashallinnon ohjelmaa.

6.1 Havainnot UML-testausmallista

Testitapausten suunnittelua varten saadaan paljon aineistoa UML-kaavioista ja niiden tekstikuvauksista, mutta pelkkä yksittäinen kaavio ei tuo testaukseen riittävää informaatiota. Yhdellä kaaviolla saadaan kuvattua järjestelmän toimintaa tai rakennetta vain yhdestä näkökulmasta. Tärkeintä on käyttää useampaa kaaviota yhdessä kokonaisvaltaisen näkemyksen hankkimiseksi. Kaavioiden piirtäminen vaatii kohdealueen ymmärtämistä sekä koulutusta UML:n käyttöön. Visuaalinen mallinnus on kuitenkin tehokasta, koska UML-kaaviot auttavat testaajaa hahmottamaan järjestelmän toimintamallin nopeammin kuin pelkät pitkät tekstikuvaukset.

Tärkeimpiä kaavioita testausta ajatellen ovat ohjelman käyttäytymistä tai suoritusta kuvaavat kaaviot (behavioral diagrams) eli käyttötapauskaaviot, sekvenssikaaviot, yhteistyökaaviot, tilakaaviot ja aktiviteettikaaviot, koska ohjelmistotestauksessa keskitytään etsimään virheitä, jotka esiintyvät ohjelman suorittamisen aikana [Wil99]. Rakenteelliset kaaviot tuovat tietoa vain siitä, mistä osista järjestelmä koostuu. Oikein käytettynä ne antavat ymmärrystä siitä, onko ohjelman modulaarinen rakenne hyvä. Käyttötapauskaaviot kuvaavat järjestelmän ja käyttäjän vuorovaikutusta. Käyttötapaukset edustavat eri tyyppisiä järjestelmän vaatimuksia, kuten toiminnallisia vaatimuksia, toiminnallisuuden jakautumista luokille tai olioille sekä olioiden vuorovaikutusta ja rajapintoja.

Käyttötapauksista tulee testata normaalien toimintopolkujen lisäksi vaihtoehtoiset ja poikkeukselliset toimintopolut. Sekvenssikaaviot kuvaavat olioiden tai komponenttien välistä vuorovaikutusta, mutta testauksen kannalta harmillista on se, että ne sisältävät vain yhden toimintaskenaarion. Vaihtoehtoisten toimintojen kuvaaminen vaatii useita sekvenssikaavioita. Yhteistyökaaviolla esitetään myös olioiden välistä vuorovaikutusta, mutta olioiden tai komponenttien sijoittelu on vapaampaa kuin sekvenssikaaviossa.

Tilakaavio kuvaa järjestelmän tiloja, tilojen välisiä siirtymiä ja siirtymiin liittyviä tapahtumia. Tilakaaviot soveltuvat hyvin testausmalliin, koska ne ovat UML:n näkökulmasta eniten formaaleja ja tarjoavat luonnollisen pohjan testitapausten luomiselle Aktiviteettikaaviot mallintavat järjestelmän sisä- tai ulkopuolella tapahtuvaa työnkulkua. Aktiviteettikaaviolla voidaan kuvata kontrollikaavion tapaan ohjelmakoodin ehtorakenteita, silmukkarakenteita sekä rinnakkaisia toimintoja.

Komponenttikaavio tarjoaa kuvauksen järjestelmän sisältämistä komponenteista ja niiden välisistä suhteista. Komponenttikaaviota voidaan käyttää järjestelmäarkkitehtuurin kuvaamiseen. Toimituskaaviolla mallinnetaan järjestelmän käyttämää laitteistoa ja laitteistokomponenttien välisiä yhteyksiä. Luokkakaavio tarjoaa mahdollisuuksia havaita ohjelman toimintojen riippuvaisuuksia kuten luokkien välinen periytyminen, assosiaatio ja osa-kokoonpano-suhteet.

UML-pohjainen testausmalli voi yksinkertaisimmalla tasolla olla sitä, että testitapaukset tehdään ja dokumentoidaan samassa järjestyksessä kuin käyttötapaukset (käyttötapauksen numero vastaa ylätason testitapauksen numeroa). Esimerkiksi ylätason testitapaus *Lisää potilas* voisi sisältää useita testitapauksia *Lisää potilas*-käyttötapauksen suorittamisesta. Testaus etenee loogisessa järjestyksessä käyttötapauksesta toiseen ja testausdokumenteista on helppoa todeta, miten kattavasti ohjelmaa on testattu.

Jäljitettävyyden korostaminen tuo koko ohjelmistotuotantoprosessiin sekä testaukseen yhtenäisyyttä ja selkeyttä. Asiakasvaatimuksista johdetaan käyttötapaukset ja käyttötapauksista ohjelmamoduulit ja testitapaukset. Jäljitettävyysketjun päässä sijaitsee toimiva ohjelmiston ominaisuus. Myös asiakas on tyytyväinen saadessaan toimittajalta hyvin jäsennellyt testausdokumentit käyttöönsä.

6.2 Havainnot toiminnallisen testauksen kokeilusta

Tutkielman teon aikana suoritetun testauskokeilun kohteena oli terveydenhuollon tietojärjestelmä, joka sisälsi muuan muassa Henkilötiedot-, Lähete- ja Ajanvaraukset-osiot. Sairaalan tietohallintoyksikkö tarjosi testiympäristön testitietokantoineen, jossa toiminnallista testausta pystyttiin tekemään.

Terveydenhuollon tietojärjestelmän testausmenetelmänä käytettiin mustalaatikkotestausta. Alustavat testitapaukset rakennettiin ohjelmiston käyttöohjeen perusteella ja testitapausten syötteet jaettiin laillisiin, kelvottomiin ja laittomiin arvoihin. Varsinaisessa testaustilanteessa käytiin läpi aiemmin laaditut testitapaukset ja lisättiin uusia testitapauksia, kun ymmärrys ohjelmasta kasvoi. Testauskokeilun tuloksena nousivat esille muun muassa seuraavat kehittämiskohteet:

- Ongelmat testausympäristön pystyttämisessä ja riittävien käyttöoikeuksien myöntämisessä voivat hidastuttaa testauksen alkamista viikoilla.
- Samojen toimintojen tulisi käynnistyä samalla tavalla järjestelmän kaikissa alisovelluksissa. Hakutoiminto käynnistyi välillä Enter-painikkeella ja välillä taas Tab-painikkeella, mikä hankaloitti käytettävyyttä.
- Virheilmoitukset ja virheenkäsittelyn tulisi olla yhtenäistä koko järjestelmässä. Normaalitilanteessa virheellisesti syötetty kenttä värjäytyi punaisella. Eräissä tapauksissa virheellinen arvo pyyhittiin kentästä pois ja kentän arvoksi asetettiin 0.
- Järjestelmän toipuminen virheistä oli hidasta. Ajonaikaisten virheiden jälkeen järjestelmä jumiutui eivätkä sovellukset käynnistyneet kunnolla ilman tietokoneen uudelleenkäynnistämistä.
- Järjestelmän käyttöohje oli rakenteeltaan hieman epäselvä. Käyttöohjeen luettavuus olisi parempi, jos jokaisen alisovelluksen käyttöohje sisältäisi oman sisällysluettelonsa ja myös lukujen sisältämät kappaleiden otsikot olisi numeroitu. Käyttöohjeessa ei saisi esiintyä turhaa asioiden toistoa.
- Käyttöohjeen tulisi sisältää vain käyttäjän kannalta olennaiset asiat. Loppukäyttäjää saattavat häiritä ohjeissa liian tekniset asiat kuten käyttöoikeuksien määrittelyssä tarvittavat parametrit.
- Ajanvaraukset-sovelluksessa voidaan sitoa joku aika siten, ettei sille voi varata potilasaikoja esim. sairaanhoitajien kokouksen tai koulutuksen vuoksi. Sitomisen syyn pitäisi näkyä kaikilla näytöillä.

- Sähköinen lähete on vasta tulossa KYS:iin ja asettaa varmasti haasteita tietojärjestelmien yhteistoiminnalle. Tällä hetkellä lähetteet saapuvat terveyskeskuksista sairaalaan paperimuodossa postitse.

Järjestelmän käytettävyyden kannalta hyvää oli kertakirjautuminen eli salasanoja ei tarvinnut syöttää uudelleen, kun siirryttiin alisovelluksesta toiseen. Pikapainikkeiden ansiosta valikkovalintoja ei tarvitse käyttää ja potilaiden tietojen käsittelyä helpottaa myös 10 viimeksi käsitellyn potilaan pikavalinta.

Järjestelmän testauksessa löydettiin virheitä muun muassa valitsemalla lähetteelle kaksi samanlaista arvoa (liitteeksi valittiin kaksi kertaa Röntgenkuvat). Syöttämällä sovelluksessa hoitopäivien määräksi negatiivinen päivä (-9) saatiin käynnistettyä virheketju, joka antoi ajonaikaisen virheen ja johti lopulta järjestelmän kaatumiseen. Varaukset-sovelluksessa esiintyi häiritsevä ylimääräinen dialogi. Koska testauksen kohteena ollut ohjelmistoversio on päivitetty jo uudempaan, edellä mainitut virheet on korjattu eikä niitä pitäisi enää esiintyä. Testauskokeilu oli hyödyllinen, koska se osoitti, että järjestelmällistä testausta lisäämällä ohjelmista voidaan löytää tehokkaammin virheitä, joista osa on merkitykseltään vakavia.

Monimutkaisen tietojärjestelmän toimittaja, esimerkiksi terveydenhuollon toimialalla, joutuu toimittamaan asiakkaalle useita ohjelmaversioita käyttöönoton aikana. Tavallisesti versioita joudutaan tekemään sitä useampia mitä keskeneräisempi tai virheellisempi ohjelmisto on kyseessä. Tällaisessa tilanteessa testitietovarasto ja hyvin suunniteltu regressiotestaus vähentävät testauksen työmäärää huomattavasti. Tietojärjestelmän toimittajalla on testauksessa käytössään ohjelmakoodi. Toimittaja testaa sovelluksen sisältämät komponentit ensin yksi kerrallaan yksikkötestauksessa. Integrointitestauksessa testataan komponenttien väliset rajapinnat eli toimivatko komponenttien väliset yhteydet. Järjestelmätestauksessa on mukana ohjelmiston lisäksi toimintaan tarvittava laitteisto ja tietokannat eli tässä vaiheessa testataan kokonaisuus. Hyväksymistestauksessa järjestelmän varsinainen käyttäjä huomaa paljon helpommin käytettävyysvirheitä kuin ohjelmiston suunnittelija.

Asiakkaalla ei ole pääsyä ohjelmakoodiin, joten ohjelmaan tulevat muutos- ja korjauspyynnöt täytyy lähettää toimittajalle. Tällä hetkellä sairaalapäässä muutospyyntöjä

kirjoitetaan ensin vihkoon, joista ne kirjataan tekstinkäsittelyohjelmaan ja lähetetään tämän jälkeen tietojärjestelmän toimittajalle. Toimittajalla on kuitenkin tekniset edellytykset ottaa vastaan muutospyyntöjä mm. weblomakkeen avulla. Asiakkaan tekemä testaus on sairaalan päässä lähinnä hyväksymistestausta, jossa ei enää ehditä testaamaan kaikkia ohjelman teknisiä yksityiskohtia. Apunaan sairaalan testaajalla on järjestelmän käyttöohje sekä Help-toiminto.

Teoriassa jokainen ohjelman syöteyhdistelmä tulee käydä testauksessa läpi. Testaus jaetaan eri ulottuvuuksiin ja tasoihin, joilla kullakin on omat testitapauksensa. Käytännössä, esimerkiksi kiireisissä ja mittavissa käyttöönottoprojekteissa, huolellista testausta rajoittavat testaukselle varatut rajalliset resurssit. Testauksen automatisointi ja testitietovarasto ovat keinoja, jotka säästävät aikaa ja rahaa testauksessa. Hyväksi testaajaksi tuskin tullaan pelkästään yhden teoreettisen kurssin kautta, vaan siihen tarvitaan paljon työkokemusta testauksen parissa sekä kärsivällinen luonne käydä järjestelmällisesti läpi satoja tai usein tuhansia testitapauksia yhtä ohjelmaa kohden. Teoriaosaaminen, kuten tieto erilaisista testitapausten suunnittelumenetelmistä (esim. ekvivalenssiluokat), on myös välttämätöntä järjestelmällisen testauksen suunnittelulle.

Testausmallin käyttäminen valmiiseen ohjelmaan tuntui ensin hiukan vaivalloiselta, koska käyttötapausten piirtäminen mm. potilaan etsimisestä valmiille ohjelmalle ei vaikuttanut järkevältä. Käyttötapaukset tulisikin laatia ohjelmistosuunnittelun alkuvaiheessa, jolloin käyttötapaukset selkeyttävät asiakasvaatimuksia ja antavat pohjaa tuotteen toiminnallisuuden jakamiselle eri ohjelmisto-osille ja osien testaukselle. Tässä tutkimuksessa laadituista käyttötapauskaavioista saatiin hyödyllistä tietoa Potilaan valinta -komponentin määritystä tekevälle PlugIT-pilotointitiimille. UML-tilakaavio oli hyvä valinta lähetteen eri tilojen kuvaamiseen. Pelkkä tekstikuvaus lähetteiden tiloista ja järjestelmän toiminnasta eri tiloissa oli vaikeasti tajuttavissa. Tilakaavio toi selkeästi esille, mitkä tapahtumat johtavat mihinkin tiloihin. Monimutkaiseksi tilakaavion teki se, että tilan tarkenteen muutokset jouduttiin käsittelemään alitiloina. Testauksessa oli ylipäänsä hankalaa hahmottaa looginen järjestys kaikkien toimintojen suorittamiselle. Toimialatuntemus hoitohenkilökunnan työprosesseista on olennaista terveydenhuollon tietojärjestelmien testauksessa.

7 Yhteenveto

Testitapausten suunnittelu tulee aloittaa aikaisessa vaiheessa ohjelmistotuotantoprosessia, koska mitä aikaisemmin virheet havaitaan ohjelmistotuotteessa, sitä halvempia ovat niille tehtävät korjaustoimenpiteet. Testaus voidaan jakaa eri testausmenetelmiin (mustalaatikko- ja lasilaatikkotestaus) sekä testaustasoihin (yksikkö-, integrointi-, järjestelmä- ja hyväksymistestaus). Eri testaustasoilla ja -menetelmissä tarvitaan erilaisia testitapauksia. Lasilaatikkotasolla testitapaukset liittyvät ohjelmakoodiin (ehto-, toisto- ja silmukkarakenteet) ja mustalaatikkotestauksessa ohjelman ulkoiseen käyttäytymiseen, koska koodia ei ole nähtävissä. Toimintojen testaamisen lisäksi tulisi kiinnittää enemmän huomiota myös ohjelmistotuotteen laadullisten ominaisuuksien tarkistamiseen kuten käytettävyyteen, suorituskykyyn, luotettavuuteen ja saatavuuteen.

Testitapaus koostuu yksikäsitteisestä tunnisteesta, esiehdoista, oletetusta tuloksesta ja todellisesta tuloksesta. Testauksessa (mustalaatikkotestauksen periaate) järjestelmän antamaa todellista tulosta verrataan määrittelyjen mukaiseen oletettuun tulokseen. Mikäli todellinen tulos vastaa oletettua tulosta, testi on hyväksytty, muussa tapauksessa testin tulos hylätään. Testitapausten etsinnän tulee olla järjestelmällistä, keskittynyttä ja automatisoitua. Näiden tavoitteiden saavuttamiseksi suoritettavaa testausta voidaan kuvata testausmallien avulla. Testausta varten testaaja voi muodostaa testausmallin, joka kuvaa testattavan kohteen käyttäytymistä ja rakennetta sekä ympäristöä. Testausmalleina voidaan käyttää tilakoneita, kombinaatiologiikkaa (päätöstaulut) tai UML-kaavioita. Määrittelypohjaisen testauksen tutkimus on keskittynyt nykyisin hyvin paljon UML-kaavioiden käyttöön testauksessa.

Testauksessa käytettävien mallien käyttö tuo monia hyötypuolia projektille, mutta samanaikaisesti saattaa esiintyä myös seuraavia ongelmia tai ennakkoasenteita malleja kohtaan: Testausmallin onnistunut rakentaminen riippuu hyvin paljon määrittely- ja suunnitteludokumenttien tasosta. On hyvä muistaa, että yhden kaavion perusteella on hankala rakentaa mitään testausmallia. Mallin rakentaminen vie projektilta aikaa ja keskittyminen olennaisten asioiden mallintamiseen voi olla hankalaa. UML:n käyttö testauksessa ja ohjelmistosuunnittelussa vaatii koulutusta. Testitapausten kirjaaminen vie runsaasti aikaa ja projektiryhmän asenteena on usein vastustaa kaikkea uutta ja

vierasta [MaK00]. UML-kaavioiden pohjalta muodostettu testausmalli ei ole täydellinen, koska kaavioista ei voi nähdä käytettävyystestaukselle tärkeitä testitapauksia, esimerkiksi miten navigointi sujuu tai mille käyttöliittymän ulkoasu näyttää. UML-kaaviot eivät myöskään kerro, että järjestelmää tulee testata useilla samanaikaisilla käyttäjillä tai sitä, miten kauan käyttötapausten suorittaminen kestää.

Testausmallin yleisinä hyötyinä voidaan vastaavasti luetella seuraavat asiat: Testauksesta tulee järjestelmällistä ja se keskitetään sinne, missä tapahtuu eniten virheitä. Testausta voidaan automatisoida vasta sitten, kun tiedetään miten testaus sujuu manuaalisesti. Testausmalli tukee suunnittelua ja kerran rakennettua testausmallia tai sen osia voidaan käyttää hyväksi tulevissa projekteissa. Testausmalli on tehokas keino kertoa asioista asiakkaille tai ei-tekniselle johdolle.

Mitä hyötyjä UML-pohjaisesta testausmallista saadaan? Yksikkötestausta varten luokkakaavio osoittaa kaikki testattavat luokat ja metodit sekä luokkien väliset suhteet. Integrointitestauksessa komponenttikaaviosta nähdään testattavat komponentit ja niiden riippuvuussuhteet toisistaan. Sekvenssikaaviot ja yhteistyökaaviot kertovat, mitä viestejä olioiden tai komponenttien välillä liikkuu. Käyttötapauksia voidaan käyttää myös integrointitestaukseen, koska ne yhdistävät korkealla tasolla eri luokat tai ohjelman osat yhteen. Järjestelmätestauksessa toimituskaaviosta havaitaan, millainen on ohjelmiston fyysinen laiteympäristö eli mitä eri laitteistoja (palvelimet, tietokannat, tulostimet) testauksessa tulee ottaa huomioon. Järjestelmän suorituskykyä voidaan arvioida mittaamalla suoritusaika käyttötapausta kohden. Hyväksymistestauksessa testaaja voi suorittaa toiminnallista testausta ohjelmalle esimerkiksi käymällä tilakaavion kaikki tilasiirtymät tai aktiviteettikaavion kaikki suorituspolut läpi.

Tarkasteltaessa testausmallin oletettuja haittoja lähemmin huomataan osan ongelmista liittyvään yleisesti testaukseen eikä niitä pitäisi yhdistää suoraan testausmalliin. Testitapauksia syntyy paljon jo pientä ohjelmaa varten ja siitä ei voida syyttää pelkästään testausmallia. Turhia testitapauksia voidaan välttää jakamalla syötteet ekvivalenssiluokkiin ja valitsemalla vain ne testitapaukset, jotka edustavat luokkaansa parhaiten. Vaikka UML-kaaviot eivät tarjoa testaukselle valmiita jalostettuja testitapauksia, testaaja saa kaavioista hyvän yleiskuvan ohjelmiston rakenteesta sekä suorituksesta ja

pystyy testaamaan järjestelmällisesti eri suoritusvaihtoehdot. Samalla pystytään arvioimaan testauksen kattavuutta (mitä on jo testattu, mitä pitäisi vielä testata).

Toiminnallisen testauksen kokeilussa havaittiin käytännössä, että järjestelmällisellä testauksella voidaan ohjelmista löytää vakavia virheitä vielä siinä vaiheessa, kun sovellus on jo asiakkaalla. Asiakaspäässä tulisi lisätä teknistä testausta, mikä on usein mahdotonta niukkojen testausresurssien vuoksi. Hyväksymistestauksessa arvioidaan, miten hyvin ohjelma soveltuu työprosessiin, joten tässä vaiheessa asiakkaan puolella testaajana tulee ehdottomasti käyttää työntekijöitä, jotka osaavat arvioida ohjelman sopivuutta työympäristöön tulevan käyttäjän näkökulmasta. Ohjelmistotoimittaja saattaa hyväksymistestausvaiheessa joutua tekemään omalla kustannuksellaan useita eri lisäversioita, jos ohjelmistoa on liian vaikeaa käyttää tai se sisältää merkittäviä virheitä.

Testaukselle on luvassa myös uusia haasteita. Mobiiliteknologiat, hajautetut web services-toteutukset ja monet muut uudet tekniikat vaativat entistä tehokkaampia testausvälineitä. Testaustyökaluilta toivotaan parempaa käytettävyyttä (käytön oppiminen ja helpompi konfigurointi omaan tuotantoympäristöön sopivaksi), halvempia lisenssihintoja (varsinkin kuormitustestaukseen) ja monipuolista tukea eri ohjelmistoalustoille ja ohjelmointikielille. Testaus on projektinhallinnan kannalta yhtä merkittävä osa kuin muutkin ohjelmistotuotannon vaiheet. Projektinhallinta vaikeutuu huomattavasti, mikäli testausta laiminlyödään. Virheiden määrä lisääntyy ja ohjelmiston laatu heikkenee. Asiakkaalle joudutaan toimittamaan useita korjausversioita, mistä johtuen järjestelmän käyttöönotto viivästyy ja asiakkaalle tulee todennäköisesti taloudellisia tappioita. Ylläpitovaiheessa virheiden korjaus maksaa toimittajalle huomattavan paljon enemmän kuin suunnitteluvaiheessa huomatun virheen korjaaminen.

Tämän tutkielman kirjoittamisen aikana saatiin hankittua monipuolista materiaalia testauksen tutkimusta varten ja todettiin, miten paljon käyttökelpoista tutkimustietoa testauksesta on saatavilla. Valitettavasti olemassa olevista tutkimustuloksista käytetään vain pientä osaa hyväksi käytännön testauksessa. Tutkielma on syventänyt tutkimusnäkökulmaa ohjelmistotestauksesta ja samalla havaittiin, miten paljon testauksen ja tarkastuksen eri muotoja esiintyy jokapäiväisessä elämässämme.

Lähteet

[AbO00] Abdurazik, A., Offutt, J.: *Using UML Collaboration Diagrams for Static Checking and Test Generation*. The Third International Conference on the Unified Modeling Language (UML '00), pages 383-395, York, UK, October 2000.

[BaL02] Baudry, B., Le Traon, Y., Sunye, G.: *Testability Analysis of a UML Class Diagram*. IRISA, Campus Universitaire de Beaulieu, France, 2002.

[Bau03] Baumhof, S.: *A Performance Testing Strategy for Continuous Tuning*. 4th ICSTEST Conference Proceedings, Cologne, 2003.

[Ber03] Bertolini, A.: *Software Testing Research and Practice*. 10th International Workshop on Abstract State Machines (ASM 2003), Springer-Verlag, 2003.

[Bei90] Beizer, B.: *Software Testing Techniques*. (2nd Edition) Van Nostrand Reinhold, 1990.

[BeP02] Bell, D., Parr, M.: *Java for Students* (3rd Edition). Prentice Hall / Pearson Education Limited, 2002.

[Bin00] Binder, D.: *Testing Object Oriented Systems*. Addison Wesley, 2000.

[Bos00] Bosch, J.: *Design & Use of Software Architectures Adopting and evolving a productd-line approach*. Addison Wesley, 2000.

[BuM01] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.*: Pattern-Oriented Software Architecture*, John Wiley & Sons, 1996-2001.

[Car99] Card, D.: *Making Statistics Work for Software Engineering*. Software Engineering Institute 1999, esitys saatavilla webmuodossa URL: http://www.sei.cmu.edu/cmm/spc/panel1999/SPC-Card.pdf

[ChT02] Chevalley, P., Thevenod-Fosse, P.: *Automated Generation of Statistical Test Cases from UML State Diagrams*. Proceedings of the 25th Annual International Computer Software and Applications Conference, 2002.

[Eer01] Eerola, A.: *Tietojärjestelmien analyysi- ja suunnittelumenetelmät*. Luentomo-niste, TKT-laitos, Kuopion yliopisto, 2001.

[Elf01] El-Far, I.: *Enjoying the Perks of Model-Based Testing*. Proceedings of the STARWEST 2001, Florida Institute of Technology.

[FoS00] Fowler, M., Scott, K.: *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley, 2000.

[Ger02] Geralds, J.: *Software bugs costs billions*. Artikkeli IT Week 27.6.2002, saatavilla webmuodossa URL: <http://www.itweek.co.uk/News/1133047>

[Ham00] Hamina-Mäki, E.: *Systeemityön nykytilan kartoitus*. SYTYKE-lehti, 2000, saatavilla webmuodossa URL: <http://www.pcuf.fi/sytyke/lehti/kirj/st20004/06-09.pdf>

[HaM97] Haikala, I., Märijärvi, J.: *Ohjelmistotuotanto* (3. painos). Gummerus Kirjapaino Oy, 1997.

[Hau03] Hauser, J.: *Software Reliability over the Life Cycle: Expectations and Reality in the Automotive Industry*. 4th ICSTEST Conference Proceedings, Cologne, 2003.

[HeS00] Herzum, P., Sims, O.: *Business Component Factory*. Wiley Computer Publishing, New York, 2000.

[Hir00] Hirvonen, M.: *Käyttötapausten hyödyntäminen ohjelmistotyössä*. Kuopion yliopisto, Pro Gradu -tutkielma, 2000.

[IEEE829] ANSI/IEEE *Standard for Software Test Documentation*. IEEE Standard 829-1983.

[Imm02] Immonen, M.: *Suunnittelumallit.* Kuopion yliopisto, erikoistyö, 2002, saatavilla webmuodossa URL:
< http://www.cs.uku.fi/research/Teho/julkaisut.html>

[Imm03] Immonen, M.: *Käytettävyyden suunnittelu ja rakentaminen ohjelmistotuotantoprosessissa.* Kuopion yliopisto, Pro Gradu -tutkielma, 2003, saatavilla webmuodossa URL:
< http://www.cs.uku.fi/research/Teho/julkaisut.html>

[Jac92] Jacobson, I.: *Object-Oriented Software Engineering- A Use Case Driven Approach*. Addison Wesley, 1992.

[Kit95] Kit, A.: *Software Testing in the Real World - improving the process*. Addison Wesley, 1995.

[Kru00] Kruchten, P.: *The Rational Unified Process An Introduction* (second edition). Addison Wesley, 2000.

[KuH98] Kung, D., Hsia, P., Gao, J.: *Testing Object-Oriented Software.* IEEE Computer Society Press, USA, 1998.

[LiZ99] Liuying, L., Zhichang, Q.: *Test Selection from UML Statecharts*. 31st International Conference on Technology of Object-Oriented Languages and Systems 22-25 September, 1999.

[May99] Mayhew, D.: *The Usability Engineering Lifecycle*. Morgan Kaufmann Publishers, 1999.

[MaK00] Marik, V., Kral, L., Marik, R.: *Software Testing & Diagnostics: Theory & Practice*. SOFSEM 2000, Springer-Verlag, Berlin Heidelberg, 2000.

[Myk00] Mykkänen, J.: *Komponentti-FixIT, Terveydenhuollon komponenttipohjainen sovellustuotanto –toiminnallisuus, arkkitehtuuri, siirtymästrategiat ja välineet.* ATK-keskus, Kuopion yliopisto, 2000.

[Myö02] Myöhänen, H.: *Jäljitettävyys ohjelmistotuotannon tukena*. Kuopion yliopisto, Pro Gradu -tutkielma, 2002, saatavilla webmuodossa URL: < http://www.cs.uku.fi/research/Teho/julkaisut.html>

[OfA99] Offut, J., Abdurazik, A: *Generating Tests from UML Specifications.* George Mason University, saatavilla webmuodossa URL: <http://isse.gmu.edu/faculty/ofut/rsrch/papers/uml99.pdf>

[Paa00] Paakki, J.: *Software Testing.* Lecture Notes (Moniste D 407), University of Helsinki, 2000.

[Par03] Partanen, A: *Olio-ohjelmien testaus.* Kuopion yliopisto, Pro Gradu -tutkielma, 2003.

[Plu03] PlugIT-projekti: *Ohjelmistotuotannon nykytila –kysely.* Kuopion yliopisto, 2003.

[Poh02] Pohjolainen, P.: *Software Testing Tools*. Kuopion yliopisto, erikoistyö, 2002, saatavilla webmuodossa URL: < http://www.cs.uku.fi/research/Teho/julkaisut.html>

[Rat03] Rational Software Finland: *Järjestelmätestauksen hallinta ja automatisointi Java-projekteissa*. Seminaarimateriaali, 2003.

[RoR99] Robertson, S., Robertson, J.: *Mastering the Requirements Process*. Addison Wesley, 1999.

[Rus03] Russell, M.: *Risk-Based Testing – a Common Language for Project Stakeholders.* 4th ICSTEST Conference Proceedings, Cologne, 2003.

[Sam97] Sametinger, J.: *Software engineering with reusable components.* Springer-Verlag, 1997.

[TeK01] Tervonen, I., Kokkoniemi, J.: *Testauksen automatisoinnista*. Luentomoniste, Oulun yliopisto, 2001.

[ToE02] Toroi, T., Eerola, A, Mykkänen, J.: *Testing business component systems.* Kuopion Yliopisto, 2002, saatavilla webmuodossa URL: < http://www.cs.uku.fi/research/Teho/julkaisut.html>

[ToJ02] Toroi, T., Jäntti, M., Pohjolainen, P., Immonen, M., Eerola, A.: How to test software – How to derive test cases from user requirements. PlugIT-Raportti 26.3.2002, saatavilla webmuodossa URL: < http://www.cs.uku.fi/research/Teho/julkaisut.html>

[ToM02] Toroi, T., Mykkänen, J. Jäntti, M, Eerola, A: *Komponenttisysteemien testaus.* SoTeTiTe2002 tutkimuspaperit, Stakes, 2002, saatavilla webmuodossa URL: < http://www.cs.uku.fi/research/Teho/julkaisut.html>

[Vir02] Virkanen, H.: *Ohjelmistojen testaus ja virheenjäljitys*. Kuopion yliopisto, Pro Gradu -tutkielma, 2002, saatavilla webmuodossa URL: < http://www.cs.uku.fi/research/Teho/julkaisut.html>

[Wil99] Williams, C.E.: *Software Testing and the UML*. Center for Software Engineering, IBM T.J Watson Research Center, 1999.

[Wol02] Wolczkiewicz, A: *Usability testing from theory to practice: remote testing vs. laboratory testing.* Kuopion yliopisto, Pro Gradu -tutkielma, 2002.

[YoC99] Yoon, H., Choi, B., Jeon, J-O.: *A UML-based Test Model for Component Integration Test.* Workshop on Software Architecture and Components, proceedings on WSAC'99, 1999.

LIITE 1: Testitapausten suunnittelu

1 Test Case Design for an Application with many Dialogs

Test case design for an application with many dialogs in a sequence needs more time and effort than designing simple test cases. A simple way to describe the procedure of test case is to refer UML (Unified Modelling Language) diagrams like use case and activity diagrams. All dialogs of the application and UML diagrams as follows are usually included in system specification documents.

1.1 Use Case Diagram

The use case diagram shows the interaction between the application and the user. The following picture (Figure 1) is a use case diagram of Patient Management application for inserting patient data. The *actors* of this use case are *nurse* and *patient system*. Use cases are for example *Insert patient* and *Update patient*.

Insert patient includes for example *Save patient* use case. A nurse is able to insert patient (fill name, personal id, address, symptoms and save them into the database) and update patient data. The nurse can also search a patient from the database or close the program.

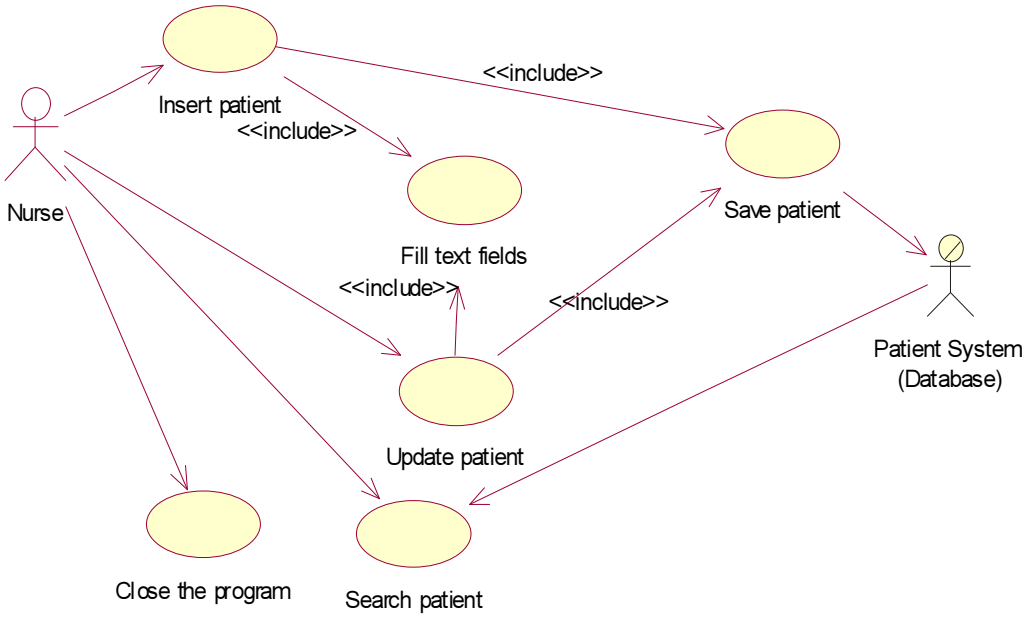


Figure 1: A Use Case Diagram of Patient Management application

1.2 Activity Diagram

The following activity diagram (Figure 2) describes the work flow and the order of displayed dialogs of Patient Management application. Arrows represent transitions from one function to the another. When a user has logged in, the main dialog of the program is displayed for him/her.

The user fills all text fields (name, personal id, address, symptoms) and presses Save button.

If text fields contain invalid data, system displays an error message (incorrect inputs) and the user can return to the main dialog (Ok button) to check values in data fields. If there were no errors, system asks whether the user would like to save data. Cancel button returns the user back to the main dialog without saving data and Yes button saves data into the database. The user receives a message while data was successfully saved.

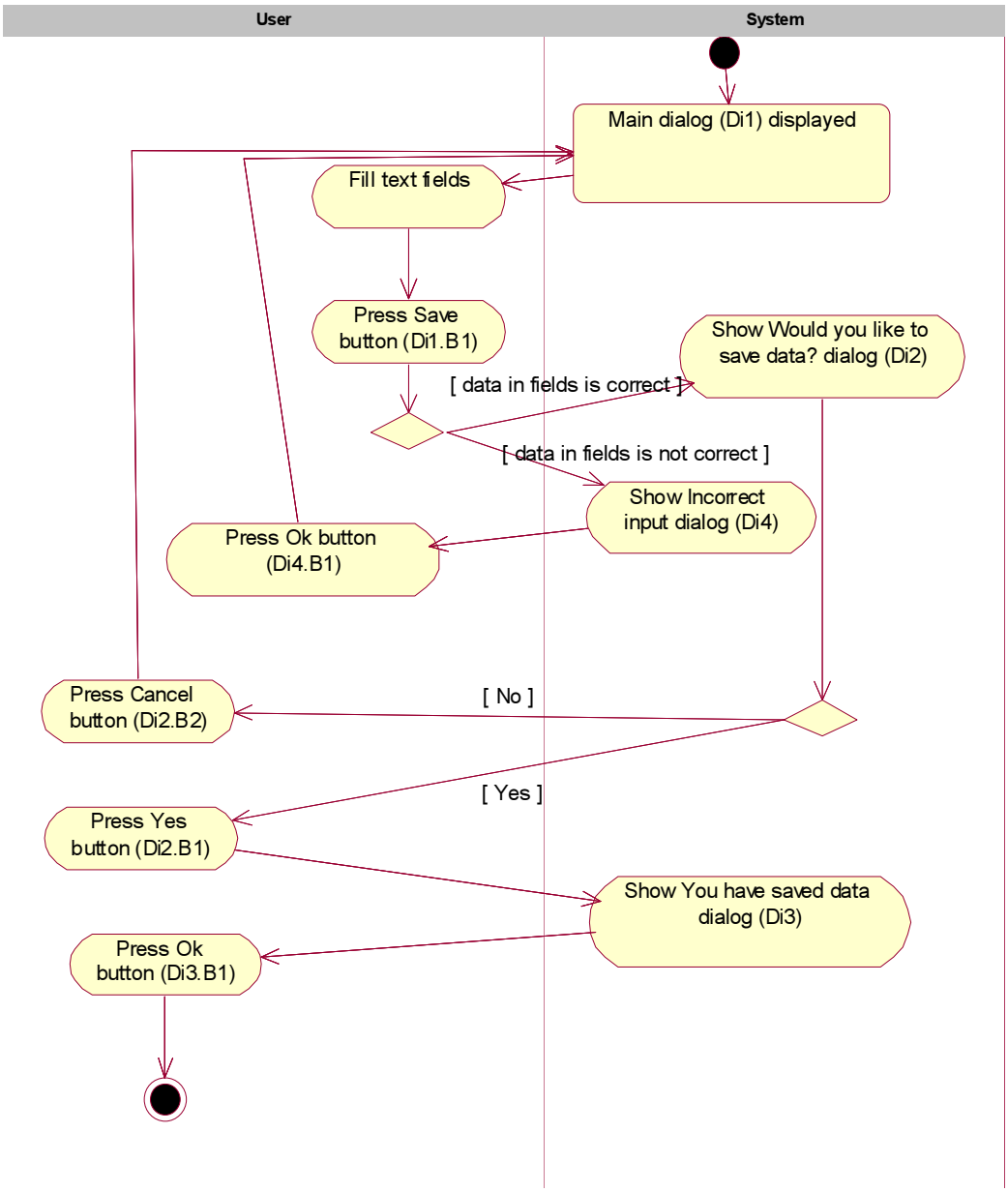


Figure 2: An activity diagram of Patient Management application

1.3 Dialogs of the Patient Management application

Patient Management application consists of following dialogs:

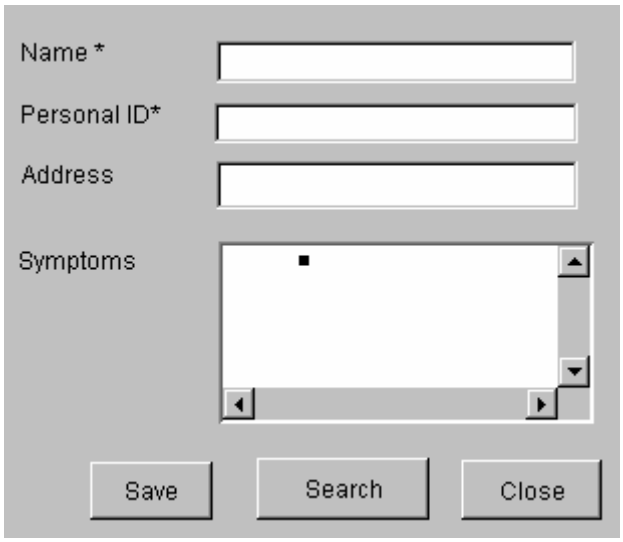


Figure 3: Main Dialog (Di1)

Notification Dialogs

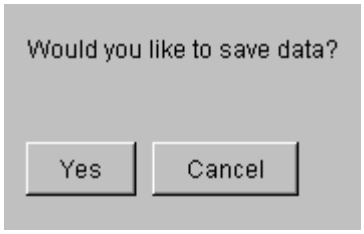


Figure 4: Would you like to save data – dialog (Di2)

Figure 5: You have saved data – dialog (Di3)

Error Message Dialogs

Figure 6: Incorrect input – dialog (Di4)

1.4 Test Cases for Patient Management application

Test Case tables can be created for example with text processing or spreadsheet programs. A test case table consists of test case id, prerequisites, step definition, input, expected outcome and special considerations. Those tables might be quite large, so a good tip is to keep the page orientation in a real document as landscape, not portrait. A test case might consist of various steps and the result of the first step can be used as input for the second step.

Test Case 1 – Insert Patient

Prerequisites:

- Login into the system has to be successfully completed
- Main Dialog is displayed for the user

Test Case 1.1: Correct saving (user has inserted valid data to the text fields)

Step	Input	Expected outcome	Special Considerations
1	Fill all text fields Name = Matt Pitt Personal ID = 120775-765R Address = unknown Symptoms= Broken arm	The text inserted by the user is visible in fields	
2	Press Save button	*Would you like to save data* dialog (Di2)	
3	Press Yes button in Di2 (fields contain valid data)	*You have saved data* dialog (Di3)	Check whether patient exists in the database

Test Case 1.2: Cancel the data saving

Step	Input	Expected outcome	Special Considerations
1	Fill all text fields: Name = Sam Personal ID = 041081-5678 Address = Wall Street 73 Symptoms= Head Ache	The text inserted by the user is visible in fields	
2	Press Save button	*Would you like to save data* dialog (Di2)	
3	Press Cancel button in Di2	Main dialog (Di1)	
4	Press Ok button in Di3	Main dialog (Di1)	Check that data was not saved to the database

Test Case 1.3 Incorrect saving (user has inserted non-valid data to the text fields)

Step	Input	Expected outcome	Special Considerations
1	Fill text fields incorrect or leave a required field empty Name *= empty Personal ID = 150984-543 Address = Symptoms=	The text inserted by the user is visible in fields (with errors)	
2	Press Save button	*Would you like to save data* dialog (Di2)	
3	Press Yes button	*Incorrect input* dialog (Di4)	
4	Press Ok button	Main dialog (Di1)	

2 Testing action flow

2.1 Test Cases for Component System

This example focuses on testing Lab Test Business Component System. In figure 7, there is Lab test business component system. At the process layer there is one business component, Lab test workflow. At the entity layer there are Lab test, Test result analyzer, Department and Patient business components. Utility layer business components include Lab test codebook and Addressbook. Database integrity manager and Performance monitor are auxiliary business components.

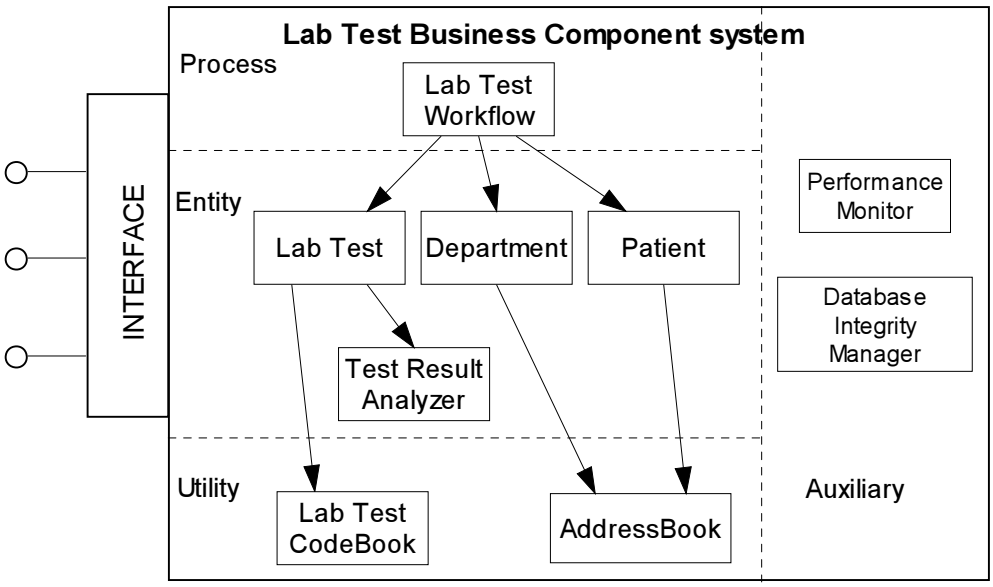


Figure 7. Lab Test BCS

Most important action flows has to be tested. The action flow describes functions between several people and several systems. Action flows are derived from requirement specification. The essential point is that one function follows another in right order. If in our example the patient has not been created in the system, it's not possible to insert medical data about patient to the system.

One action flow of Lab Test BCS is following:

- Create or choose a patient; (human and Patient BC)
- Examine patient
- Create a lab order; (human and Lab Test BC)
- Send the lab order to the lab; (human, Lab Test and Department BC)
- Reception
- Take a sample; (human)
- Analyze the sample and return results; (Test Result Analyzer BC)
- Derive reference values; (Lab Test Codebook BC)
- Save lab test results and reference values; (Lab Test BC)

This action flow is shown as a Use Case diagram in Figure 8.

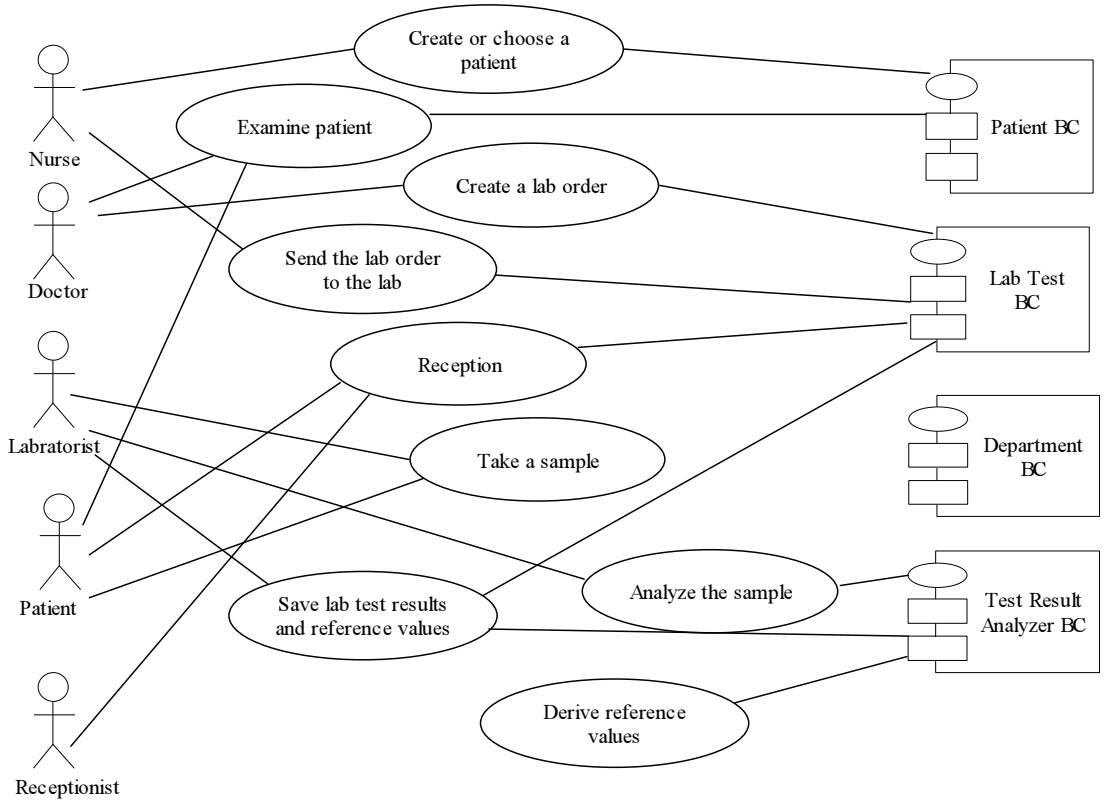


Figure 8: A Use Case diagram for Lab Test BCS

Following table shows a test case designed for this action flow. Test case is based on the action flow and use case diagram of Lab Test BCS. It is possible to utilize scenarios, too.

One test case for Lab Test BCS

Step	**Input**	**Expected outcome**	**Special Considerations**
1 Create patient	Create patient button	The basic information about patient updated	If patient exists, it won't be created
2a Examine patient	Search patient	Patient record on the screen	-
2b Examine patient	Examination data about patient	Patient record updated with examination data	Check that updated data exists in the database
3 Create a lab order	Choose lab tests	Lab order saved	Check that lab tests have been saved
4 Send the lab order to the lab	Reserve time from lab. Send the lab order	Time reserved. Lab order sent	Check that the lab order came to the right place
5 Reception	Choose a patient	Lab test order on the screen	Check the time reservation

Step	Input	Expected outcome	Special Considerations
6 Take a sample	-	-	-
7 Analyze the sample	-	-	Check that the analyser works correctly
8 Derive reference values	-	-	Check that the analyser works correctly
9 Save lab test results	Lab test results + Save button	Lab test results saved	Check that lab test results have been saved

Test cases for other action flows are derived similarly.

2.2 Testing Components

Next we consider as an example an operation sequence of Lab Test BC:

- Input a patient number;(human and user DC)
- Find lab test results with reference values;(enterprise and resource DC)
- Output lab test results with reference values;(user DC)
- Evaluate results;(human)
- Decide further actions;(human)
- Send lab test results and advice for further actions to the department;(human, user and enterprise DC)

The operation sequence is shown as a Use Case Diagram in Figure 9.

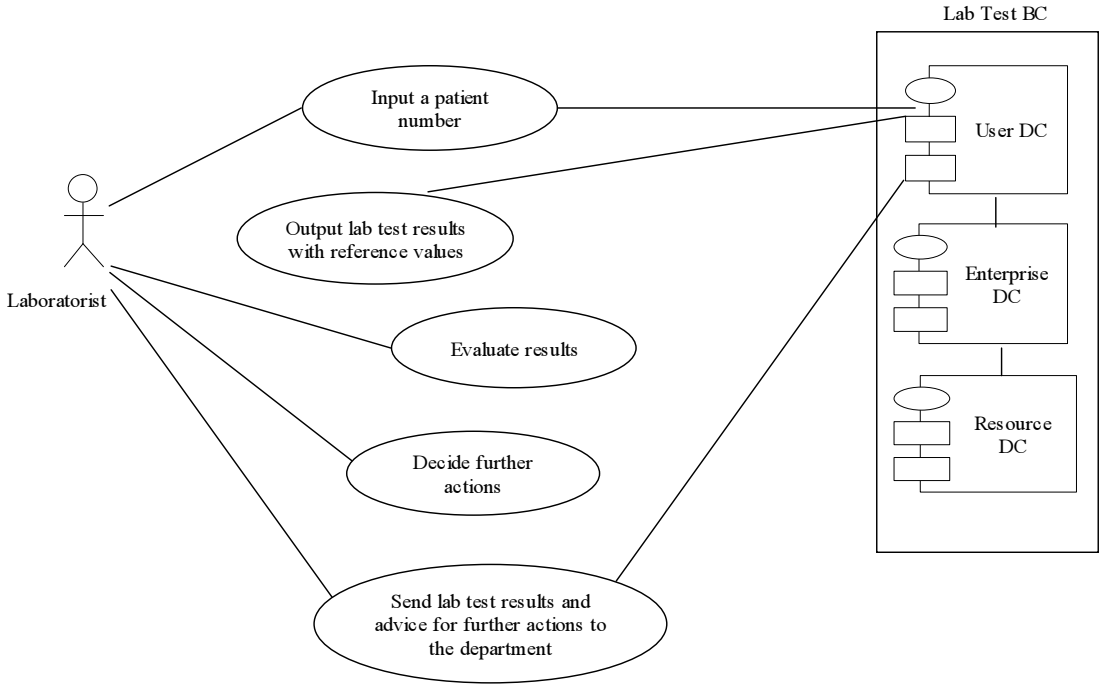


Figure 9: Save lab test results and reference values – a lower level description

This operation sequence can be tested with following test case:

One test case for Lab Test BC

Step	**Input**	**Expected outcome**	**Special Considerations**
1 Input a patient number	Patient number	Patient number has been entered	This step will be done together with step 2
2 Output lab test results…	Press Find button when patient number has been entered.	Lab test results with reference values on the screen	-
3 Evaluate results	-	-	-
4 Decide further actions	-	-	-
5 Send lab test results …	Lab test results + Send button	Lab test results sent to the department	Check that department received lab test results

At the lowest level we test the interfaces of the distributed component. This can be done in the same way like testing an application with one dialog (See chapter 3).

3 Test Report

A test report is a document that a tester follows while testing applications. At the beginning of the test report there is general information about the project and test cases, for example: project name, test case description and id, test level, test environment, test technique and referred documents, names of the tester and the person who fixes errors.

The test report table includes columns of test case table inserted with actual result, information about whether the test case was passed or failed, possible defect, severity of the defect, the date when the defect was fixed and person who fixed the defect.

Test Case Report was made for

- Patient Management application

Project:	TEHO	**Test Level:**	Functional
Test Case:	TC1 Insert Patient	**Environment:**	WinNT
Description:	Insert patient information to the database TC1.1 Correct saving TC1.2 Cancel saving TC1.3 Incorrect saving	**Technique:**	Equivalence Partitioning
		Referred Documents:	Requirements Specification 1.0: Activity Diagram
Author:	Marko Jäntti		
Date:	1.2.2002		

Id	Prerequisites	Test Input	Expected result	Actual result	Pass/Fail	Defect and severity*	Fixed (Date)
TC1.1 Step1	Main Dialog Displayed	Fill all text fields: Name = Sam, Personal ID = 041081-5678, Address = Wall Street 73 Symptoms= Head Ache	The text inserted by the user is visible in fields	Ok	P		
Step2	Text fields contain data	Press Save-button (Di1.B1)	*Would you like to save data* dialog (Di2)	Ok	P		
Step3	Di2 displayed	Press Cancel button in Di2	*Main dialog (Di1)*	Ok	P		

* Severity of defect: S1 = Critical, S2 = Major, S3 =Average, S4 = Minor

Id	Prerequisites	Test Input	Expected result	Actual result	Pass/Fail	Defect and Severity *	Fixed (Date)
TC1.2 Step1	Main Dialog displayed	Fill all text fields: Name = Sam, Personal ID = 041081-5678, Address = Unknown Symptoms= Head Ache	The text inserted by the user is visible in fields	Ok	P		
Step2	Fields contain valid data	Press Save-button (Di1.B1)	*Would you like to save data* dialog (Di2)	Ok	P		
Step3	Di2 displayed	Press Yes button in Di2	You have saved data dialog (Di3)	Ok	P		
Step 4		Press Ok in Di3	Main dialog (Di1)	Di3 still displayed	F	Button listener doesn't work, S3	DD 16.3

Special Considerations: TC 1.2 Step 3 Check whether patient with valid values exists in the database

* Severity of defect: S1 = Critical, S2 = Major, S3 =Average, S4 = Minor

Id	Prerequisites	Test Input	Expected result	Actual result	Pass/Fail	Defect and severity*	Fixed (Date)
TC1.3 Step1	Main Dialog displayed	Fill text fields with invalid values Name = empty Personal ID = 041081-5678, Address = Unknown Symptoms=""	The text inserted by the user is visible in fields (with errors)	Ok	P		
Step2	Fields contain invalid data	Press Save-button (Di1.B1)	*Incorrect input* dialog Di4	Dr.Watson Fatal error	F	System went down, S1	DD 15.3
Step3	Di4 displayed	Press Ok button in Di4	*Main dialog (Di1)*	Ok	P		

Special Considerations:

* Severity of defect: S1 = Critical, S2 = Major, S3 =Average, S4 = Minor